Sealed Types

Changes to the Java® Language Specification • Version 14-internal+0-adhoc.gbierman.20200228

Chapter 3: Lexical Structure
  3.8 Identifiers
  3.9 Keywords

Chapter 4: Types, Values, and Variables
  4.11 Where Types Are Used

Chapter 5: Conversions and Contexts
  5.1 Kinds of Conversion
    5.1.6.1 Allowed Narrowing Reference Conversion

Chapter 8: Classes
  8.1 Class Declarations
    8.1.1 Class Modifiers
      8.1.1.2 sealed, non-sealed, and final Classes
    8.1.4 Superclasses and Subclasses
    8.1.5 Superinterfaces
    8.1.6 permits Clause
  8.9 Enum Types

Chapter 9: Interfaces
  9.1 Interface Declarations
    9.1.1 Interface Modifiers
      9.1.1.3 sealed and non-sealed Interfaces
    9.1.3 Superinterfaces and Subinterfaces
    9.1.4 Permitted Subtypes

Chapter 13: Binary Compatibility
  13.4 Evolution of Classes
    13.4.2 final Classes sealed, non-sealed and final Classes
    13.4.2.1 sealed Classes
    13.4.2.2 non-sealed Classes
    13.4.2.3 final Classes
    13.4.4 Superclasses and Superinterfaces

Chapter 14: Blocks and Statements
  14.3 Local Class Declarations

Chapter 15: Expressions
  15.9 Class Instance Creation Expressions
    15.9.5 Anonymous Class Declarations
  15.27 Lambda Expressions
    15.27.3 Type of a Lambda Expression

This document describes changes to the Java Language Specification to support sealed types. See JEP 360.
A companion document describes the changes needed to the Java Virtual Machine Specification to support sealed types.

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 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.

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, sealed, and permits 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.

A type identifier is an identifier that not one of the following character sequences: var, permits, sealed, or yield.

TypeIdentifier:
   Identifier but not var, permits, sealed or yield

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, permits, sealed, or yield (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

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

Keyword:

- abstract
double
int
strictfp
char
for
interface
super
assert
else
long
switch
boolean
enum
native
synchronized
break
extends
new
this
byte
final
non-sealed
throw
The keywords\texttt{const} and\texttt{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.

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

- \texttt{true} and \texttt{false} are not keywords, but rather boolean literals (3.10.3).\footnote{3.10.3}
- \texttt{null} is not a keyword, but rather the null literal (3.10.7).\footnote{3.10.7}
- \texttt{The restricted identifiers} \texttt{var, and yield, sealed and permits} are not keywords. \texttt{var} has special meaning as the type of a local variable declaration (14.4, 14.1.1, 14.1.2, 14.20.3) and the type of a lambda formal parameter (15.27.1). \texttt{yield} has special meaning in a\texttt{yield} statement (14.21). All invocations of a method called \texttt{yield} must be qualified so as to be distinguished from a\texttt{yield} statement. \texttt{sealed and permits} are used in class and interface declarations to limit subclassing and extension, respectively.

A further ten character sequences are \textit{restricted keywords}: \texttt{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 \texttt{ModuleDeclaration}, \texttt{ModuleDirective}, and \texttt{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 \texttt{requires} in the \texttt{ModuleDirective} production, the character sequence \texttt{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 \textit{type contexts} where types are used:

- In declarations:
  1. A type in the \texttt{extends} or \texttt{implements} or \texttt{permits} clause of a class declaration (8.1.4, 8.1.5, 8.5, 9.5)
  2. A type in the \texttt{extends} or \texttt{permits} 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⇗)


10. The type in an exception parameter declaration (14.20⇗)

- 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 ρ, 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 ρ, for class and interface types that are raw
• 4.9 ρ, for intersection types in the bounds of type parameters
• 6.5 ρ, for class and interface types in contexts where genericity is unimportant (6.1 ρ)
• 10.1 ρ, 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 ρ) 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 ρ), 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 ρ, 15.9 ρ, 15.12 ρ, 15.13 ρ).

• 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 ρ)
  • As the type that follows the instanceof relational operator (15.20.2 ρ)

• 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 ρ):
  • As the type of an exception that can be thrown by a method or constructor (8.4.6 ρ, 8.8.5 ρ, 9.4 ρ)
  • In an exception parameter declaration (14.20 ρ)

