Derived Record Creation (Preview)

Changes to the Java® Language Specification • Version 23-internal-adhoc.gbierman.20240326

This document describes changes to the Java Language Specification to support *Derived Record Creation*, a preview feature of Java SE 23. See a JEP 468 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:

- *March 2024: First draft released.*

Chapter 4: Types, Values, and Variables

4.12 Variables

4.12.4 `final` Variables

A variable can be declared `final`. A `final` variable may only be assigned to once. It is a compile-time error if a `final` variable is assigned to unless it is definitely unassigned immediately prior to the assignment (16 ⇗).

Once a `final` variable has been assigned, it always contains the same value. If a `final` variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object. This applies also to arrays, because arrays are objects; if a `final` variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

A *blank `final`* is a `final` variable whose declaration lacks an initializer.

A *constant variable* is a `final` variable of primitive type or type `String` that is initialized with a constant expression (15.29 ⇗). Whether a variable is a constant variable or not may have implications with respect to class initialization (12.4.1 ⇗), binary compatibility (13.1 ⇗), reachability (14.22 ⇗), and definite assignment (16.1.1 ⇗).

Three kinds of variable are implicitly declared `final`: a field of an interface (9.3 ⇗), a local variable declared as a resource of a `try`-with-resources statement (14.20.3 ⇗), and an exception parameter of a multi-`catch` clause (14.20 ⇗). An exception parameter of a uni-`catch` clause is never implicitly declared `final`, but may be effectively final.

> **Example 4.12.4-1. Final Variables**
>
> *Declaring a variable `final` can serve as useful documentation that its value will not change and can help avoid programming errors. In this program:*
>
> ```
> class Point {
>     int x, y;
>     int useCount;
>     Point(int x, int y) { this.x = x; this.y = y; }
>     static final Point origin = new Point(0, 0);
> }
> ```
>
> *the class `Point` declares a `final` class variable `origin`. The `origin` variable holds a reference to an object that is an instance of class `Point` whose coordinates are (0, 0). The value of the variable `Point.origin` can never change, so it always refers to the same `Point` object, the one created by its initializer. However, an operation on this `Point` object might change its state - for example, modifying its `useCount` or even, misleadingly, its `x` or `y` coordinate.*

Certain variables that are not declared `final` are instead considered *effectively final*:

- A local variable declared by a statement and whose declarator has an initializer (14.4 ⇗), ~~or~~ a local variable declared by a pattern (14.30.1 ⇗), or a local variable implicitly declared in the block of a derived record creation expression (15.30), is *effectively final* if all of the following are true:
  - It is not declared `final`.
  - It never occurs as the left hand side in an assignment expression (15.26). (Note

that the local variable declarator containing the initializer is *not* an assignment expression.)

- It never occurs as the operand of a prefix or postfix increment or decrement operator (15.14 ⇗, 15.15 ⇗).

- A local variable declared by a statement and whose declarator lacks an initializer is *effectively final* if all of the following are true:
  - It is not declared `final`.
  - Whenever it occurs as the left hand side in an assignment expression, it is definitely unassigned and not definitely assigned before the assignment; that is, it is definitely unassigned and not definitely assigned after the right hand side of the assignment expression (16 ⇗).
  - It never occurs as the operand of a prefix or postfix increment or decrement operator.
- A method, constructor, lambda, or exception parameter (8.4.1 ⇗, 8.8.1 ⇗, 9.4 ⇗, 15.27.1 ⇗, 14.20 ⇗) is treated, for the purpose of determining whether it is *effectively final*, as a local variable whose declarator has an initializer.

If a variable is effectively final, adding the `final` modifier to its declaration will not introduce any compile-time errors. Conversely, a local variable or parameter that is declared `final` in a valid program becomes effectively final if the `final` modifier is removed.

Chapter 6: Names

6.1 Declarations

A *declaration* introduces one of the following entities into a program:

- A module, declared in a `module` declaration (7.7 ⇗)
- A package, declared in a `package` declaration (7.4 ⇗)
- An imported class or interface, declared in a single-type-import declaration or a type-import-on-demand declaration (7.5.1 ⇗, 7.5.2 ⇗)
- An imported `static` member, declared in a single-static-import declaration or a static-import-on-demand declaration (7.5.3 ⇗, 7.5.4 ⇗)
- A class, declared by a normal class declaration (8.1 ⇗), an enum declaration (8.9 ⇗), or a record declaration (8.10 ⇗)
- An interface, declared by a normal interface declaration (9.1 ⇗) or an annotation interface declaration (9.6 ⇗).
- A type parameter, declared as part of the declaration of a generic class, interface, method, or constructor (8.1.2 ⇗, 9.1.2 ⇗, 8.4.4 ⇗, 8.8.4 ⇗)
- A member of a reference type (8.2 ⇗, 9.2 ⇗, 8.9.3 ⇗, 9.6 ⇗, 10.7 ⇗), one of the following:
  - A member class (8.5 ⇗, 9.5 ⇗)
  - A member interface (8.5 ⇗, 9.5 ⇗)
  - A field, one of the following:

