Record Classes

Changes to the Java® Language Specification • Version 16-internal+0-adhoc.gbierman.20200923

This document describes changes to the Java Language Specification, as modified by Consistent Class and Interface Terminology and Local Static Interfaces and Enum Classes, to support *Record Classes*, a feature of Java SE 16. See the draft JEP for an overview of the feature.

The changes are the same as those in the second preview of Records in Java SE 15, except for minor editorial changes and the following:

- To relax the current restriction on an inner class from declaring a member that is explicitly or implicitly static. This will now be permitted and, in particular, will allow an inner class to declare a record class member. (These changes are detailed in the document on Local Static Interfaces and Enum Classes.)
- Add text to explicitly rule out using C-style array declaration of record components.
- Clarify that any annotations on record components that apply to the implicitly declared accessor method must satisfy the existing rules for annotating a method declaration.

A companion document describes the changes needed to the Java Virtual Machine Specification to support record classes.

A further companion document describes changes to the Java Object Serialization Specification to support serializable record classes.

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.

Chapter 1: Introduction

1.1 Organization of the Specification

...

Chapter 8 describes classes. The members of classes are classes, interfaces, fields (variables) and methods. Class variables exist once per class. Class methods operate without reference to a specific object. Instance variables are dynamically created in objects that are instances of classes. Instance methods are invoked on instances of classes; such instances become the current object `this` during their execution, supporting the object-oriented programming style.

Classes support single inheritance, in which each class has a single superclass. Each class inherits members from its superclass, and ultimately from the class `Object`. Variables of a class type can reference an instance of that class or of any subclass of that class, allowing new types to be used with existing methods, polymorphically.

Classes support concurrent programming with `synchronized` methods. Methods declare the checked exceptions that can arise from their execution, which allows compile-time checking to ensure that exceptional conditions are handled. Objects can declare a `finalize` method that will be invoked before the objects are discarded by the garbage collector, allowing the objects to clean up their state.

For simplicity, the language has neither declaration "headers" separate from the implementation of a class nor separate type and class hierarchies.

~~A special form of classes, enum classes, support the definition of small sets of values and their manipulation in a type safe manner.~~ Enum classes are a special kind of class that support the definition of small sets of values which can then be used in a type safe manner. Unlike

enumerations in other languages, enum constants are objects and may have their own methods.

Record classes are another special kind of class that support the compact expression of simple objects that serve as aggregates of values.

...

1.5 Preview Features

> *The following text will be added to the description of the preview feature: Record Types.*

The following are essential API elements associated with Record Types:

- The class `java.lang.Record`.
- The enum constant `RECORD_COMPONENT` in `java.lang.annotation.ElementType`.

Chapter 3: Lexical Structure

3.8 Identifiers

An *identifier* is an unlimited-length sequence of *Java letters* and *Java digits*, the first of which must be a *Java letter*.

Identifier:
IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral

IdentifierChars:
JavaLetter {*JavaLetterOrDigit*}

JavaLetter:
any Unicode character that is a "Java letter"

JavaLetterOrDigit:
any Unicode character that is a "Java letter-or-digit"

A "Java letter" is a character for which the method `Character.isJavaIdentifierStart(int)` returns true.

A "Java letter-or-digit" is a character for which the method `Character.isJavaIdentifierPart(int)` returns true.

The "Java letters" include uppercase and lowercase ASCII Latin letters `A-Z` (`\u0041-\u005a`), and `a-z` (`\u0061-\u007a`), and, for historical reasons, the ASCII dollar sign (`$`, or `\u0024`) and underscore (`_`, or `\u005f`). The dollar sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems. The underscore may be used in identifiers formed of two or more characters, but it cannot be used as a one-character identifier due to being a keyword.

The "Java digits" include the ASCII digits `0-9` (`\u0030-\u0039`).

Letters and digits may be drawn from the entire Unicode character set, which supports most writing scripts in use in the world today, including the large sets for Chinese, Japanese, and Korean. This allows programmers to use identifiers in their programs that are written in their native languages.

An identifier cannot have the same spelling (Unicode character sequence) as a keyword (3.9), boolean literal (3.10.3⇗), or the null literal (3.10.7⇗), or a compile-time error occurs.

Two identifiers are the same only if, after ignoring characters that are ignorable, the identifiers have the same Unicode character for each letter or digit. An ignorable character is a character for which the method `Character.isIdentifierIgnorable(int)` returns true. Identifiers that have the same external appearance may yet be different.

For example, the identifiers consisting of the single letters LATIN CAPITAL LETTER A (`A`, `\u0041`), LATIN SMALL LETTER A (`a`, `\u0061`), GREEK CAPITAL LETTER ALPHA (`A`, `\u0391`), CYRILLIC SMALL LETTER A (`a`, `\u0430`) and MATHEMATICAL BOLD ITALIC SMALL A (`a`, `\ud835\udc82`) are all different.

Unicode composite characters are different from their canonical equivalent decomposed characters. For example, a LATIN CAPITAL LETTER A ACUTE (`Á`, `\u00c1`) is different from a LATIN CAPITAL LETTER A (`A`, `\u0041`) immediately followed by a NON-SPACING ACUTE (`´`, `\u0301`) in identifiers. See The Unicode Standard, Section 3.11 "Normalization Forms".

Examples of identifiers are:

- *`String`*
- *`i3`*
- *`αρετη`*
- *`MAX_VALUE`*
- *`isLetterOrDigit`*

The identifiers `var`, ~~and~~ `yield`, and `record` are *restricted identifiers* because they are not allowed in some contexts.

A *type identifier* is ~~an identifier that is not the character sequence `var` or the character sequence `yield`~~ any identifier other than the character sequences `var`, `yield`, and `record`.

TypeIdentifier:
Identifier but not `var`, ~~*or*~~ `yield` or `record`

*Type identifiers are used in certain contexts involving the declaration or use of types. For example, the name of a class must be a TypeIdentifier, so it is illegal to declare a class named `var`, ~~*or*~~ `yield`, or `record` (8.1⇗).*

An *unqualified method identifier* is an identifier that is not the character sequence `yield`.

UnqualifiedMethodIdentifier:
Identifier but not `yield`

This restriction allows `yield` to be used in a `yield` statement (14.21⇗) and still also be used as a (qualified) method name for compatibility reasons.

3.9 Keywords

51 character sequences, formed from ASCII letters, are reserved for use as keywords and cannot be used as identifiers (3.8).

Keyword:
(one of)

```
abstract continue for new switch
assert default if package synchronized
boolean do goto private this
break double implements protected throw
byte else import public throws
case enum instanceof return transient
```

```
catch extends int short try
char final interface static void
class finally long strictfp volatile
const float native super while
```
_ (underscore)

The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs. The keyword _ (underscore) is reserved for possible future use in parameter declarations.

A variety of character sequences are sometimes assumed, incorrectly, to be keywords:

- *`true` and `false` are not keywords, but rather boolean literals (3.10.3⇗).*
- *`null` is not a keyword, but rather the null literal (3.10.7⇗).*
- *`var`, ~~and~~ `yield`, and `record` are not keywords, but rather restricted identifiers (3.8). `var` has special meaning as the type of a local variable declaration (14.4⇗, 14.14.1⇗, 14.14.2⇗, 14.20.3⇗) and the type of a lambda formal parameter (15.27.1⇗). `yield` has special meaning in a `yield` statement (14.21⇗). All invocations of a method named `yield` must be qualified so as to be distinguished from a `yield` statement. `record` has special meaning in a record declaration (8.10).*

A further ten character sequences are *restricted keywords*: `open`, `module`, `requires`, `transitive`, `exports`, `opens`, `to`, `uses`, `provides`, and `with`. These character sequences are tokenized as keywords solely where they appear as terminals in the *ModuleDeclaration*, *ModuleDirective*, and *RequiresModifier* productions (7.7⇗). They are tokenized as identifiers everywhere else, for compatibility with programs written before the introduction of restricted keywords. There is one exception: immediately to the right of the character sequence `requires` in the *ModuleDirective* production, the character sequence `transitive` is tokenized as a keyword unless it is followed by a separator, in which case it is tokenized as an identifier.

Chapter 4: Types, Values, and Variables

4.11 Where Types Are Used

Types are used in most kinds of declaration and in certain kinds of expression. Specifically, there are ~~16~~ 17 *type contexts* where types are used:

- In declarations:
    1. A type in the `extends` or `implements` clause of a class declaration (8.1.4, 8.1.5⇗, 8.5, 9.5⇗)
    2. A type in the `extends` clause of an interface declaration (9.1.3⇗, 8.5, 9.5⇗)
    3. The return type of a method (including the type of an element of an annotation type) (8.4.5⇗, 9.4⇗, 9.6.1⇗)
    4. A type in the `throws` clause of a method or constructor (8.4.6⇗, 8.8.5⇗, 9.4⇗)
    5. A type in the `extends` clause of a type parameter declaration of a generic class, interface, method, or constructor (8.1.2⇗, 9.1.2⇗, 8.4.4⇗, 8.8.4⇗)
    6. The type in a field declaration of a class or interface (including an enum constant) (8.3⇗, 9.3⇗, 8.9.1⇗)

7. The type in a formal parameter declaration of a method, constructor, or lambda expression (8.4.1, 8.8.1, 9.4, 15.27.1)
8. The type of the receiver parameter of a method (8.4)
9. The type in a local variable declaration (14.4, 14.14.1, 14.14.2, 14.20.3)
10. The type in an exception parameter declaration (14.20)
11. The type of a record component in a record declaration (8.10.1)

- In expressions:
  1. A type in the explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (8.8.7.1, 15.9, 15.12)
  2. In an unqualified class instance creation expression, as the class type to be instantiated (15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5)
  3. The element type in an array creation expression (15.10.1)
  4. The type in the cast operator of a cast expression (15.16)
  5. The type that follows the `instanceof` relational operator (15.20.2)
  6. In a method reference expression (15.13), as the reference type to search for a member method or as the class type or array type to construct.

Also, types are used as:

- The element type of an array type in any of the above contexts; and
- A non-wildcard type argument, or a bound of a wildcard type argument, of a parameterized type in any of the above contexts.

