Flexible Constructor Bodies (Third Preview)

Changes to the Java® Language Specification • Version 24-internal-adhoc.gbierman.20241101

This document describes changes to the Java Language Specification ⇗ to support *Flexible Constructor Bodies*, which is a proposed feature of Java SE 24. See JEP 492 ⇗ for an overview of the feature.

Changes are described with respect to existing sections of the JLS. New text is indicated like this and deleted text is indicated ~~like this~~. Explanation and discussion, as needed, is set aside in grey boxes.

Changelog:

2024-11-01: First draft of the third preview. Only small bugfixes made, including:

- *Moved definition of early construction context from 8.8.7.1 to 8.8.7.*
- *Assignment to an instance variable in an early construction context is not permitted if it appears in a lambda expression or inner class declaration. (6.5.6.1, 15.8.3, 15.8.4)*
- *Unqualified and qualified `this` expressions treated the same with respect to early construction contexts. (15.8.3, 15.8.4)*
- *The runtime semantics of superclass constructor invocations erroneously contained compile-time calculation of the enclosing instance. This has now been moved. (8.8.7.1)*

Chapter 6: Names

6.5 Determining the Meaning of a Name

6.5.6 Meaning of Expression Names

6.5.6.1 Simple Expression Names

If an expression name consists of a single *Identifier*, then there must be exactly one declaration denoting either a local variable, formal parameter, exception parameter, or field in scope at the point at which the identifier occurs. Otherwise, a compile-time error occurs.

If the declaration denotes an instance variable of a class *C* (8.3.1.1 ⇗), then ~~both~~ all of the following must be true, or a compile-time error occurs:

- The expression name does not occur in a static context (8.1.3).
- If the expression name appears in an early construction context of *C* (8.8.7), then it is the left-hand operand of a simple assignment expression (15.26 ⇗), the declaration of the named variable lacks an initializer, and the simple assignment expression is not enclosed in a lambda expression or inner class declaration that is contained in the early construction context of *C*.
- If the expression name appears in a nested class or interface declaration of *C*, then the immediately enclosing class or interface declaration of the expression name is an inner class of *C*.

For example, the expression name must not appear in the body of a `static` method declared by C, nor in the body of an instance method of a `static` class nested within C.

If the declaration denotes a local variable, formal parameter, or exception parameter, let *X* be the innermost method declaration, constructor declaration, instance initializer, static initializer,

field declaration, or explicit constructor invocation ~~statement~~ which encloses the local variable or parameter declaration. If the expression name appears directly or indirectly in the body of a local class, local interface, or anonymous class *D* declared directly in *X*, then both of the following must be true, or a compile-time error occurs:

- The expression name does not occur in a static context.
- *D* is an inner class, and the immediately enclosing class or interface declaration of the expression name is *D* or an inner class of *D*.

For example, the expression name must not appear in the body of a `static` method declared by D, nor (if D is a local interface) in the body of a default method of D.

If the declaration denotes a local variable, formal parameter, or exception parameter that is neither `final` nor effectively final (4.12.4 ⇗), it is a compile-time error if the expression name appears either in an inner class enclosed directly or indirectly by *X*, or in a lambda expression contained by *X* (15.27 ⇗).

The net effect of these rules is that a local variable, formal parameter, or exception parameter can only be referenced from a nested class or interface declared within its scope if (i) the reference is not within a static context, (ii) there is a chain of inner (non-`static`) classes from the reference to the variable declaration, and (iii) the variable is `final` or effectively final. References from lambda expressions also require the variable to be `final` or effectively final.

If the declaration declares a `final` variable which is definitely assigned before the simple expression, the meaning of the name is the value of that variable. Otherwise, the meaning of the expression name is the variable declared by the declaration.

If the expression name appears in an assignment context, invocation context, or casting context, then the type of the expression name is the declared type of the field, local variable, or parameter after capture conversion (5.1.10 ⇗).

Otherwise, the type of the expression name is the declared type of the field, local variable or parameter.

That is, if the expression name appears "on the right hand side", its type is subject to capture conversion. If the expression name is a variable that appears "on the left hand side", its type is not subject to capture conversion.

Example 6.5.6.1-1. Simple Expression Names

```
class Test {
    static int v;
    static final int f = 3;
    public static void main(String[] args) {
        int i;
        i = 1;
        v = 2;
        f = 33;  // compile-time error
        System.out.println(i + " " + v + " " + f);
    }
}
```

In this program, the names used as the left-hand-sides in the assignments to `i`, `v`, and `f` denote the local variable `i`, the field `v`, and the value of `f` (not the variable `f`, because `f` is a `final` variable). The example therefore produces an error at compile time because the last assignment does not have a variable as its left-hand side. If the erroneous assignment is removed, the modified code can be compiled and it will produce the output:

```
1 2 3
```

Example 6.5.6.1-2. References to Instance Variables

```
class Test {
    static String a;
    String b;

    String concat1() {
        return a + b;
    }

    static String concat2() {
        return a + b;  // compile-time error
    }

    int index() {
        interface I {
            class Matcher {
                void check() {
                    if (a == null ||
                        b == null) {  // compile-time error
                        throw new IllegalArgumentException();
                    }
                }
                int match(String s, String t) {
                    return s.indexOf(t);
                }
            }
    }

    I.Matcher matcher = new I.Matcher();
    matcher.check();
    return matcher.match(a, b);
    }
}
```

The fields `a` and `b` are in scope throughout the body of class `Test`. However, using the name `b` in the static context of the `concat2` method, or in the declaration of the nested class `Matcher` that is not an inner class of `Test`, is illegal.

Example 6.5.6.1-3. References to Local Variables and Formal Parameters

```
class Test {
    public static void main(String[] args) {
        String first = args[0];

        class Checker {
            void checkWhitespace(int x) {
                String arg = args[x];
                if (!arg.trim().equals(arg)) {
                    throw new IllegalArgumentException();
                }
            }

            static void checkFlag(int x) {
                String arg = args[x];  // compile-time error
                if (!arg.startsWith("-")) {
```

```
                        throw new IllegalArgumentException();
                    }
                }

                static void checkFirst() {
                    Runnable r = new Runnable() {
                        public void run() {
                            if (first == null) {  // compile-time error
                                throw new IllegalArgumentException();
                            }
                        }
                    };
                    r.run();
                }
            }

            final Checker c = new Checker();
            c.checkFirst();
            for (int i = 1; i < args.length; i++) {
                Runnable r = () -> {
                    c.checkWhitespace(i);  // compile-time error
                    c.checkFlag(i);  // compile-time error
                };
            }
        }
    }
```

The formal parameter `args` is in scope throughout the body of method `main`. `args` is effectively final, so the name `args` can be used in the instance method `checkWhitespace` of local class `Checker`. However, using the name `args` in the static context of the `checkFlag` method of local class `Checker` is illegal.

The local variable `first` is in scope for the remainder of the body of method `main`. `first` is also effectively final. However, the anonymous class declared in `checkFirst` is not an inner class of `Checker`, so using the name `first` in the anonymous class body is illegal. (A lambda expression in the body of `checkFirst` would similarly be unable to refer to `first`, because the lambda expression would occur in a static context.)

The local variable `c` is in scope for the last few lines of the body of method `main`, and is declared `final`, so the name `c` can be used in the body of the lambda expression.

The local variable `i` is in scope throughout the `for` loop. However, `i` is not effectively final, so using the name `i` in the body of the lambda expression is illegal.

6.5.7 Meaning of Method Names

6.5.7.1 Simple Method Names

A simple method name appears in the context of a method invocation expression (15.12 ⇗). The simple method name consists of a single *UnqualifiedMethodIdentifier* which specifies the name of the method to be invoked. The rules of method invocation require that the *UnqualifiedMethodIdentifier* denotes a method that is in scope at the point of the method invocation. ~~The rules also prohibit (15.12.3) a reference to an instance method occurring in a static context (8.1.3), or in a nested class or interface other than an inner class of the class or interface which declares the instance method.~~ The rules (15.12.3) also prohibit a reference to an instance method occurring in any one of the following:

1. a static context (8.1.3),

2. a nested class or interface other than an inner class of the innermost class or interface of which the instance method is a member, or
3. an early construction context (8.8.7) of a class where the instance method is a member.

Editorial: The struck-out sentence above offered an incorrect summary of the rules appearing in (15.12.3). Whilst it correctly asserted that the following is prohibited:

```
class T {
    void foo() { ... }
    static class Mid {
        class L {
            ... foo() ... // Compile-time error as L is not an inner class
of T
        }
    }
}
```

It suggested that the following code is not allowed by the rules in (15.12.3), although it is.

```
class B {
    void foo() { ... }
}
class T extends B {
    class Mid {
        class L {
            ... foo() ... // No compile-time error even though L is not an
inner class of B
        }
    }
 }
```

Example 6.5.7.1-1. Simple Method Names

The following program demonstrates the role of scoping when determining which method to invoke.

```
    class Super {
        void f2(String s)       {}
        void f3(String s)       {}
        void f3(int i1, int i2) {}
    }

    class Test {
        void f1(int i) {}
        void f2(int i) {}
        void f3(int i) {}

        void m() {
            new Super() {
                {
                    f1(0);  // OK, resolves to Test.f1(int)
                    f2(0);  // compile-time error
                    f3(0);  // compile-time error
                }
```

```
                };
            }
        }
```

For the invocation `f1(0)`, only one method named `f1` is in scope. It is the method `Test.f1(int)`, whose declaration is in scope throughout the body of `Test` including the anonymous class declaration. 15.12.1 chooses to search in class `Test` since the anonymous class declaration has no member named `f1`. Eventually, `Test.f1(int)` is resolved.

For the invocation `f2(0)`, two methods named `f2` are in scope. First, the declaration of the method `Super.f2(String)` is in scope throughout the anonymous class declaration. Second, the declaration of the method `Test.f2(int)` is in scope throughout the body of `Test` including the anonymous class declaration. (Note that neither declaration shadows the other, because at the point where each is declared, the other is not in scope.) 15.12.1 chooses to search in class `Super` because it has a member named `f2`. However, `Super.f2(String)` is not applicable to `f2(0)`, so a compile-time error occurs. Note that class `Test` is not searched.

For the invocation `f3(0)`, three methods named `f3` are in scope. First and second, the declarations of the methods `Super.f3(String)` and `Super.f3(int,int)` are in scope throughout the anonymous class declaration. Third, the declaration of the method `Test.f3(int)` is in scope throughout the body of `Test` including the anonymous class declaration. 15.12.1 chooses to search in class `Super` because it has a member named `f3`. However, `Super.f3(String)` and `Super.f3(int,int)` are not applicable to `f3(0)`, so a compile-time error occurs. Note that class `Test` is not searched.

Choosing to search a nested class's superclass hierarchy before the lexically enclosing scope is called the "comb rule" (15.12.1).

Chapter 8: Classes

A class declaration defines a new class and describes how it is implemented (8.1).

A *top level class* (7.6) is a class declared directly in a compilation unit.

A *nested class* is any class whose declaration occurs within the body of another class or interface declaration. A nested class may be a member class (8.5, 9.5), a local class (14.3), or an anonymous class (15.9.5).

~~Some kinds of nested class are an *inner class* (8.1.3), which is a class that can refer to enclosing class instances, local variables, and type variables.~~

A nested class may be *inner* (8.1.3), in which case it may (depending on precisely where its declaration occurs) be able to refer to enclosing class instances, local variables, and type variables.

An *enum class* (8.9) is a class declared with abbreviated syntax that defines a small set of named class instances.

A *record class* (8.10) is a class declared with abbreviated syntax that defines a simple aggregate of values.

This chapter discusses the common semantics of all classes. Details that are specific to particular kinds of classes are discussed in the sections dedicated to these constructs.

A class may be declared `public` (8.1.1) so it can be referred to from code in any package of its module and potentially from code in other modules.