        - A field declared in a class (8.3 ⇗)
        - A field declared in an interface (9.3 ⇗)
        - An implicitly declared field of a class corresponding to an enum constant or a record component
        - The field `length`, which is implicitly a member of every array type (10.7 ⇗)
    - A method, one of the following:
        - A method (`abstract` or otherwise) declared in a class (8.4 ⇗)
        - A method (`abstract` or otherwise) declared in an interface (9.4 ⇗)
        - An implicitly declared accessor method corresponding to a record component
- An enum constant (8.9.1 ⇗)
- A record component (8.10.3 ⇗)
- A formal parameter, one of the following:
    - A formal parameter of a method of a class or interface (8.4.1 ⇗)
    - A formal parameter of a constructor of a class (8.8.1 ⇗)
    - A formal parameter of a lambda expression (15.27.1 ⇗)
- An exception parameter of an exception handler declared in a `catch` clause of a `try` statement (14.20 ⇗)
- A local variable, one of the following:
    - A local variable declared by a local variable declaration statement in a block (14.4.2 ⇗)
    - A local variable declared by a `for` statement or a `try`-with-resources statement (14.14 ⇗, 14.20.3 ⇗)
    - A local variable declared by a pattern (14.30.1 ⇗)
    - A component local variable implicitly declared in the block of a derived record creation expression (15.30)
- A local class or interface (14.3 ⇗), one of the following:
    - A local class declared by a normal class declaration
    - A local class declared by an enum declaration
    - A local class declared by an record declaration
    - A local interface declared by a normal interface declaration

Constructors (8.8 ⇗, 8.10.4 ⇗) are also introduced by declarations, but use the name of the class in which they are declared rather than introducing a new name.

A declaration commonly includes an identifier (3.8 ⇗) that can be used in a name to refer to the declared entity. The identifier is constrained to avoid certain contextual keywords when the entity being introduced is a class, interface, or type parameter.

If a declaration does not include an identifier, but instead includes the keyword _ (underscore), then the entity cannot be referred to by name. The following kinds of entity may be declared using an underscore:

- A local variable, one of the following:
  - A local variable declared by a local variable declaration statement (14.4.2 ⇗)
  - A local variable declared by a `for` statement or a `try`-with-resources statement (14.14 ⇗, 14.20.3 ⇗)
  - A local variable declared by a pattern (14.30.1 ⇗)
- An exception parameter of an exception handler declared in a `catch` clause of a `try` statement (14.20 ⇗)
- A formal parameter of a lambda expression (15.27.1 ⇗)

A local variable, exception parameter, or lambda parameter that is declared using an underscore is called an *unnamed local variable*, *unnamed exception parameter*, or *unnamed lambda parameter*, respectively.

> *The rest of this section is unchanged.*

6.3 Scope of a Declaration

The *scope* of a declaration is the region of the program within which the entity declared by the declaration can be referred to using a simple name, provided it is not shadowed ([6.4.1]).

A declaration is said to be *in scope* at a particular point in a program if and only if the declaration's scope includes that point.

The scope of the declaration of an observable top level package (7.4.3 ⇗) is all observable compilation units associated with modules to which the package is uniquely visible (7.4.3 ⇗).

The declaration of a package that is not observable is never in scope.

The declaration of a subpackage is never in scope.

The package `java` is always in scope.

The scope of a class or interface imported by a single-type-import declaration (7.5.1 ⇗) or a type-import-on-demand declaration (7.5.2 ⇗) is the module declaration (7.7 ⇗) and all the class and interface declarations (8.1 ⇗, 9.1 ⇗) of the compilation unit in which the `import` declaration appears, as well as any annotations on the module declaration or package declaration of the compilation unit.

The scope of a member imported by a single-static-import declaration (7.5.3 ⇗) or a static-import-on-demand declaration (7.5.4 ⇗) is the module declaration and all the class and interface declarations of the compilation unit in which the `import` declaration appears, as well as any annotations on the module declaration or package declaration of the compilation unit.

The scope of a top level class or interface (7.6 ⇗) is all class and interface declarations in the package in which the top level class or interface is declared.

The scope of a declaration of a member *m* declared in or inherited by a class or interface *C* (8.2 ⇗, 9.2 ⇗) is the entire body of *C*, including any nested class or interface declarations. If *C* is a record class, then the scope of *m* additionally includes the header of the record declaration of *C*.

The scope of a formal parameter of a method (8.4.1 ⇗), constructor (8.8.1 ⇗), or lambda expression (15.27 ⇗) is the entire body of the method, constructor, or lambda expression.

The scope of a class's type parameter (8.1.2 ⇗) is the type parameter section of the class declaration, and the type parameter section of any superclass type or superinterface type of the

class declaration, and the class body. If the class is a record class (8.10 ⇗), then the scope of the type parameter additionally includes the header of the record declaration (8.10.1 ⇗).

The scope of an interface's type parameter (9.1.2 ⇗) is the type parameter section of the interface declaration, and the type parameter section of any superinterface type of the interface declaration, and the interface body.

The scope of a method's type parameter (8.4.4 ⇗) is the entire declaration of the method, including the type parameter section, but excluding the method modifiers.