Finally, there are three special terms in the Java programming language which denote the use of a type:

- An unbounded wildcard (4.5.1)
- The `...` in the type of a variable arity parameter (8.4.1), to indicate an array type
- The simple name of a type in a constructor declaration (8.8), to indicate the class of the constructed object

The meaning of types in type contexts is given by:

- 4.2, for primitive types
- 4.4, for type parameters
- 4.5, for class and interface types that are parameterized, or appear either as type arguments in a parameterized type or as bounds of wildcard type arguments in a parameterized type
- 4.8, for class and interface types that are raw
- 4.9, for intersection types in the bounds of type parameters
- 6.5, for class and interface types in contexts where genericity is unimportant (6.1)
- 10.1, for array types

Some type contexts restrict how a reference type may be parameterized:

- The following type contexts require that if a type is a parameterized reference type, it has no wildcard type arguments:
  - In an `extends` or `implements` clause of a class declaration (8.1.4, 8.1.5)
  - In an `extends` clause of an interface declaration (9.1.3)
  - In an unqualified class instance creation expression, as the class type to be instantiated (15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5)
  - In a method reference expression (15.13), as the reference type to search for a member method or as the class type or array type to construct.

  In addition, no wildcard type arguments are permitted in the explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression or method reference expression (8.8.7.1, 15.9, 15.12, 15.13).
- The following type contexts require that if a type is a parameterized reference type, it has only unbounded wildcard type arguments (i.e. it is a reifiable type) :
  - As the element type in an array creation expression (15.10.1)
  - As the type that follows the `instanceof` relational operator (15.20.2)
- The following type contexts disallow a parameterized reference type altogether, because they involve exceptions and the type of an exception is non-generic (6.1):
  - As the type of an exception that can be thrown by a method or constructor (8.4.6, 8.8.5, 9.4)
  - In an exception parameter declaration (14.20)

In any type context where a type is used, it is possible to annotate the keyword denoting a primitive type or the Identifier denoting the simple name of a reference type. It is also possible to annotate an array type by writing an annotation to the left of the `[` at the desired level of nesting in the array type. Annotations in these locations are called type annotations, and are specified in 9.7.4. Here are some examples:

- *`@Foo int[] f;` annotates the primitive type `int`*
- *`int @Foo [] f;` annotates the array type `int[]`*
- *`int @Foo [][] f;` annotates the array type `int[][]`*
- *`int[] @Foo [] f;` annotates the array type `int[]` which is the component type of the array type `int[][]`*

~~Five~~ Six of the type contexts which appear in declarations occupy the same syntactic real estate as a number of declaration contexts (9.6.4.1):

- *The return type of a method (including the type of an element of an annotation type)*
- *The type in a field declaration of a class or interface (including an enum constant)*
- *The type in a record component declaration of a record class*
- *The type in a formal parameter declaration of a method, constructor, or lambda expression*
- *The type in a local variable declaration*
- *The type in an exception parameter declaration*

The fact that the same syntactic location in a program can be both a type context and a declaration

context arises because the modifiers for a declaration immediately precede the type of the declared entity. 9.7.4 explains how an annotation in such a location is deemed to appear in a type context or a declaration context or both.

Example 4.11-1. Usage of a Type

```
import java.util.Random;
import java.util.Collection;
import java.util.ArrayList;

class MiscMath<T extends Number> {
    int divisor;
    MiscMath(int divisor) { this.divisor = divisor; }
    float ratio(long l) {
        try {
            l /= divisor;
        } catch (Exception e) {
            if (e instanceof ArithmeticException)
                l = Long.MAX_VALUE;
            else
                l = 0;
        }
        return (float)l;
    }
    double gausser() {
        Random r = new Random();
        double[] val = new double[2];
        val[0] = r.nextGaussian();
        val[1] = r.nextGaussian();
        return (val[0] + val[1]) / 2;
    }
    Collection<Number> fromArray(Number[] na) {
        Collection<Number> cn = new ArrayList<Number>();
        for (Number n : na) cn.add(n);
        return cn;
    }
    <S> void loop(S s) { this.<S>loop(s); }
}
```

In this example, types are used in declarations of the following:

- *Imported types (7.5⇗); here the type `Random`, imported from the type `java.util.Random` of the package `java.util`, is declared*
- *Fields, which are the class variables and instance variables of classes (8.3⇗), and constants of interfaces (9.3⇗); here the field `divisor` in the class `MiscMath` is declared to be of type `int`*
- *Method parameters (8.4.1⇗); here the parameter `l` of the method `ratio` is declared to be of type `long`*
- *Method results (8.4⇗); here the result of the method `ratio` is declared to be of type `float`, and the result of the method `gausser` is declared to be of type `double`*
- *Constructor parameters (8.8.1⇗); here the parameter of the constructor for `MiscMath` is declared to be of type `int`*
- *Local variables (14.4⇗, 14.14⇗); the local variables `r` and `val` of the method `gausser` are declared to be of types `Random` and `double[]` (array of `double`)*
- *Exception parameters (14.20⇗); here the exception parameter `e` of the `catch` clause is declared to be of type `Exception`*

- *Type parameters (4.4⇗); here the type parameter of `MiscMath` is a type variable `T` with the type `Number` as its declared bound*
- *In any declaration that uses a parameterized type; here the type `Number` is used as a type argument (4.5.1⇗) in the parameterized type `Collection<Number>`.*

and in expressions of the following kinds:

- *Class instance creations (15.9⇗); here a local variable `r` of method `gausser` is initialized by a class instance creation expression that uses the type `Random`*
- *Generic class (8.1.2⇗) instance creations (15.9⇗); here `Number` is used as a type argument in the expression `new ArrayList<Number>()`*
- *Array creations (15.10.1⇗); here the local variable `val` of method `gausser` is initialized by an array creation expression that creates an array of `double` with size 2*
- *Generic method (8.4.4⇗) or constructor (8.8.4⇗) invocations (15.12⇗); here the method `loop` calls itself with an explicit type argument `S`*
- *Casts (15.16⇗); here the `return` statement of the method `ratio` uses the `float` type in a cast*
- *The `instanceof` operator (15.20.2⇗); here the `instanceof` operator tests whether `e` is assignment-compatible with the type `ArithmeticException`*

Chapter 6: Names

6.1 Declarations

A *declaration* introduces an entity into a program and includes an identifier (3.8) that can be used in a name to refer to this entity. The identifier is constrained to be a type identifier when the entity being introduced is a class, interface, or type parameter.

A declared entity is one of the following:

- 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 in a class declaration (8.1⇗) or an enum declaration (8.9⇗)
- An interface, declared in an interface declaration (9.1⇗) or an annotation 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 (8.9.3) or a record component (8.10.3)
        - 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 enum constant (8.9.1)
- A record component (8.10.3)
- A formal parameter of a method of a class or interface (8.4.1), a constructor of a class (8.8.1), or 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 in a block (14.4)
    - A local variable declared in a `for` statement (14.14)
- A local class or interface (14.3), declared in one of the following:
    - A class declaration
    - An enum declaration
    - A record declaration
    - An interface declaration

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

...

6.5 Determining the Meaning of a Name

6.5.1 Syntactic Classification of a Name According to Context

A name is syntactically classified as a *ModuleName* in these contexts:

- In a `requires` directive in a module declaration (7.7.1)
- To the right of `to` in an `exports` or `opens` directive in a module declaration (7.7.2)

A name is syntactically classified as a *PackageName* in these contexts:

- To the right of `exports` or `opens` in a module declaration
- To the left of the "." in a qualified *PackageName*

A name is syntactically classified as a *TypeName* in these contexts:

- The first eleven non-generic contexts (6.1):
    1. In a `uses` or `provides` directive in a module declaration (7.7.1)
    2. In a single-type-import declaration (7.5.1)

3. To the left of the `.` in a single-static-import declaration (7.5.3)
4. To the left of the `.` in a static-import-on-demand declaration (7.5.4)
5. To the left of the `(` in a constructor declaration (8.8)
6. After the `@` sign in an annotation (9.7)
7. To the left of `.class` in a class literal (15.8.2)
8. To the left of `.this` in a qualified `this` expression (15.8.4)
9. To the left of `.super` in a qualified superclass field access expression (15.11.2)
10. To the left of *.Identifier* or `.super.`*Identifier* in a qualified method invocation expression (15.12)
11. To the left of `.super::` in a method reference expression (15.13)

- As the *Identifier* or dotted *Identifier* sequence that constitutes any *ReferenceType* (including a *ReferenceType* to the left of the brackets in an array type, or to the left of the < in a parameterized type, or in a non-wildcard type argument of a parameterized type, or in an `extends` or `super` clause of a wildcard type argument of a parameterized type) in the ~~16~~ 17 contexts where types are used (4.11):
    1. In an `extends` or `implements` clause of a class declaration (8.1.4, 8.1.5, 8.5, 9.5)
    2. In an `extends` clause of an interface declaration (9.1.3)
    3. The return type of a method (8.4, 9.4) (including the type of an element of an annotation type (9.6.1))
    4. In the `throws` clause of a method or constructor (8.4.6, 8.8.5, 9.4)
    5. In an `extends` clause of a type parameter declaration of a generic class, interface, method, or constructor (8.1.2, 9.1.2, 8.4.4, 8.8.4)
    6. The type in a field declaration of a class or interface (8.3, 9.3)
    7. The type in a formal parameter declaration of a method, constructor, or lambda expression (8.4.1, 8.8.1, 9.4, 15.27.1)
    8. The type of the receiver parameter of a method (8.4)
    9. The type in a local variable declaration (14.4, 14.14.1, 14.14.2, 14.20.3)
    10. A type in an exception parameter declaration (14.20)
    11. The type of a record component (8.10.1)

    ~~11.~~ 12. In an explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (8.8.7.1, 15.9, 15.12)

    ~~12.~~ 13. In an unqualified class instance creation expression, either as the class type to be instantiated (15.9) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5)

    ~~13.~~ 14. The element type in an array creation expression (15.10.1)

    ~~14.~~ 15. The type in the cast operator of a cast expression (15.16)

    ~~15.~~ 16. The type that follows the `instanceof` relational operator (15.20.2)

    ~~16.~~ 17. In a method reference expression (15.13), as the reference type to search for a

member method or as the class type or array type to construct.

The extraction of a TypeName from the identifiers of a ReferenceType in the 16 contexts above is intended to apply recursively to all sub-terms of the ReferenceType, such as its element type and any type arguments.

For example, suppose a field declaration uses the type `p.q.Foo[]`. The brackets of the array type are ignored, and the term `p.q.Foo` is extracted as a dotted sequence of Identifiers to the left of the brackets in an array type, and classified as a TypeName. A later step determines which of `p`, `q`, and `Foo` is a type name or a package name.

