Unnamed Variables and Patterns

Changes to the Java® Language Specification • Version 22-internal-adhoc.abimpoudis.20231128

This document describes changes to the Java Language Specification to support unnamed patterns and variables, which is a feature of Java SE 22. See JEP:456 for overview of the feature.

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

Changelog:

2023-11-28: Add missing form for lambda parameters with _.

2023-10-17: Update description and links. JLS changes are the same as in JLS 443.

2023-03-22: Initial spec draft.

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 (3.9 ⇗).

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.

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 (`Α`, `\u0391`), CYRILLIC SMALL LETTER A (`а`, `\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`*

An identifier never has the same spelling (Unicode character sequence) as a reserved keyword (3.9 ⇗), a boolean literal (3.10.3 ⇗) or the null literal (3.10.8 ⇗), due to the rules of tokenization (3.5 ⇗). However, an identifier may have the same spelling as a contextual keyword, because the tokenization of a sequence of input characters as an identifier or a contextual keyword depends on where the sequence appears in the program.

To facilitate the recognition of contextual keywords, the syntactic grammar (2.3 ⇗) sometimes disallows certain identifiers by defining a production to accept only a subset of identifiers. The

subsets are as follows:

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

UnqualifiedMethodIdentifier:
Identifier but not `yield`

TypeIdentifier is used in the declaration of classes, interfaces, and type parameters (8.1 ⇗, 9.1 ⇗, 4.4 ⇗), and when referring to types (6.5 ⇗). For example, the name of a class must be a TypeIdentifier, so it is illegal to declare a class named `permits`, `record`, `sealed`, `var`, *or* `yield`.

UnqualifiedMethodIdentifier is used when a method invocation expression refers to a method by its simple name (6.5.7.1 ⇗). Since the term `yield` *is excluded from UnqualifiedMethodIdentifier, any invocation of a method named* `yield` *must be qualified, thus distinguishing the invocation from a* `yield` *statement (14.21 ⇗).*

3.9 Keywords

51 character sequences, formed from ASCII characters, are reserved for use as keywords and cannot be used as identifiers (3.8 ⇗). Another 16 character sequences, also formed from ASCII characters, may be interpreted as keywords or as other tokens, depending on the context in which they appear.

Keyword:
ReservedKeyword
ContextualKeyword

ReservedKeyword:
(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)

ContextualKeyword:
(one of)

```
exports    permits  sealed     var
module     provides to         when
non-sealed record   transitive with
open       requires uses       yield
opens
```

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 `strictfp` *is obsolete and should not be used in new code.*

~~*The keyword _ (underscore) is reserved for possible future use in parameter declarations.*~~

The keyword _ (underscore) may be used in declarations in place of an identifier (6.1).

`true` *and* `false` *are not keywords, but rather boolean literals (3.10.3 ⇗).*

`null` is not a keyword, but rather the null literal (3.10.8 ⇗).

During the reduction of input characters to input elements (3.5 ⇗), a sequence of input characters that notionally matches a contextual keyword is reduced to a contextual keyword if and only if both of the following conditions hold:

1. The sequence is recognized as a terminal specified in a suitable context of the syntactic grammar (2.3 ⇗), as follows:
    - For `module` and `open`, when recognized as a terminal in a *ModuleDeclaration* (7.7 ⇗).
    - For `exports`, `opens`, `provides`, `requires`, `to`, `uses`, and `with`, when recognized as a terminal in a *ModuleDirective*.
    - For `transitive`, when recognized as a terminal in a *RequiresModifier*.

        For example, recognizing the sequence `requires transitive ;` does not make use of RequiresModifier, so the term `transitive` is reduced here to an identifier and not a contextual keyword.

    - For `var`, when recognized as a terminal in a *LocalVariableType* (14.4) or a *LambdaParameterType* (15.27.1).

        In other contexts, attempting to use `var` as an identifier will cause an error, because `var` is not a TypeIdentifier (3.8 ⇗).

    - For `yield`, when recognized as a terminal in a *YieldStatement* (14.21 ⇗).

        In other contexts, attempting to use the `yield` as an identifier will cause an error, because `yield` is neither a TypeIdentifier nor a UnqualifiedMethodIdentifier.

    - For `record`, when recognized as a terminal in a *RecordDeclaration* (8.10).
    - For `non-sealed`, `permits`, and `sealed`, when recognized as a terminal in a *NormalClassDeclaration* (8.1 ⇗) or a *NormalInterfaceDeclaration* (9.1 ⇗).
    - For `when`, when recognized as a terminal in a *Guard* (14.11.1).
2. The sequence is not immediately preceded or immediately followed by an input character that matches *JavaLetterOrDigit*.

In general, accidentally omitting white space in source code will cause a sequence of input characters to be tokenized as an identifier, due to the "longest possible translation" rule (3.2 ⇗). For example, the sequence of twelve input characters `p u b l i c s t a t i c` is always tokenized as the identifier `publicstatic`, rather than as the reserved keywords `public` and `static`. If two tokens are intended, they must be separated by white space or a comment.