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 ρ. 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 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 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

```java
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 5: Conversions and Contexts

#### 5.1 Kinds of Conversion

##### 5.1.6.1 Allowed Narrowing Reference Conversion

A **narrowing reference conversion** exists from reference type `S` to reference type `T` if all of the following are true:

- `S` is not a subtype of `T` (4.10)
If there exists a parameterized type $X$ that is a supertype of $T$, and a parameterized type $Y$ that is a supertype of $S$, such that the erasures of $X$ and $Y$ are the same, then $X$ and $Y$ are not provably distinct (4.5⇗).

Using types from the java.util package as an example, no narrowing reference conversion exists from ArrayList<String> to ArrayList<Object>, or vice versa, because the type arguments String and Object are provably distinct. For the same reason, no narrowing reference conversion exists from ArrayList<String> to List<Object>, or vice versa. The rejection of provably distinct types is a simple static gate to prevent "stupid" narrowing reference conversions.

One of the following cases applies:

1. $S$ and $T$ are class types, and either $|S| <: |T|$ or $|T| <: |S|$.
2. $S$ and $T$ are interface types.
3. $S$ is a class type, $T$ is an interface type, and $S$ does not name a final class (8.1.1).
4. $S$ is a class type, $T$ is an interface type, and $S$ names a final class that implements the interface named by $T$.
5. $S$ is an interface type, $T$ is a class type, and $T$ does not name a final class.
6. $S$ is an interface type, $T$ is a class type, and $T$ names a final class that implements the interface named by $S$.

   1. Both $S$ and $T$ are class or interface types, and $S$ names a class or interface that is not disjoint from the class or interface named by $T$.

   2. $S$ is the class type Object or the interface type java.io.Serializable or Cloneable (the only interfaces implemented by arrays (10.8⇗)), and $T$ is an array type.

   3. $S$ is an array type $SC[]$, that is, an array of components of type $SC$; $T$ is an array type $TC[]$, that is, an array of components of type $TC$; and a narrowing reference conversion exists from $SC$ to $TC$.

   4. $S$ is a type variable, and a narrowing reference conversion exists from the upper bound of $S$ to $T$.

   5. $T$ is a type variable, and either a widening reference conversion or a narrowing reference conversion exists from $S$ to the upper bound of $T$.

   6. $S$ is an intersection type $S_1 \& \ldots \& S_n$, and for all $i (1 \leq i \leq n)$, either a widening reference conversion or a narrowing reference conversion exists from $S_i$ to $T$.

   7. $T$ is an intersection type $T_1 \& \ldots \& T_n$, and for all $i (1 \leq i \leq n)$, either a widening reference conversion or a narrowing reference conversion exists from $S$ to $T_i$.

A class or interface is said to be disjoint from another class or interface if it can be determined statically that they can have no instances in common (other than the null value). This is is the underlying notion used in the definition of narrowing reference conversion. It is defined as follows:

1. A class named $C$ is disjoint from an interface named $I$ if it is not the case that $C <: I$. 

A class or interface is said to be disjoint from another class or interface if it can be determined statically that they can have no instances in common (other than the null value). This is is the underlying notion used in the definition of narrowing reference conversion. It is defined as follows:

1. A class named $C$ is disjoint from an interface named $I$ if it is not the case that $C <: I$.
and one of the following cases applies:

a. \( C \) is final.

b. \( C \) is sealed, and all of the permitted subclasses of \( C \) are disjoint from \( I \).

c. \( C \) is extensible (8.1.1.2), and \( I \) is sealed, and \( C \) is disjoint from all of the permitted subtypes of \( I \).

Rules (b) and (c) simply unpack the sealed hierarchies; the important case is rule (a).

Consider the following declarations:

\[\text{interface } I \{\}\]
\[\text{final class } C \{\}\]

As class \( C \) is final and does not implement \( I \), there can be no instances of \( C \) that are also an instance of \( I \), so the types are disjoint. Hence there is no narrowing reference conversion from \( C \) to \( I \).

In contrast, consider the following declarations:

\[\text{interface } J \{\}\]
\[\text{class } D \{\}\]

Even though class \( D \) does not implement \( J \), it is still possible that an instance of \( D \) is an instance of \( J \); for example, if the following declaration has been made (possibly at a later time):

\[\text{class } E \text{ extends } D \text{ implements } J \{\}\]

For this reason \( D \) is not disjoint from \( J \), and there is a narrowing reference conversion from \( D \) to \( J \).

2. An interface named \( I \) is disjoint from a class named \( C \) if \( C \) is disjoint from \( I \).

3. A class named \( C \) is disjoint from another class named \( D \) if it is not the case that \( C <: D \), and it is not the case that \( D <: C \).

4. An interface named \( I \) is disjoint from another interface named \( J \) if it is not that case that \( I <: J \), and it is not the case that \( J <: I \), and additionally one of the following cases applies:

   a. \( I \) is sealed, and all of the permitted subtypes of \( I \) are disjoint from \( J \).

   b. \( J \) is sealed, and \( I \) is disjoint from all the permitted subtypes of \( J \).

Chapter 8: Classes

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

A top level class is a class that is not a nested class.