As another example, suppose a cast operator uses the type `p.q.Foo<? extends String>`. The term `p.q.Foo` is again extracted as a dotted sequence of Identifier terms, this time to the left of the `<` in a parameterized type, and classified as a TypeName. The term `String` is extracted as an Identifier in an `extends` clause of a wildcard type argument of a parameterized type, and classified as a TypeName.

A name is syntactically classified as an *ExpressionName* in these contexts:

- As the qualifying expression in a qualified superclass constructor invocation (8.8.7.1↗)
- As the qualifying expression in a qualified class instance creation expression (15.9↗)
- As the array reference expression in an array access expression (15.10.3↗)
- As a *PostfixExpression* (15.14↗)
- As the left-hand operand of an assignment operator (15.26↗)
- As a *VariableAccess* in a `try`-with-resources statement (14.20.3↗)

A name is syntactically classified as a *MethodName* in this context:

- Before the "`(`" in a method invocation expression (15.12↗)

A name is syntactically classified as a *PackageOrTypeName* in these contexts:

- To the left of the "`.`" in a qualified *TypeName*
- In a type-import-on-demand declaration (7.5.2↗)

A name is syntactically classified as an *AmbiguousName* in these contexts:

- To the left of the "`.`" in a qualified *ExpressionName*
- To the left of the rightmost `.` that occurs before the "`(`" in a method invocation expression
- To the left of the "`.`" in a qualified *AmbiguousName*
- In the default value clause of an annotation type element declaration (9.6.2↗)
- To the right of an "`=`" in an an element-value pair (9.7.1↗)
- To the left of `::` in a method reference expression (15.13↗)

The effect of syntactic classification is to restrict certain kinds of entities to certain parts of expressions:

- *The name of a field, parameter, or local variable may be used as an expression (15.14.1↗).*
- *The name of a method may appear in an expression only as part of a method invocation expression (15.12↗).*
- *The name of a class or interface type may appear in an expression only as part of a class literal*

(15.8.2), a qualified `this` expression (15.8.4), a class instance creation expression (15.9), an array creation expression (15.10.1), a cast expression (15.16), an `instanceof` expression (15.20.2), an enum constant (8.9), or as part of a qualified name for a field or method.

- *The name of a package may appear in an expression only as part of a qualified name for a class or interface type.*

Chapter 7: Packages and Modules

7.6 Top Level Class and Interface Declarations

A *top level class or interface declaration* declares a top level class (8.1), which may be an enum class (8.9) or a record class (8.10), or a top level interface (9.1), which may be an annotation interface (9.6).

TopLevelClassOrInterfaceDeclaration:
ClassOrInterfaceDeclaration
;

ClassOrInterfaceDeclaration:
ClassDeclaration
EnumDeclaration
RecordDeclaration
InterfaceDeclaration
AnnotationDeclaration

Extra ";" tokens appearing at the level of class or interface declarations in a compilation unit have no effect on the meaning of the compilation unit. Stray semicolons are permitted in the Java programming language solely as a concession to C++ programmers who are used to placing ";" after a class declaration. They should not be used in new Java code.

In the absence of an access modifier, a top level class or interface has package access: it is accessible only within ordinary compilation units of the package in which it is declared (6.6.1). A class or interface may be declared `public` to grant access to the class or interface from code in other packages of the same module, and potentially from code in packages of other modules.

It is a compile-time error if a top level class or interface declaration contains any one of the following access modifiers: `protected`, `private`, or `static`.

It is a compile-time error if the name of a top level class or interface appears as the name of any other top level class or interface declared in the same package.

The scope and shadowing of a top level class or interface is specified in 6.3 and 6.4.

The fully qualified name of a top level class or interface is specified in 6.7.

Example 7.6-1. Conflicting Top Level Class or Interface Declarations

```
package test;
import java.util.Vector;
class Point {
    int x, y;
}
interface Point {  // compile-time error #1
    int getR();
    int getTheta();
}
```

```
class Vector { Point[] pts; }  // compile-time error #2
```

Here, the first compile-time error is caused by the duplicate declaration of the name `Point` as both a class and an interface in the same package. A second compile-time error is the attempt to declare the name `Vector` both by a class declaration and by a single-type-import declaration.

Note, however, that it is not an error for the name of a class to also name a class or interface that otherwise might be imported by a type-import-on-demand declaration (7.5.2⇗) in the compilation unit (7.3⇗) containing the class declaration. Thus, in this program:

```
package test;
import java.util.*;
class Vector {}  // not a compile-time error
```

the declaration of the class `Vector` is permitted even though there is also a class `java.util.Vector`. Within this compilation unit, the simple name `Vector` refers to the class `test.Vector`, not to `java.util.Vector` (which can still be referred to by code within the compilation unit, but only by its fully qualified name).

Example 7.6-2. Scope of Top Level Classes and Interfaces

```
package points;
class Point {
    int x, y;           // coordinates
    PointColor color;   // color of this point
    Point next;         // next point with this color
    static int nPoints;
}
class PointColor {
    Point first;        // first point with this color
    PointColor(int color) { this.color = color; }
    private int color;  // color components
}
```

This program defines two classes that use each other in the declarations of their class members. Because the classes `Point` and `PointColor` have all the class declarations in package `points`, including all those in the current compilation unit, as their scope, this program compiles correctly. That is, forward reference is not a problem.

Example 7.6-3. Fully Qualified Names

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

In this code, the class `Point` is declared in a compilation unit with no `package` declaration, and thus `Point` is its fully qualified name, whereas in the code:

```
package vista;
class Point { int x, y; }
```

the fully qualified name of the class `Point` is `vista.Point`. (The package name `vista` is suitable for local or personal use; if the package were intended to be widely distributed, it would be better to give it a unique package name (6.1).)

An implementation of the Java SE Platform must keep track of classes and interfaces within packages by the combination of their enclosing module names and their binary names (13.1). Multiple ways of naming a class or interface must be expanded to binary names to make sure that such names are understood as referring to the same class or interface.

For example, if a compilation unit contains the single-type-import declaration (7.5.1⇗):

`import java.util.Vector;`

then within that compilation unit, the simple name `Vector` and the fully qualified name `java.util.Vector` refer to the same class.

If and only if packages are stored in a file system (7.2⇗), the host system may choose to enforce the restriction that it is a compile-time error if a class or interface is not found in a file under a name composed of the class or interface name plus an extension (such as `.java` or `.jav`) if either of the following is true:

- The class or interface is referred to by code in other ordinary compilation units of the package in which the class or interface is declared.
- The class or interface is declared `public` (and therefore is potentially accessible from code in other packages).

This restriction implies that there must be at most one such class or interface per compilation unit. This restriction makes it easy for a Java compiler to find a named class or interface within a package. In practice, many programmers choose to put each class or interface in its own compilation unit, whether or not it is `public` or is referred to by code in other compilation units.

For example, the source code for a `public` type `wet.sprocket.Toad` would be found in a file `Toad.java` in the directory `wet/sprocket`, and the corresponding object code would be found in the file `Toad.class` in the same directory.

Chapter 8: Classes

Class declarations define new reference types and describe how they are implemented (8.1⇗).

A *top level class* (7.6) is a class that is declared at the top level of a compilation unit.

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

An *inner class* (8.1.3⇗) is a nested class that can refer to enclosing class instances, local variables, and type variables.

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

A *record class* (8.10) is a class declared with special 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 named 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. A class may be declared `final` (8.1.1.2⇗), in which case it cannot have subclasses. If a class is declared `public`, then it can be referred to from code in any package of its module and potentially from code in other modules. 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⇗). Classes may be *generic* (8.1.2⇗), that is, they may declare type variables whose bindings may differ among different instances of the class.

Classes 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 (8.1.6↗). 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 (6.6↗) `public`, `protected`, or `private`. 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 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` or they may be inner classes (8.1.3↗).

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

8.1.1 Class Modifiers

8.1.1.4 `static` Classes

The `static` keyword indicates that a nested class is not an inner class (8.1.3↗). The class has no immediately enclosing instance and cannot directly reference enclosing type variables (6.5.5.1↗); enclosing instance variables, local variables, formal parameters, or exception parameters (6.5.6.1↗); or enclosing instance methods (15.12.3↗).

A local class declaration may not use the `static` keyword (14.3).

Nested enum classes and record classes are implicitly declared `static`. A member enum class or record class may redundantly specify the `static` modifier; a local enum class or record class may not (8.9↗).

8.1.4 Superclasses and Subclasses

The optional `extends` clause in a normal class declaration specifies the *direct superclass* of the current class.

Superclass:
`extends` *ClassType*

The `extends` clause must not appear in the definition of the class `Object`, or a compile-time error occurs, because it is the primordial class and has no direct superclass.

The *ClassType* must name an accessible class type (6.6⇗), or a compile-time error occurs.

It is a compile-time error if the *ClassType* names a class that is `final`, because `final` classes are not allowed to have subclasses (8.1.1.2⇗).

It is a compile-time error if the *ClassType* names a type of the class `Enum` ~~or any invocation of `Enum` (8.9⇗)~~ or the class type `Record`.

If the *ClassType* has type arguments, it must denote a well-formed parameterized type (4.5⇗), and none of the type arguments may be wildcard type arguments, or a compile-time error occurs.

Given a (possibly generic) class declaration $C<F_1,...,F_n>$ ($n \geq 0$, $C \neq$ `Object`), the *direct superclass* of the class type $C<F_1,...,F_n>$ is the type given in the `extends` clause of the declaration of *C* if an `extends` clause is present, or `Object` otherwise.

Given a generic class declaration $C<F_1,...,F_n>$ ($n > 0$), the *direct superclass* of the parameterized class type $C<T_1,...,T_n>$, where T_i ($1 \leq i \leq n$) is a type, is $D<U_1\ \theta,...,U_k\ \theta>$, where $D<U_1,...,U_k>$ is the direct superclass of $C<F_1,...,F_n>$ and θ is the substitution `[F1:=T1,...,Fn:=Tn]`.

A class is said to be a *direct subclass* of its direct superclass. The direct superclass is the class from whose implementation the implementation of the current class is derived.

The *subclass* relationship is the transitive closure of the direct subclass relationship. A class *A* is a subclass of class *C* if either of the following is true:

- *A* is the direct subclass of *C*
- There exists a class *B* such that *A* is a subclass of *B*, and *B* is a subclass of *C*, applying this definition recursively.

Class *C* is said to be a *superclass* of class *A* whenever *A* is a subclass of *C*.