The rule above works in tandem with the "longest possible translation" rule to produce an intuitive result in contexts where contextual keywords may appear. For example, the sequence of eleven input characters `v a r f i l e n a m e` is usually tokenized as the identifier `varfilename`, but in a local variable declaration, the first three input characters are tentatively recognized as the contextual keyword `var` by the first condition of the rule above. However, it would be confusing to overlook the lack of white space in the sequence by recognizing the next eight input characters as the identifier `filename`. (This would mean that the sequence undergoes different tokenization in different contexts: an identifier in most contexts, but a contextual keyword and an identifier in local variable declarations.) Accordingly, the second condition prevents recognition of the contextual keyword `var` on the grounds that the immediately following input character `f` is a JavaLetterOrDigit. The sequence `v a r f i l e n a m e` is therefore tokenized as the identifier `varfilename` in a local variable declaration.

As another example of the careful recognition of contextual keywords, consider the sequence of 15 input characters `n o n - s e a l e d c l a s s`. This sequence is usually translated to three tokens - the identifier `non`, the operator `-`, and the identifier `sealedclass` - but in a normal class declaration, where the first condition holds, the first ten input characters are tentatively recognized as the contextual keyword `non-sealed`. To avoid translating the sequence to two keyword tokens (`non-sealed` and `class`) rather than three non-keyword tokens, and to avoid rewarding the programmer for omitting white space before

`class`, the second condition prevents recognition of the contextual keyword. The sequence `non-sealed class` is therefore tokenized as three tokens in a class declaration.

In the rule above, the first condition depends on details of the syntactic grammar, but a compiler for the Java programming language can implement the rule without fully parsing the input program. For example, a heuristic could be used to track the contextual state of the tokenizer, as long as the heuristic guarantees that valid uses of contextual keywords are tokenized as keywords, and valid uses of identifiers are tokenized as identifiers. Alternatively, a compiler could always tokenize a contextual keyword as an identifier, leaving it to a later phase to recognize special uses of these identifiers.

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 avoid certain contextual keywords when the entity being introduced is a class, interface, or type parameter.~~

~~A declared entity is one of the following:~~

A *declaration* introduces an entity into a program, 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 by a normal class declaration (8.1 ⇗), an enum declaration (8.9 ⇗), or a record declaration (8.10)
- An interface, declared by a normal interface declaration (9.1 ⇗) or an annotation interface declaration (9.6 ⇗).
- A type parameter, declared as part of the declaration of a generic class, interface, method, or constructor (8.1.2 ⇗, 9.1.2 ⇗, 8.4.4 ⇗, 8.8.4 ⇗)
- A member of a reference type (8.2 ⇗, 9.2 ⇗, 8.9.3 ⇗, 9.6 ⇗, 10.7 ⇗), one of the following:
  - A member class (8.5 ⇗, 9.5 ⇗)
  - A member interface (8.5 ⇗, 9.5 ⇗)
  - A field, one of the following:
    - A field declared in a class (8.3)
    - A field declared in an interface (9.3)
    - An implicitly declared field of a class corresponding to an enum constant or a record component
    - The field `length`, which is implicitly a member of every array type (10.7 ⇗)
  - A method, one of the following:
    - A method (`abstract` or otherwise) declared in a class (8.4 ⇗)
    - A method (`abstract` or otherwise) declared in an interface (9.4 ⇗)
    - An implicitly declared accessor method corresponding to a record component
- An enum constant (8.9.1 ⇗)
- A record component (8.10.3 ⇗)
- A formal parameter, one of the following:

    - A formal parameter of a method of a class or interface (8.4.1)
    - A formal parameter of a constructor of a class (8.8.1 ⇗)
    - A formal parameter of a lambda expression (15.27.1)
- An exception parameter of an exception handler declared in a `catch` clause of a `try` statement (14.20)
- A local variable, one of the following:
    - A local variable declared by a local variable declaration statement in a block (14.4.2 ⇗)
    - A local variable declared by a `for` statement or a `try`-with-resources statement (14.14, 14.20.3)
    - A local variable declared by a pattern (14.30.1)
- A local class or interface (14.3 ⇗), one of the following:
    - A local class declared by a normal class declaration
    - A local class declared by an enum declaration
    - A local class declared by an record declaration
    - A local interface declared by a normal interface declaration

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

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

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

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

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

> *The declaration of a generic class or interface (`class` C<T> ... or `interface` C<T> ...) introduces both a class named C and a family of types: the raw type C, the parameterized type C<Foo>, the parameterized type C<Bar>, etc.*
>
> *When a reference to C occurs where genericity is unimportant, identified below as one of the non-generic contexts, the reference to C denotes the class or interface C. In other contexts, the reference to C denotes a type, or part of a type, introduced by C.*
>
> *The 15 non-generic contexts are as follows:*
>
> 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. *In a `permits` clause of a `sealed` class or interface declaration (8.1.6, 9.1.4).*
6. *To the left of the `(` in a constructor declaration (8.8)*
7. *After the `@` sign in an annotation (9.7)*
8. *To the left of `.class` in a class literal (15.8.2)*
9. *To the left of `.this` in a qualified `this` expression (15.8.4)*
10. *To the left of `.super` in a qualified superclass field access expression (15.11.2)*
11. *To the left of .Identifier or `.super`.Identifier in a qualified method invocation expression (15.12)*
12. *To the left of `.super::` in a method reference expression (15.13)*
13. *In a qualified expression name in a postfix expression or a `try`-with-resources statement (15.14.1, 14.20.3)*
14. *In a `throws` clause of a method or constructor (8.4.6, 8.8.5, 9.4)*
15. *In an exception parameter declaration (14.20)*

The first twelve non-generic contexts correspond to the first twelve syntactic contexts for a TypeName in [6.5.1]. The thirteenth non-generic context is where a qualified ExpressionName such as `C.x` may include a TypeName `C` to denote static member access. The common use of TypeName in these thirteen contexts is significant: it indicates that these contexts involve a less-than-first-class use of a type. In contrast, the fourteenth and fifteenth non-generic contexts employ ClassType, indicating that `throws` and `catch` clauses use types in a first-class way, in line with, for example, field declarations. The characterization of these two contexts as non-generic is due to the fact that an exception type cannot be parameterized (8.1.2).