A class may be declared `abstract` (8.1.1.1), and must be declared `abstract` if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. The degree to which a class can be extended can be controlled explicitly

(8.1.1.2 ⇗): it may be declared `sealed` to limit its subclasses, or it may be declared `final` to ensure no subclasses. Each class except `Object` is an extension of (that is, a subclass of) a single existing class (8.1.4 ⇗) and may implement interfaces (8.1.5 ⇗).

A class may be *generic* (8.1.2 ⇗), that is, its declaration may introduce type variables whose bindings differ among different instances of the class.

Class declarations may be decorated with annotations (9.7 ⇗) just like any other kind of declaration.

The body of a class declares members (fields, methods, classes, and interfaces), instance and static initializers, and constructors (8.1.7 ⇗). The scope (6.3 ⇗) of a member (8.2 ⇗) is the entire body of the declaration of the class to which the member belongs. Field, method, member class, member interface, and constructor declarations may include the access modifiers `public`, `protected`, or `private` (6.6 ⇗). The members of a class include both declared and inherited members (8.2 ⇗). Newly declared fields can hide fields declared in a superclass or superinterface. Newly declared member classes and member interfaces can hide member classes and member interfaces declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

Field declarations (8.3 ⇗) describe class variables, which are incarnated once, and instance variables, which are freshly incarnated for each instance of the class. A field may be declared `final` (8.3.1.2 ⇗), in which case it can be assigned to only once. Any field declaration may include an initializer.

Member class declarations (8.5 ⇗) describe nested classes that are members of the surrounding class. Member classes may be `static`, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes.

Member interface declarations (8.5 ⇗) describe nested interfaces that are members of the surrounding class.

Method declarations (8.4 ⇗) describe code that may be invoked by method invocation expressions (15.12 ⇗). A class method is invoked relative to the class; an instance method is invoked with respect to some particular object that is an instance of a class. A method whose declaration does not indicate how it is implemented must be declared `abstract`. A method may be declared `final` (8.4.3.3 ⇗), in which case it cannot be hidden or overridden. A method may be implemented by platform-dependent `native` code (8.4.3.4 ⇗). A `synchronized` method (8.4.3.6 ⇗) automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a `synchronized` statement (14.19 ⇗), thus allowing its activities to be synchronized with those of other threads (17 ⇗).

Method names may be overloaded (8.4.9 ⇗).

Instance initializers (8.6 ⇗) are blocks of executable code that may be used to help initialize an instance when it is created (15.9 ⇗).

Static initializers (8.7 ⇗) are blocks of executable code that may be used to help initialize a class.

Constructors (8.8 ⇗) are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded (8.8.8 ⇗).

8.1 Class Declarations

8.1.3 Inner Classes and Enclosing Instances

An *inner class* is a nested class that is not explicitly or implicitly `static`.

An inner class is one of the following:

- a member class that is not explicitly or implicitly `static` (8.5 ⇗)
- a local class that is not implicitly `static` (14.3 ⇗)
- an anonymous class (15.9.5 ⇗)

The following nested classes are implicitly `static`, so are not inner classes:

- a member enum class (8.9 ⇗)
- a local enum class (14.3 ⇗)
- a member record class (8.10 ⇗)
- a local record class (14.3 ⇗)
- a member class of an interface (9.5 ⇗)

All of the rules that apply to nested classes apply to inner classes. In particular, an inner class may declare and inherit `static` members (8.2 ⇗), and declare static initializers (8.7 ⇗), even though the inner class itself is not `static`.

There are no "inner interfaces" because every nested interface is implicitly `static` (9.1.1.3 ⇗).

Example 8.1.3-1. Inner Class Declarations and Static Members

```
class HasStatic {
    static int j = 100;
}

class Outer {
    class Inner extends HasStatic {
        static {
            System.out.println("Hello from Outer.Inner");
        }

        static       int x = 3;
        static final int y = 4;

        static void hello() {
            System.out.println("Hello from Outer.Inner.hello");
        }

        static class VeryNestedButNotInner
            extends NestedButNotInner {}
    }

    static class NestedButNotInner {
        int z = Inner.x;
    }

    interface NeverInner {}  // Implicitly static, so never inner
}
```

Prior to Java SE 16, an inner class could not declare static initializers, and could only declare `static` members that were constant variables (4.12.4 ⇗).

A construct (statement, local variable declaration statement, local class declaration, local

interface declaration, or expression) *occurs in a static context* if the innermost:

- method declaration,
- field declaration,
- constructor declaration,
- instance initializer, or
- static initializer~~, or~~
- ~~explicit constructor invocation statement~~

which encloses the construct is one of the following:

- a `static` method declaration (8.4.3.2, 9.4)
- a `static` field declaration (8.3.1.1, 9.3)
- a static initializer (8.7)
- ~~an explicit constructor invocation statement (8.8.7.1)~~

Note that a construct which appears in a constructor declaration or an instance initializer does not occur in a static context.

The purpose of a static context is to demarcate code ~~that must not refer explicitly or implicitly to the current instance of the class whose declaration lexically encloses the static context.~~ for which there is no current instance defined of the class whose declaration lexically encloses the static context. Consequently, code that occurs in a static context is restricted in the following ways:

- *`this` expressions (both unqualified and qualified) are disallowed (15.8.3, 15.8.4).*
- *Field accesses, method invocations, and method references may not be qualified by `super` (15.11.2, 15.12.3, 15.13.1).*
- *Unqualified references to instance variables of any lexically enclosing class or interface declaration are disallowed (6.5.6.1).*
- *Unqualified invocations of instance methods of any lexically enclosing class or interface declaration are disallowed (15.12.3).*
- *References to type parameters of any lexically enclosing class or interface declarations are disallowed (6.5.5.1).*
- *References to type parameters, local variables, formal parameters, and exception parameters declared by methods or constructors of any lexically enclosing class or interface declaration that is outside the immediately enclosing class or interface declaration are disallowed (6.5.5.1, 6.5.6.1).*
- *Declarations of local normal classes (as opposed to local enum classes) and declarations of anonymous classes both specify classes that are inner, yet when instantiated have no immediately enclosing instances (15.9.2).*
- *Class instance creation expressions that instantiate inner member classes must be qualified (15.9.2).*

An inner class *C* is a *direct inner class of a class or interface O* if *O* is the immediately enclosing class or interface declaration of *C* and the declaration of *C* does not occur in a static context.

If an inner class is a local class or an anonymous class, it may be declared in a static context, and in that case is not considered an inner class of any enclosing class or interface.

A class *C* is an *inner class of class or interface O* if it is either a direct inner class of *O* or an

inner class of an inner class of *O*.

> *It is unusual, but possible, for the immediately enclosing class or interface declaration of an inner class to be an interface. This only occurs if the class is a local or anonymous class declared in a* `default` *or* `static` *method body (9.4 ⇗).*

A class or interface *O* is the *zeroth lexically enclosing class or interface declaration of itself*.

A class *O* is the *n'th lexically enclosing class declaration of a class C* if it is the immediately enclosing class declaration of the *n-1*'th lexically enclosing class declaration of *C*.

An instance *i* of a direct inner class *C* of a class or interface *O* ~~is~~ may be associated with an instance of *O*, known as the *immediately enclosing instance of i*. The immediately enclosing instance of an object, if any, is determined when the object is created (15.9.2).

An object *o* is the *zeroth lexically enclosing instance of itself*.

An object *o* is the *n'th lexically enclosing instance of an instance i* if it is the immediately enclosing instance of the *n-1*'th lexically enclosing instance of *i*.

An instance of an inner local class or an anonymous class whose declaration occurs in a static context has no immediately enclosing instance. Also, an instance of a `static` nested class (8.1.1.4 ⇗) has no immediately enclosing instance.

> *Editorial: Note that in the existing JLS, where static context is used to also cover what we now call an early constructor context, this means that, for example, an anonymous class declared as an argument to an explicit constructor invocation is considered to appear in a static context. For example:*
>
> ```
> class Outer {
>     int x;
>     class Inner extends Thread {
>         Inner() {
>             super(new Runnable() {
>                 @Override
>                 public void run() {
>                     System.out.println("hello from " + Outer.this.x); //
> error??
>                 }
>             });
>         }
>     }
> }
> ```
>
> *The existing JLS is unclear whether the reference* `Outer.this.x` *is valid, as the anonymous class does not have an immediately enclosing instance. (The reference compiler accepts this code.)*
>
> *Local and anonymous classes that occur in an early construction context are now defined to always have an immediately enclosing instance. Rules specify what can be referenced from the early construction context of a particular class. In the example above, the anonymous class is not in the early construction context of the class* `Outer` *and so the field* `x` *can be referenced.*

For every superclass *S* of *C* which is itself a direct inner class of a class or interface *SO*, there is an instance of *SO* associated with *i*, known as the *immediately enclosing instance of i with*

respect to S. The immediately enclosing instance of an object with respect to its class' direct superclass, if any, is determined when the superclass constructor is invoked via an explicit constructor invocation ~~statement~~ (8.8.7.1).

~~When an inner class (whose declaration does not occur in a static context) refers to an instance variable that is a member of a lexically enclosing class or interface declaration, the variable of the corresponding lexically enclosing instance is used.~~

When an inner class contains a valid reference to an instance variable that is a member of a lexically enclosing class or interface declaration, the variable of the corresponding lexically enclosing instance is used.

Any local variable, formal parameter, or exception parameter used but not declared in an inner class must either be `final` or effectively final (4.12.4 ⇗), as specified in 6.5.6.1.

Any local variable used but not declared in an inner class must be definitely assigned (16) before the body of the inner class, or a compile-time error occurs.

Similar rules on variable use apply in the body of a lambda expression (15.27.2 ⇗).

A blank `final` field (4.12.4 ⇗) of a lexically enclosing class or interface declaration may not be assigned within an inner class, or a compile-time error occurs.

Example 8.1.3-2. Inner Class Declarations

```
class Outer {
    int i = 100;
    static void classMethod() {
        final int l = 200;
        class LocalInStaticContext {
            int k = i;  // Compile-time error
            int m = l;  // OK
        }
    }
    void foo() {
        class Local {  // A local class
            int j = i;
        }
    }
}
```

The declaration of class `LocalInStaticContext` occurs in a static context due to being within the static method `classMethod`. Instance variables of class `Outer` are not available within the body of a static method. In particular, instance variables of `Outer` are not available inside the body of `LocalInStaticContext`. However, local variables from the surrounding method may be referred to without error (provided they are declared `final` or are effectively final).

Inner classes whose declarations do not occur in a static context may freely refer to the instance variables of their enclosing class declaration. An instance variable is always defined with respect to an instance. In the case of instance variables of an enclosing class declaration, the instance variable must be defined with respect to an enclosing instance of the inner class. For example, the class `Local` above has an enclosing instance of class `Outer`. As a further example:

```
class WithDeepNesting {
    boolean toBe;
    WithDeepNesting(boolean b) { toBe = b; }

    class Nested {
        boolean theQuestion;
```

```
            class DeeplyNested {
                DeeplyNested(){
                    theQuestion = toBe || !toBe;
                }
            }
        }
    }
```

Here, every instance of `WithDeepNesting.Nested.DeeplyNested` has an enclosing instance of class `WithDeepNesting.Nested` (its immediately enclosing instance) and an enclosing instance of class `WithDeepNesting` (its 2nd lexically enclosing instance).

8.8 Constructor Declarations

8.8.7 Constructor Body

A constructor body is a block of code that is executed as part of the process of creating a new instance of a class (12.5). A constructor body may contain an explicit invocation of another constructor of the same class or of the direct superclass (8.8.7.1).

~~The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass (8.8.7.1).~~

ConstructorBody:

~~{ [*ExplicitConstructorInvocation*] [*BlockStatements*] }~~

{ [*BlockStatements*] }

{ [*BlockStatements*] *ExplicitConstructorInvocation* [*BlockStatements*] }

~~It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving `this`.~~

> *Editorial: The paragraph above has been moved to 8.8.7.1*

If a constructor body contains an explicit constructor invocation, the *BlockStatements* preceding the explicit constructor invocation are called the *prologue* of the constructor body. The prologue of a constructor body may be empty. The *BlockStatements* in a constructor with no explicit constructor invocation and the *BlockStatements* following an explicit constructor invocation in a constructor body are called the *epilogue*. The epilogue of a constructor body may also be empty.