The scope of a constructor's type parameter (8.8.4 ⇗) is the entire declaration of the constructor, including the type parameter section, but excluding the constructor modifiers.

The scope of a local class or interface declaration immediately enclosed by a block (14.2 ⇗) is the rest of the immediately enclosing block, including the local class or interface declaration itself.

The scope of a local class or interface declaration immediately enclosed by a switch block statement group (14.11 ⇗) is the rest of the immediately enclosing switch block statement group, including the local class or interface declaration itself.

The scope of a local variable declared in a block by a local variable declaration statement (14.4.2 ⇗) is the rest of the block, starting with the declaration's own initializer and including any further declarators to the right in the local variable declaration statement.

The scope of a local variable declared in the *ForInit* part of a basic `for` statement (14.14.1 ⇗) includes all of the following:

- Its own initializer
- Any further declarators to the right in the *ForInit* part of the `for` statement
- The *Expression* and *ForUpdate* parts of the `for` statement
- The contained *Statement*

The scope of a local variable declared in the header of an enhanced `for` statement (14.14.2 ⇗) is the contained *Statement*.

The scope of a local variable declared in the resource specification of a `try`-with-resources statement (14.20.3 ⇗) is from the declaration rightward over the remainder of the resource specification and the entire `try` block associated with the `try`-with-resources statement.

The translation of a `try`-with-resources statement implies the rule above.

The scope of a parameter of an exception handler that is declared in a `catch` clause of a `try` statement (14.20 ⇗) is the entire block associated with the `catch`.

The scope of a component local variable implicitly declared in the block of a derived record creation expression (15.30) is the block itself.

> *The rest of this section is unchanged.*

6.4 Shadowing and Obscuring

6.4.1 Shadowing

Some declarations may be *shadowed* in part of their scope by another declaration of the same name, in which case a simple name cannot be used to refer to the declared entity.

Shadowing is distinct from hiding (8.3 ⇗, [8.4.8.2], 8.5 ⇗, 9.3 ⇗, 9.5 ⇗), which applies only to members which would otherwise be inherited but are not because of a declaration in a subclass. Shadowing is also distinct from obscuring (6.4.2 ⇗).

A declaration *d* of a type named *n* shadows the declarations of any other types named *n* that are in scope at the point where *d* occurs throughout the scope of *d*.

A declaration *d* of a field or formal parameter named *n* shadows, throughout the scope of *d*, the declarations of any other variables named *n* that are in scope at the point where *d* occurs.

A declaration *d* of a local variable or exception parameter named *n* shadows, throughout the scope of *d*, (a) the declarations of any other fields named *n* that are in scope at the point where *d* occurs, and (b) the declarations of any other variables named *n* that are in scope at the point where *d* occurs but are *not* declared in the innermost class in which *d* is declared.

The (implicit) declaration *d* of a component local variable named *n* in the block of a derived record creation expression (15.30) shadows, throughout the scope of *d*, the declarations of any other variables named *n* that are in scope at the point where *d* occurs.

A declaration *d* of a method named *n* shadows the declarations of any other methods named *n* that are in an enclosing scope at the point where *d* occurs throughout the scope of *d*.

A package declaration never shadows any other declaration.

A type-import-on-demand declaration never causes any other declaration to be shadowed.

A static-import-on-demand declaration never causes any other declaration to be shadowed.

A single-type-import declaration *d* in a compilation unit *c* of package *p* that imports a type named *n* shadows, throughout *c*, the declarations of:

- any top level type named *n* declared in another compilation unit of *p*
- any type named *n* imported by a type-import-on-demand declaration in *c*
- any type named *n* imported by a static-import-on-demand declaration in *c*

A single-static-import declaration *d* in a compilation unit *c* of package *p* that imports a field named *n* shadows the declaration of any static field named *n* imported by a static-import-on-demand declaration in *c*, throughout *c*.

A single-static-import declaration *d* in a compilation unit *c* of package *p* that imports a method named *n* with signature *s* shadows the declaration of any static method named *n* with signature *s* imported by a static-import-on-demand declaration in *c*, throughout *c*.

A single-static-import declaration *d* in a compilation unit *c* of package *p* that imports a type named *n* shadows, throughout *c*, the declarations of:

- any static type named *n* imported by a static-import-on-demand declaration in *c*;
- any top level type (7.6 ⇗) named *n* declared in another compilation unit (7.3 ⇗) of *p*;
- any type named *n* imported by a type-import-on-demand declaration (7.5.2 ⇗) in *c*.