Note that the ClassType production allows annotations, so it is possible to annotate the use of a type in a `throws` or `catch` clause, whereas the TypeName production disallows annotations, so it is not possible to annotate the name of a type in, for example, a single-type-import declaration.

Naming Conventions

The class libraries of the Java SE Platform attempt to use, whenever possible, names chosen according to the conventions presented below. These conventions help to make code more readable and avoid certain kinds of name conflicts.

We recommend these conventions for use in all programs written in the Java programming language. However, these conventions should not be followed slavishly if long-held conventional usage dictates otherwise. So, for example, the `sin` and `cos` methods of the class `java.lang.Math` have mathematically conventional names, even though these method names flout the convention suggested here because they are short and are not verbs.

Package Names and Module Names

Programmers should take steps to avoid the possibility of two published packages having the same name by choosing unique package names for packages that are widely distributed. This allows packages to be easily and automatically installed and catalogued. This section specifies a suggested convention for generating such unique package names. Implementations of the Java SE Platform are encouraged to provide automatic support for converting a set of packages from local and casual package names to the unique name format described here.

If unique package names are not used, then package name conflicts may arise far from the point of creation of either of the conflicting packages. This may create a situation that is difficult or impossible for the user or programmer to resolve. The classes `ClassLoader` and `ModuleLayer` can be used to isolate packages with the same name from each other in those cases where the packages will have constrained interactions, but not in a way that is transparent to a naïve program.

You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as `oracle.com`. You then reverse this name, component by component, to obtain, in this example, `com.oracle`, and use this as a prefix for your package names, using a convention developed within your organization to further administer package names. Such a convention might specify that certain package name components be division, department, project, machine, or login names.

Example 6.1-1. Unique Package Names

```
com.nighthacks.scrabble.dictionary
org.openjdk.compiler.source.tree
net.jcip.annotations
edu.cmu.cs.bovik.cheese
gov.whitehouse.socks.mousefinder
```

The first component of a unique package name is always written in all-lowercase ASCII letters and should be one of the top level domain names, such as `com`, `edu`, `gov`, `mil`, `net`, or `org`, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166.

In some cases, the Internet domain name may not be a valid package name. Here are some suggested conventions for dealing with these situations:

- *If the domain name contains a hyphen, or any other special character not allowed in an identifier (3.8 ⇗), convert it into an underscore.*
- *If any of the resulting package name components are keywords (3.9 ⇗), append an underscore to them.*
- *If any of the resulting package name components start with a digit, or any other character that is not allowed as an initial character of an identifier, have an underscore prefixed to the component.*

The name of a module should correspond to the name of its principal exported package. If a module does not have such a package, or if for legacy reasons it must have a name that does not correspond to one of its exported packages, then its name should still start with the reversed form of an Internet domain with which its author is associated.

Example 6.1-2. Unique Module Names

```
com.nighthacks.scrabble
org.openjdk.compiler
net.jcip.annotations
```

The first component of a package or module name must not be the identifier `java`. Package and module names that start with the identifier `java` are reserved for packages and modules of the Java SE Platform.

The name of a package or module is not meant to imply where the package or module is stored on the Internet. For example, a package named `edu.cmu.cs.bovik.cheese` is not necessarily obtainable from the host `cmu.edu` or `cs.cmu.edu` or `bovik.cs.cmu.edu`. The suggested convention for generating unique package and module names is merely a way to piggyback a package and module naming convention on top of an existing, widely known unique name registry instead of having to create a separate registry for package and module names.

Class and Interface Names

Names of class should be descriptive nouns or noun phrases, not overly long, in mixed case with the first letter of each word capitalized.

Example 6.1-3. Descriptive Class Names

```
`ClassLoader`
SecurityManager
`Thread`
Dictionary
BufferedInputStream
```

Likewise, names of interface should be short and descriptive, not overly long, in mixed case with the first letter of each word capitalized. The name may be a descriptive noun or noun phrase, which is appropriate when an interface is used as if it were an abstract superclass, such as interfaces `java.io.DataInput` and `java.io.DataOutput`; or it may be an adjective describing a behavior, as for the interfaces `Runnable` and `Cloneable`.

Type Variable Names

Type variable names should be pithy (single character if possible) yet evocative, and should not include lower case letters. This makes it easy to distinguish type parameters from ordinary classes and interfaces.

Container classes and interfaces should use the name `E` for their element type. Maps should use `K` for the type of their keys and `V` for the type of their values. The name `X` should be used for arbitrary exception types. We use `T` for type, whenever there is not anything more specific about the type to distinguish it. (This is often the case in generic methods.)