A nested class is any class whose declaration occurs within the body of another class or interface.

This chapter discusses the common semantics of all classes - top level (7.6) and nested (including member classes (8.5), local classes (14.3), and anonymous 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) and must be declared abstract if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses.

The degree to which a class can be extended can be explicitly controlled (8.1.1.2). A class may be declared sealed, in which case there is a fixed set of classes that directly extend the sealed class. A class may be declared final, in which case it cannot have any 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 and methods and nested 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 class members and interface members can hide class or interface members declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

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

Member class declarations (8.5) describe nested classes that are members of the surrounding class. Member classes may be static, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes (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 type; an instance method is invoked with respect to some particular object that is an instance of a class type. 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

A class declaration specifies a new named reference type.
There are two kinds of class declarations: normal class declarations and enum declarations.

ClassDeclaration:
   NormalClassDeclaration
   EnumDeclaration

NormalClassDeclaration:
   { ClassModifier } class TypeIdentifier [ TypeParameters ]
   [ Superclass ] [ Superinterfaces ]
   [ PermittedSubclasses ] ClassBody

The rules in this section apply to all class declarations, including enum declarations. However, special rules apply to enum declarations with regard to class modifiers, inner classes, and superclasses; these rules are stated in 8.9.

The TypeIdentifier in a class declaration specifies the name of the class.
It is a compile-time error if a class has the same simple name as any of its enclosing classes or interfaces.
The scope and shadowing of a class declaration is specified in 6.3⇗ and 6.4⇗.

8.1.1 Class Modifiers

A class declaration may include class modifiers.

ClassModifier:
   (one of)
   Annotation public protected private
   abstract static sealed non-sealed final strictfp

The rules for annotation modifiers on a class declaration are specified in 9.7.4⇗ and 9.7.5⇗.
The access modifier public (6.6⇗) pertains only to top level classes (7.6⇗) and member classes (8.5⇗), not to local classes (14.3⇗) or anonymous classes (15.9.5⇗).
The access modifiers protected and private pertain only to member classes within a directly enclosing class declaration (8.5⇗).
The modifier static pertains only to member classes (8.5.1), not to top level or local or anonymous classes.

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

It is a compile-time error if a class declaration has more than one of the class modifiers sealed, non-sealed and final.

If two or more (distinct) class modifiers appear in a class declaration, then it is customary, though not required, that they appear in the order consistent with that shown above in the production for ClassModifier.

8.1.1.2 sealed, non-sealed, and final Classes

Extensibility of class hierarchies is an important feature of object-oriented programming. However, there are circumstances where explicit control of this extensibility is desirable.

A class can be declared sealed when it is useful to restrict its direct subclasses to a fixed set of classes.

A class can be declared final if its definition is complete and no subclasses are desired or required.

A class can be declared non-sealed when it is useful to allow unrestricted direct subclasses.

A class is said to be extensible if it is not declared sealed and is not declared final. (A class that is declared non-sealed is then extensible.)

If a class \( C \) has a sealed direct superclass (8.1.4), or a sealed direct superinterface (8.1.5), one of the following must apply otherwise a compile-time error occurs:

- The class is an enum type (8.9).
- The class is explicitly declared final or sealed.
- The class is explicitly declared non-sealed, meaning that there are no restrictions on the subclasses of \( C \).

JEP 359 proposes extending the Java language with Records, a special kind of class type, which is implicitly final. In this case, we would add the following rule to the preceding list.

- The class is a record type.

It is a compile-time error if a class that does not have a sealed direct superclass, or a sealed direct superinterface, is declared non-sealed.

It is a compile-time error if the name of a final class appears in the extends clause (8.1.4) of another class declaration; this implies that a final class cannot have any subclasses.

It is a compile-time error if a class is declared both final and abstract, because the implementation of such a class could never be completed (8.1.1).
Because a final class never has any subclasses, the methods of a final class are never overridden (8.4.8.1).  

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 sealed class and the current class is not a permitted subclass of ClassType (8.1.6).

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 the class Enum or any invocation of Enum (8.9).

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 $C_{<F_1,...,F_n>}$ ($n \geq 0$, $C \neq \text{Object}$), 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 $\theta$ is the substitution $[F_i:=T_i,...,F_n:=T_n]$.

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; }
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**