Example 6.4.1-1. Shadowing of a Field Declaration by a Local Variable Declaration

```
class Test {
    static int x = 1;
    public static void main(String[] args) {
        int x = 0;
        System.out.print("x=" + x);
        System.out.println(", Test.x=" + Test.x);
```

```
        }
    }
```

This program produces the output:

```
x=0, Test.x=1
```

This program declares:

- *a class* `Test`
- *a class (*`static`*) variable* `x` *that is a member of the class* `Test`
- *a class method* `main` *that is a member of the class* `Test`
- *a parameter* `args` *of the* `main` *method*
- *a local variable* `x` *of the* `main` *method*

Since the scope of a class variable includes the entire body of the class (8.2 ⇗), the class variable `x` *would normally be available throughout the entire body of the method* `main`*. In this example, however, the class variable* `x` *is shadowed within the body of the method* `main` *by the declaration of the local variable* `x`*.*

A local variable has as its scope the rest of the block in which it is declared (6.3); in this case this is the rest of the body of the `main` *method, namely its initializer "*`0`*" and the invocations of* `System.out.print` *and* `System.out.println`*.*

This means that:

- *The expression* `x` *in the invocation of* `print` *refers to (denotes) the value of the local variable* `x`*.*
- *The invocation of* `println` *uses a qualified name (6.6 ⇗)* `Test.x`*, which uses the class type name* `Test` *to access the class variable* `x`*, because the declaration of* `Test.x` *is shadowed at this point and cannot be referred to by its simple name.*

The keyword `this` *can also be used to access a shadowed field* `x`*, using the form* `this.x`*. Indeed, this idiom typically appears in constructors (8.8 ⇗):*

```
class Pair {
    Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
}
```

Here, the constructor takes parameters having the same names as the fields to be initialized. This is simpler than having to invent different names for the parameters and is not too confusing in this stylized context. In general, however, it is considered poor style to have local variables with the same names as fields.

Example 6.4.1-2. Shadowing of a Type Declaration by Another Type Declaration

```
import java.util.*;

class Vector {
    int[] val = { 1 , 2 };
}

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

```
            Vector v = new Vector();
            System.out.println(v.val[0]);
        }
    }
```

The program compiles and prints:

```
1
```

using the class `Vector` declared here in preference to the generic class `java.util.Vector` (8.1.2 ⇗) that might be imported on demand.

Example 6.4.1-3. Shadowing of a Variables by A Component Local Variable

```
record A(int i, B b) {}
record B(int i, int j) {}
class Test {
    public static A updateBComponents(A arg) {
        return arg with {
                                    // Implicit declaration of i and b
            b = b with {
                                    // Implicit declaration of i and j
                i++;                // Refers to the nested variable i
                j++;
            }
        };
    }
}
```

Here the record class `A` shares a component name (`i`) with its nested record class `B`. In the outer derived record creation expression the local component variables `i` and `b` are implicitly declared in the block, and in the nested derived record creation expression, the local component variables `i` and `j` are declared. The declaration of the nested local component variable `i` shadows the outer declaration. The value of outer component is available using the accessor method, i.e. `arg.i()`.

Chapter 14: Blocks, Statements, and Patterns

14.15 The `break` Statement

A `break` statement transfers control out of an enclosing statement.

BreakStatement:
`break` [*Identifier*] `;`

There are two kinds of `break` statement:

- A `break` statement with no label.
- A `break` statement with the label *Identifier*.

A `break` statement with no label attempts to transfer control to the innermost enclosing `switch`, `while`, `do`, or `for` statement; this enclosing statement, which is called the *break target*, then immediately completes normally.

A `break` statement with label *Identifier* attempts to transfer control to the enclosing labeled statement (14.7 ⇗) that has the same *Identifier* as its label; this enclosing statement, which is called the *break target*, then immediately completes normally. In this case, the break target

need not be a `switch`, `while`, `do`, or `for` statement.

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

It is a compile-time error if the break target contains any method, constructor, instance initializer, static initializer, lambda expression, ~~or~~ switch expression, or derived record creation expression that encloses the `break` statement. That is, there are no non-local jumps.

Execution of a `break` statement with no label always completes abruptly, the reason being a `break` with no label.

Execution of a `break` statement with label *Identifier* always completes abruptly, the reason being a `break` with label *Identifier*.

It can be seen, then, that a `break` 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 break target whose `try` blocks or `catch` clauses contain the `break` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the break target. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `break` statement.

Example 14.15-1. The `break` Statement

In the following example, a mathematical graph is represented by an array of arrays. A graph consists of a set of nodes and a set of edges; each edge is an arrow that points from some node to some other node, or from a node to itself. In this example it is assumed that there are no redundant edges; that is, for any two nodes P and Q, where Q may be the same as P, there is at most one edge from P to Q.

Nodes are represented by integers, and there is an edge from node i to node `edges[i][j]` for every i and j for which the array reference `edges[i][j]` does not throw an `ArrayIndexOutOfBoundsException`.

The task of the method `loseEdges`, given integers i and j, is to construct a new graph by copying a given graph but omitting the edge from node i to node j, if any, and the edge from node j to node i, if any:

```
class Graph {
    int[][] edges;
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
        for (int k = 0; k < n; ++k) {
edgelist:
{
            int z;
search:
{
            if (k == i) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == j) break search;
                }
            } else if (k == j) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == i) break search;
                }
            }
```

```
                // No edge to be deleted; share this list.
                newedges[k] = edges[k];
                break edgelist;
        } //search