If there are multiple type parameters that denote arbitrary types, one should use letters that neighbor `T` in the alphabet, such as `S`. Alternately, it is acceptable to use numeric subscripts (e.g., `T1`, `T2`) to distinguish among the different type variables. In such cases, all the variables with the same prefix should be subscripted.

If a generic method appears inside a generic class, it is a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

Example 6.1-4. Conventional Type Variable Names

```
public class HashSet<E> extends AbstractSet<E> { ... }
public class HashMap<K,V> extends AbstractMap<K,V> { ... }
public class ThreadLocal<T> { ... }
public interface Functor<T, X extends Throwable> {
    T eval() throws X;
}
```

When type parameters do not fall conveniently into one of the categories mentioned, names should be chosen to be as meaningful as possible within the confines of a single letter. The names mentioned above (`E`, `K`, `V`, `X`, `T`) should not be used for type parameters that do not fall into the designated categories.

Method Names

Method names should be verbs or verb phrases, in mixed case, with the first letter lowercase and the first letter of any subsequent words capitalized. Here are some additional specific conventions for method names:

- *Methods to get and set an attribute that might be thought of as a variable V should be named `getV` and `setV`. An example is the methods `getPriority` and `setPriority` of class `Thread`.*
- *A method that returns the length of something should be named `length`, as in class `String`.*
- *A method that tests a boolean condition V about an object should be named `isV`. An example is the method `isInterrupted` of class `Thread`.*
- *A method that converts its object to a particular format F should be named `toF`. Examples are the method `toString` of class `Object` and the methods `toLocaleString` and `toGMTString` of class `java.util.Date`.*

Whenever possible and appropriate, basing the names of methods in a new class on names in an existing class that is similar, especially a class from the Java SE Platform API, will make it easier to use.

Field Names

Names of fields that are not `final` should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized. Note that well-designed classes have very few `public` or `protected` fields, except for fields that are constants (`static final` fields).

Fields should have names that are nouns, noun phrases, or abbreviations for nouns.

Examples of this convention are the fields `buf`, `pos`, and `count` of the class `java.io.ByteArrayInputStream` and the field `bytesTransferred` of the class `java.io.InterruptedIOException`.

Constant Names

The names of constants in interfaces should be, and `final` variables of classes may conventionally be, a sequence of one or more words, acronyms, or abbreviations, all uppercase, with components separated by underscore "_" characters. Constant names should be descriptive and not unnecessarily abbreviated. Conventionally they may be any appropriate part of speech.

Examples of names for constants include `MIN_VALUE`, `MAX_VALUE`, `MIN_RADIX`, and `MAX_RADIX` of the class `Character`.

A group of constants that represent alternative values of a set, or, less frequently, masking bits in an integer value, are sometimes usefully specified with a common acronym as a name prefix.

For example:

```
interface ProcessStates {
    int PS_RUNNING   = 0;
    int PS_SUSPENDED = 1;
}
```

Local Variable and Parameter Names

Local variable and parameter names should be short, yet meaningful. They are often short sequences of lowercase letters that are not words, such as:

- *Acronyms, that is the first letter of a series of words, as in `cp` for a variable holding a reference to a `ColoredPoint`*
- *Abbreviations, as in `buf` holding a pointer to a buffer of some kind*
- *Mnemonic terms, organized in some way to aid memory and understanding, typically by using a set of local variables with conventional names patterned after the names of parameters to widely used classes. For example:*
    - *`in` and `out`, whenever some kind of input and output are involved, patterned after the fields of `System`*
    - *`off` and `len`, whenever an offset and length are involved, patterned after the parameters to the `read` and `write` methods of the interfaces `DataInput` and `DataOutput` of `java.io`*

One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type. Conventional one-character names are:

- *`b` for a `byte`*
- *`c` for a `char`*
- *`d` for a `double`*
- *`e` for an `Exception`*
- *`f` for a `float`*
- *`i`, `j`, and `k` for `int`s*
- *`l` for a `long`*
- *`o` for an `Object`*
- *`s` for a `String`*
- *`v` for an arbitrary value of some type*

Local variable or parameter names that consist of only two or three lowercase letters should not conflict with the initial country codes and domain names that are the first component of unique package names.

Chapter 8: Classes

8.3 Field Declarations

The variables of a class are introduced by *field declarations*.

FieldDeclaration:
{*FieldModifier*} *UnannType VariableDeclaratorList* ;

VariableDeclaratorList:
VariableDeclarator {, *VariableDeclarator*}

VariableDeclarator:
VariableDeclaratorId [= *VariableInitializer*]

~~*VariableDeclaratorId:*~~
~~*Identifier* [*Dims*]~~

VariableDeclaratorId:
Identifier [*Dims*]
=

VariableInitializer:
Expression
ArrayInitializer

UnannType:
UnannPrimitiveType
UnannReferenceType

UnannPrimitiveType:
NumericType
`boolean`

UnannReferenceType:
UnannClassOrInterfaceType
UnannTypeVariable
UnannArrayType

UnannClassOrInterfaceType:
UnannClassType
UnannInterfaceType

UnannClassType:
TypeIdentifier [*TypeArguments*]
PackageName . {*Annotation*} *TypeIdentifier* [*TypeArguments*]
UnannClassOrInterfaceType . {*Annotation*} *TypeIdentifier* [*TypeArguments*]

UnannInterfaceType:
UnannClassType

UnannTypeVariable:
TypeIdentifier

UnannArrayType:
UnannPrimitiveType Dims
UnannClassOrInterfaceType Dims
UnannTypeVariable Dims

The following production from 4.3 is shown here for convenience:

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

Each declarator in a *FieldDeclaration* declares one field. The declarator must include an *Identifier*, or a compile-time error occurs. The *Identifier* ~~in a declarator~~ may be used in a name to refer to the field.