```java
class Colored3dPoint extends ColoredPoint { int z; } // error
```

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**

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

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

### 8.1.5 Superinterfaces

The optional implements clause in a class declaration lists the names of interfaces that are
direct superinterfaces of the class being declared.

Superinterfaces:
  implements InterfaceTypeList

InterfaceTypeList:
  InterfaceType { , InterfaceType }

Each InterfaceType must name an accessible interface type (6.6 ⇨), or a compile-time error occurs.

It is a compile-time error if an InterfaceType names a sealed interface, and the class being declared is not contained in the permits clause of InterfaceType (9.1.1.3).

If an InterfaceType 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.

It is a compile-time error if the same interface is mentioned as a direct superinterface more than once in a single implements clause. This is true even if the interface is named in different ways.

Example 8.1.5-1. Illegal Superinterfaces

```
class Redundant implements java.lang.Cloneable, Cloneable {
    int x;
}
```

This program results in a compile-time error because the names java.lang.Cloneable and Cloneable refer to the same interface.

Given a (possibly generic) class declaration $C_{<F_1,\ldots,F_n>}$ ($n \geq 0$, $C \neq \text{Object}$), the direct superinterfaces of the class type $C_{<F_1,\ldots,F_n>}$ are the types given in the implements clause of the declaration of $C$, if an implements clause is present.

Given a generic class declaration $C_{<F_1,\ldots,F_n>}$ ($n > 0$), the direct superinterfaces of the parameterized class type $C_{<T_1,\ldots,T_n>}$, where $T_i$ ($1 \leq i \leq n$) is a type, are all types $<\theta_{U_1},\ldots,U_k\theta>$, where $<\theta_{U_1},\ldots,U_k\theta>$ is a direct superinterface of $C_{<F_1,\ldots,F_n>}$ and $\theta$ is the substitution $[F_1 := T_1,\ldots,F_n := T_n]$.

An interface type $I$ is a superinterface of class type $C$ if any of the following is true:

- $I$ is a direct superinterface of $C$.
- $C$ has some direct superinterface $J$ for which $I$ is a superinterface, using the definition of "superinterface of an interface" given in 9.1.3.
- $I$ is a superinterface of the direct superclass of $C$.

A class can have a superinterface in more than one way.

A class is said to implement all its superinterfaces.

A class may not at the same time be a subtype of two interface types which are different parameterizations of the same generic interface (9.1.2 ⇨), or a subtype of a
parameterization of a generic interface and a raw type naming that same generic interface, or a compile-time error occurs.

This requirement was introduced in order to support translation by type erasure (4.6).
superinterface, because a class that is not abstract is not permitted to have abstract methods (8.1.1.1).

Each default method (9.4.3) of a superinterface of the class may optionally be overridden by a method in the class; if not, the default method is typically inherited and its behavior is as specified by its default body.

It is permitted for a single method declaration in a class to implement methods of more than one superinterface.

**Example 8.1.5-3. Implementing Methods of a Superinterface**

```java
interface Colorable {
    void setColor(int color);
    int getColor();
}

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

This program causes a compile-time error, because ColoredPoint is not an abstract class but fails to provide an implementation of methods setColor and getColor of the interface Colorable.

In the following program:

```java
interface Fish {
    int getNumberOfScales();
}
interface Piano {
    int getNumberOfScales();
}
class Tuna implements Fish, Piano {
    // You can tune a piano, but can you tuna fish?
    public int getNumberOfScales() { return 91; }
}
```

the method getNumberOfScales in class Tuna has a name, signature, and return type that matches the method declared in interface Fish and also matches the method declared in interface Piano; it is considered to implement both.

On the other hand, in a situation such as this:

```java
interface Fish {
    int getNumberOfScales();
}
interface StringBass {
    double getNumberOfScales();
}
class Bass implements Fish, StringBass {
    // This declaration cannot be correct,
    // no matter what type is used.
    public ?? getNumberOfScales() { return 91; }
}
```

it is impossible to declare a method named getNumberOfScales whose signature and return type are compatible with those of both the methods declared in interface Fish and in interface StringBass, because a class cannot have multiple methods with the same signature and different primitive return types (8.4). Therefore, it is impossible for a single class to implement both interface Fish and interface StringBass (8.4.8).

**8.1.6 permits Clause**
A class can only be an extension of a sealed class if it is permitted by the sealed class. The permitted subclasses of a sealed class C are declared in a permits clause. A sealed class C may have an explicitly declared permits clause, which provides a non-empty list of the permitted subclasses of C.

```
PermittedSubclasses
   permits TypeName { , TypeName }.
```

Every TypeName must denote a class type that is accessible (6.6 ⇨). It is a compile-time error if any TypeName denotes a class type that is not accessible, or denotes a type variable.

It is a compile-time error if any TypeName denotes the class C, or a class that is a superclass of C (8.1.4), or any interface type (8.1.5).

It is a compile-time error if the denotation of a class type appears more than once in a single permits clause. This is true even if the class type is named in different ways.

A sealed class C without an explicitly declared permits clause, has an implicitly declared permits clause that lists as permitted subclasses all the classes in the same compilation unit (7.3 ⇨) as C that declare C as their direct superclass.

It is a compile-time error for a sealed class to have an empty permits clause.

This means that it is a compile-time error to declare a sealed class that has no permits clause, and has no direct subclasses in the same compilation unit.

It is a compile-time error if any permitted subclass of a sealed class C is not a member of the same module as C. If the sealed class C is a member of the unnamed module, then it is a compile-time error if any permitted subclass is not in the same package as C.