Example 8.1.4-1. Direct Superclasses and Subclasses

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; }  // error
```

Here, the relationships are as follows:

- *The class `Point` is a direct subclass of `Object`.*
- *The class `Object` is the direct superclass of the class `Point`.*
- *The class `ColoredPoint` is a direct subclass of class `Point`.*
- *The class `Point` is the direct superclass of class `ColoredPoint`.*

The declaration of class `Colored3dPoint` causes a compile-time error because it attempts to extend the final class `ColoredPoint`.

Example 8.1.4-2. Superclasses and Subclasses

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
final class Colored3dPoint extends ColoredPoint { int z; }
```

Here, the relationships are as follows:

- *The class `Point` is a superclass of class `ColoredPoint`.*
- *The class `Point` is a superclass of class `Colored3dPoint`.*
- *The class `ColoredPoint` is a subclass of class `Point`.*
- *The class `ColoredPoint` is a superclass of class `Colored3dPoint`.*
- *The class `Colored3dPoint` is a subclass of class `ColoredPoint`.*
- *The class `Colored3dPoint` is a subclass of class `Point`.*

A class *C directly depends* on a type *T* if *T* is mentioned in the `extends` or `implements` clause of *C* either as a superclass or superinterface, or as a qualifier in the fully qualified form of a superclass or superinterface name.

A class *C depends* on a reference type *T* if any of the following is true:

- *C* directly depends on *T*.
- *C* directly depends on an interface *I* that depends (9.1.3↗) on *T*.
- *C* directly depends on a class *D* that depends on *T* (using this definition recursively).

It is a compile-time error if a class depends on itself.

If circularly declared classes are detected at run time, as classes are loaded, then a `ClassCircularityError` is thrown (12.2.1↗).

Example 8.1.4-3. Class Depends on Itself

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

This program causes a compile-time error because class `Point` depends on itself.

8.5 Member Class and Interface Declarations

A *member class* is a class whose declaration is directly enclosed in the body of another class or interface declaration (8.1.6↗, 9.1.4↗). A member class may be an enum class (8.9↗) or a record class (8.10).

A *member interface* is an interface whose declaration is directly enclosed in the body of another class or interface declaration (8.1.6↗, 9.1.4↗). A member interface may be an annotation interface (9.6↗).

The accessibility of a member class or interface declaration in a class is specified by its access modifier, or by 6.6↗ if lacking an access modifier.

If a class declares a member class or interface with a certain name, then the declaration of that class or interface is said to *hide* any and all accessible declarations of member classes and interfaces with the same name in superclasses and superinterfaces of the class.

In this respect, hiding of member classes and interfaces is similar to hiding of fields (8.3↗).

A class inherits from its direct superclass and direct superinterfaces all the non-`private` member classes and interfaces of the superclass and superinterfaces that are both accessible to code in the class and not hidden by a declaration in the class.

It is possible for a class to inherit more than one member class or interface with the same name, either from its superclass and superinterfaces or from its superinterfaces alone. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the class to refer to any such member class or interface by its simple name will result in a compile-time error, because the reference is ambiguous.

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

8.8 Constructor Declarations

A *constructor* is used in the creation of an object that is an instance of a class (12.5⇗, 15.9⇗).

ConstructorDeclaration:
{*ConstructorModifier*} *ConstructorDeclarator* [*Throws*] *ConstructorBody*

ConstructorDeclarator:
[*TypeParameters*] *SimpleTypeName*
([*ReceiverParameter* ,] [*FormalParameterList*])

SimpleTypeName:
TypeIdentifier

The rules in this section apply to constructors in all class declarations, including enum declarations and record declarations. However, special rules apply to enum declarations with regard to constructor modifiers, constructor bodies, and default constructors; these rules are stated in 8.9.2⇗. Special rules also apply to record declarations with regard to constructors, including a special compact declaration form; the details are given in 8.10.4.

The *SimpleTypeName* in the *ConstructorDeclarator* must be the simple name of the class that contains the constructor declaration, or a compile-time error occurs.

In all other respects, a constructor declaration looks just like a method declaration that has no result (8.4.5⇗).

Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.

Constructors are invoked by class instance creation expressions (15.9⇗), by the conversions and concatenations caused by the string concatenation operator + (15.18.1⇗), and by explicit constructor invocations from other constructors (8.8.7⇗). Access to constructors is governed by access modifiers (6.6⇗), so it is possible to prevent class instantiation by declaring an inaccessible constructor (8.8.10⇗).

Constructors are never invoked by method invocation expressions (15.12⇗).

Example 8.8-1. Constructor Declarations

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

8.10 Record Declarations

A *record declaration* specifies a new *record class*, a special kind of class that defines a simple aggregate of values.

RecordDeclaration:
{*ClassModifier*} `record` *TypeIdentifier* [*TypeParameters*] *RecordHeader* [*SuperInterfaces*] *RecordBody*

The *TypeIdentifier* in the record declaration specifies the name of the record class.

A record declaration may specify a top level record class (7.6), a member record class (8.5, 9.5), or a local record class (14.3).

It is a compile-time error if a record declaration has the modifier `abstract`.

A record declaration is implicitly `final`. It is permitted for the declaration of a record class to redundantly specify the `final` modifier.

> *JEP 360 proposes extending Java to support sealed classes. In this case, the following text would be added:*
>
> *It is a compile-time error if a record declaration has the modifier `sealed`.*

A nested record declaration is implicitly `static`. It is permitted for the declaration of a member record class to redundantly specify the `static` modifier. A local record declaration may not redundantly specify the `static` modifier (14.3).

It is a compile-time error if the same keyword appears more than once as a modifier for a record declaration, or if a record declaration has more than one of the access modifiers `public`, `protected`, and `private` (6.6).

The direct superclass type of a record class is `Record` (8.1.4).

A record declaration has no `extends` clause, so it is not possible to explicitly declare that the direct superclass type of a record class is `Record`.

The serialization mechanism treats instances of a record class differently than ordinary serializable or externalizable objects. In particular, a record object is deserialized using the canonical constructor (8.10.4).

8.10.1 Record Components

The header of a record declaration consists of a *record component list*, which is a possibly empty list of *record components*. Each record component consists of a type (optionally preceded by one or more annotations) and an identifier that specifies the name of the record component. If a record class has no record components, then the record header consists of an empty pair of parentheses.

Each record component corresponds to an implicitly declared field and an accessor method (declared either explicitly or implicitly) of the record class (8.10.3).

RecordHeader:
`(` [*RecordComponentList*] `)`

RecordComponentList:
RecordComponent { `,` *RecordComponent* }

RecordComponent:
{ *Annotation* } *UnannType Identifier*
VariableArityRecordComponent

VariableArityRecordComponent:
{ *Annotation* } *UnannType* { *Annotation* } ... *Identifier*

A record component may be a variable arity record component, indicated by an ellipsis following the type. At most one variable arity record component is permitted for a record type. It is a compile-time error if a variable arity record component appears anywhere in the list of record components except the last position.

It is a compile-time error for a record declaration to declare two record components with the same name.

It is a compile-time error for a record declaration to declare a record component with the name `clone`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, or `wait` (8.10.3).

The rules for annotation modifiers on a record component are specified in 9.7.4. Annotations on a record component of a record class may be propagated to members and constructors of the record class as specified in 8.10.3. An annotation on a record component only remains on the record component if its annotation type is applicable in the record component context (9.6.4.1).

The declared type of a record component depends on whether it is a variable arity record component:

- If the record component is not a variable arity record component, then the declared type is denoted by *UnannType*.
- If the record component is a variable arity record component, then the declared type is an array type specified by 10.2.

If the declared type of a variable arity record component has a non-reifiable element type (4.7), then a compile-time unchecked warning occurs for the declaration of the variable arity record component, unless the canonical constructor (8.10.4) is annotated with `@SafeVarargs` (9.6.4.7) or the warning is suppressed by `@SuppressWarnings` (9.6.4.5).

8.10.2 Record Bodies

The body of a record declaration may contain constructor and member declarations as well as static initializers.

RecordBody:
`{` {*RecordBodyDeclaration*} `}`

RecordBodyDeclaration:
ClassBodyDeclaration
CompactConstructorDeclaration

The following productions from 8.1.6 are shown here for convenience:

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:

FieldDeclaration
MethodDeclaration
ClassOrInterfaceDeclaration
.

It is a compile-time error for the body of a record declaration to contain a non-`static` field declaration (8.3.1.1).

It is a compile-time error for the body of a record declaration to contain a `native` method declaration (8.4.3.4).

It is a compile-time error for the body of a record declaration to contain an instance initializer (8.6).

8.10.3 Record Members

A record class has for each record component appearing in the record component list an implicitly declared field with the same name as the record component and the same type as the declared type of the record component. This field is declared `private` and `final`. The field is annotated with the annotations, if any, that appear on the corresponding record component and whose annotation types are applicable in the field declaration context, or in type contexts, or both.

In a record declaration, an *accessor method for a record component* is a method whose name is the same as the name of the given record component, and whose formal parameter list is empty.

If an accessor method for a record component is declared explicitly, then it must satisfy the following:

- The return type of the accessor method must be the same as the declared type of the corresponding record component.
- The accessor method must not be generic (8.4.4).
- The accessor method must be declared `public`.
- The accessor method must not be declared `static`.
- The accessor method must not have a `throws` clause.
- All other rules for a method in a normal class declaration must be satisfied (8.4).

Otherwise, a compile-time error occurs.

If an accessor method for a record component is declared explicitly, and the corresponding record component is annotated with an annotation that is only applicable in the method declaration context, then a compile-time error occurs.

> *This captures the contradictory case where a programmer both declares that an annotation can only propagate to the accessor method, and also ensures that the annotation will not be propagated (as the accessor method has been declared explicitly).*

If an accessor method for a record component is not explicitly declared, then one is implicitly declared with the following properties:

- The name is the same as the name of the record component.
- The return type is the same as the declared type of the record component.
- It is not generic.
- It has an empty formal parameter list.

- It is declared `public`.
- It is annotated with the annotations, if any, that appear on the corresponding record component and whose annotation types are applicable in the method declaration context, or in type contexts, or both. The rules for these annotation modifiers, if any, on the accessor method are the same as for a method declaration, and are specified in 9.7.4 and 9.7.5⇗.
- The body of the implicitly declared accessor method simply returns the value of the field corresponding to the record component.

An implicitly declared accessor method must satisfy all the rules for a method in a normal class declaration (8.4⇗).