More than one field may be declared in a single *FieldDeclaration* by using more than one declarator; the *FieldModifier*s and *UnannType* apply to all the declarators in the declaration.

The *FieldModifier* clause is described in [8.3.1].

The declared type of a field is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and is specified by 10.2 otherwise.

The scope and shadowing of a field declaration is specified in 6.3 and 6.4.1.

It is a compile-time error for the body of a class declaration to declare two fields with the same name.

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

> *In this respect, hiding of fields differs from hiding of methods (8.4.8.3 ⇗), for there is no distinction drawn between `static` and non-`static` fields in field hiding whereas a distinction is drawn between `static` and non-`static` methods in method hiding.*

A hidden field can be accessed by using a qualified name (6.5.6.2 ⇗) if it is `static`, or by using a field access expression that contains the keyword `super` (15.11.2 ⇗) or a cast to a superclass type.

> *In this respect, hiding of fields is similar to hiding of methods.*

If a field declaration hides the declaration of another field, the two fields need not have the same type.

A class inherits from its direct superclass and direct superinterfaces all the non-`private` fields of the superclass and superinterfaces that are both accessible (6.6 ⇗) to code in the class and not hidden by a declaration in the class.

A `private` field of a superclass might be accessible to a subclass - for example, if both classes are members of the same class. Nevertheless, a `private` field is never inherited by a subclass.

It is possible for a class to inherit more than one field 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 field 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 field declaration is inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

> **Example 8.3-1. Multiply Inherited Fields**
>
> *A class may inherit two or more fields with the same name, either from its superclass and a superinterface or from two superinterfaces. A compile-time error occurs on any attempt to refer to any ambiguously inherited field by its simple name. A qualified name or a field access expression that contains the keyword `super` (15.11.2 ⇗) may be used to access such fields unambiguously. In the program:*
>
> ```
> interface Frob  { float v = 2.0f; }
> class SuperTest { int   v = 3; }
> class Test extends SuperTest implements Frob {
>     public static void main(String[] args) {
>         new Test().printV();
>     }
>     void printV() { System.out.println(v); }
> }
> ```
>
> *the class `Test` inherits two fields named `v`, one from its superclass `SuperTest` and one from its superinterface `Frob`. This in itself is permitted, but a compile-time error occurs because of the use of the simple name `v` in method `printV`: it cannot be determined which `v` is intended.*
>
> *The following variation uses the field access expression `super.v` to refer to the field named `v` declared in class `SuperTest` and uses the qualified name `Frob.v` to refer to the field named `v` declared in interface `Frob`:*
>
> ```
> interface Frob  { float v = 2.0f; }
> class SuperTest { int   v = 3; }
> class Test extends SuperTest implements Frob {
>     public static void main(String[] args) {
>         new Test().printV();
>     }
>     void printV() {
>         System.out.println((super.v + Frob.v)/2);
> ```

```
    }
}
```

It compiles and prints:

```
2.5
```

Even if two distinct inherited fields have the same type, the same value, and are both `final`, any reference to either field by simple name is considered ambiguous and results in a compile-time error. In the program:

```
interface Color        { int RED=0, GREEN=1,  BLUE=2;  }
interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }
class Test implements Color, TrafficLight {
    public static void main(String[] args) {
        System.out.println(GREEN);  // compile-time error
        System.out.println(RED);    // compile-time error
    }
}
```

it is not astonishing that the reference to `GREEN` should be considered ambiguous, because class `Test` inherits two different declarations for `GREEN` with different values. The point of this example is that the reference to `RED` is also considered ambiguous, because two distinct declarations are inherited. The fact that the two fields named `RED` happen to have the same type and the same unchanging value does not affect this judgment.

Example 8.3-2. Re-inheritance of Fields

If the same field declaration is inherited from an interface by multiple paths, the field is considered to be inherited only once. It may be referred to by its simple name without ambiguity. For example, in the code:

```
interface Colorable {
    int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
interface Paintable extends Colorable {
    int MATTE = 0, GLOSSY = 1;
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {}
class PaintedPoint extends ColoredPoint implements Paintable {
    int p = RED;
}
```

the fields `RED`, `GREEN`, and `BLUE` are inherited by the class `PaintedPoint` both through its direct superclass `ColoredPoint` and through its direct superinterface `Paintable`. The simple names `RED`, `GREEN`, and `BLUE` may nevertheless be used without ambiguity within the class `PaintedPoint` to refer to the fields declared in interface `Colorable`.

8.4 Method Declarations

8.4.1 Formal Parameters

The *formal parameters* of a method or constructor, if any, are specified by a list of comma-separated parameter specifiers. Each parameter specifier consists of a type (optionally preceded by the `final` modifier and/or one or more annotations) and an identifier (optionally followed by brackets) that specifies the name of the parameter.