It is a compile-time error if any permitted subclass of a sealed class C does not declare C as a direct superclass.

It is a compile-time error if a class that is not sealed has a permits clause.

### 8.9 Enum Types

An enum declaration specifies a new enum type, a special kind of class type.

```
EnumDeclaration:
   { ClassModifier } enum TypeIdentifier [ Superinterfaces ] EnumBody
```

It is a compile-time error if an enum declaration has the modifier abstract, or final, sealed or non-sealed.

An enum declaration is implicitly final unless it contains at least one enum constant that has a class body (8.9.1 ⇨).

A nested enum type is implicitly static. It is permitted for the declaration of a nested enum type to redundantly specify the static modifier.
This implies that it is impossible to declare an enum type in the body of an inner class (8.1.3), because an inner class cannot have static members except for constant variables.

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

The direct superclass of an enum type $E$ is Enum$<E>$ (8.1.4).

An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type (15.9.1).

In addition to the compile-time error, three further mechanisms ensure that no instances of an enum type exist beyond those defined by its enum constants:

- The final clone method in Enum ensures that enum constants can never be cloned.
- Reflective instantiation of enum types is prohibited.
- Special treatment by the serialization mechanism ensures that duplicate instances are never created as a result of deserialization.

Chapter 9: Interfaces

An interface declaration introduces a new reference type whose members are classes, interfaces, constants, and methods. This type has no instance variables, and typically declares one or more abstract methods; otherwise unrelated classes can implement the interface by providing implementations for its abstract methods. Interfaces may not be directly instantiated.

A nested interface is any interface whose declaration occurs within the body of another class or interface.

A top level interface is an interface that is not a nested interface.

We distinguish between two kinds of interfaces - normal interfaces and annotation types.

This chapter discusses the common semantics of all interfaces - normal interfaces, both top level (7.6) and nested (8.5, 9.5), and annotation types (9.6). Details that are specific to particular kinds of interfaces are discussed in the sections dedicated to these constructs.

Programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to Object.

An interface may be declared to be a direct extension of one or more other interfaces, meaning that it inherits all the member types, instance methods, and constants of the interfaces it extends, except for any members that it may override or hide.

A class may be declared to directly implement one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct superclasses and direct superinterfaces do. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing a superclass.

Unlike a class, an interface cannot be declared final. However, an interface may be declared sealed (9.1.1.3), in which case it specifies a fixed set of classes and interfaces that
may directly implement or extend the sealed interface.

A variable whose declared type is an interface type may have as its value a reference to any instance of a class which implements the specified interface. It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface, or else the class is not considered to implement the interface.

9.1 Interface Declarations

An interface declaration specifies a new named reference type. There are two kinds of interface declarations - normal interface declarations and annotation type declarations (9.6 ⇧).

InterfaceDeclaration:
  NormalInterfaceDeclaration
  AnnotationTypeDeclaration

NormalInterfaceDeclaration:
  {InterfaceModifier} interface TypeIdentifier [TypeParameters]
  [ExtendsInterfaces] [PermittedSubtypes] InterfaceBody

The TypeIdentifier in an interface declaration specifies the name of the interface.

It is a compile-time error if an interface has the same simple name as any of its enclosing classes or interfaces.

The scope and shadowing of an interface declaration is specified in 6.3 ⇧ and 6.4 ⇧.

9.1.1 Interface Modifiers

An interface declaration may include interface modifiers.

InterfaceModifier:
  (one of)
  Annotation public protected private
  abstract static sealed non-sealed strictfp

The rules for annotation modifiers on an interface declaration are specified in 9.7.4 ⇧ and 9.7.5 ⇧.

The access modifier public (6.6 ⇧) pertains to every kind of interface declaration.

The access modifiers protected and private pertain only to member interfaces whose declarations are directly enclosed by a class declaration (8.5.1 ⇧).

The modifier static pertains only to member interfaces (8.5.1 ⇧, 9.5 ⇧), not to top level interfaces (7.6 ⇧).

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

It is a compile-time error if an interface declaration has more than one of the interface modifiers sealed, and non-sealed.
9.1.1.3 sealed and non-sealed Interfaces

An interface can be declared sealed when it is useful to restrict its direct subtypes to a fixed set of classes and interfaces.

In certain circumstances, an interface can be declared non-sealed to allow unrestricted direct subtypes.

If an interface \( I \) has a sealed direct superinterface (9.1.3), one of the following must apply otherwise a compile-time error occurs:

- The interface \( I \) is explicitly declared sealed.
- The interface \( I \) is explicitly declared non-sealed, meaning that there are no restrictions on the subtypes of \( I \).

It is a compile-time error if an interface that does not have a sealed superinterface is declared non-sealed.

9.1.3 Superinterfaces and Subinterfaces

If an extends clause is provided, then the interface being declared extends each of the other named interfaces and therefore inherits the member types, instance methods, and constants of each of the other named interfaces.