> *A record class will thus have accessor methods for all record components appearing in the record component list.*
>
> *Annotations that appear on a record component are not propagated to an explicitly declared accessor method for that record component. This is in contrast to an implicitly declared accessor method which is annotated with the applicable annotations from the corresponding record component.*
>
> *Annotations that are propagated to an implicitly declared accessor method must result in a correctly annotated method. For example, in the following program, the implicitly declared accessor method would be annotated with the `@SafeVarargs` annotation, but this method is not correctly annotated as it is a fixed arity method (9.6.4.7⇗).*
>
> ```
> record BadRecord(@SafeVarargs int x) {}  // Error!
> ```

An accessor method (explicitly or implicitly declared) may override or overload methods declared in superinterfaces of the record class.

> *The restrictions on the record component names (8.10.1) mean that no implicitly declared accessor method will have a signature that is override-equivalent with a non-`private` method of the class `Object`.*

All record classes provide an implementation of the `abstract` methods declared in the class `Record`. For each of the following methods, if a record class does not explicitly declare a method with the same signature (8.4.2⇗), then the method is declared implicitly:

- A method `public final boolean equals(Object obj)` that returns `true` if and only if the argument is an instance of the same record class, and each record component of this record is equal to the corresponding record component of the argument, according to the `equals` method of class `Object` for record components whose types are reference types, and `==` for record components whose types are primitive types.
- A method `public final int hashCode()` that returns a hash code value derived by combining the hash code value for all the record components, according to the `hashCode` method of class `Object` for record components whose types are reference types, or the `hashCode` method of the wrapper class (5.1.7⇗) for record components whose types are primitive types.
- A method `public final String toString()` that returns a string that is derived from the name of the record class and the names and string representations of the record components, according to `toString` method of class `Object` for components whose types are reference types, and the `toString` method of the wrapper class for record components whose types are primitive types.

Record classes may declare or inherit other members, as described in 8.2⇗. All members of record

classes, including the implicitly declared members, are subject to the usual rules for member declarations in a class (8.3, 8.4, 8.5).

For example, a record class can inherit default methods from its direct superinterfaces. Given the declarations:

```
interface Logging{
    default void logAction() { ... }
}

record Point(int i, int j) implements Logging {}
```

Then the following code works as expected:

```
Point p = new Point(42,37);
p.logAction();
```

8.10.4 Record Constructor Declarations

To support 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 either explicitly or implicitly, that initializes all of the fields corresponding to the record components.

A record class *R* has a *derived constructor signature* that consists of the name *R*, no type parameters, and the formal parameter types derived from the record component list of *R* by taking the declared type of each record component.

A record class *R* has a *derived formal parameter list* that is constructed by taking each record component in the record component list and deriving a formal parameter with the same name and the declared type of the record component.

There are two ways to explicitly declare a canonical constructor in a record declaration: either by declaring a constructor with a particular signature, or by declaring a compact constructor.

A constructor in a declaration of a record class *R* is said to be a canonical constructor if its signature is override-equivalent (8.4.2) to the derived constructor signature of *R*.

As a canonical constructor has a signature that is override-equivalent to the derived constructor signature of a record class, there can only be one explicitly declared canonical constructor.

The access modifier of a canonical constructor for a record class 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.

A constructor declaration that is not a compact constructor but is a canonical constructor must satisfy the following conditions:

- Each formal parameter in the formal parameter list of the constructor must have the same name and type as the corresponding record component. The formal parameter must be a

variable arity parameter if and only if the corresponding record component is a variable arity record component.

- The constructor must not be generic (8.8.4⇗).
- The constructor must not have a `throws` clause.
- The body of the constructor must not contain an explicit constructor invocation statement (8.8.7.1⇗).
- All the rules for constructor declarations in a normal class declaration must be satisfied (8.8).

Otherwise, a compile-time error occurs.

If a canonical constructor that is not a compact constructor is declared explicitly, and one of record components is annotated with an annotation that is only applicable in the parameter context, then a compile-time error occurs.

A consequence of these rules is that that the annotations on a record component can differ from the annotations on the corresponding formal parameter of an explicitly declared canonical constructor. For example, the following is valid:

```
@Target(ElementType.TYPE_USE)
@interface DevAnnotation{ String value(); }

record R(@DevAnnotation("devA") String s) {
    R(@DevAnnotation("devB") String s) {
        // Explicitly declared canonical constructor
        ...
    }
}
```

However, if the annotation `@DevAnnotation` was only applicable in the formal parameter context, then a compile-time error would occur. This is because the annotation with value `"devA"` would not be propagated anywhere (as the canonical constructor is explicitly declared) and would be lost.

```
@Target(ElementType.PARAMETER)
@interface DevAnnotation{ String value(); }

record R(@DevAnnotation("devA") String s) {     // Error!
    R(@DevAnnotation("devB") String s) {
        // Explicitly declared canonical constructor
        ...
    }
}
```

The second way to explicitly declare a canonical constructor in a record declaration is to provide a *compact constructor declaration*, which is a special, succinct form of constructor declaration only available in a record declaration.

CompactConstructorDeclaration:
{ *ConstructorModifier* } *SimpleTypeName ConstructorBody*

In a record class *R*, the formal parameter list for a compact constructor declaration is implicitly declared and given by the derived formal parameter list of *R*.

Thus, given a record declaration with a record component named c and a compact constructor declaration, in the body of the compact constructor an occurrence of an unqualified name c denotes

the implicit formal parameter c.

In a record class R, the signature of a compact constructor declaration is the derived constructor signature of R.

It is a compile-time error to declare more than one compact constructor in a record class.

The rules concerning signatures of constructors in class declarations (8.8.2) mean that it is also a compile-time error if a record declaration contains a compact constructor declaration and a standard constructor declaration that is a canonical constructor. A compact constructor is a canonical constructor.

A compact constructor declaration must satisfy all of the following conditions; otherwise a compile-time error occurs.

- The body of a compact constructor must not contain a `return` statement (14.17).
- The body of a compact constructor must not contain an explicit constructor invocation statement (8.8.7.1).
- All the other rules for a constructor in a normal class declaration must be satisfied (8.8), except the requirement that the fields corresponding to the record components of the record class must be definitely assigned and moreover not definitely unassigned at the end of the compact constructor (8.3.1.2).

It is a compile-time error if an assignment occurs (16) to a field corresponding to a record component of the record class in the body of the compact constructor.

All fields corresponding to the record components of the record class are implicitly initialized to the value of the corresponding formal parameter after the body of the compact constructor. These fields are implicitly initialized in the order that they are declared in the record component list.

The intention of a compact constructor declaration is that only validation and/or normalization code need be given in the constructor body; the remaining initialization code is supplied by the compiler. Here is a simple example:

```
record Rational(int num, int denom) {
    Rational {
        int gcd = gcd(num, denom);
        num /= gcd;
        denom /= gcd;
    }
}
```

This declaration is equivalent to the following declaration:

```
record Rational(int num, int denom) {
    Rational(int num, int demon) {
        int gcd = gcd(num, denom);
        num /= gcd;
        denom /= gcd;
        this.num = num;
        this.denom = denom;
    }
}
```

In the declaration of a record class *R*, if a canonical constructor is not explicitly declared, then one is implicitly declared with the following properties:

- The signature of the implicitly declared canonical constructor has no type parameters, and

has a formal parameter list derived as follows from each record component in the record component list:

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

In both cases, the derived formal parameter is annotated with the annotations, if any, that appear on the corresponding record component whose annotation interfaces are applicable in the formal parameter context, or in type contexts, or both.

- The implicitly declared canonical constructor has the same access modifier as the record class *R*, unless the record class lacks an access modifier, in which case the canonical constructor has package access (6.6⇗).
- The implicitly declared canonical constructor does not have a `throws` clause.
- The body of the implicitly declared canonical constructor initializes each field corresponding to a record component with the corresponding formal parameter in the order that they appear in the record component list.

An implicitly declared canonical constructor must satisfy all the rules for constructor declarations in a normal class declaration (8.8).

If in a declaration of a record class *R* there all other declarations of constructors that are not canonical constructors (if any) must satisfy the following:

- The constructor body must start with an alternate constructor invocation (8.8.7.1⇗).

Otherwise, a compile-time error occurs.

Chapter 9: Interfaces

9.6 Annotation Interfaces

9.6.4 Predefined Annotation Interfaces

9.6.4.1 `@Target`

An annotation of type `java.lang.annotation.Target` is used on the declaration of an annotation interface *T* to specify the contexts in which *T* is *applicable*. `java.lang.annotation.Target` has a single element, `value`, of type `java.lang.annotation.ElementType[]`, to specify contexts.

Annotation interfaces may be applicable in *declaration contexts*, where annotations apply to declarations, or in *type contexts*, where annotations apply to types used in declarations and expressions.

There are ~~nine~~ ten declaration contexts, each corresponding to an enum constant of `java.lang.annotation.ElementType`:

1. Module declarations (7.7⇗)

   Corresponds to `java.lang.annotation.ElementType.MODULE`

2. Package declarations (7.4.1⇗)

   Corresponds to `java.lang.annotation.ElementType.PACKAGE`

3. Type declarations: class, interface, enum, record, and annotation declarations (8.1.1⇗, 9.1.1⇗, 8.5, 9.5⇗, 8.9⇗, 8.10, 9.6⇗)

   Corresponds to `java.lang.annotation.ElementType.TYPE`

   Additionally, annotation declarations correspond to `java.lang.annotation.ElementType.ANNOTATION_TYPE`

4. Method declarations (including elements of annotation interfaces) (8.4.3⇗, 9.4⇗, 9.6.1⇗)

   Corresponds to `java.lang.annotation.ElementType.METHOD`

5. Constructor declarations (8.8.3⇗)

   Corresponds to `java.lang.annotation.ElementType.CONSTRUCTOR`

6. Type parameter declarations of generic classes, interfaces, methods, and constructors (8.1.2⇗, 9.1.2⇗, 8.4.4⇗, 8.8.4⇗)

   Corresponds to `java.lang.annotation.ElementType.TYPE_PARAMETER`

7. Field declarations (including enum constants) (8.3.1⇗, 9.3⇗, 8.9.1⇗)

   Corresponds to `java.lang.annotation.ElementType.FIELD`

8. Formal and exception parameter declarations (8.4.1⇗, 9.4⇗, 14.20⇗)

   Corresponds to `java.lang.annotation.ElementType.PARAMETER`

9. Local variable declarations (including loop variables of `for` statements and resource variables of `try`-with-resources statements) (14.4⇗, 14.14.1⇗, 14.14.2⇗, 14.20.3⇗)

   Corresponds to `java.lang.annotation.ElementType.LOCAL_VARIABLE`

10. Record component declarations (8.10.1)

    Corresponds to `java.lang.annotation.ElementType.RECORD_COMPONENT`

There are ~~16~~ 17 type contexts (4.11), all represented by the enum constant `TYPE_USE` of `java.lang.annotation.ElementType`.