An expression *occurs in the early construction context of a class C* if it is contained in either the prologue of a constructor body of *C*, or it is nested in the explicit constructor invocation (8.8.7.1) of a constructor body of *C*.

If a constructor body does not ~~begin with~~ contain an explicit constructor invocation and the constructor being declared is not part of the primordial class `Object`, then the constructor body implicitly begins with a superclass constructor invocation "`super();`", an invocation of the constructor of ~~its~~ the direct superclass that takes no arguments.

Except for the possibility of explicit or implicit constructor invocations, and the ~~prohibition on explicitly returning a value~~ prohibitions on `return` statements (14.17), the body of a constructor is like the body of a method (8.4.7 ⇗).

> *Note that a constructor body contains at most one explicit constructor invocation. The grammar makes it impossible, for example, to place explicit constructor invocations in different branches of an `if` statement.*

~~A `return` statement (14.17) may be used in the body of a constructor if it does not include an expression.~~

Example 8.8.7-1. Constructor Bodies

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, WHITE);
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```

Here, the first constructor of `ColoredPoint` invokes the second, providing an additional argument; the second constructor of `ColoredPoint` invokes the constructor of its superclass `Point`, passing along the coordinates.

8.8.7.1 Explicit Constructor Invocations

ExplicitConstructorInvocation:
[*TypeArguments*] `this` `(` [*ArgumentList*] `)` `;`
[*TypeArguments*] `super` `(` [*ArgumentList*] `)` `;`
ExpressionName `.` [*TypeArguments*] `super` `(` [*ArgumentList*] `)` `;`
Primary `.` [*TypeArguments*] `super` `(` [*ArgumentList*] `)` `;`

The following productions from 4.5.1 ⇗ and 15.12 ⇗ are shown here for convenience:

TypeArguments:
`<` *TypeArgumentList* `>`

ArgumentList:
Expression {`,` *Expression*}

Explicit constructor ~~invocation statements~~ invocations are divided into two kinds:

- *Alternate constructor invocations* begin with the keyword `this` (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.
- *Superclass constructor invocations* begin with either the keyword `super` (possibly prefaced with explicit type arguments) or a *Primary* expression or an *ExpressionName*. They are used to invoke a constructor of the direct superclass. They are further divided:
    - *Unqualified superclass constructor invocations* begin with the keyword `super` (possibly prefaced with explicit type arguments).
    - *Qualified superclass constructor invocations* begin with a *Primary* expression or an *ExpressionName*. They allow a subclass constructor to explicitly specify the newly created object's immediately enclosing instance with respect to the direct

superclass (8.1.3). This may be necessary when the superclass is an inner class.

It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more alternate constructor invocations.

~~An explicit constructor invocation statement introduces a static context (8.1.3), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context (15.8.3, 15.11.2), as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations (6.5.5.1, 6.5.6.1, 15.12.3).~~

An explicit constructor invocation introduces an early construction context (8.8.7), which limits the use of constructs that refer to the current object. Notably, references to the current object using `this` and `super` are restricted in an early construction context (15.8.3, 15.11.2), as are references to instance variables, and instance methods (6.5.6.1, 6.5.7.1).

If *TypeArguments* is present to the left of `this` or `super`, then it is a compile-time error if any of the type arguments are wildcards (4.5.1).

The rules for a superclass constructor invocation, where ~~Let~~ *C* ~~be~~ is the class being instantiated, and ~~let~~ *S* ~~be~~ is the direct superclass of *C*, are as follows:

- If a superclass constructor invocation ~~statement~~ is unqualified, then:
  - If *S* is an inner member class, but *S* is not a member of a class enclosing *C*, then a compile-time error occurs.

    Otherwise, let *O* be the innermost enclosing class of *C* of which *S* is a member. *C* must be an inner class of *O* (8.1.3), or a compile-time error occurs. If the superclass constructor invocation occurs in an early construction context of the class *O*, then a compile-time error occurs.
  - If *S* is an inner local class, and *S* does not occur in a static context, let *O* be the immediately enclosing class or interface declaration of *S*. *C* must be an inner class of *O*, or a compile-time error occurs.
- If a superclass constructor invocation ~~statement~~ is qualified, then:
  - If *S* is not an inner class, or if the declaration of *S* occurs in a static context, then a compile-time error occurs.
  - Otherwise, let *p* be the *Primary* expression or the *ExpressionName* immediately preceding "`.super`", and let *O* be the immediately enclosing class of *S*. It is a compile-time error if the type of *p* is not *O* or a subclass of *O*, or if the type of *p* is not accessible (6.6).

The exception types that an explicit constructor invocation ~~statement~~ can throw are specified in 11.2.2.

Let *i* be the instance being created by a superclass constructor invocation. If *S* is an inner class, then *i* may have an immediately enclosing instance with respect to *S*, determined as follows:

- If *S* is not an inner class, or if the declaration of *S* occurs in a static context, then no immediately enclosing instance of *i* with respect to *S* exists. Let *N* be the nearest `static` method declaration, `static` field declaration, or static initializer that encloses the declaration of *S*. If *S* is a local class and the nearest `static` method declaration, `static`

field declaration, or static initializer that encloses the class instance creation expression is not *N*, then a compile-time error occurs.

- Otherwise, if the superclass constructor invocation is unqualified, then *S* is necessarily an inner local class or an inner member class.

  If *S* is an inner local class, let *O* be the immediately enclosing class or interface declaration of *S*.

  If *S* is an inner member class, let *O* be the innermost enclosing class of *C* of which *S* is a member. If the superclass constructor invocation occurs in the early construction context of the class *O*, then a compile-time error occurs.

  Let *n* be an integer ($n \geq 1$) such that *O* is the *n*'th lexically enclosing class or interface declaration of *C*.

  The immediately enclosing instance of *i* with respect to *S* is the *n*'th lexically enclosing instance of `this`.

  > *While it may be the case that S is a member of C due to inheritance, the zeroth lexically enclosing instance of `this` (that is, `this` itself) is never used as the immediately enclosing instance of i with respect to S.*

- Otherwise, if the superclass constructor invocation is qualified, then the immediately enclosing instance of *i* with respect to *S* is the object that is the value of the *Primary* expression or the *ExpressionName*.

If the superclass constructor invocation has an immediately enclosing instance then this instance is taken to be the first actual argument to the constructor invocation, and the subsequent actual arguments to the constructor invocation are taken to be the arguments in the argument list of superclass constructor invocation, if any, in the order that they appear. Otherwise the actual arguments to the constructor invocation are taken to be the arguments in the superclass constructor invocation, if any, in the order that they appear.

Evaluation of an alternate constructor invocation ~~statement~~ proceeds by first evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.

Evaluation of a superclass constructor invocation proceeds as follows:

1. ~~Let *i* be the instance being created. The immediately enclosing instance of *i* with respect to *S* (if any) must be determined:~~
   - ~~If *S* is not an inner class, or if the declaration of *S* occurs in a static context, then no immediately enclosing instance of *i* with respect to *S* exists.~~
   - ~~Otherwise, if the superclass constructor invocation is unqualified, then *S* is necessarily an inner local class or an inner member class.~~

     ~~If *S* is an inner local class, let *O* be the immediately enclosing class or interface declaration of *S*.~~

     ~~If *S* is an inner member class, let *O* be the innermost enclosing class of *C* of which *S* is a member.~~

     ~~Let *n* be an integer ($n \geq 1$) such that *O* is the *n*'th lexically enclosing class or interface declaration of *C*.~~

     ~~The immediately enclosing instance of *i* with respect to *S* is the *n*'th lexically enclosing instance of `this`.~~

~~*While it may be the case that S is a member of C due to inheritance, the zeroth lexically enclosing instance of `this` (that is, `this` itself) is never used as the immediately enclosing instance of i with respect to S.*~~

- ~~Otherwise, if the superclass constructor invocation is qualified, then the *Primary* expression or the *ExpressionName* immediately preceding "`.super`", *p*, is evaluated.~~

  ~~If *p* evaluates to `null`, a `NullPointerException` is raised, and the superclass constructor invocation completes abruptly.~~

  ~~Otherwise, the result of this evaluation is the immediately enclosing instance of *i* with respect to *S*.~~

2. ~~After determining the immediately enclosing instance of *i* with respect to *S* (if any), evaluation of the superclass constructor invocation statement proceeds by evaluating the arguments to the constructor, left-to-right, as in an ordinary method invocation; and then invoking the constructor.~~
3. ~~Finally, if the superclass constructor invocation statement completes normally, then all instance variable initializers of *C* and all instance initializers of *C* are executed. If an instance initializer or instance variable initializer *I* textually precedes another instance initializer or instance variable initializer *J*, then *I* is executed before *J*.~~

   ~~Execution of instance variable initializers and instance initializers is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation statement or is provided implicitly. (An alternate constructor invocation does not perform this additional implicit execution.)~~

Editorial: Evaluation of a superclass constructor invocation should not include calculation of an enclosing instance; this clearly takes place at compile-time.

1. First, if the superclass constructor invocation is qualified, the qualifying primary expression is evaluated. If the qualifying primary expression evaluates to `null`, a `NullPointerException` is raised and the superclass constructor invocation completes abruptly. If the qualifying expression completes abruptly, then superclass constructor invocation completes abruptly for the same reason.
2. The actual arguments to the constructor invocation are evaluated, left-to-right, as in an ordinary method invocation; then the constructor is invoked.

   Recall that the first actual argument to the constructor invocation may be an enclosing instance.

3. Finally, if the superclass constructor invocation completes normally, then all instance variable initializers of *C* and all instance initializers of *C* are executed. If an instance initializer or instance variable initializer *I* textually precedes another instance initializer or instance variable initializer *J*, then *I* is executed before *J*.

   Execution of instance variable initializers and instance initializers is performed regardless of whether the superclass constructor invocation actually appears as an explicit constructor invocation or is provided implicitly. (An alternate constructor invocation does not perform this additional implicit execution.)

Example 8.8.7.1-1. Restrictions on Explicit Constructor ~~Invocation Statements~~ Invocations

If the first constructor of `ColoredPoint` in the example from 8.8.7 were changed as follows:

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
    static final int WHITE = 0, BLACK = 1;
    int color;
    ColoredPoint(int x, int y) {
        this(x, y, color);  // Changed to color from WHITE
    }
    ColoredPoint(int x, int y, int color) {
        super(x, y);
        this.color = color;
    }
}
```

then a compile-time error would occur, because the instance variable `color` cannot be used by a explicit constructor invocation ~~statement~~.

Example 8.8.7.1-2. Qualified Superclass Constructor Invocation

In the code below, `ChildOfInner` has no lexically enclosing class or interface declaration, so an instance of `ChildOfInner` has no enclosing instance. However, the superclass of `ChildOfInner` (`Inner`) has a lexically enclosing class declaration (`Outer`), and an instance of `Inner` must have an enclosing instance of `Outer`. The enclosing instance of `Outer` is set when an instance of `Inner` is created. Therefore, when we create an instance of `ChildOfInner`, which is implicitly an instance of `Inner`, we must provide the enclosing instance of `Outer` via a qualified superclass invocation ~~statement~~ in `ChildOfInner`'s constructor. The instance of `Outer` is called the immediately enclosing instance of `ChildOfInner` with respect to `Inner`.

```
class Outer {
    class Inner {}
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner() { (new Outer()).super(); }
}
```

Perhaps surprisingly, the same instance of `Outer` may serve as the immediately enclosing instance of `ChildOfInner` with respect to `Inner` for multiple instances of `ChildOfInner`. These instances of `ChildOfInner` are implicitly linked to the same instance of `Outer`. The program below achieves this by passing an instance of `Outer` to the constructor of `ChildOfInner`, which uses the instance in a qualified superclass constructor invocation ~~statement~~. The rules for an explicit constructor invocation ~~statement~~ do not prohibit using formal parameters of the constructor that contains the ~~statement~~ invocation.