These other named interfaces are the direct superinterfaces of the interface being declared.

Any class that implements the declared interface is also considered to implement all the interfaces that this interface extends.

```
ExtendsInterfaces:
    extends InterfaceTypeList
```

The following production from 8.1.5 is shown here for convenience:

```
InterfaceTypeList:
    InterfaceType { , InterfaceType }
```

Each InterfaceType in the extends clause of an interface declaration must name an accessible interface type (6.6), or a compile-time error occurs.

It is a compile-time error if an InterfaceType names an interface that is sealed, and the interface being declared is not a permitted subtype of InterfaceType (9.1.4).

If an InterfaceType 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) interface declaration \( \langle F_1, \ldots, F_n \rangle \) \((n \geq 0)\), the direct superinterfaces of the interface type \( \langle F_1, \ldots, F_n \rangle \) are the types given in the extends clause of
the declaration of \( I \), if an extends clause is present.

Given a generic interface declaration \(<F_1,...,F_n>\) \((n > 0)\), the direct superinterfaces of the parameterized interface type \(<T_1,...,T_n>\), where \( T_i \) \(1 \leq i \leq n\) is a type, are all types \(<U_1,\ldots,U_k,\theta>\), where \(<U_1,\ldots,U_k>\) is a direct superinterface of \(<F_1,...,F_n>\) and \( \theta \) is the substitution \([F_1:=T_1,\ldots,F_n:=T_n]\).

The superinterface relationship is the transitive closure of the direct superinterface relationship. An interface \( K \) is a superinterface of interface \( I \) if either of the following is true:

- \( K \) is a direct superinterface of \( I \).
- There exists an interface \( J \) such that \( K \) is a superinterface of \( J \), and \( J \) is a superinterface of \( I \), applying this definition recursively.

Interface \( I \) is said to be a subinterface of interface \( K \) whenever \( K \) is a superinterface of \( I \).

While every class is an extension of class \( \text{Object} \), there is no single interface of which all interfaces are extensions.

An interface \( I \) directly depends on a type \( T \) if \( T \) is mentioned in the extends clause of \( I \) either as a superinterface or as a qualifier in the fully qualified form of a superinterface name.

An interface \( I \) depends on a reference type \( T \) if any of the following is true:

- \( I \) directly depends on \( T \).
- \( I \) directly depends on a class \( C \) that depends on \( T \)(8.1.5).
- \( I \) directly depends on an interface \( J \) that depends on \( T \) (using this definition recursively).

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

If circularly declared interfaces are detected at run time, as interfaces are loaded, then a \texttt{ClassCircularityError} is thrown (12.2.1angkan).

### 9.1.4 Permitted Subtypes

This is a new subsection. The existing subsection 9.1.4 “Interface Body and Member Declarations” is renumbered to 9.1.5.

A class can only be a direct implementation of a sealed interface if it is permitted by the sealed interface. Similarly, an interface can only be a direct extension of a sealed interface if it is permitted by the sealed interface. The permitted subtypes of a sealed interface are declared in a permits clause. A sealed interface \( I \) may have an explicit permits clause, which provides a non-empty list of the permitted subtypes of \( I \).

\[
\text{PermittedSubtypes}
\]

\[\text{permits \_TypeName} {, \_TypeName} .\]

Every \_TypeName must denote a class or interface type that is accessible (6.6angkan). It is a compile-time error if any \_TypeName denotes a class or interface type that is not accessible, or denotes a type variable.
It is a compile-time error if any `TypeName` denotes the interface `I`, or an interface that is a superinterface of `I`(9.1.3).

If a sealed interface `I` does not have an explicit `permits` clause, then it has an implicitly declared `permits` clause that lists all the classes and interfaces in the same compilation unit (7.3) as `I` that declare `I` as their direct superinterface.

It is a compile-time error if a sealed interface has an empty `permits` clause.

It is a compile-time error if any permitted subtype of a sealed interface `I` is not a member of the same module as `I`. If the sealed interface `I` is a member of the unnamed module, then it is a compile-time error if any permitted subtype is not in the same package as the interface `I`.

It is a compile-time error if any permitted subtype of a sealed interface `I` does not declare `I` as a direct superinterface.

It is a compile-time error if an interface declaration has an explicit `permits` clause but is not sealed.

**Chapter 13: Binary Compatibility**

**13.4 Evolution of Classes**

**13.4.2 final Classes sealed, non-sealed and final Classes**

This subsection has been renamed. Existing references to 13.4.2 should be renumbered to 13.4.2.3.

**13.4.2.1 sealed Classes**

If a class that was not declared `sealed` is changed to be declared `sealed`, then a `VerifyError` is thrown if a binary of a pre-existing subclass of this class is loaded that is not contained in its `permits` clause; such a change is not recommended for widely distributed classes.