It is a compile-time error if the same enum constant appears more than once in the `value` element of an annotation of type `java.lang.annotation.Target`.

If an annotation of type `java.lang.annotation.Target` is not present on the declaration of an annotation interface *T*, then *T* is applicable in all ~~nine~~ ten declaration contexts and in all ~~16~~ 17 type contexts.

9.6.4.4 `@Override`

Programmers occasionally overload a method declaration when they mean to override it, leading to subtle problems. The annotation interface `Override` supports early detection of such problems.

The classic example concerns the `equals` method. Programmers write the following in class `Foo`:

```
public boolean equals(Foo that) { ... }
```

when they mean to write:

```
public boolean equals(Object that) { ... }
```

This is perfectly legal, but class `Foo` inherits the `equals` implementation from `Object`, which can cause some subtle bugs.

~~If a method declaration in class or interface *T* is annotated with `@Override`, but the method does not override from *T* a method declared in a supertype of *T* (8.4.8.1, 9.4.1.1), or is not override-equivalent to a `public` method of `Object` (4.3.2, 8.4.2), then a compile-time error occurs.~~

It is a compile-time error for a method declaration in a class or interface *T* to be annotated with `@Override` unless one of the following conditions holds:

- the method overrides from *T* a method declared in a supertype of *T* (8.4.8.1, 9.4.1.1)
- the method is override-equivalent to a `public` method of `Object` (4.3.2, 8.4.2)
- *T* is a record class (8.10), and the method is an accessor method for a record component of *T* (8.10.3).

This behavior differs from Java SE 5.0, where `@Override` only caused a compile-time error if applied to a method that implemented a method from a superinterface that was not also present in a superclass.

The clause about overriding a `public` method is motivated by use of `@Override` in an interface. Consider the following declarations:

```
class Foo     { @Override public int hashCode() {..} }
interface Bar { @Override int hashCode(); }
```

The use of `@Override` in the class declaration is legal by the first clause, because `Foo.hashCode` overrides from `Foo` the method `Object.hashCode`.

For the interface declaration, consider that while an interface does not have `Object` as a supertype, an interface does have `public abstract` members that correspond to the `public` members of `Object` (9.2). If an interface chooses to declare them explicitly (that is, to declare members that are override-equivalent to `public` methods of `Object`), then the interface is deemed to override them, and use of `@Override` is allowed.

However, consider an interface that attempts to use `@Override` on a `clone` method: (`finalize` could also be used in this example)

```
interface Quux { @Override Object clone(); }
```

Because `Object.clone` is not `public`, there is no member called `clone` implicitly declared in `Quux`. Therefore, the explicit declaration of `clone` in `Quux` is not deemed to "implement" any other method, and it is erroneous to use `@Override`. (The fact that `Quux.clone` is `public` is not relevant.)

In contrast, a class declaration that declares `clone` is simply overriding `Object.clone`, so is able to use `@Override`:

```
class Beep { @Override protected Object clone() {..} }
```

The `@Override` annotation has a special meaning in a record declaration, where it can be used to specify that a method declaration is an accessor method (8.10.3) for a record component. Consider the following declaration:

```
record R(int x) {
    @Override
    public int x() {
        return Math.abs(x);
    }
    ...
}
```

The `@Override` annotation on the accessor method `x` ensures that if the record component `x` is modified or removed, then the corresponding accessor method must be modified or removed too.

9.7 Annotations

9.7.4 Where Annotations May Appear

A *declaration annotation* is an annotation that applies to a declaration, and whose own type is applicable in the declaration context (9.6.4.1) represented by that declaration; or an annotation that applies to a class, interface, enum, record, annotation type, or type parameter declaration, and whose own type is applicable in type contexts (4.11).

A *type annotation* is an annotation that applies to a type (or any part of a type), and whose own type is applicable in type contexts.

For example, given the field declaration:

```
@Foo int f;
```

`@Foo` is a declaration annotation on `f` if `Foo` is meta-annotated by `@Target(ElementType.FIELD)`, and a type annotation on `int` if `Foo` is meta-annotated by `@Target(ElementType.TYPE_USE)`. It is possible for `@Foo` to be both a declaration annotation and a type annotation simultaneously.

Type annotations can apply to an array type or any component type thereof (10.1). For example, assuming that `A`, `B`, and `C` are annotation types meta-annotated with `@Target(ElementType.TYPE_USE)`, then given the field declaration:

```
@C int @A [] @B [] f;
```

`@A` applies to the array type `int[][]`, `@B` applies to its component type `int[]`, and `@C` applies to the element type `int`. For more examples, see 10.2.

An important property of this syntax is that, in two declarations that differ only in the number of array levels, the annotations to the left of the type refer to the same type. For example, `@C` applies to the type `int` in all of the following declarations:

```
@C int f;
@C int[] f;
@C int[][] f;
```

It is customary, though not required, to write declaration annotations before all other modifiers, and type annotations immediately before the type to which they apply.

It is possible for an annotation to appear at a syntactic location in a program where it could plausibly apply to a declaration, or a type, or both. This can happen in any of the ~~five~~ six declaration contexts where modifiers immediately precede the type of the declared entity:

- Method declarations (including elements of annotation types)
- Constructor declarations
- Field declarations (including enum constants)
- Formal and exception parameter declarations
- Local variable declarations (including loop variables of `for` statements and resource variables of `try`-with-resources statements)
- Record component declarations

The grammar of the Java programming language unambiguously treats annotations at these

locations as modifiers for a declaration (8.3⇗), but that is purely a syntactic matter. Whether an annotation applies to the declaration or to the type of the declared entity - and thus, whether the annotation is a *declaration annotation* or a *type annotation* - depends on the applicability of the annotation's type:

- If the annotation's type is applicable in the declaration context corresponding to the declaration, and not in type contexts, then the annotation is deemed to apply only to the declaration.
- If the annotation's type is applicable in type contexts, and not in the declaration context corresponding to the declaration, then the annotation is deemed to apply only to the type which is closest to the annotation.
- If the annotation's type is applicable in the declaration context corresponding to the declaration *and* in type contexts, then the annotation is deemed to apply to both the declaration *and* the type which is closest to the annotation.

In the second and third cases above, the type which is *closest* to the annotation is determined as follows:

- If the annotation appears before a `void` method declaration or a local variable declaration that uses `var` (14.4⇗, 14.14.2⇗, 14.20.3⇗), then there is no closest type. If the annotation's type is deemed to apply only to the type which is closest to the annotation, a compile-time error occurs.
- If the annotation appears before a constructor declaration, then the closest type is the type of the newly constructed object. The type of the newly constructed object is the fully qualified name of the type immediately enclosing the constructor declaration. Within that fully qualified name, the annotation applies to the simple type name indicated by the constructor declaration.
- In all other cases, the closest type is the type written in source code for the declared entity; if that type is an array type, then the element type is deemed to be closest to the annotation.

  For example, in the field declaration `@Foo public static String f;`, the type which is closest to `@Foo` is `String`. (If the type of the field declaration had been written as `java.lang.String`, then `java.lang.String` would be the type closest to `@Foo`, and later rules would prohibit a type annotation from applying to the package name `java`.) In the generic method declaration `@Foo <T> int[] m() {...}`, the type written for the declared entity is `int[]`, so `@Foo` applies to the element type `int`.

  Local variable declarations which do not use `var` are similar to formal parameter declarations of lambda expressions, in that both allow declaration annotations and type annotations in source code, but only the type annotations can be stored in the `class` file.

It is a compile-time error if an annotation of type *T* is syntactically a modifier for:

- a module declaration, but *T* is not applicable to module declarations.
- a package declaration, but *T* is not applicable to package declarations.
- a class, interface, ~~or~~ enum, or record declaration, but *T* is not applicable to type declarations or type contexts; or an annotation type declaration, but *T* is not applicable to annotation type declarations or type declarations or type contexts.
- a method declaration (including an element of an annotation type), but *T* is not applicable

to method declarations or type contexts.

- a constructor declaration, but *T* is not applicable to constructor declarations or type contexts.
- a type parameter declaration of a generic class, interface, method, or constructor, but *T* is not applicable to type parameter declarations or type contexts.
- a field declaration (including an enum constant), but *T* is not applicable to field declarations or type contexts.
- a formal or exception parameter declaration, but *T* is not applicable to either formal and exception parameter declarations or type contexts.
- a receiver parameter, but *T* is not applicable to type contexts.
- a local variable declaration (including a loop variable of a `for` statement or a resource variable of a `try`-with-resources statement), but *T* is not applicable to local variable declarations or type contexts.
- a record component but *T* is not applicable to record component declarations, field declarations, method declarations, formal and exception parameter declarations, or type contexts.

~~Five~~ Six of these ~~nine~~ eleven clauses mention "... or type contexts" because they characterize the ~~five~~ six syntactic locations where an annotation could plausibly apply either to a declaration or to the type of a declared entity. Furthermore, two of the ~~nine~~ eleven clauses - for class, interface, enum, record, and annotation type declarations, and for type parameter declarations - mention "... or type contexts" because it may be convenient to apply an annotation whose type is meta-annotated with `@Target(ElementType.TYPE_USE)` (thus, applicable in type contexts) to a type declaration.

A type annotation is *admissible* if both of the following are true:

- The simple name to which the annotation is closest is classified as a *TypeName*, not a *PackageName*.
- If the simple name to which the annotation is closest is followed by "." and another *TypeName* - that is, the annotation appears as `@Foo T.U` - then `U` denotes an inner class of `T`.

The intuition behind the second clause is that if `Outer.this` is legal in a nested class enclosed by `Outer`, then `Outer` may be annotated because it represents the type of some object at run time. On the other hand, if `Outer.this` is not legal - because the class where it appears has no enclosing instance of `Outer` at run time - then `Outer` may not be annotated because it is logically just a name, akin to components of a package name in a fully qualified type name.