If a method or constructor has no formal parameters, and no receiver parameter, then an empty pair of parentheses appears in the declaration of the method or constructor.

FormalParameterList:
  FormalParameter {`,` *FormalParameter*}

FormalParameter:

{VariableModifier} UnannType VariableDeclaratorId
VariableArityParameter

VariableArityParameter:
{VariableModifier} UnannType {Annotation} ... Identifier

VariableModifier:
Annotation
`final`

The following productions from 8.3 and 4.3 are shown here for convenience:

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

A formal parameter of a method or constructor may be a *variable arity parameter*, indicated by an ellipsis following the type. At most one variable arity parameter is permitted for a method or constructor. It is a compile-time error if a variable arity parameter appears anywhere in the list of parameter specifiers except the last position.

In the grammar for VariableArityParameter, note that the ellipsis (...) is a token unto itself (3.11). It is possible to put whitespace between it and the type, but this is discouraged as a matter of style.

If the last formal parameter of a method is a variable arity parameter, the method is a *variable arity method*. Otherwise, it is a *fixed arity method*.

The rules concerning annotation modifiers for a formal parameter declaration and for a receiver parameter are specified in 9.7.4 and 9.7.5.

It is a compile-time error if `final` appears more than once as a modifier for a formal parameter declaration.

The scope and shadowing of a formal parameter is specified in 6.3 and 6.4.

References to a formal parameter from a nested class or interface, or a lambda expression, are restricted, as specified in 6.5.6.1.

Every declaration of a formal parameter of a method or constructor must include an *Identifier*, otherwise a compile-time error occurs.

It is a compile-time error for a method or constructor to declare two formal parameters with the same name. (That is, their declarations mention the same *Identifier*.)

It is a compile-time error if a formal parameter that is declared `final` is assigned to within the body of the method or constructor.

The declared type of a formal parameter depends on whether it is a variable arity parameter:

- If the formal parameter is not a variable arity parameter, then the declared type is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and specified by 10.2 otherwise.
- If the formal parameter is a variable arity parameter, then the declared type is an array type specified by 10.2.

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

When the method or constructor is invoked (15.12 ⇗), the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before execution of the body of the method or constructor. The *Identifier* that appears in the *FormalParameter* may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

Invocations of a variable arity method may contain more actual argument expressions than formal parameters. All the actual argument expressions that do not correspond to the formal parameters preceding the variable arity parameter will be evaluated and the results stored into an array that will be passed to the method invocation (15.12.4.2 ⇗).

Here are some examples of receiver parameters in instance methods and inner classes' constructors:

```
class Test {
    Test(/* ?? ?? */) {}
      // No receiver parameter is permitted in the constructor of
      // a top level class, as there is no conceivable type or name.

    void m(Test this) {}
      // OK: receiver parameter in an instance method

    static void n(Test this) {}
      // Illegal: receiver parameter in a static method

    class A {
        A(Test Test.this) {}
          // OK: the receiver parameter represents the instance
          // of Test which immediately encloses the instance
          // of A being constructed.

        void m(A this) {}
          // OK: the receiver parameter represents the instance
          // of A for which A.m() is invoked.

        class B {
            B(Test.A A.this) {}
              // OK: the receiver parameter represents the instance
              // of A which immediately encloses the instance of B
              // being constructed.

            void m(Test.A.B this) {}
              // OK: the receiver parameter represents the instance
              // of B for which B.m() is invoked.
        }
    }
}
```

`B`'s constructor and instance method show that the type of the receiver parameter may be denoted with a qualified TypeName like any other type; but that the name of the receiver parameter in an inner class's constructor must use the simple name of the enclosing class.

8.10 Record Classes

8.10.1 Record Components

The *record components* of a record class, if any, are specified in the header of a record declaration. 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. A record component corresponds to two members of the record class: a `private` field declared implicitly, and a `public` accessor method declared explicitly or implicitly (8.10.3 ⇗).

If a record class has no record components, then an empty pair of parentheses appears in the header of the record declaration.

RecordHeader:
([*RecordComponentList*])

RecordComponentList:
RecordComponent {, *RecordComponent*}

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

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

RecordComponentModifier:
Annotation

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 class. It is a compile-time error if a variable arity record component appears anywhere in the list of record components except the last position.

The rules concerning annotation modifiers for a record component declaration are specified in 9.7.4 and 9.7.5.

> *Annotations on a record component declaration are available via reflection if their annotation interfaces are applicable in the record component context (9.6.4.1). Independently, annotations on a record component declaration are propagated to the declarations of members and constructors of the record class if their annotation interfaces are applicable in other contexts (8.10.3, 8.10.4).*

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`.

> *These are the names of the no-args `public` and `protected` methods in `Object`. Disallowing them as the names of record components avoids confusion in a number of ways. First, every record class provides implementations of `hashCode` and `toString` that return representations of a record object as a whole; they cannot serve as accessor methods (8.10.3) for record components called `hashCode` or `toString`, and there would be no way to access such record components from outside the record class. Similarly, some record classes may provide implementations of `clone` and (regrettably) `finalize`, so a record component called `clone` or `finalize` could not be accessed via an accessor method. Finally, the `getClass`, `notify`, `notifyAll`, and `wait` methods in `Object` are `final`, so record components with the same names could not have accessor methods. (The accessor methods would have the same signatures as the `final` methods, and would thus attempt, unsuccessfully, to override them.)*

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

Every declaration of a record component of a record declaration must include an *Identifier*, otherwise a compile-time error occurs.