                // Copy the list, omitting the edge at position z.
                int   m  = edges[k].length - 1;
                int[] ne = new int[m];
                System.arraycopy(edges[k], 0, ne, 0, z);
                System.arraycopy(edges[k], z+1, ne, z, m-z);
                newedges[k] = ne;
        } //edgelist
            }
            return new Graph(newedges);
        }
    }
```

Note the use of two statement labels, `edgelist` and `search`, and the use of `break` statements. This allows the code that copies a list, omitting one edge, to be shared between two separate tests, the test for an edge from node i to node j, and the test for an edge from node j to node i.

14.16 The `continue` Statement

A `continue` statement may occur only in a `while`, `do`, or `for` statement; statements of these three kinds are called *iteration statements*. Control passes to the loop-continuation point of an iteration statement.

ContinueStatement:
`continue` [*Identifier*] `;`

There are two kinds of `continue` statement:

- A `continue` statement with no label.
- A `continue` statement with the label *Identifier*.

A `continue` statement with no label attempts to transfer control to the innermost enclosing `while`, `do`, or `for` statement; this enclosing statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one.

A `continue` statement with label *Identifier* attempts to transfer control to the enclosing labeled statement (14.7 ⇗) that has the same *Identifier* as its label; this enclosing statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one. In this case, the continue target must be a `while`, `do`, or `for` statement, or a compile-time error occurs.

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

It is a compile-time error if the continue target contains any method, constructor, instance initializer, static initializer, lambda expression, ~~or~~ switch expression, or derived record creation expression that encloses the `continue` statement. That is, there are no non-local jumps.

Execution of a `continue` statement with no label always completes abruptly, the reason being a `continue` with no label.

Execution of a `continue` statement with label *Identifier* always completes abruptly, the reason

being a `continue` with label *Identifier*.

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

See the descriptions of the `while` statement (14.12 ⇗), `do` statement (14.13 ⇗), and `for` statement (14.14 ⇗) for a discussion of the handling of abrupt termination because of `continue`.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any `try` statements (14.20 ⇗) within the continue target whose `try` blocks or `catch` clauses contain the `continue` statement, then any `finally` clauses of those `try` statements are executed, in order, innermost to outermost, before control is transferred to the continue target. Abrupt completion of a `finally` clause can disrupt the transfer of control initiated by a `continue` statement.

Example 14.16-1. The `continue` Statement

In the `Graph` class in 14.15, one of the `break` statements is used to finish execution of the entire body of the outermost `for` loop. This break can be replaced by a `continue` if the `for` loop itself is labeled:

```
class Graph {
    int[][] edges;
    public Graph(int[][] edges) { this.edges = edges; }

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
edgelists:
        for (int k = 0; k < n; ++k) {
            int z;
search:
            {
            if (k == i) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == j) break search;
                }
            } else if (k == j) {
                for (z = 0; z < edges[k].length; ++z) {
                    if (edges[k][z] == i) break search;
                }
            }

            // No edge to be deleted; share this list.
            newedges[k] = edges[k];
            continue edgelists;
} //search

            // Copy the list, omitting the edge at position z.
            int   m  = edges[k].length - 1;
            int[] ne = new int[m];
            System.arraycopy(edges[k], 0, ne, 0, z);
            System.arraycopy(edges[k], z+1, ne, z, m-z);
            newedges[k] = ne;
        } //edgelists
        return new Graph(newedges);
    }
}
```

Which to use, if either, is largely a matter of programming style.

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 initializer, a ~~or~~ static initializer ~~that encloses the `return` statement~~, ~~or (ii)~~ a `switch` expression, or a derived record creation 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 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.*

14.21 The `yield` Statement

A `yield` statement transfers control by causing an enclosing `switch` expression (15.28 ⇗) to

produce a specified value.

YieldStatement:
`yield` *Expression* `;`

A `yield` statement attempts to transfer control to the innermost enclosing `switch` expression; this enclosing expression, which is called the *yield target*, then immediately completes normally and the value of the *Expression* becomes the value of the `switch` expression.

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

It is a compile-time error if the yield target contains any method, constructor, instance initializer, static initializer, ~~or~~ lambda expression, or derived record creation expression that encloses the `yield` statement. That is, there are no non-local jumps.

It is a compile-time error if the *Expression* of a `yield` statement is void (15.1 ⇗).

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

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

Example 14.21-1. The `yield` Statement

In the following example, a `yield` statement is used to produce a value for the enclosing `switch` expression.

```
class Test {
    enum Day {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
        SATURDAY, SUNDAY
    }