For example, in the following program, it is not possible to write `A.this` in the body of `B`, as `B` has no lexically enclosing instances (8.5.1). Therefore, it is not possible to apply `@Foo` to `A` in the type `A.B`, because `A` is logically just a name, not a type.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
  class A {
    static class B {}
  }

  @Foo A.B x;  // Illegal
```

```
}
```

On the other hand, in the following program, it is possible to write `C.this` in the body of `D`. Therefore, it is possible to apply `@Foo` to `C` in the type `C.D`, because `C` represents the type of some object at run time.

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
  static class C {
    class D {}
  }

  @Foo C.D x;  // Legal
}
```

Finally, note that the second clause looks only one level deeper in a qualified type. This is because a `static` class may only be nested in a top level class or another `static` nested class. It is not possible to write a nest like:

```
@Target(ElementType.TYPE_USE)
@interface Foo {}

class Test {
  class E {
    class F {
      static class G {}
    }
  }

  @Foo E.F.G x;
}
```

Assume for a moment that the nest was legal. In the type of field `x`, `E` and `F` would logically be names qualifying `G`, as `E.F.this` would be illegal in the body of `G`. Then, `@Foo` should not be legal next to `E`. Technically, however, `@Foo` would be admissible next to `E` because the next deepest term `F` denotes an inner class; but this is moot as the class nest is illegal in the first place.

It is a compile-time error if an annotation of type *T* applies to the outermost level of a type in a type context, and *T* is not applicable in type contexts or the declaration context (if any) which occupies the same syntactic location.

It is a compile-time error if an annotation of type *T* applies to a part of a type (that is, not the outermost level) in a type context, and *T* is not applicable in type contexts.

It is a compile-time error if an annotation of type *T* applies to a type (or any part of a type) in a type context, and *T* is applicable in type contexts, and the annotation is not admissible.

For example, assume an annotation type `TA` which is meta-annotated with just `@Target(ElementType.TYPE_USE)`. The terms `@TA java.lang.Object` and `java.@TA lang.Object` are illegal because the simple name to which `@TA` is closest is classified as a package name. On the other hand, `java.lang.@TA Object` is legal.

Note that the illegal terms are illegal "everywhere". The ban on annotating package names applies broadly: to locations which are solely type contexts, such as `class ... extends @TA java.lang.Object {...}`, and to locations which are both declaration and type contexts, such as `@TA java.lang.Object f;`. (There are no locations which are solely declaration contexts where a package name could be annotated, as class, package, and type parameter declarations use only simple names.)

If `TA` is additionally meta-annotated with `@Target(ElementType.FIELD)`, then the term `@TA java.lang.Object` is legal in locations which are both declaration and type contexts, such as a field declaration `@TA java.lang.Object f;`. Here, `@TA` is deemed to apply to the declaration of `f` (and not to the type `java.lang.Object`) because `TA` is applicable in the field declaration context.

Chapter 10: Arrays

10.2 Array Variables

A variable of array type holds a reference to an object. Declaring a variable of array type does not create an array object or allocate any space for array components. It creates only the variable itself, which can contain a reference to an array. However, the initializer part of a declarator (8.3⇗, 9.3⇗, 14.4.1⇗) may create an array, a reference to which then becomes the initial value of the variable.

Example 10.2-1. Declarations of Array Variables

```
int[]     ai;         // array of int
short[][] as;         // array of array of short
short     s,          // scalar short
          aas[][];    // array of array of short
Object[]  ao,         // array of Object
          otherAo;    // array of Object
Collection<?>[] ca;   // array of Collection of unknown type
```

The declarations above do not create array objects. The following are examples of declarations of array variables that do create array objects:

```
Exception ae[]  = new Exception[3];
Object aao[][]  = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[]       = { 'n', 'o', 't', ' ', 'a', ' ',
                    'S', 't', 'r', 'i', 'n', 'g' };
String[] aas    = { "array", "of", "String", };
```

The array type of a variable depends on the bracket pairs that may appear as part of the type at the beginning of a variable declaration, or as part of the declarator for the variable, or both. Specifically, in the declaration of a field, formal parameter, ~~or~~ local variable, or record component (8.3⇗, 8.4.1⇗, 9.3⇗, 9.4⇗, 14.4.1⇗, 14.14.2⇗, 15.27.1⇗, 8.10.1), the array type of the variable is denoted by:

- the element type that appears at the beginning of the declaration; then,
- any bracket pairs that follow the variable's *Identifier* in the declarator (not applicable for a variable arity parameter or variable arity record component); then,
- any bracket pairs that appear in the type at the beginning of the declaration (where the ellipsis of a variable arity parameter or variable arity record component is treated as a bracket pair).

The return type of a method (8.4.5⇗) may be an array type. The precise array type depends on the bracket pairs that may appear as part of the type at the beginning of the method declaration, or after the method's formal parameter list, or both. The array type is denoted by:

- the element type that appears in the *Result*; then,
- any bracket pairs that follow the formal parameter list; then,

- any bracket pairs that appear in the *Result*.

We do not recommend "mixed notation" in array variable declarations, where bracket pairs appear on both the type and in declarators; nor in method declarations, where bracket pairs appear both before and after the formal parameter list.

Example 10.2-2. Array Variables and Array Types

The local variable declaration statement:

```
byte[] rowvector, colvector, matrix[];
```

is equivalent to:

```
byte rowvector[], colvector[], matrix[][];
```

because the array type of each local variable is unchanged. Similarly, the local variable declaration statement:

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

is equivalent to the series of declaration statements:

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

Brackets are allowed in declarators as a nod to the tradition of C and C++. The general rules for variable declaration, however, permit brackets to appear on both the type and in declarators, so that the local variable declaration statement:

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

is equivalent to the series of declarations:

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

Because of how array types are formed, the following parameter declarations have the same array type:

```
void m(int @A [] @B []  x) {}
void n(int @A [] @B ... y) {}
```

And perhaps surprisingly, the following field declarations have the same array type:

```
int @A [] f @B [];
int @B [] @A [] g;
```

Once an array object is created, its length never changes. To make an array variable refer to an array of different length, a reference to a different array must be assigned to the variable.

A single variable of array type may contain references to arrays of different lengths, because an array's length is not part of its type.

If an array variable *v* has type *A*`[]`, where *A* is a reference type, then *v* can hold a reference to an instance of any array type *B*`[]`, provided *B* can be assigned to *A* (5.2⇗). This may result in a run-time exception on a *later* assignment; see 10.5⇗ for a discussion.

Chapter 13: Binary Compatibility

13.1 The Form of a Binary

> *A companion document describes the changes needed to the Java Virtual Machine Specification⇗ to support records.*

Programs must be compiled either into the `class` file format specified by *The Java Virtual Machine Specification, Java SE 14 Edition*, or into a representation that can be mapped into that format by a class loader written in the Java programming language.

A `class` file corresponding to a class or interface declaration must have certain properties. A number of these properties are specifically chosen to support source code transformations that preserve binary compatibility. The required properties are:

1. The class or interface must be named by its *binary name*, which must meet the following constraints:
    - The binary name of a top level class or interface (7.6) is its canonical name (6.7⇗).
    - The binary name of a member class or interface (8.5, 9.5⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by the simple name of the member.
    - The binary name of a local class or interface (14.3) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by a non-empty sequence of digits, followed by the simple name of the local class.
    - The binary name of an anonymous class (15.9.5⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by a non-empty sequence of digits.
    - The binary name of a type variable declared by a generic class or interface (8.1.2⇗, 9.1.2⇗) is the binary name of its immediately enclosing class or interface, followed by `$`, followed by the simple name of the type variable.
    - The binary name of a type variable declared by a generic method (8.4.4⇗) is the binary name of the class or interface declaring the method, followed by `$`, followed by the descriptor of the method (JVMS §4.3.3), followed by `$`, followed by the simple name of the type variable.
    - The binary name of a type variable declared by a generic constructor (8.8.4⇗) is the binary name of the class declaring the constructor, followed by `$`, followed by the descriptor of the constructor (JVMS §4.3.3), followed by `$`, followed by the simple name of the type variable.
2. A reference to another class or interface must be symbolic, using the binary name of the class or interface.
3. A reference to a field that is a constant variable ([4.12.4]) must be resolved at compile time to the value *V* denoted by the constant variable's initializer.

   If such a field is `static`, then no reference to the field should be present in the code in a binary file, including the class or interface which declared the field. Such a field must always appear to have been initialized (12.4.2⇗); the default initial value for the field (if different than *V*) must never be observed.

If such a field is non-`static`, then no reference to the field should be present in the code in a binary file, except in the class containing the field. (It will be a class rather than an interface, since an interface has only `static` fields.) The class should have code to set the field's value to *V* during instance creation (12.5⇗).

4. Given a legal expression denoting a field access in a class *C*, referencing a field named *f* that is not a constant variable and is declared in a (possibly distinct) class or interface *D*, we define the *qualifying type of the field reference* as follows:
    - If the expression is referenced by a simple name, then if *f* is a member of the current class or interface, *C*, then let *T* be *C*. Otherwise, let *T* be the innermost lexically enclosing class or interface declaration of which *f* is a member. In either case, *T* is the qualifying type of the reference.
    - If the reference is of the form *TypeName*`.`*f*, where *TypeName* denotes a class or interface, then the class or interface denoted by *TypeName* is the qualifying type of the reference.
    - If the expression is of the form *ExpressionName*`.`*f* or *Primary*`.`*f*, then:
        - If the compile-time type of *ExpressionName* or *Primary* is an intersection type V_1 `&` ... `&` V_n (4.9⇗), then the qualifying type of the reference is V_1.
        - Otherwise, the compile-time type of *ExpressionName* or *Primary* is the qualifying type of the reference.
    - If the expression is of the form `super`.*f*, then the superclass of *C* is the qualifying type of the reference.
    - If the expression is of the form *TypeName*`.super.`*f*, then the superclass of the class denoted by *TypeName* is the qualifying type of the reference.