Changing a class that is declared `sealed` to no longer be declared `sealed` does not break compatibility with pre-existing binaries.

**13.4.2.2 non-sealed Classes**

Changing a class that is not declared `non-sealed` to be declared `non-sealed` does not break compatibility with pre-existing binaries.

Changing a class that is declared `non-sealed` to no longer be declared `non-sealed` does not, in itself, break compatibility with pre-existing binaries.

Although removing a `non-sealed` modifier from a class declaration does not break compatibility with pre-existing binaries, it is highly likely to not compile successfully.

**sealed interface I permits C { }**

**non-sealed class C implements I { }**

Suppose that a new version of `C` is produced:
Whilst this is compatible with the existing binary for \( I \), recompiling \( C \) but not \( I \) results in an error as \( C \) implements a sealed interface but is neither sealed, non-sealed nor final (8.1.1.2).

13.4.2.3 final Classes

If a class that was not declared final is changed to be declared final, then a VerifyError is thrown if a binary of a pre-existing subclass of this class is loaded, because final classes can have no subclasses; such a change is not recommended for widely distributed classes.

Changing a class that is declared final to no longer be declared final does not break compatibility with pre-existing binaries.

13.4.4 Superclasses and Superinterfaces

Supersclasses, Superinterfaces and Permitted Subtypes

This subsection has been renamed.

A ClassCircularityError is thrown at load time if a class would be a superclass of itself. Changes to the class hierarchy that could result in such a circularity when newly compiled binaries are loaded with pre-existing binaries are not recommended for widely distributed classes.

Changing the direct superclass or the set of direct superinterfaces of a class type will not break compatibility with pre-existing binaries, provided that the total set of superclasses or superinterfaces, respectively, of the class type loses no members.

If a change to the direct superclass or the set of direct superinterfaces results in any class or interface no longer being a superclass or superinterface, respectively, then linkage errors may result if pre-existing binaries are loaded with the binary of the modified class. Such changes are not recommended for widely distributed classes.

Example 13.4.4-1. Changing A Superclass

Suppose that the following test program:

```java
class Hyper { char h = 'h'; }
class Super extends Hyper { char s = 's'; }
class Test extends Super {
    public static void printH(Hyper h) {
        System.out.println(h.h);
    }
    public static void main(String[] args) {
        printH(new Super());
    }
}
```

is compiled and executed, producing the output:

```
h
```

Suppose that a new version of class Super is then compiled:

```java
class Super { char s = 's'; }
```
This version of class Super is not a subclass of Hyper. If we then run the existing binaries of Hyper and Test with the new version of Super, then a VerifyError is thrown at link time. The verifier objects because the result of new Super() cannot be passed as an argument in place of a formal parameter of type Hyper, because Super is not a subclass of Hyper.

It is instructive to consider what might happen without the verification step: the program might run and print:

```
s
```

This demonstrates that without the verifier, the Java type system could be defeated by linking inconsistent binary files, even though each was produced by a correct Java compiler.

The lesson is that an implementation that lacks a verifier or fails to use it will not maintain type safety and is, therefore, not a valid implementation.

The requirement that alternatives in a multi-catch clause (14.20) not be subclasses or superclasses of each other is only a source restriction. Assuming the following client code is legal:

```java
try {
    throwAorB();
} catch(ExceptionA | ExceptionB e) {
    ...
}
```

where ExceptionA and ExceptionB do not have a subclass/superclass relationship when the client is compiled, it is binary compatible with respect to the client for ExceptionA and ExceptionB to have such a relationship when the client is executed.

This is analogous to other situations where a class transformation that is binary compatible for a client might not be source compatible for the same client.

Changing the set of permitted subclasses of a class (8.1.6) or permitted subtypes of an interface (9.1.4) may result in linkage errors if pre-existing binaries are loaded with the binary of the modified class or interface. Such changes are not recommended for widely distributed classes or interfaces.

Chapter 14: Blocks and Statements

14.3 Local Class Declarations

A local class is a nested class (8) that is not a member of any class and that has a name (6.2, 6.7).

All local classes are inner classes (8.1.3).

Every local class declaration statement is immediately contained by a block (14.2). Local class declaration statements may be intermixed freely with other kinds of statements in the block.

It is a compile-time error if a local class declaration contains any of the access modifiers public, protected, or private (6.6), or any of the modifier modifiers static (8.1.1), sealed or non-sealed (8.1.1.2).