```
class Outer {
    int secret = 5;
    class Inner {
        int  getSecret()      { return secret; }
        void setSecret(int s) { secret = s; }
    }
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner(Outer x) { x.super(); }
}

public class Test {
    public static void main(String[] args) {
```

```
            Outer x = new Outer();
            ChildOfInner a = new ChildOfInner(x);
            ChildOfInner b = new ChildOfInner(x);
            System.out.println(b.getSecret());
            a.setSecret(6);
            System.out.println(b.getSecret());
        }
    }
```

This program produces the output:

```
5
6
```

The effect is that manipulation of instance variables in the common instance of `Outer` is visible through references to different instances of `ChildOfInner`, even though such references are not aliases in the conventional sense.

8.10 Record Classes

8.10.4 Record Constructor Declarations

To ensure proper initialization of its record components, a record class does not implicitly declare a default constructor (8.8.9 ⇗). Instead, a record class has a *canonical constructor*, declared explicitly or implicitly, that initializes all the component fields of the record class.

There are two ways to explicitly declare a canonical constructor in a record declaration: by declaring a normal constructor with a suitable signature (8.10.4.1 ⇗) or by declaring a compact constructor (8.10.4.2 ⇗).

> *Given the signature of a normal constructor that qualifies as canonical, and the signature derived for a compact constructor, the rules of constructor signatures (8.8.2 ⇗) mean it is a compile-time error if a record declaration has both a normal constructor that qualifies as canonical and a compact constructor.*

Either way, an explicitly declared canonical constructor must provide at least as much access as the record class, as follows:

- If the record class is `public`, then the canonical constructor must be `public`; otherwise, a compile-time error occurs.
- If the record class is `protected`, then the canonical constructor must be `protected` or `public`; otherwise, a compile-time error occurs.
- If the record class has package access, then the canonical constructor must *not* be `private`; otherwise, a compile-time error occurs.
- If the record class is `private`, then the canonical constructor may be declared with any accessibility.

An explicitly declared canonical constructor may be a fixed arity constructor or a variable arity constructor (8.8.1 ⇗).

If a canonical constructor is not explicitly declared in the declaration of a record class *R*, then a canonical constructor *r* is implicitly declared in *R* with the following properties:

- The signature of *r* has no type parameters, and has formal parameters given by the derived formal parameter list of *R*, defined below.
- *r* has the same access modifier as *R*, unless *R* lacks an access modifier, in which case *r*

has package access.

- *r* has no `throws` clause.
- The body of *r* initializes each component field of the record class with the corresponding formal parameter of *r*, in the order that record components (corresponding to the component fields) appear in the record header.

The *derived formal parameter list* of a record class is formed by deriving a formal parameter from each record component in the record header, in order, as follows:

- If the record component is not a variable arity record component, then the derived formal parameter has the same name and declared type as the record component.

  If the record component is a variable arity record component, then the derived formal parameter is a variable arity parameter (8.4.1 ⇗) with the same name and declared type as the record component.
- The derived formal parameter is annotated with the annotations, if any, that appear on the record component and whose annotation interfaces are applicable in the formal parameter context, or in type contexts, or both (9.7.4 ⇗).

A record declaration may contain declarations of constructors that are not canonical constructors. The body of every non-canonical constructor in a record declaration must ~~start with~~ contain an alternate constructor invocation (8.8.7.1), or a compile-time error occurs.

Chapter 11: Exceptions

11.2 Compile-Time Checking of Exceptions

11.2.2 Exception Analysis of Statements and Explicit Constructor Invocations

A `throw` statement (14.18 ⇗) whose thrown expression has static type *E* and is not a final or effectively final exception parameter can throw *E* or any exception class that the thrown expression can throw.

> *For example, the statement* `throw new java.io.FileNotFoundException();` *can throw* `java.io.FileNotFoundException` *only. Formally, it is not the case that it "can throw" a subclass or superclass of* `java.io.FileNotFoundException`*.*

A `throw` statement whose thrown expression is a final or effectively final exception parameter of a `catch` clause *C* can throw an exception class *E* iff:

- *E* is an exception class that the `try` block of the `try` statement which declares *C* can throw; and
- *E* is assignment compatible with any of *C*'s catchable exception classes; and
- *E* is not assignment compatible with any of the catchable exception classes of the `catch` clauses declared to the left of *C* in the same `try` statement.

A `try` statement (14.20 ⇗) can throw an exception class *E* iff either:

- The `try` block can throw *E*, or an expression used to initialize a resource (in a `try`-with-resources statement) can throw *E*, or the automatic invocation of the `close()` method of a resource (in a `try`-with-resources statement) can throw *E*, and *E* is not assignment

compatible with any catchable exception class of any `catch` clause of the `try` statement, and either no `finally` block is present or the `finally` block can complete normally; or

- Some `catch` block of the `try` statement can throw *E* and either no `finally` block is present or the `finally` block can complete normally; or
- A `finally` block is present and can throw *E*.

~~An explicit constructor invocation statement (8.8.7.1) can throw an exception class *E* iff either:~~

- ~~Some expression of the constructor invocation's parameter list can throw *E*; or~~
- ~~*E* is determined to be an exception class of the `throws` clause of the constructor that is invoked (15.12.2.6 ⇗).~~

A `switch` statement (14.11 ⇗) can throw an exception class *E* iff either:

- The selector expression can throw *E*; or
- Some switch rule expression, switch rule block, switch rule `throw` statement, or switch labeled statement group in the switch block can throw *E*.

Any other statement *S* can throw an exception class *E* iff an expression or statement immediately contained in *S* can throw *E*.

An explicit constructor invocation (8.8.7.1) can throw an exception class *E* iff either:

- Some expression of the constructor invocation's argument list can throw *E*; or
- *E* is determined to be an exception class of the `throws` clause of the constructor that is invoked (15.12.2.6 ⇗).

Chapter 12: Execution

12.5 Creation of New Class Instances

A new class instance is explicitly created when evaluation of a class instance creation expression (15.9 ⇗) causes a class to be instantiated.

A new class instance may be implicitly created in the following situations:

- Loading of a class or interface that contains a string literal (3.10.5 ⇗) or a text block (3.10.6 ⇗) may create a new `String` object to denote the string represented by the string literal or text block. (This object creation will not occur if an instance of `String` denoting the same sequence of Unicode code points as the string represented by the string literal or text block has previously been interned.)
- Execution of an operation that causes boxing conversion (5.1.7 ⇗). Boxing conversion may create a new object of a wrapper class (`Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`) associated with one of the primitive types.
- Execution of a string concatenation operator `+` (15.18.1 ⇗) that is not part of a constant expression (15.29 ⇗) always creates a new `String` object to represent the result. String concatenation operators may also create temporary wrapper objects for a value of a primitive type.

- Evaluation of a method reference expression (15.13.3 ⇗) or a lambda expression (15.27.4 ⇗) may require that a new instance be created of a class that implements a functional interface type (9.8 ⇗).

Each of these situations identifies a particular constructor (8.8 ⇗) to be called with specified arguments (possibly none) as part of the class instance creation process.

Whenever a new class instance is created, memory space is allocated for it with room for all the instance variables declared in the class and all the instance variables declared in each superclass of the class, including all the instance variables that may be hidden (8.3 ⇗).

If there is not sufficient space available to allocate memory for the object, then creation of the class instance completes abruptly with an `OutOfMemoryError`. Otherwise, all the instance variables in the new object, including those declared in superclasses, are initialized to their default values (4.12.5 ⇗).

Just before a reference to the newly created object is returned as the result, the indicated constructor is processed to initialize the new object using the following procedure:

1. ~~Assign the arguments for the constructor to newly created parameter variables for this constructor invocation.~~
2. ~~If this constructor begins with an explicit constructor invocation (8.8.7.1) of another constructor in the same class (using `this`), then evaluate the arguments and process that constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason; otherwise, continue with step 5.~~
3. ~~This constructor does not begin with an explicit constructor invocation of another constructor in the same class (using `this`). If this constructor is for a class other than `Object`, then this constructor will begin with an explicit or implicit invocation of a superclass constructor (using `super`). Evaluate the arguments and process that superclass constructor invocation recursively using these same five steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, continue with step 4.~~
4. ~~Execute the instance initializers and instance variable initializers for this class, assigning the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class. If execution of any of these initializers results in an exception, then no further initializers are processed and this procedure completes abruptly with that same exception. Otherwise, continue with step 5.~~
5. ~~Execute the rest of the body of this constructor. If that execution completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, this procedure completes normally.~~

1. Assign the arguments for the constructor to newly created parameter variables for this constructor invocation.
2. If this constructor does not contain an explicit constructor invocation (8.8.7.1) then continue from step 5.
3. Execute the *BlockStatements*, if any, of the prologue of the constructor body. If execution of any statement completes abruptly, then execution of the constructor completes abruptly for the same reason, otherwise continue with the next step.

4. The explicit constructor invocation is either an invocation of another constructor in the same class (using `this`) or an invocation of a superclass constructor (using `super`). Evaluate the arguments of the constructor invocation and process the constructor invocation recursively using these same seven steps. If the constructor invocation completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, continue from step 7 if the invocation is of another constructor in the same class, and continue from step 6 if the invocation is of a superclass constructor.
5. If this constructor is for a class other than `Object`, then this constructor contains an implicit invocation of a superclass constructor with no arguments. In this case, process the implicit constructor invocation recursively using these same seven steps. If that constructor invocation completes abruptly, then this procedure completes abruptly for the same reason, otherwise continue with the next step.
6. Execute the instance initializers and instance variable initializers for this class, assigning the values of instance variable initializers to the corresponding instance variables, in the left-to-right order in which they appear textually in the source code for the class. If execution of any of these initializers results in an exception, then no further initializers are processed and this procedure completes abruptly with that same exception, otherwise continue with the next step.
7. Execute the *BlockStatements*, if any, of the epilogue of this constructor. If execution of any statement completes abruptly, then this procedure completes abruptly for the same reason. Otherwise, this procedure completes normally.

Unlike C++, the Java programming language does not specify altered rules for method dispatch during the creation of a new class instance. If methods are invoked that are overridden in subclasses in the object being initialized, then these overriding methods are used, even before the new object is completely initialized. Classes can avoid unwanted exposure of uninitialized state by assigning to their fields in the prologue of the constructor body.

Example 12.5-1. Evaluation of Instance Creation

```
class Point {
    int x, y;
    Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
    int color = 0xFF00FF;
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        System.out.println(cp.color);
    }
}
```

Here, a new instance of `ColoredPoint` is created. First, space is allocated for the new `ColoredPoint`, to hold the fields `x`, `y`, and `color`. All these fields are then initialized to their default values (in this case, `0` for each field). Next, the `ColoredPoint` constructor with no arguments is first invoked. Since `ColoredPoint` declares no constructors, a default constructor of the following form is implicitly declared:

```
ColoredPoint() { super(); }
```

This constructor then invokes the `Point` constructor with no arguments. The `Point` constructor does

not begin with an invocation of a constructor, so the Java compiler provides an implicit invocation of its superclass constructor of no arguments, as though it had been written:

```
Point() { super(); x = 1; y = 1; }
```

Therefore, the constructor for `Object` which takes no arguments is invoked.

The class `Object` has no superclass, so the recursion terminates here. Next, any instance initializers and instance variable initializers of `Object` are invoked. Next, the body of the constructor of `Object` that takes no arguments is executed. No such constructor is declared in `Object`, so the Java compiler supplies a default one, which in this special case is:

```
Object() { }
```

This constructor executes without effect and returns.

Next, all initializers for the instance variables of class `Point` are executed. As it happens, the declarations of `x` and `y` do not provide any initialization expressions, so no action is required for this step of the example. Then the body of the `Point` constructor is executed, setting `x` to `1` and `y` to `1`.

Next, the initializers for the instance variables of class `ColoredPoint` are executed. This step assigns the value `0xFF00FF` to `color`. Finally, the ~~rest of the body~~ epilogue of the `ColoredPoint` constructor is executed (the part after the invocation of `super`); there happen to be no statements in the ~~rest of the body~~ epilogue, so no further action is required and initialization is complete.

Example 12.5-2. Dynamic Dispatch During Instance Creation

```
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}
class Test extends Super {
    int three = (int)Math.PI;  // That is, 3
    void printThree() { System.out.println(three); }

    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
}
```

This program produces the output:

```
0
3
```

This shows that the invocation of `printThree` in the constructor for class `Super` does not invoke the definition of `printThree` in class `Super`, but rather invokes the overriding definition of `printThree` in class `Test`. This method therefore runs before the field initializers of `Test` have been executed, which is why the first value output is `0`, the default value to which the field `three` of `Test` is initialized. The later invocation of `printThree` in method `main` invokes the same definition of `printThree`, but by that point the initializer for instance variable `three` has been executed, and so the value `3` is printed.

Example 12.5-3. Initialization of Fields in the Prologue

```
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}
class Test extends Super {
```

```
    int three;

    public Test() {
        three = (int)Math.PI;  // That is, 3
        super();
    }

    void printThree() { System.out.println(three); }

    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
}
```

This alternative to Example 12.5-2 produces the output:

```
3
3
```

Because the field `three` is initialized in the prologue of the `Test` class's constructor, its assignment occurs in Step 3 of the object initialization procedure, before evaluation of the `Super` class's constructor body in Step 4. When `three` is initialized in this way, it is impossible to observe it with the default value `0`.

Chapter 14: Blocks, Statements, and Patterns

14.17 The `return` Statement

A `return` statement returns control to the invoker of a method (8.4 ⇗, 15.12 ⇗) or constructor (8.8 ⇗, 15.9 ⇗).

ReturnStatement:
`return` [*Expression*] `;`

There are two kinds of `return` statement:

- A `return` statement with no value.
- A `return` statement with value *Expression*.

A `return` statement attempts to transfer control to the invoker of the innermost enclosing constructor, method, or lambda expression; this enclosing declaration or expression is called the *return target*. In the case of a `return` statement with value *Expression*, the value of the *Expression* becomes the value of the invocation.

It is a compile-time error if a `return` statement has no return target.

It is a compile-time error if the return target contains either (i) an instance or static initializer that encloses the `return` statement, or (ii) a `switch` expression that encloses the `return` statement.

It is a compile-time error if the return target of a `return` statement with no value is a method, and that method is not declared `void`.

It is a compile-time error if the return target of a `return` statement is a constructor, and the

`return` statement appears in the prologue of this constructor (8.8.7).

It is a compile-time error if the return target of a `return` statement with value *Expression* is either a constructor, or a method that is declared `void`.

It is a compile-time error if the return target of a `return` statement with value *Expression* is a method with declared return type *T*, and the type of *Expression* is not assignable compatible (5.2 ⇗) with *T*.

Execution of a `return` statement with no value always completes abruptly, the reason being a return with no value.

Execution of a `return` statement with value *Expression* first evaluates the *Expression*. If the evaluation of the *Expression* completes abruptly for some reason, then the `return` statement completes abruptly for that reason. If evaluation of the *Expression* completes normally, producing a value *V*, then the `return` statement completes abruptly, the reason being a return with value *V*.

> *It can be seen, then, that a `return` statement always completes abruptly.*
>
> *The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (14.20 ⇗) within the method or constructor whose `try` blocks or `catch` clauses contain the `return` statement, then any `finally` clauses of those `try` statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `return` statement.*

14.22 Unreachable Statements

It is a compile-time error if a statement cannot be executed because it is *unreachable*.

> *This section is devoted to a precise explanation of the word "reachable." The idea is that there must be some possible execution path from the beginning of the constructor, method, instance initializer, or static initializer that contains the statement to the statement itself. The analysis takes into account the structure of statements. Except for the special treatment of `while`, `do`, and `for` statements whose condition expression has the constant value `true`, the values of expressions are not taken into account in the flow analysis.*
>
> *For example, a Java compiler will accept the code:*

```
{
    int n = 5;
    while (n > 7) k = 2;
}
```

> *even though the value of `n` is known at compile time and in principle it can be known at compile time that the assignment to `k` can never be executed.*

The rules in this section define two technical terms:

- whether a statement is *reachable*
- whether a statement *can complete normally*

The rules allow a statement to complete normally only if it is reachable.

Two further technical terms are used:

- A reachable `break` statement *exits a statement* if, within the break target, either there

are no `try` statements whose `try` blocks contain the `break` statement, or there are `try` statements whose `try` blocks contain the `break` statement and all `finally` clauses of those `try` statements can complete normally.

This definition is based on the logic around "attempts to transfer control" in [14.15].

- A `continue` statement *continues a `do` statement* if, within the `do` statement, either there are no `try` statements whose `try` blocks contain the `continue` statement, or there are `try` statements whose `try` blocks contain the `continue` statement and all `finally` clauses of those `try` statements can complete normally.

The rules are as follows:

- The block that is the body of a constructor, method, instance initializer, static initializer, lambda expression, or `switch` expression is reachable.
- An empty block that is not a switch block can complete normally iff it is reachable.

  A non-empty block that is not a switch block or a constructor body containing an explicit constructor invocation can complete normally iff the last statement in it can complete normally.

  The first statement in a non-empty block that is not a switch block or a constructor body containing an explicit constructor invocation is reachable iff the block is reachable.

  Every other statement *S* in a non-empty block that is not a switch block or a constructor body containing an explicit constructor invocation is reachable iff the statement preceding *S* can complete normally.

- A non-empty block that is the body of a constructor containing an explicit constructor invocation can complete normally iff the last statement in it can complete normally.

  The first statement in a non-empty prologue of a constructor body containing an explicit constructor invocation is reachable iff the block is reachable.

  Every other statement *S* in the prologue of a constructor body containing an explicit constructor invocation is reachable iff the statement preceding *S* can complete normally.

  The first statement in a non-empty epilogue of a constructor containing an explicit constructor invocation and an empty prologue is reachable iff the block is reachable.

  The first statement in a non-empty epilogue of a constructor containing an explicit constructor invocation and a non-empty prologue is reachable iff the last statement of the prologue of the constructor can complete normally.

  Every other statement *S* in the epilogue of a constructor containing an explicit constructor invocation is reachable iff the statement preceding *S* can complete normally.

Editorial: The rest of section 14.22 is unchanged.

Chapter 15: Expressions

15.8 Primary Expressions

15.8.3 `this`

The keyword `this` may be used as an expression in the following contexts:

- in the body of an instance method of a class (8.4.3.2 ⇗)
- in the body of a constructor of a class (8.8.7)
- in an instance initializer of a class (8.6 ⇗)
- in the initializer of an instance variable of a class (8.3.2 ⇗)
- in the body of an instance method of an interface, that is, a default method or a non-`static` `private` interface method (9.4 ⇗)

When used as an expression, the keyword `this` denotes a value that is a reference either to the object for which the instance method was invoked (15.12 ⇗), or to the object being constructed. The value denoted by `this` in a lambda body (15.27.2 ⇗) is the same as the value denoted by `this` in the surrounding context.

> *The keyword `this` is also used in explicit constructor ~~invocation statements~~ invocations (8.8.7.1), and to denote the receiver parameter of a method or constructor (8.4 ⇗).*

It is a compile-time error if a `this` expression occurs in a static context (8.1.3).

Let *C* by the innermost enclosing class or interface declaration of a `this` expression.

It is a compile-time error if a `this` expression occurs in an early construction context (8.8.7) of *C*, unless it appears as the qualifier of a field access expression (15.11 ⇗) appearing as the left-hand operand of a simple assignment expression (15.26 ⇗), and the simple assignment expression is not enclosed in a lambda expression or inner class declaration that is contained in the early construction context of *C*.

If *C* is generic, with type parameters $F_1,...,F_n$, the type of `this` is $C<F_1,...,F_n>$. Otherwise, the type of `this` is *C*.

At run time, the class of the actual object referred to may be *C* or a subclass of *C* (8.1.5 ⇗).

Example 15.8.3-1. The `this` Expression