    The reference to *f* must be compiled into a symbolic reference to the erasure (4.6⇗) of the qualifying type of the reference, plus the simple name of the field, *f*. The reference must also include a symbolic reference to the erasure of the declared type of the field so that the verifier can check that the type is as expected.
5. Given a method invocation expression or a method reference expression in a class or interface *C*, referencing a method named *m* declared (or implicitly declared (9.2⇗)) in a (possibly distinct) class or interface *D*, we define the *qualifying type of the method invocation* as follows:
    - If *D* is `Object` then the qualifying type of the expression is `Object`.
    - Otherwise:
        - If the method is referenced by a simple name, then if *m* is a member of the current class or interface *C*, let *T* be *C*; otherwise, let *T* be the innermost lexically enclosing class or interface declaration of which *m* is a member. In either case, *T* is the qualifying type of the method invocation.
        - If the expression is of the form *TypeName*`.`*m* or *ReferenceType*`::`*m*, then the type denoted by *TypeName* or *ReferenceType* is the qualifying type of the method invocation.
        - If the expression is of the form *ExpressionName*`.`*m* or *Primary*`.`*m* or *ExpressionName*`::`*m* or *Primary*`::`*m*, then:
            - If the compile-time type of *ExpressionName* or *Primary* is an intersection

type V_1 `&` ... `&` V_n (4.9⇗), then the qualifying type of the method invocation is V_1.

- Otherwise, the compile-time type of *ExpressionName* or *Primary* is the qualifying type of the method invocation.

- If the expression is of the form `super.`*m* or `super::`*m*, then the superclass of *C* is the qualifying type of the method invocation.
- If the expression is of the form *TypeName*`.super.`*m* or *TypeName*`.super::`*m*, then if *TypeName* denotes a class *X*, the superclass of *X* is the qualifying type of the method invocation; if *TypeName* denotes an interface *X*, *X* is the qualifying type of the method invocation.

A reference to a method must be resolved at compile time to a symbolic reference to the erasure (4.6⇗) of the qualifying type of the invocation, plus the erasure of the signature (8.4.2⇗) of the method. The signature of a method must include all of the following as determined by 15.12.3⇗:

- The simple name of the method
- The number of parameters to the method
- A symbolic reference to the type of each parameter

A reference to a method must also include either a symbolic reference to the erasure of the return type of the denoted method or an indication that the denoted method is declared `void` and does not return a value.

6. Given a class instance creation expression (15.9⇗) or an explicit constructor invocation statement (8.8.7.1⇗) or a method reference expression of the form *ClassType* `::` `new` (15.13⇗) in a class or interface *C* referencing a constructor *m* declared in a (possibly distinct) class or interface *D*, we define the qualifying type of the constructor invocation as follows:

   - If the expression is of the form `new` *D*`(...)` or *ExpressionName*`.new` *D*`(...)` or *Primary*`.new` *D*`(...)` or *D* `::` `new`, then the qualifying type of the invocation is *D*.
   - If the expression is of the form `new` *D*`(...){...}` or *ExpressionName*`.new` *D*`(...){...}` or *Primary*`.new` *D*`(...){...}`, then the qualifying type of the expression is the compile-time type of the expression.
   - If the expression is of the form `super(...)` or *ExpressionName*`.super(...)` or *Primary*`.super(...)`, then the qualifying type of the expression is the direct superclass of *C*.
   - If the expression is of the form `this(...)`, then the qualifying type of the expression is *C*.

   A reference to a constructor must be resolved at compile time to a symbolic reference to the erasure (4.6⇗) of the qualifying type of the invocation, plus the signature of the constructor (8.8.2⇗). The signature of a constructor must include both:

   - The number of parameters of the constructor
   - A symbolic reference to the type of each formal parameter

A binary representation for a class or interface must also contain all of the following:

1. If it is a class and is not `Object`, then a symbolic reference to the erasure of the direct superclass of this class.
2. A symbolic reference to the erasure of each direct superinterface, if any.
3. A specification of each field declared in the class or interface, given as the simple name of the field and a symbolic reference to the erasure of the type of the field.
4. If it is a class, then the erased signature of each constructor, as described above.
5. For each method declared in the class or interface (excluding, for an interface, its implicitly declared methods (9.2⇗)), its erased signature and return type, as described above.
6. The code needed to implement the class or interface:
    - For an interface, code for the field initializers and the implementation of each method with a block body (9.4.3⇗).
    - For a class, code for the field initializers, the instance and static initializers, the implementation of each method with a block body (8.4.7⇗), and the implementation of each constructor.
7. Every class or interface must contain sufficient information to recover its canonical name (6.7⇗).
8. Every member class or interface must have sufficient information to recover its source-level access modifier.
9. Every nested class or interface must have a symbolic reference to its immediately enclosing class or interface (8.1.3⇗).
10. Every class or interface must contain symbolic references to all of its member classes and interfaces (8.5, 9.5⇗), and to all other nested classes and interfaces declared within its body.
11. A construct emitted by a Java compiler must be marked as *synthetic* if it does not correspond to a construct declared explicitly or implicitly in source code, unless the emitted construct is a class initialization method (JVMS §2.9).
12. A construct emitted by a Java compiler must be marked as *mandated* if it corresponds to a formal parameter declared implicitly in source code (8.8.1⇗, 8.8.9⇗, 8.9.3⇗, 15.9.5.1⇗).

The following formal parameters are declared implicitly in source code:

- *The first formal parameter of a constructor of a non-`private` inner member class (8.8.1⇗, 8.8.9⇗).*
- *The first formal parameter of an anonymous constructor of an anonymous class whose superclass is an inner class (not in a static context) (15.9.5.1⇗).*
- *The formal parameter `name` of the `valueOf` method which is implicitly declared in an enum class (8.9.3⇗).*
- *The formal parameters of a compact constructor of a record class (8.10.4).*

For reference, the following constructs are declared implicitly in source code, but are not marked as mandated because only formal parameters can be so marked in a `class` file (JVMS §4.7.24):

- *Default constructors of classes (8.8.9⇗, 8.9.2⇗)*
- *Canonical constructors of record classes (8.10.4)*
- *Anonymous constructors (15.9.5.1⇗)*
- *The `values` and `valueOf` methods of enum classes (8.9.3⇗)*

- *Certain `public` fields of enum classes (8.9.3)*
- *Certain `private` fields and `public` methods of record classes (8.10.3)*
- *Certain `public` methods of interfaces (9.2)*
- *Container annotations (9.7.5)*

A `class` file corresponding to a module declaration must have the properties of a `class` file for a class whose binary name is `module-info` and which has no superclass, no superinterfaces, no fields, and no methods. In addition, the binary representation of the module must contain all of the following:

- A specification of the name of the module, given as a symbolic reference to the name indicated after `module`. Also, the specification must include whether the module is normal or open (7.7).
- A specification of each dependence denoted by a `requires` directive, given as a symbolic reference to the name of the module indicated by the directive (7.7.1). Also, the specification must include whether the dependence is `transitive` and whether the dependence is `static`.
- A specification of each package denoted by an `exports` or `opens` directive, given as a symbolic reference to the name of the package indicated by the directive (7.7.2). Also, if the directive was qualified, the specification must give symbolic references to the names of the modules indicated by the directive's `to` clause.
- A specification of each service denoted by a `uses` directive, given as a symbolic reference to the name of the class or interface indicated by the directive (7.7.3).
- A specification of the service providers denoted by a `provides` directive, given as symbolic references to the names of the classes and interfaces indicated by the directive's `with` clause (7.7.4). Also, the specification must give a symbolic reference to the name of the class or interface indicated as the service by the directive.

The following sections discuss changes that may be made to class and interface declarations without breaking compatibility with pre-existing binaries. Under the translation requirements given above, the Java Virtual Machine and its `class` file format support these changes. Any other valid binary format, such as a compressed or encrypted representation that is mapped back into `class` files by a class loader under the above requirements, will necessarily support these changes as well.

13.4 Evolution of Classes

13.4.27 Evolution of Record Classes

Adding, deleting, changing, or reordering record components in a record class declaration may break compatibility with pre-existing binaries that are not recompiled; such a change is not recommended for widely distributed record classes.

More precisely, adding, deleting, changing, or reordering record components may change the corresponding implicit field declarations and corresponding accessor method declarations, as well as changing the signature and implementation of the canonical constructor and other supporting methods, with consequences described in 13.4.8 and 13.4.12.

In all other respects, the binary compatibility rules for record classes are identical to those for normal classes.

Chapter 14: Blocks and Statements

14.3 Local Class Declarations

A *local class* or a *local interface* is a nested class or interface (8, 9⇗) whose declaration is immediately contained by a block (14.2⇗).

LocalClassOrInterfaceDeclaration:
ClassDeclaration
EnumDeclaration
RecordDeclaration
InterfaceDeclaration

An annotation interface (9.6⇗) may not be declared as a local interface.

Local class and interface declarations may be intermixed freely with statements in the block.

A local class or interface is not a member of any package, class, or interface. Unlike an anonymous class (15.9.5⇗), a local class or interface has a simple name (6.2⇗, 6.7⇗).

Local enum classes, record classes, and ~~local~~ interfaces are implicitly `static` (8.1.1.4, 9.1.1.3⇗). A local class that is not implicitly `static` is an inner class (8.1.3⇗).

It is a compile-time error if a local class or interface is declared with any of the access modifiers `public`, `protected`, or `private` (6.6⇗), or the modifier `static` (8.1.1⇗).

The scope and shadowing of a local class or interface declaration is specified in 6.3⇗ and 6.4⇗.

Example 14.3-1. Local Class and Interface Declarations

Here is an example that illustrates several aspects of the rules given above:

```
class Global {
    class Cyclic {}

    void foo() {
        new Cyclic(); // create a Global.Cyclic
        class Cyclic extends Cyclic {} // circular definition

        {
            class Local {}
            {
                class Local {} // compile-time error
            }
            class Local {} // compile-time error
            class AnotherLocal {
                void bar() {
                    class Local {} // ok
                }
            }
        }
        class Local {} // ok, not in scope of prior Local
    }
}
```

The first statement of method `foo` creates an instance of the member class `Global.Cyclic` rather than an instance of the local class `Cyclic`, because the statement appears prior to the scope of the local class declaration.

The fact that the scope of a local class declaration encompasses its whole declaration (not only its body) means that the definition of the local class `Cyclic` is indeed cyclic because it extends itself rather than `Global.Cyclic`. Consequently, the declaration of the local class `Cyclic` is rejected at compile time.

Since local class names cannot be redeclared within the same method (or constructor or initializer, as the case may be), the second and third declarations of `Local` result in compile-time errors. However, `Local` can be redeclared in the context of another, more deeply nested, class such as `AnotherLocal`.

The final declaration of `Local` is legal, since it occurs outside the scope of any prior declaration of `Local`.

Chapter 16: Definite Assignment

Each local variable (14.4⇗) 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 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~~ *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.

> *This is an editorial change to italicize the notion of an assignment occurring.*

...

DRAFT 16-internal+0-adhoc.gbierman.20200923