    public int calculate(Day d) {
        return switch (d) {
            case SATURDAY, SUNDAY -> d.ordinal();
            default -> {
                int len = d.toString().length();
                yield len*len;
            }
        };
    }
}
```

Chapter 15: Expressions

15.8 Primary Expressions

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

Primary:
PrimaryNoNewArray

ArrayCreationExpression

PrimaryNoNewArray:
Literal
ClassLiteral
`this`
TypeName `.` `this`
`(` *Expression* `)`
ClassInstanceCreationExpression
DerivedRecordCreationExpression
FieldAccess
ArrayAccess
MethodInvocation
MethodReference

This part of the grammar of the Java programming language is unusual, in two ways. First, one might expect simple names, such as names of local variables and method parameters, to be primary expressions. For technical reasons, names are grouped together with primary expressions a little later when postfix expressions are introduced (15.14 ⇗).

The technical reasons have to do with allowing left-to-right parsing of Java programs with only one-token lookahead. Consider the expressions `(z[3])` and `(z[])`. The first is a parenthesized array access (15.10.3 ⇗) and the second is the start of a cast (15.16 ⇗). At the point that the look-ahead symbol is `[`, a left-to-right parse will have reduced the `z` to the nonterminal Name. In the context of a cast we prefer not to have to reduce the name to a Primary, but if Name were one of the alternatives for Primary, then we could not tell whether to do the reduction (that is, we could not determine whether the current situation would turn out to be a parenthesized array access or a cast) without looking ahead two tokens, to the token following the `[`. The grammar presented here avoids the problem by keeping Name and Primary separate and allowing either in certain other syntax rules (those for ClassInstanceCreationExpression, MethodInvocation, ArrayAccess, and PostfixExpression, though not FieldAccess because it uses an identifier directly). This strategy effectively defers the question of whether a Name should be treated as a Primary until more context can be examined.

The second unusual feature avoids a potential grammatical ambiguity in the expression "`new int[3][3]`" which in Java always means a single creation of a multidimensional array, but which, without appropriate grammatical finesse, might also be interpreted as meaning the same as "`(new int[3])[3]`".

This ambiguity is eliminated by splitting the expected definition of Primary into Primary and PrimaryNoNewArray. (This may be compared to the splitting of Statement into Statement and StatementNoShortIf (14.5 ⇗) to avoid the "dangling else" problem.)

15.14 Postfix Expressions

15.14.2 Postfix Increment Operator ++

A postfix expression followed by a `++` operator is a postfix increment expression.

PostIncrementExpression:
PostfixExpression `++`

The result of the postfix expression must be a variable of a type that is convertible (5.1.8 ⇗) to a numeric type, or a compile-time error occurs.

The type of the postfix increment expression is the type of the variable. The result of the postfix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value `1` is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (5.6 ⇗) is performed on the value `1` and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (5.1.3 ⇗) and/or subjected to boxing conversion (5.1.7 ⇗) to the type of the variable before it is stored. The value of the postfix increment expression is the value of the variable *before* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (5.1.8 ⇗).

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

If the operand expression is a named variable then it is a compile-time error if the postfix increment expression is contained in the block of a derived record creation expression (15.30) and the declaration of the variable is not contained in the *Block*.

It is intended that code appearing in the block of a derived record creation expression does not assign to variables declared outside the derived record creation expression. For example:

```
record Point(int x, int y) { }
int i = 42;
Point p = new Point(1, 0);
p = p with {
    x = x + 1;      // Ok
    y = y + 1;      // Ok
    i = i++;        // Error
};
```

15.14.3 Postfix Decrement Operator --

A postfix expression followed by a `--` operator is a postfix decrement expression.

PostDecrementExpression:
PostfixExpression `--`

The result of the postfix expression must be a variable of a type that is convertible (5.1.8 ⇗) to a numeric type, or a compile-time error occurs.

The type of the postfix decrement expression is the type of the variable. The result of the postfix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the postfix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value `1` is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (5.6 ⇗) is performed on the value `1` and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (5.1.3 ⇗) and/or subjected to boxing conversion (5.1.7 ⇗) to the type of the variable before it is stored. The value of the postfix decrement expression is the value of the variable *before* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (5.1.8 ⇗).

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

used as the operand of a postfix decrement operator.

If the operand expression is a named variable then it is a compile-time error if the postfix decrement expression is contained in the *Block* of a derived record creation expression (15.30) and the declaration of the variable is not contained in the *Block*.

It is intended that code appearing in the Block of a derived record creation expression does not assign to variables declared outside the derived record creation expression. For example:

```
record Point(int x, int y) { }
int i = 42;
Point p = new Point(1, 0);
p = p with {
    x = x + 1;      // Ok
    y = y + 1;      // Ok
    i = i--;        // Error
};
```

15.15 Unary Operators

15.15.1 Prefix Increment Operator ++

A unary expression preceded by a `++` operator is a prefix increment expression.

The result of the unary expression must be a variable of a type that is convertible (5.1.8 ⇗) to a numeric type, or a compile-time error occurs.

The type of the prefix increment expression is the type of the variable. The result of the prefix increment expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix increment expression completes abruptly for the same reason and no incrementation occurs. Otherwise, the value `1` is added to the value of the variable and the sum is stored back into the variable. Before the addition, binary numeric promotion (5.6 ⇗) is performed on the value `1` and the value of the variable. If necessary, the sum is narrowed by a narrowing primitive conversion (5.1.3 ⇗) and/or subjected to boxing conversion (5.1.7 ⇗) to the type of the variable before it is stored. The value of the prefix increment expression is the value of the variable *after* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (5.1.8 ⇗).

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

If the operand expression is a named variable then it is a compile-time error if the prefix increment expression is contained in the *Block* of a derived record creation expression (15.30) and the declaration of the variable is not contained in the *Block*.

It is intended that code appearing in the Block of a derived record creation expression does not assign to variables declared outside the derived record creation expression. For example:

```
record Point(int x, int y) { }
int i = 42;
Point p = new Point(1, 0);
p = p with {
    x = x + 1;      // Ok
    y = y + 1;      // Ok
    i = ++i;        // Error
```