```
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;
    }
}
```

> *Here, 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`.*

15.8.4 Qualified `this`

Any lexically enclosing instance (8.1.3) can be referred to by explicitly qualifying the keyword `this`.

Let *n* be an integer such that *TypeName* denotes the *n*'th lexically enclosing class or interface declaration of the class or interface whose declaration immediately encloses the qualified `this` expression.

The value of a qualified `this` expression *TypeName*`.this` is the *n*'th lexically enclosing instance of `this`.

If *TypeName* denotes a generic class, with type parameters $F_1,...,F_n$, the type of the qualified `this` expression is *TypeName*$<F_1,...,F_n>$. Otherwise, the type of the qualified `this` expression is *TypeName*.

It is a compile-time error if a qualified `this` expression occurs in a static context (8.1.3).

It is a compile-time error if a qualified `this` expression occurs in an early construction context (8.8.7) of the class named by *TypeName*, unless it appears as the qualifier of a field access expression (15.11) appearing as the left-hand operand of a simple assignment expression (15.26), and the simple assignment expression is not enclosed in a lambda expression or inner class declaration that is contained in the early construction context of the class named by *TypeName*.

It is a compile-time error if the class or interface whose declaration immediately encloses a qualified `this` expression is not an inner class of *TypeName* or *TypeName* itself.

15.9 Class Instance Creation Expressions

15.9.2 Determining Enclosing Instances

Let *C* be the class being instantiated, and let *i* be the instance being created. If *C* is an inner class, then *i* may have an *immediately enclosing instance* (8.1.3), determined as follows:

- If *C* is an anonymous class, then:
  - If the class instance creation expression occurs in a static context, then *i* has no immediately enclosing instance. Let *S* be the nearest `static` method declaration, `static` field declaration, or static initializer that encloses the declaration of *C*. If the nearest `static` method declaration, `static` field declaration, or static initializer that encloses the class instance creation expression is not *S*, then a compile-time error occurs.
  - Otherwise, the immediately enclosing instance of *i* is `this`.
- If *C* is an inner local class, then:
  - If *C* occurs in a static context, then *i* has no immediately enclosing instance. Let *S* be the nearest `static` method declaration, `static` field declaration, or static initializer that encloses the declaration of *C*. If the nearest `static` method declaration, `static` field declaration, or static initializer that encloses the class instance creation expression is not *S*, then a compile-time error occurs.
  - Otherwise, if the class instance creation expression occurs in a static context, then a compile-time error occurs.
  - Otherwise, let *O* be the immediately enclosing class or interface declaration of *C*, and let *U* be the immediately enclosing class or interface declaration of the class

instance creation expression.

If *U* is not an inner class of *O* or *O* itself, then a compile-time error occurs.

Let *n* be an integer such that *O* is the *n*'th lexically enclosing class or interface declaration of *U*.

The immediately enclosing instance of *i* is the *n*'th lexically enclosing instance of `this`.

- If *C* is an inner member class, then:
  - If the class instance creation expression is unqualified, then:
    - If the class instance creation expression occurs in a static context, then a compile-time error occurs.
    - Otherwise, if *C* is not a member of any class whose declaration lexically encloses the class instance creation expression, then a compile-time error occurs.
    - Otherwise, let *O* be the innermost enclosing class declaration of which *C* is a member, and let *U* be the immediately enclosing class or interface declaration of the class instance creation expression.

      If *U* is not an inner class of *O* or *O* itself, then a compile-time error occurs.

      If the class instance creation expression occurs in the early construction context (8.8.7) of *O*, then a compile-time error occurs.

      Let *n* be an integer such that *O* is the *n*'th lexically enclosing class or interface declaration of *U*

      The immediately enclosing instance of *i* is the *n*'th lexically enclosing instance of `this`.
  - If the class instance creation expression is qualified, then the immediately enclosing instance of *i* is the object that is the value of the *Primary* expression or the *ExpressionName*.

If *C* is an anonymous class, and its direct superclass *S* is an inner class, then *i* may have an *immediately enclosing instance with respect to S*, determined as follows:

- If *S* is an inner local class, then:
  - If *S* occurs in 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, let *O* be the immediately enclosing class or interface declaration of *S*, and let *U* be the immediately enclosing class or interface declaration of the class instance creation expression.

    If *U* is not an inner class of *O* or *O* itself, then a compile-time error occurs.

    Let *n* be an integer such that *O* is the *n*'th lexically enclosing class or interface declaration of *U*.

    The immediately enclosing instance of *i* with respect to *S* is the *n*'th lexically enclosing instance of `this`.

- If *S* is an inner member class, then:
  - If the class instance creation expression is unqualified, then:
    - If the class instance creation expression occurs in a static context, then a compile-time error occurs.
    - Otherwise, if *S* is not a member of any class whose declaration encloses the class instance creation expression, then a compile-time error occurs.
    - Otherwise, let *O* be the innermost enclosing class declaration of which *S* is a member, and let *U* be the immediately enclosing class or interface declaration of the class instance creation expression.

      If *U* is not an inner class of *O* or *O* itself, then a compile-time error occurs.

      If the class instance creation expression occurs in the early construction context of *O*, then a compile-time error occurs.

      Let *n* be an integer such that *O* is the *n*'th lexically enclosing class or interface declaration of *U*.

      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.
  - If the class instance creation expression is qualified, then the immediately enclosing instance of *i* with respect to *S* is the object that is the value of the *Primary* expression or the *ExpressionName*.

15.11 Field Access Expressions

15.11.2 Accessing Superclass Members using `super`

The form `super`.*Identifier* refers to the field named *Identifier* of the current object, but with the current object viewed as an instance of the superclass of the current class.

The form *T*.`super`.*Identifier* refers to the field named *Identifier* of the lexically enclosing instance corresponding to *T*, but with that instance viewed as an instance of the superclass of *T*.

The forms using the keyword `super` may be used in the locations within a class declaration that allow the keyword `this` as an expression (15.8.3).

It is a compile-time error if a field access expression using the keyword `super` appears in a static context (8.1.3) or in an early construction context (8.8.7) of the current class.

For a field access expression of the form `super`.*Identifier*:

- It is a compile-time error if the immediately enclosing class or interface declaration of the field access expression is the class `Object` or an interface.

For a field access expression of the form *T*.`super`.*Identifier*:

- It is a compile-time error if *T* is the class `Object` or an interface.
- Let *U* be the immediately enclosing class or interface declaration of the field access expression. It is a compile-time error if *U* is not an inner class of *T* or *T* itself.

Suppose that a field access expression `super.`*f* appears within class *C*, and the immediate superclass of *C* is class *S*. If *f* in *S* is accessible from class *C* (6.6 ⇗), then `super.`*f* is treated as if it had been the expression `this.`*f* in the body of class *S*. Otherwise, a compile-time error occurs.

Thus, `super.`f can access the field f that is accessible in class S, even if that field is hidden by a declaration of a field f in class C.

Suppose that a field access expression *T*`.super.`*f* appears within class *C*, and the immediate superclass of the class denoted by *T* is a class whose fully qualified name is *S*. If *f* in *S* is accessible from *C*, then *T*`.super.`*f* is treated as if it had been the expression `this.`*f* in the body of class *S*. Otherwise, a compile-time error occurs.

Thus, T`.super.`f can access the field f that is accessible in class S, even if that field is hidden by a declaration of a field f in class T.

Example 15.11.2-1. The `super` Expression

```
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();
    }
}
```

This program 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` has the same effect as `((T2)this).x` when `x` has package access. Note that `super.x` is not specified in terms of a cast, due to difficulties around access to `protected` members of the superclass.

15.12 Method Invocation Expressions

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.

It is a compile-time error if an argument to a method invocation is not compatible with its target type, as derived from the invocation type of the compile-time declaration.

If the compile-time declaration is applicable by variable arity invocation, then where the last formal parameter type of the invocation type of the method is F_n`[]`, it is a compile-time error if the type which is the erasure of F_n is not accessible (6.6 ⇗) at the point of invocation.

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

In addition, whether the compile-time declaration is appropriate may depend on the form of the method invocation expression before the left parenthesis, as follows:

- If the form is *MethodName* - that is, just an *Identifier* - and the compile-time declaration is an instance method, then:
  - It is a compile-time error if the method invocation occurs in a static context (8.1.3).
  - Otherwise, let *T* be the class or interface to search (15.12.1 ⇗). It is a compile-time error if either the method invocation occurs in an early construction context (8.8.7) of class *T*, or the innermost enclosing class or interface declaration of the method invocation is neither *T* nor an inner class of *T*.
- If the form is *TypeName* `.` *[TypeArguments] Identifier*, then the compile-time declaration must be `static`, or a compile-time error occurs.
- If the form is *ExpressionName* `.` *[TypeArguments] Identifier* or *Primary* `.` *[TypeArguments] Identifier*, then the compile-time declaration must not be a `static` method declared in an interface, or a compile-time error occurs.
- If the form is `super` `.` *[TypeArguments] Identifier*, then:
  - It is a compile-time error if the compile-time declaration is `abstract`.
  - It is a compile-time error if the method invocation occurs in a static context or in an early construction context of the current class.
- If the form is *TypeName* `.` `super` `.` *[TypeArguments] Identifier*, then:
  - It is a compile-time error if the compile-time declaration is `abstract`.
  - It is a compile-time error if the method invocation occurs in a static context or in an early construction context of the current class.
  - If *TypeName* denotes a class *C*, then if the class or interface declaration immediately enclosing the method invocation is not *C* or an inner class of *C*, a compile-time error occurs.
  - If *TypeName* denotes an interface, let *E* be the class or interface declaration immediately enclosing the method invocation. A compile-time error occurs if there exists a method, distinct from the compile-time declaration, that overrides (9.4.1 ⇗) the compile-time declaration from a direct superclass or direct superinterface of *E*.

    In the case that a superinterface overrides a method declared in a grandparent interface, this rule prevents the child interface from "skipping" the override by simply adding the grandparent to its list of direct superinterfaces. The appropriate way to access functionality of a grandparent is through the direct superinterface, and only if that interface chooses to expose the desired behavior. (Alternately, the programmer is free to define an additional superinterface that exposes the desired behavior with a `super`

method invocation.)

The *compile-time parameter types* and *compile-time result* are determined as follows:

- If the compile-time declaration for the method invocation is *not* a signature polymorphic method, then:
  - The compile-time parameter types are the types of the formal parameters of the compile-time declaration.
  - The compile-time result is the result of the invocation type of the compile-time declaration (15.12.2.6 ⇗).
- If the compile-time declaration for the method invocation is a signature polymorphic method, then:
  - The compile-time parameter types are the types of the actual argument expressions. An argument expression which is the null literal `null` (3.10.8 ⇗) is treated as having the type `Void`.
  - The compile-time result is determined as follows:
    - If the signature polymorphic method is either `void` or has a return type other than `Object`, the compile-time result is the result of the invocation type of the compile-time declaration (15.12.2.6 ⇗).
    - Otherwise, if the method invocation expression is an expression statement, the compile-time result is `void`.
    - Otherwise, if the method invocation expression is the operand of a cast expression (15.16 ⇗), the compile-time result is the erasure of the type of the cast expression (4.6 ⇗).
    - Otherwise, the compile-time result is the signature polymorphic method's return type, `Object`.

A method is *signature polymorphic* if all of the following are true:

- It is declared in the `java.lang.invoke.MethodHandle` class or the `java.lang.invoke.VarHandle` class.
- It has a single variable arity parameter (8.4.1 ⇗) whose declared type is `Object[]`.
- It is `native`.

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

- The name of the method.
- The qualifying class or interface of the method invocation (13.1 ⇗).
- The number of parameters and the compile-time parameter types, in order.
- The compile-time result.
- The invocation mode, computed as follows:
  - If the compile-time declaration has the `static` modifier, then the invocation mode is `static`.

- Otherwise, if the part of the method invocation before the left parenthesis is of the form `super` . *Identifier* or of the form *TypeName* . `super` . *Identifier*, then the invocation mode is `super`.
- Otherwise, if the qualifying class or interface of the method invocation is in fact an interface, then the invocation mode is `interface`.
- Otherwise, the invocation mode is `virtual`.

If the result of the invocation type of the compile-time declaration is not `void`, then the type of the method invocation expression is obtained by applying capture conversion (5.1.10 ⇗) to the return type of the invocation type of the compile-time declaration.

15.13 Method Reference Expressions

A method reference expression is used to refer to the invocation of a method without actually performing the invocation. Certain forms of method reference expression also allow class instance creation (15.9 ⇗) or array creation (15.10 ⇗) to be treated as if it were a method invocation.

MethodReference:
ExpressionName `::` [*TypeArguments*] *Identifier*
Primary `::` [*TypeArguments*] *Identifier*
ReferenceType `::` [*TypeArguments*] *Identifier*
`super` `::` [*TypeArguments*] *Identifier*
TypeName `.` `super` `::` [*TypeArguments*] *Identifier*
ClassType `::` [*TypeArguments*] `new`
ArrayType `::` `new`

If *TypeArguments* is present to the right of `::`, then it is a compile-time error if any of the type arguments are wildcards (4.5.1 ⇗).

If a method reference expression has the form *ExpressionName* `::` *[TypeArguments] Identifier* or *Primary* `::` *[TypeArguments] Identifier*, it is a compile-time error if the type of the *ExpressionName* or *Primary* is not a reference type.

If a method reference expression has the form `super` `::` *[TypeArguments] Identifier*, let *E* be the class or interface declaration immediately enclosing the method reference expression. It is a compile-time error if *E* is the class `Object` or if *E* is an interface.

If a method reference expression has the form *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, then:

- If *TypeName* denotes a class, *C*, then it is a compile-time error if *C* is not a lexically enclosing class of the current class, or if *C* is the class `Object`.
- If *TypeName* denotes an interface, *I*, then let *E* be the class or interface declaration immediately enclosing the method reference expression. It is a compile-time error if *I* is not a direct superinterface of *E*, or if there exists some other direct superclass or direct superinterface of *E*, *J*, such that *J* is a subclass or subinterface of *I*.
- If *TypeName* denotes a type variable, then a compile-time error occurs.

If a method reference expression has the form `super` `::` *[TypeArguments] Identifier* or

TypeName `.` `super` `::` *[TypeArguments] Identifier*, it is a compile-time error if the expression occurs in a static context (8.1.3) or in an early construction context (8.8.7) of the current class.

If a method reference expression has the form *ClassType* `::` *[TypeArguments]* `new`, then:

- *ClassType* must name a class that is accessible (6.6 ⇗), non-`abstract`, and not an enum class, or a compile-time error occurs.
- If *ClassType* denotes a parameterized type (4.5 ⇗), then it is a compile-time error if any of its type arguments are wildcards.
- If *ClassType* denotes a raw type (4.8 ⇗), then it is a compile-time error if *TypeArguments* is present after the `::`.

If a method reference expression has the form *ArrayType* `::` `new`, then *ArrayType* must denote a type that is reifiable (4.7 ⇗), or a compile-time error occurs.

The target reference of an instance method (15.12.4.1 ⇗) may be provided by the method reference expression using an *ExpressionName*, a *Primary*, or `super`, or it may be provided later when the method is invoked. The immediately enclosing instance of a new inner class instance (15.9.2) is provided by a lexically enclosing instance of `this` (8.1.3).

When more than one member method of a type has the same name, or when a class has more than one constructor, the appropriate method or constructor is selected based on the functional interface type targeted by the method reference expression, as specified in 15.13.1.

If a method or constructor is generic, the appropriate type arguments may either be inferred or provided explicitly. Similarly, the type arguments of a generic type mentioned by the method reference expression may be provided explicitly or inferred.

Method reference expressions are always poly expressions (15.2 ⇗).

It is a compile-time error if a method reference expression occurs in a program in someplace other than an assignment context (5.2 ⇗), an invocation context (5.3 ⇗), or a casting context (5.5 ⇗).

Evaluation of a method reference expression produces an instance of a functional interface type (9.8 ⇗). This does *not* cause the execution of the corresponding method; instead, the execution may occur at a later time when an appropriate method of the functional interface is invoked.

Here are some method reference expressions, first with no target reference and then with a target reference:

```
String::length             // instance method
System::currentTimeMillis  // static method
List<String>::size  // explicit type arguments for generic type
List::size          // inferred type arguments for generic type
int[]::clone
T::tvarMember

System.out::println
"abc"::length
foo[x]::bar
(test ? list.replaceAll(String::trim) : list) :: iterator
super::toString
```

Here are some more method reference expressions:

```
String::valueOf       // overload resolution needed
Arrays::sort          // type arguments inferred from context
Arrays::<String>sort  // explicit type arguments
```

Here are some method reference expressions that represent a deferred creation of an object or an array:

```
ArrayList<String>::new     // constructor for parameterized type
ArrayList::new             // inferred type arguments
                           // for generic class
Foo::<Integer>new          // explicit type arguments
                           // for generic constructor
Bar<String>::<Integer>new  // generic class, generic constructor
Outer.Inner::new           // inner class constructor
int[]::new                 // array creation
```

It is not possible to specify a particular signature to be matched, for example, `Arrays::sort(int[])`. Instead, the functional interface provides argument types that are used as input to the overload resolution algorithm (15.12.2 ⇗). This should satisfy the vast majority of use cases; when the rare need arises for more precise control, a lambda expression can be used.

The use of type argument syntax in the class name before a delimiter (`List<String>::size`) raises the parsing problem of distinguishing between `<` as a type argument bracket and `<` as a less-than operator. In theory, this is no worse than allowing type arguments in cast expressions; however, the difference is that the cast case only comes up when a `(` token is encountered; with the addition of method reference expressions, the start of every expression is potentially a parameterized type.

15.13.1 Compile-Time Declaration of a Method Reference

The *compile-time declaration* of a method reference expression is the method to which the expression refers. In special cases, the compile-time declaration does not actually exist, but is a notional method that represents a class instance creation or an array creation. The choice of compile-time declaration depends on a function type targeted by the expression, just as the compile-time declaration of a method invocation depends on the invocation's arguments (15.12.3).

The search for a compile-time declaration mirrors the process for method invocations in 15.12.1 ⇗ and 15.12.2 ⇗, as follows:

- First, a type to search is determined:
  - If the method reference expression has the form *ExpressionName* `::` *[TypeArguments] Identifier* or *Primary* `::` *[TypeArguments] Identifier*, the type to search is the type of the expression preceding the `::` token.
  - If the method reference expression has the form *ReferenceType* `::` *[TypeArguments] Identifier*, the type to search is the result of capture conversion (5.1.10 ⇗) applied to *ReferenceType*.
  - If the method reference expression has the form `super` `::` *[TypeArguments] Identifier*, the type to search is the superclass type of the immediately enclosing class or interface declaration of the method reference expression.

    Let *T* be the class or interface declaration immediately enclosing the method reference expression. It is a compile-time error if *T* is the class `Object` or an interface.
  - If the method reference expression has the form *TypeName* `.` `super` `::`

[TypeArguments] Identifier, then if *TypeName* denotes a class, the type to search is the superclass type of the named class; otherwise, *TypeName* denotes an interface to search.

It is a compile-time error if *TypeName* is neither a lexically enclosing class or interface declaration of the method reference expression, nor a direct superinterface of the immediately enclosing class or interface declaration of the method reference expression.

It is a compile-time error if *TypeName* is the class Object.

It is a compile-time error if *TypeName* is an interface, and there exists some other direct superclass or direct superinterface of the immediately enclosing class or interface declaration of the method reference expression, *J*, such that *J* is a subclass or subinterface of *TypeName*.

- For the two other forms (involving `::` `new`), the referenced method is notional and there is no type to search.

- Second, given a targeted function type with *n* parameters, a set of potentially applicable methods is identified:
  - If the method reference expression has the form *ReferenceType* `::` *[TypeArguments] Identifier*, then the potentially applicable methods are:
    - the member methods of the type to search that would be potentially applicable (15.12.2.1 ⇗) for a method invocation which names *Identifier*, has arity *n*, has type arguments *TypeArguments*, and appears in the same class as the method reference expression; plus
    - the member methods of the type to search that would be potentially applicable for a method invocation which names *Identifier*, has arity *n*-1, has type arguments *TypeArguments*, and appears in the same class as the method reference expression.

    Two different arities, n and n-1, are considered, to account for the possibility that this form refers to either a `static` *method or an instance method.*

  - If the method reference expression has the form *ClassType* `::` *[TypeArguments]* `new`, then the potentially applicable methods are a set of notional methods corresponding to the constructors of *ClassType*.

    If *ClassType* is a raw type, but is not a non-`static` member type of a raw type, the candidate notional member methods are those specified in 15.9.3 ⇗ for a class instance creation expression that uses `<>` to elide the type arguments to a class. Otherwise, the candidate notional member methods are the constructors of *ClassType*, treated as if they were methods with return type *ClassType*.

    Among these candidates, the potentially applicable methods are the notional methods that would be potentially applicable for a method invocation which has arity *n*, has type arguments *TypeArguments*, and appears in the same class as the method reference expression.

  - If the method reference expression has the form *ArrayType* `::` `new`, a single notional method is considered. The method has a single parameter of type `int`, returns the *ArrayType*, and has no `throws` clause. If *n* = 1, this is the only potentially applicable method; otherwise, there are no potentially applicable

methods.

- For all other forms, the potentially applicable methods are the member methods of the type to search that would be potentially applicable for a method invocation which names *Identifier*, has arity *n*, has type argument *TypeArguments*, and appears in the same class as the method reference expression.

- Finally, if there are no potentially applicable methods, then there is no compile-time declaration.

  Otherwise, given a targeted function type with parameter types P_1, ..., P_n and a set of potentially applicable methods, the compile-time declaration is selected as follows:

  - If the method reference expression has the form *ReferenceType* `::` *[TypeArguments] Identifier*, then two searches for a most specific applicable method are performed. Each search is as specified in 15.12.2.2 through 15.12.2.5, with the clarifications below. Each search produces a set of applicable methods and, possibly, designates a most specific method of the set. In the case of an error as specified in 15.12.2.4, the set of applicable methods is empty. In the case of an error as specified in 15.12.2.5, there is no most specific method.

    In the first search, the method reference is treated as if it were an invocation with argument expressions of types P_1, ..., P_n. Type arguments, if any, are given by the method reference expression.

    In the second search, if P_1, ..., P_n is not empty and P_1 is a subtype of *ReferenceType*, then the method reference expression is treated as if it were a method invocation expression with argument expressions of types P_2, ..., P_n. If *ReferenceType* is a raw type, and there exists a parameterization of this type, *G*<...>, that is a supertype of P_1, the type to search is the result of capture conversion (5.1.10) applied to *G*<...>; otherwise, the type to search is the same as the type of the first search. Type arguments, if any, are given by the method reference expression.

    If the first search produces a most specific method that is `static`, and the set of applicable methods produced by the second search contains no non-`static` methods, then the compile-time declaration is the most specific method of the first search.

    Otherwise, if the set of applicable methods produced by the first search contains no `static` methods, and the second search produces a most specific method that is non-`static`, then the compile-time declaration is the most specific method of the second search.

    Otherwise, there is no compile-time declaration.

  - For all other forms of method reference expression, one search for a most specific applicable method is performed. The search is as specified in 15.12.2.2 through 15.12.2.5, with the clarifications below.

    The method reference is treated as if it were an invocation with argument expressions of types P_1, ..., P_n; the type arguments, if any, are given by the method reference expression.

    If the search results in an error as specified in 15.12.2.2 through 15.12.2.5, or if the most specific applicable method is `static`, there is no compile-time

declaration.

Otherwise, the compile-time declaration is the most specific applicable method.

It is a compile-time error if a method reference expression has the form *ReferenceType* `::` *[TypeArguments] Identifier*, and the compile-time declaration is `static`, and *ReferenceType* is not a simple or qualified name (6.2 ⇗).

It is a compile-time error if the method reference expression has the form `super` `::` *[TypeArguments] Identifier* or *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, and the compile-time declaration is `abstract`.

It is a compile-time error if the method reference expression has the form `super` `::` *[TypeArguments] Identifier* or *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, and the method reference expression occurs in a static context (8.1.3) or in an early construction context (8.8.7) of the current class.

It is a compile-time error if the method reference expression has the form *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, and *TypeName* denotes a class *C*, and the immediately enclosing class or interface declaration of the method reference expression is not *C* or an inner class of *C*.

It is a compile-time error if the method reference expression has the form *TypeName* `.` `super` `::` *[TypeArguments] Identifier*, and *TypeName* denotes an interface, and there exists a method, distinct from the compile-time declaration, that overrides the compile-time declaration from a direct superclass or direct superinterface of the class or interface whose declaration immediately encloses the method reference expression (8.4.8 ⇗, 9.4.1 ⇗).

It is a compile-time error if the method reference expression is of the form *ClassType* `::` *[TypeArguments]* `new` and a compile-time error would occur when determining an enclosing instance for *ClassType* as specified in 15.9.2 (treating the method reference expression as if it were an unqualified class instance creation expression).

A method reference expression of the form ReferenceType `::` [TypeArguments] Identifier can be interpreted in different ways. If Identifier refers to an instance method, then the implicit lambda expression has an extra parameter compared to if Identifier refers to a `static` method. It is possible for ReferenceType to have both kinds of applicable methods, so the search algorithm described above identifies them separately, since there are different parameter types for each case.

An example of ambiguity is:

```
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
          // Error: instance method size()
          // or static method size(Object)?
    }
}
```

This ambiguity cannot be resolved by providing an applicable instance method which is more specific than an applicable `static` method:

```
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    static int size(Object arg) { return 0; }
    int size(C arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
          // Error: instance method size()
          // or static method size(Object)?
    }
}
```

The search is smart enough to ignore ambiguities in which all the applicable methods (from both searches) are instance methods:

```
interface Fun<T,R> { R apply(T arg); }

class C {
    int size() { return 0; }
    int size(Object arg) { return 0; }
    int size(C arg) { return 0; }

    void test() {
        Fun<C, Integer> f1 = C::size;
          // OK: reference is to instance method size()
    }
}
```

For convenience, when the name of a generic type is used to refer to an instance method (where the receiver becomes the first parameter), the target type is used to determine the type arguments. This facilitates usage like `Pair::first` in place of `Pair<String,Integer>::first`. Similarly, a method reference like `Pair::new` is treated like a "diamond" instance creation (`new Pair<>()`). Because the "diamond" is implicit, this form does not instantiate a raw type; in fact, there is no way to express a reference to the constructor of a raw type.

For some method reference expressions, there is only one possible compile-time declaration with only one possible invocation type (15.12.2.6 ⇗), regardless of the targeted function type. Such method reference expressions are said to be *exact*. A method reference expression that is not exact is said to be *inexact*.

A method reference expression ending with *Identifier* is exact if it satisfies all of the following:

- If the method reference expression has the form *ReferenceType* `::` *[TypeArguments] Identifier*, then *ReferenceType* does not denote a raw type.
- The type to search has exactly one member method with the name *Identifier* that is accessible to the class or interface in which the method reference expression appears.
- This method is not variable arity (8.4.1 ⇗).
- If this method is generic (8.4.4 ⇗), then the method reference expression provides *TypeArguments*.

A method reference expression of the form *ClassType* `::` *[TypeArguments]* `new` is exact if it satisfies all of the following:

- The type denoted by *ClassType* is not raw, or is a non-`static` member type of a raw type.
- The type denoted by *ClassType* has exactly one constructor that is accessible to the class or interface in which the method reference expression appears.
- This constructor is not variable arity.
- If this constructor is generic, then the method reference expression provides *TypeArguments*.

A method reference expression of the form *ArrayType* `::` `new` is always exact.

Chapter 16: Definite Assignment

Every local variable declared by a statement (14.4.2 ⇗, 14.14.1 ⇗, 14.14.2 ⇗, 14.20.3 ⇗) and every blank `final` field (4.12.4 ⇗, 8.3.1.2 ⇗) must have a *definitely assigned* value when any access of its value occurs.

An access to its value consists of the simple name of the variable (or, for a field, the simple name of the field qualified by `this`) occurring anywhere in an expression except as the left-hand operand of the simple assignment operator `=` (15.26.1 ⇗).

For every access of a local variable declared by a statement *x*, or blank `final` field *x*, *x* must be definitely assigned before the access, or a compile-time error occurs.

Similarly, every blank `final` variable must be assigned at most once; it must be *definitely unassigned* when an assignment to it occurs.

Such an assignment is defined to occur if and only if either the simple name of the variable (or, for a field, its simple name qualified by `this`) occurs on the left hand side of an assignment operator.

For every assignment to a blank `final` variable, the variable must be definitely unassigned before the assignment, or a compile-time error occurs.

Similarly, for every alternate constructor invocation (8.8.7.1) occurring in a constructor of a class *C*, every blank `final` instance variable of *C* declared in *C* must be definitely unassigned after the argument list of the alternate constructor invocation, or a compile-time error occurs.

Note that local variables declared by a pattern (14.30 ⇗) are not subject to the rules of definite assignment. Every local variable declared by a pattern is initialized by the process of pattern matching and so always has a value when accessed.

The remainder of this chapter is devoted to a precise explanation of the words "definitely assigned before" and "definitely unassigned before".

...

16.2 Definite Assignment and Statements

16.2.2 Blocks

> *Editorial: Note that the body of a constructor cannot be properly considered a block, especially now that it is split into separate prologue and epilogue sections. The treatment of constructor bodies is now covered in 16.9.*

- A blank `final` member field *V* is definitely assigned (and moreover is not definitely unassigned) before the block (14.2 ⇗) that is the body of any method in the scope of *V* and before the declaration of any class declared within the scope of *V*.
- A local variable *V* declared by a statement *S* is definitely unassigned (and moreover is not definitely assigned) before the block that is the body of the ~~constructor,~~ method, instance initializer or static initializer which contains *S*.
- Let *C* be a class declared within the scope of *V*. Then *V* is definitely assigned before the block that is the body of any ~~constructor,~~ method, instance initializer, or static initializer declared in *C* iff *V* is definitely assigned before the declaration of *C*.

  Note that there are no rules that would allow us to conclude that V is definitely unassigned before the block that is the body of any ~~constructor,~~ method, instance initializer, or static initializer declared in C. We can informally conclude that V is not definitely unassigned before the block that is the body of any constructor, method, instance initializer, or static initializer declared in C, but there is no need for such a rule to be stated explicitly.

- *V* is [un]assigned after an empty block iff *V* is [un]assigned before the empty block.
- *V* is [un]assigned after a non-empty block iff *V* is [un]assigned after the last statement in the block.
- *V* is [un]assigned before the first statement of the block iff *V* is [un]assigned before the block.
- *V* is [un]assigned before any other statement *S* of the block iff *V* is [un]assigned after the statement immediately preceding *S* in the block.

We say that *V* is definitely unassigned everywhere in a block *B* iff:

- *V* is definitely unassigned before *B*.
- *V* is definitely assigned after *e* in every assignment expression *V* `=` *e*, *V* `+=` *e*, *V* `-=` *e*, *V* `*=` *e*, *V* `/=` *e*, *V* `%=` *e*, *V* `<<=` *e*, *V* `>>=` *e*, *V* `>>>=` *e*, *V* `&=` *e*, *V* `|=` *e*, or *V* `^=` *e* that occurs in *B*.
- *V* is definitely assigned before every expression `++`*V*, `--`*V*, *V*`++`, or *V*`--` that occurs in *B*.

These conditions are counterintuitive and require some explanation. Consider a simple assignment V = e. If V is definitely assigned after e, then either:

- *The assignment occurs in dead code, and V is vacuously definitely assigned. In this case, the assignment will not actually take place, and we can assume that V is not being assigned by the assignment expression. Or:*
- *V was already assigned by an earlier expression prior to e. In this case the current assignment will cause a compile-time error.*

So, we can conclude that if the conditions are met by a program that causes no compile time error, then any assignments to V in B will not actually take place at run time.

16.9 Definite Assignment, Constructors, and Instance Initializers

Let *C* be a class declared within the scope of *V*. Then:

- *V* is definitely assigned before a constructor declaration (8.8.7) or ~~an~~ instance variable initializer (8.3.2 ⇗) of *C* iff *V* is definitely assigned before the declaration of *C*.

  Note that there are no rules that would allow us to conclude that V is definitely unassigned before ~~an~~ the constructor declaration or instance variable initializer. We can informally conclude

that V is not definitely unassigned before any constructor declaration or instance variable initializer of C, but there is no need for such a rule to be stated explicitly.

Let *C* be a class, and let *V* be a blank `final` non-`static` member field of *C*, declared in *C*. Then:

- *V* is definitely unassigned (and moreover is not definitely assigned) before the declaration of any constructor in *C*.
- *V* is definitely unassigned (and moreover is not definitely assigned) before the leftmost instance initializer (8.6 ⇗) or instance variable initializer of *C* iff *V* is definitely unassigned after every superclass constructor invocation (8.8.7.1) in the constructors of *C*.
- *V* is definitely assigned (and moreover is not definitely unassigned) before the leftmost instance initializer (8.6 ⇗) or instance variable initializer of *C* iff *C* declares at least one constructor, every constructor of *C* has an explicit constructor invocation, and *V* is definitely assigned after every superclass constructor invocation in these constructors.
- *V* is [un]assigned before an instance initializer or instance variable initializer of *C* other than the leftmost iff *V* is [un]assigned after the preceding instance initializer or instance variable initializer of *C*.

Let *C* be a class, and let *V* be a blank `final` non-`static` member field of *C*, declared in a superclass of *C*. Then:

- *V* is definitely unassigned (and moreover is not definitely assigned) before the declaration of any constructor in *C*.
- *V* is definitely assigned (and moreover is not definitely unassigned) before every instance initializer and instance variable initializer of *C*.

Let *C* be a class, and let *V* be a blank `final static` member field of *C*. Then:

- *V* is definitely assigned (and moreover is not definitely unassigned) before every constructor declaration, instance initializer, and instance variable initializer of *C*.

Let *C* be a class, and let *V* be a local variable declared by a statement *S* contained by a constructor or instance variable initializer of *C*. Then:

- *V* is definitely unassigned (and moreover is not definitely unassigned) before the constructor declaration or instance variable initializer.

The following rules hold within the constructors ~~(8.8.7)~~ of class *C*:

- ~~*V* is definitely assigned (and moreover is not definitely unassigned) after an alternate constructor invocation (8.8.7.1).~~
- ~~*V* is definitely unassigned (and moreover is not definitely assigned) before an explicit or implicit superclass constructor invocation (8.8.7.1).~~
- ~~If *C* has no instance initializers or instance variable initializers, then *V* is not definitely assigned (and moreover is definitely unassigned) after an explicit or implicit superclass constructor invocation.~~
- ~~If *C* has at least one instance initializer or instance variable initializer then *V* is [un]assigned after an explicit or implicit superclass constructor invocation iff *V* is [un]assigned after the rightmost instance initializer or instance variable initializer of *C*.~~

- *V* is [un]assigned before the prologue of the constructor body (8.8.7) iff *V* is [un]assigned before the constructor declaration.
- *V* is [un]assigned after an empty prologue iff *V* is [un]assigned before the prologue.
- *V* is [un]assigned after a non-empty prologue iff *V* is [un]assigned after the last statement in the prologue.
- *V* is [un]assigned before an explicit constructor invocation (8.8.7.1) iff *V* is [un]assigned after the prologue.
- *V* is [un]assigned before the argument list of a qualified super constructor invocation iff *V* is [un]assigned after the qualifier expression.
- *V* is [un]assigned before the argument list of any other explicit constructor invocation iff *V* is [un]assigned before the invocation.
- *V* is [un]assigned after the argument list of an explicit constructor invocation iff either *V* is [un]assigned after the rightmost argument expression in the argument list, or the argument list is empty and *V* is [un]assigned before the argument list.
- A blank `final` non-`static` member field of *C*, declared in a superclass of *C*, is definitely assigned (and moreover is not definitely unassigned) after a superclass constructor invocation.
- Any other variable *V* is [un]assigned after a superclass constructor invocation iff *V* is [un]assigned after the argument list of the invocation.
- A blank `final` non-`static` member field of *C*, declared in *C* or a superclass of *C*, is definitely assigned (and moreover is not definitely unassigned) after an alternate constructor invocation.
- Any other variable *V* is [un]assigned after an alternate constructor invocation iff *V* is [un]assigned after the argument list of the invocation.
- For the epilogue of a constructor body with no explicit constructor invocation:
    - A blank `final` non-`static` member field *V* of *C*, declared in *C*, is [un]assigned before the epilogue iff either *V* is [un]assigned after the rightmost instance initialization or instance variable initializer of *C*, or *C* declares no instance initializer or instance variable initializer, and *V* is [un]assigned after the prologue of the constructor body.
    - A blank `final` non-`static` member field of *C*, declared in a superclass of *C*, is definitely assigned (and moreover is not definitely unassigned) before the epilogue.
    - Any other variable *V* is [un]assigned before the epilogue iff *V* is [un]assigned after the prologue of the constructor body.
- For the epilogue of a constructor body with a superclass constructor invocation:
    - A blank `final` non-`static` member field *V* of *C*, declared in *C*, is [un]assigned before the epilogue iff either *V* is [un]assigned after the rightmost instance initialization or instance variable initializer of *C*, or *C* declares no instance initializer or instance variable initializer, and *V* is [un]assigned after the superclass constructor invocation.
    - Any other variable *V* is [un]assigned before the epilogue iff *V* is [un]assigned after the superclass constructor invocation.
- For the epilogue of a constructor body with an alternate constructor invocation, *V* is

[un]assigned before the epilogue iff *V* is [un]assigned after the alternate constructor invocation.

- *V* is [un]assigned after an empty epilogue iff *V* is [un]assigned before the epilogue.
- *V* is [un]assigned after a non-empty epilogue iff *V* is [un]assigned after the last statement in the epilogue.
- *V* is [un]assigned before the first statement of the prologue or epilogue iff *V* is [un]assigned before the prologue or epilogue, respectively.
- *V* is [un]assigned before any other statement *S* in the prologue or epilogue iff *V* is [un]assigned after the statement immediately preceding *S* in the prologue or epilogue.
- *V* is [un]assigned before the qualifier expression of a super constructor invocation iff *V* is [un]assigned before the super constructor invocation.
- *V* is [un]assigned before the leftmost argument expression of an explicit constructor invocation iff *V* is [un]assigned before the argument list.
- *V* is [un]assigned before any other argument expression *x* of an explicit constructor invocation iff *V* is [un]assigned after the argument expression to the left of *x*.

~~Let *C* be a class, and let *V* be a blank `final` member field of *C*, declared in a superclass of *C*. Then:~~

- ~~*V* is definitely assigned (and moreover is not definitely unassigned) before the block that is the body of a constructor or instance initializer of *C*.~~
- ~~*V* is definitely assigned (and moreover is not definitely unassigned) before every instance variable initializer of *C*.~~

DRAFT 24-internal-adhoc.gbierman.20241101