The scope and shadowing of a local class declaration is specified in 6.3 and 6.4.
Example 14.3-1. Local Class Declarations

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

```java
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 15: Expressions

15.9 Class Instance Creation Expressions

15.9.5 Anonymous Class Declarations

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

An anonymous class is never abstract (8.1.1.9).
An anonymous class is never final (8.1.1.2).

An anonymous class is never sealed (8.1.1.2).

The fact that an anonymous class is not final is relevant in casting, in particular the narrowing reference conversion allowed for the cast operator (5.5). It is also of interest in subclassing, in that it is impossible to declare a subclass of an anonymous class, despite an anonymous class being non-final, because an anonymous class cannot be named by an extends clause (8.1.4).

An anonymous class is always an inner class (8.1.3); it is never static (8.1.1, 8.5.1),

The superclass or superinterface of an anonymous class is given by the class instance creation expression (15.9.1), with type arguments inferred as necessary while choosing a constructor (15.9.3).

It is a compile-time error if the superclass or superinterface of an anonymous class is sealed (8.1.1.2, 9.1.1.3).

If the class instance creation expression uses <> with an anonymous class, then for all non-private methods declared in the anonymous class body, it is as if the method declaration is annotated with @Override (9.6.4.4).

When <> is used, the inferred type arguments may not be as anticipated by the programmer. Consequently, the supertype of the anonymous class may not be as anticipated, and methods declared in the anonymous class may not override supertype methods as intended. Treating such methods as if annotated with @Override (if they are not explicitly annotated with @Override) helps avoid silently incorrect programs.

## 15.27 Lambda Expressions

### 15.27.3 Type of a Lambda Expression

A lambda expression is compatible in an assignment context, invocation context, or casting context with a target type \( T \) if \( T \) is a functional interface type (9.8) and the expression is congruent with the function type of the ground target type derived from \( T \).

The ground target type is derived from \( T \) as follows:

- If \( T \) is a wildcard-parameterized functional interface type and the lambda expression is explicitly typed, then the ground target type is inferred as described in 18.5.3.
- If \( T \) is a wildcard-parameterized functional interface type and the lambda expression is implicitly typed, then the ground target type is the non-wildcard parameterization (9.9) of \( T \).
- Otherwise, the ground target type is \( T \).

A lambda expression is congruent with a function type if all of the following are true:

- The function type has no type parameters.
- The number of lambda parameters is the same as the number of parameter types of the function type.
- If the lambda expression is explicitly typed, its formal parameter types are the same as the parameter types of the function type.
If the lambda parameters are assumed to have the same types as the function type's parameter types, then:

- If the function type's result is void, the lambda body is either a statement expression (14.8) or a void-compatible block.

- If the function type's result is a (non-void) type $R$, then either
  1. the lambda body is an expression that is compatible with $R$ in an assignment context, or
  2. the lambda body is a value-compatible block, and each result expression (15.27.2) is compatible with $R$ in an assignment context.

If a lambda expression is compatible with a target type $T$, then the type of the expression, $U$, is the ground target type derived from $T$.

It is a compile-time error if any class or interface mentioned by either $U$ or the function type of $U$ is not accessible (6.6) from the class or interface in which the lambda expression appears.

It is a compile-time error if the target type $T$ is sealed (9.1.1.3).

For each non-static member method $m$ of $U$, if the function type of $U$ has a subsignature of the signature of $m$, then a notional method whose method type is the function type of $U$ is deemed to override $m$, and any compile-time error or unchecked warning specified in 8.4.8.3 may occur.

A checked exception that can be thrown in the body of the lambda expression may cause a compile-time error, as specified in 11.2.3.

The parameter types of explicitly typed lambdas are required to exactly match those of the function type. While it would be possible to be more flexible - allow boxing or contravariance, for example - this kind of generality seems unnecessary, and is inconsistent with the way overriding works in class declarations. A programmer ought to know exactly what function type is being targeted when writing a lambda expression, so he should thus know exactly what signature must be overridden. (In contrast, this is not the case for method references, and so more flexibility is allowed when they are used.) In addition, more flexibility with parameter types would add to the complexity of type inference and overload resolution.

Note that while boxing is not allowed in a strict invocation context, boxing of lambda result expressions is always allowed - that is, the result expression appears in an assignment context, regardless of the context enclosing the lambda expression. However, if an explicitly typed lambda expression is an argument to an overloaded method, a method signature that avoids boxing or unboxing the lambda result is preferred by the most specific check (15.12.2.5).

If the body of a lambda is a statement expression (that is, an expression that would be allowed to stand alone as a statement), it is compatible with a void-producing function type; any result is simply discarded. So, for example, both of the following are legal:

```java
// Predicate has a `boolean` result
java.util.function.Predicate<String> p = s -> list.add(s);
// Consumer has a `void` result
java.util.function.Consumer<String> c = s -> list.add(s);
```

Generally speaking, a lambda of the form ( ) -> expr, where expr is a statement expression, is interpreted as either ( ) -> { return expr; } or ( ) -> { expr; }, depending on the target type.
Sealed Types

Copyright © 1993, 2020, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA. All rights reserved. Use is subject to license terms and the documentation redistribution policy.

DRAFT 14-internal+0-adhoc.gbierman.20200228