```
};
```

15.15.2 Prefix Decrement Operator --

A unary expression preceded by a -- operator is a prefix decrement expression.

The result of the unary expression must be a variable of a type that is convertible (5.1.8 ⇗) to a numeric type, or a compile-time error occurs.

The type of the prefix decrement expression is the type of the variable. The result of the prefix decrement expression is not a variable, but a value.

At run time, if evaluation of the operand expression completes abruptly, then the prefix decrement expression completes abruptly for the same reason and no decrementation occurs. Otherwise, the value `1` is subtracted from the value of the variable and the difference is stored back into the variable. Before the subtraction, binary numeric promotion (5.6 ⇗) is performed on the value `1` and the value of the variable. If necessary, the difference is narrowed by a narrowing primitive conversion (5.1.3 ⇗) and/or subjected to boxing conversion (5.1.7 ⇗) to the type of the variable before it is stored. The value of the prefix decrement expression is the value of the variable *after* the new value is stored.

Note that the binary numeric promotion mentioned above may include unboxing conversion (5.1.8 ⇗).

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

If the operand expression is a named variable then it is a compile-time error if the prefix decrement expression is contained in the *Block* of a derived record creation expression (15.30) and the declaration of the variable is not contained in the *Block*.

It is intended that code appearing in the Block of a derived record creation expression does not assign to variables declared outside the derived record creation expression. For example:

```
record Point(int x, int y) { }
int i = 42;
Point p = new Point(1, 0);
p = p with {
    x = x + 1;      // Ok
    y = y + 1;      // Ok
    i = --i;        // Error
};
```

15.26 Assignment Operators

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

AssignmentExpression:
    ConditionalExpression
    Assignment

Assignment:
    LeftHandSide AssignmentOperator Expression

LeftHandSide:
    ExpressionName

FieldAccess
ArrayAccess

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

The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs.

This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access (15.11 ⇗) or an array access (15.10.3 ⇗).

The type of the assignment expression is the type of the variable after capture conversion (5.1.10 ⇗).

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

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

If the first operand of an assignment operator is a named variable then it is a compile-time error if the assignment expression is contained in the *Block* of a derived record creation expression (15.30) and the declaration of the variable is not contained in the *Block*.

It is intended that code appearing in the Block of a derived record creation expression does not assign to variables declared outside the derived record creation expression. For example:

```
record Point(int x, int y) { }
int i = 42;
Point p = new Point(1, 0);
p = p with {
    x = x + 1;      // Ok
    y = y + 1;      // Ok
    i = i + 1;      // Error
};
```

15.30 Derived Record Creation Expression

A derived record creation expression provides a compact means for creating a new instance of a record class whose state is derived from that of another instance of the same record class utilizing a block of transformation code.

DerivedRecordCreationExpression:
PrimaryNoNewArray `with` *Block*

The expression on the left-hand side is known as the *origin expression*.

Restrictions on `break`, `continue`, `return`, and `yield` statements (14.15, 14.16, 14.17, 14.21) ensure that it is only possible to transfer control out of the Block of a derived record creation expression by completing normally or completing abruptly because an exception has been thrown.

Before the first statement, if any, in the *Block* a number of local variables are declared implicitly. For each record component, if any, in the header of the record class named by the type of the origin expression, a local variable with the same name and type is declared. These

local variables are known as the *component local variables*.

Restrictions to assignment expressions (15.26), postfix expressions (15.14), prefix increment expressions (15.15.1) and prefix decrement expressions (15.15.2) ensure that any assignment to variables with a simple name in the Block of a derived record creation expression is only permitted if the variable is declared in the Block.

Without these restrictions, the following code would not only initialize the `nextPoint` record value but, as a side-effect, double the value of `counter`.

```
record Point(int x, int y, int z) { }

int counter = 42;
Point startingPoint = new Point(0, 0, 0);
Point nextPoint = startingPoint with {
    x++;
    y++;
    z++;
    counter = counter * 2;      // Error
};
```

In contrast, the following code is allowed:

```
Point finalPoint = nextPoint with {
    int counter = x;
    if (is.Even(y))
        counter = counter * 2;   // Assignment to local allowed
    z = counter;
};
```

The type of the origin expression must be a record class type, or a compile-time error occurs.

The type of a derived record creation expression is given by the type of the origin expression.

At run time, evaluation of a derived record creation expression proceeds as follows:

1. The origin expression, whose type names a record class *R*, is evaluated. If evaluation of the origin expression completes abruptly, then evaluation of the derived record creation expression completes abruptly for the same reason.
2. If the value of the origin expression is `null` then evaluation of the derived record creation expression completes abruptly with a `NullPointerException`.
3. Before executing the *Block*, the component local variables, if any, are initialized. The component local variables are initialized in the order that they appear in the header of the record class *R*, and are initialized to the value given as if by invoking the corresponding component accessor method on the value of the origin expression.

   If invoking the corresponding component accessor method on the value of the origin expression would complete abruptly, then evaluation of the derived instance creation expression completes abruptly for the same reason.
4. The *Block* is executed. If execution of the *Block* completes abruptly, then evaluation of the derived record creation expression completes abruptly for the same reason.
5. After the final statement, if any, of the *Block* has completed normally, a new instance of record class *R* is created as if by evaluating a new class instance creation expression (15.9) with the compile-time type of the origin expression and an argument list containing the component local variables, if any, in the order that they appear in the header of the record class *R*.

The resulting instance of the record class *R* is taken as the overall value of the derived record creation expression. If evaluating the new class creation expression would complete abruptly, then evaluation of the derived instance creation expression completes abruptly for the same reason.

The Block need only mention the component values of the origin record value that are being modified, the other values are simply copied. For example:

```
Point originalPoint = new Point(3, 4, 5);
Point translatedPoint = originalPoint with {
    y = 0;
};
```

The values of the `x` and `z` components of `translatedPoint` are `3` and `5`, respectively.

If the Block is empty then the result of the derived record creation expression is a copy of the origin record value.

Chapter 16: Definite Assignment

16.1 Definite Assignment and Expressions

16.1.10 Other Expressions

If an expression is not a boolean constant expression, and is not a preincrement expression `++`*a*, predecrement expression `--`*a*, postincrement expression *a*`++`, postdecrement expression *a*`--`, logical complement expression `!`*a*, conditional-and expression *a* `&&` *b*, conditional-or expression *a* `||` *b*, conditional expression *a* `?` *b* `:` *c*, assignment expression, ~~or~~ lambda expression, or derived record creation expression, then the following rules apply:

- If the expression has no subexpressions, *V* is [un]assigned after the expression iff *V* is [un]assigned before the expression.

  This case applies to literals, names, `this` (both qualified and unqualified), unqualified class instance creation expressions with no arguments, array creation expressions with initializers that contain no expressions, superclass field access expressions, unqualified and type-qualified method invocation expressions with no arguments, superclass method invocation expressions with no arguments, and superclass and type-qualified method reference expressions.

- If the expression has subexpressions, *V* is [un]assigned after the expression iff *V* is [un]assigned after its rightmost immediate subexpression.

There is a piece of subtle reasoning behind the assertion that a variable V can be known to be definitely unassigned after a method invocation expression. Taken by itself, at face value and without qualification, such an assertion is not always true, because an invoked method can perform assignments. But it must be remembered that, for the purposes of the Java programming language, the concept of definite unassignment is applied only to blank `final` variables. If V is a blank `final` local variable, then only the method to which its declaration belongs can perform assignments to V. If V is a blank `final` field, then only a constructor or an initializer for the class containing the declaration for V can perform assignments to V; no method can perform assignments to V. Finally, explicit constructor invocations (8.8.7.1 ⇗) are handled specially (16.9 ⇗); although they are syntactically similar to expression statements containing method invocations, they are not expression statements and therefore the rules of this section do not apply to explicit constructor invocations.

If an expression is a lambda expression, then the following rules apply:

- *V* is [un]assigned after the expression iff *V* is [un]assigned before the expression.

- *V* is definitely assigned before the expression or block that is the lambda body (15.27.2) iff *V* is definitely assigned before the lambda expression.

  No rule allows V to be definitely unassigned before a lambda body. This is by design: a variable that was definitely unassigned before the lambda body may end up being assigned to later on, so we cannot conclude that the variable will be unassigned when the body is executed.

If an expression is a derived record creation expression, then the following rules apply:

- *V* is [un]assigned before the origin expression iff *V* is [un]assigned before the expression.
- *V* is [un]assigned after the expression iff *V* is [un]assigned after the origin expression.

  Restrictions on the Block of a derived record creation expression (15.30), mean that no assignment to a variable declared outside the Block need be considered.

  The component local variables are declared and initialized implicitly before the first statement, if any, in the Block of a derived record creation expression. This means that these variables are all definitely assigned before any statement in the Block.

For any immediate subexpression *y* of an expression *x*, where *x* is not a lambda expression or a derived record creation expression, *V* is [un]assigned before *y* iff one of the following is true:

- *y* is the leftmost immediate subexpression of *x* and *V* is [un]assigned before *x*.
- *y* is the right-hand operand of a binary operator and *V* is [un]assigned after the left-hand operand.
- *x* is an array access, *y* is the subexpression within the brackets, and *V* is [un]assigned after the subexpression before the brackets.
- *x* is a primary method invocation expression, *y* is the first argument expression in the method invocation expression, and *V* is [un]assigned after the primary expression that computes the target object.
- *x* is a method invocation expression or a class instance creation expression; *y* is an argument expression, but not the first; and *V* is [un]assigned after the argument expression to the left of *y*.
- *x* is a qualified class instance creation expression, *y* is the first argument expression in the class instance creation expression, and *V* is [un]assigned after the primary expression that computes the qualifying object.
- *x* is an array creation expression; *y* is a dimension expression, but not the first; and *V* is [un]assigned after the dimension expression to the left of *y*.
- *x* is an array creation expression initialized via an array initializer; *y* is the array initializer in *x*; and *V* is [un]assigned after the dimension expression to the left of *y*.

DRAFT 23-internal-adhoc.gbierman.20240326