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).

Chapter 9: Interfaces

9.3 Field (Constant) Declarations

ConstantDeclaration:
{*ConstantModifier*} *UnannType VariableDeclaratorList* ;

ConstantModifier:
(one of)
Annotation `public`
`static final`

See 8.3 for UnannType. The following productions from 4.3 and 8.3 are shown here for convenience:

VariableDeclaratorList:
VariableDeclarator {, *VariableDeclarator*}

VariableDeclarator:
VariableDeclaratorId [= *VariableInitializer*]

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]

Dims:
{*Annotation*} `[ ]` {{*Annotation*} `[ ]`}

VariableInitializer:
Expression
ArrayInitializer

The rules concerning annotation modifiers for an interface field declaration are specified in 9.7.4 and 9.7.5.

Every field declaration in the body of an interface declaration is implicitly `public`, `static`, and `final`. It is permitted to redundantly specify any or all of these modifiers for such fields.

It is a compile-time error if the same keyword appears more than once as a modifier for a field declaration.

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

The declared type of a field is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and is specified by 10.2 otherwise.

Every declaration of a field must include an *Identifier*, or a compile-time error occurs.

The scope and shadowing of an interface field declaration is specified in 6.3 and 6.4.1.

Because an interface field is `static`, its declaration introduces a static context (8.1.3), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context (15.8.3, 15.11.2), as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations (6.5.5.1, 6.5.6.1, 15.12.3).

It is a compile-time error for the body of an interface declaration to declare two fields with the same name.

If the interface declares a field with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of fields with the same name in superinterfaces of the interface.

It is possible for an interface to inherit more than one field with the same name. Such a situation does not in itself cause a compile-time error. However, any attempt within the body of the interface declaration to refer to any such field 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 field declaration is inherited from an interface. In such a situation, the field is considered to be inherited only once, and it may be referred to by its simple name without ambiguity.

Example 9.3-1. Ambiguous Inherited Fields

If two fields with the same name are inherited by an interface because, for example, two of its direct superinterfaces declare fields with that name, then a single ambiguous member results. Any use of this ambiguous member will result in a compile-time error. In the program:

```
interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors {
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
interface PrintColors extends BaseColors {
    int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors {
    int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

the interface `LotsOfColors` inherits two fields named `YELLOW`. This is all right as long as the interface does not contain any reference by simple name to the field `YELLOW`. (Such a reference could occur within a variable initializer for a field.)

Even if interface `PrintColors` were to give the value `3` to `YELLOW` rather than the value `8`, a reference to field `YELLOW` within interface `LotsOfColors` would still be considered ambiguous.

Example 9.3-2. Multiply Inherited Fields

If a single field is inherited multiple times from the same interface because, for example, both this interface and one of this interface's direct superinterfaces extend the interface that declares the field, then only a single member results. This situation does not in itself cause a compile-time error.

In the previous example, the fields `RED`, `GREEN`, and `BLUE` are inherited by interface `LotsOfColors` in more than one way, through interface `RainbowColors` and also through interface `PrintColors`, but the reference to field `RED` in interface `LotsOfColors` is not considered ambiguous because only one actual declaration of the field `RED` is involved.

Chapter 14: Blocks, Statements, and Patterns

14.4 Local Variable Declarations

A *local variable declaration* declares and optionally initializes one or more local variables (4.12.3 ⇗).

LocalVariableDeclaration:
{*VariableModifier*} *LocalVariableType VariableDeclaratorList*

LocalVariableType:
UnannType
`var`

See 8.3 for UnannType. The following productions from 4.3 ⇗, 8.3, and 8.4.1 are shown here for convenience:

VariableModifier:
Annotation
`final`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} [ ] {{Annotation} [ ]}

VariableInitializer:
Expression
ArrayInitializer

A local variable declaration can appear in the following locations:

- a local variable declaration statement in a block (14.4.2 ⇗)
- the header of a basic `for` statement (14.14.1 ⇗)
- the header of an enhanced `for` statement (14.14.2)
- the resource specification of a `try`-with-resources statement (14.20.3)
- a pattern (14.30.1)

The rules concerning annotation modifiers for a local variable declaration are specified in 9.7.4 ⇗ and 9.7.5 ⇗.

If the keyword `final` appears as a modifier for a local variable declaration, then the local variable is a `final` variable (4.12.4 ⇗).

It is a compile-time error if `final` appears more than once as a modifier for a local variable declaration.

It is a compile-time error if a local variable declaration that does not include an *Identifier* and does not have an initializer is used in the following locations:

- a local variable declaration statement in a block (14.4.2 ⇗)
- the header of a basic `for` statement (14.14.1 ⇗)

It is a compile-time error if the *LocalVariableType* is `var` and any of the following are true:

- More than one *VariableDeclarator* is listed.
- The *VariableDeclaratorId* has one or more bracket pairs.
- The *VariableDeclarator* lacks an initializer.
- The initializer of the *VariableDeclarator* is an *ArrayInitializer*.
- The initializer of the *VariableDeclarator* contains a reference to the variable.

Example 14.4-1. Local Variables Declared With `var`

The following code illustrates these rules restricting the use of `var`:

```
var a = 1;            // Legal
var b = 2, c = 3.0;   // Illegal: multiple declarators
var d[] = new int[4]; // Illegal: extra bracket pairs
var e;                // Illegal: no initializer
```

```
var f = { 6 };        // Illegal: array initializer
var g = (g = 7);      // Illegal: self reference in initializer
```

These restrictions help to avoid confusion about the type being represented by `var`*.*

14.11 The `switch` Statement

The `switch` statement transfers control to one of several statements or expressions, depending on the value of an expression.

SwitchStatement:
`switch (` *Expression* `)` *SwitchBlock*

The *Expression* is called the *selector expression*. The type of the selector expression must be `char`, `byte`, `short`, `int`, or a reference type, or a compile-time error occurs.

14.11.1 Switch Blocks

The body of both a `switch` statement and a `switch` expression (15.28 ⇗) is called a *switch block*. This subsection presents general rules which apply to all switch blocks, whether they appear in `switch` statements or `switch` expressions. Other subsections present additional rules which apply either to switch blocks in `switch` statements (14.11.2 ⇗) or to switch blocks in `switch` expressions (15.28.1 ⇗).

SwitchBlock:
`{` *SwitchRule* {*SwitchRule*} `}`
`{` {*SwitchBlockStatementGroup*} {*SwitchLabel* `:`} `}`

SwitchRule:
SwitchLabel `->` *Expression* `;`
SwitchLabel `->` *Block*
SwitchLabel `->` *ThrowStatement*

SwitchBlockStatementGroup:
SwitchLabel `:` { *SwitchLabel* `:`} *BlockStatements*

~~*SwitchLabel:*~~
~~`case` *CaseConstant* {~~`,` *~~CaseConstant~~*~~}~~
~~`case null` [~~`, default`~~]~~
~~`case` *CasePattern* [*Guard*]~~
~~`default`~~

SwitchLabel:
`case` *CaseConstant* {`,` *CaseConstant*}
`case null` [`, default`]
`case` *CasePattern*{`,` *CasePattern* } [*Guard*]
`default`

CaseConstant:
ConditionalExpression

CasePattern:
Pattern

Guard:
`when` *Expression*

A switch block can consist of either:

- *Switch rules*, which use `->` to introduce either a *switch rule expression*, a *switch rule block*, or a *switch rule `throw` statement*; or
- *Switch labeled statement groups*, which use `:` to introduce *switch labeled block statements*.

Every switch rule and switch labeled statement group starts with a *switch label*, which is either a `case` label or a `default` label. Multiple switch labels are permitted for a switch labeled statement group.

A `case` label consists of either a list of `case` constants or ~~a single~~ one or more `case` pattern**s**.

- ~~Every~~For a case label with case constants, every `case` constant must be either (1) the `null` literal, (2) a constant expression (15.29 ⇗), or (3) the (simple or qualified) name of an enum constant (8.9.1 ⇗); otherwise a compile-time error occurs. A single `null` case constant may also be paired with the `default` keyword.
- For a case label with case patterns, it is a compile-time error if any of its case patterns declares one or more pattern variables.

  > *If a `case` label had more than one `case` pattern that declared pattern variables, then it would not be clear which variable would be initialized if the `case` label were to apply. For example, in the code fragment `switch (obj) { case Integer i, Boolean b -> { ... } ... }` it is not clear in the block to the right of the `->` which of the pattern variables `i` or `b` will be initialized, so this is a compile-time error. The fragment `switch(obj) { case Integer i, Boolean _ -> { ... } ... }` still results in a compile-time error, as it is still not clear if the pattern variable `i` will be initialized. The fragment `switch (obj) { case Integer _, Boolean _ -> { ... } ... }` does not result in a compile-time error.*

A `case` label with a `case` pattern may have an optional `when` expression, known as a *guard*, which represents a further test on values that match the pattern. A `case` label is said to be *unguarded* if either (i) it has no guard, or (ii) it has a guard that is a constant expression (15.29 ⇗) with value `true`; and *guarded* otherwise.

Switch labels and their `case` constants and `case` patterns are said to be *associated* with the switch block.

For a given switch block both of the following must be true, otherwise a compile-time error occurs:

- No two of the `case` constants associated with a switch block may have the same value.
- No more than one `default` label may be associated with a switch block.

Any guard associated with a `case` label must have type `boolean` or `Boolean`. Any variable that is used but not declared by a guard must either be `final` or effectively final (4.12.4 ⇗) and cannot be assigned to (15.26 ⇗), incremented ([15.14.2]), or decremented ([15.14.3]), otherwise a compile-time error occurs. It is a compile-time error if a guard is a constant expression (15.29 ⇗) with the value `false`.

The switch block of a `switch` statement or a `switch` expression is *switch compatible* with the type of the selector expression, *T*, if all of the following are true:

- If any `null` constant is associated with the switch block, then *T* is a reference type.
- If *T* is an enum type, then every `case` constant associated with the switch block that is the name of an enum constant is the name of an enum constant of type *T*.
- If *T* is not an enum type, then every `case` constant associated with the switch block that is the name of of an enum constant is the qualified name of an enum constant that is assignment compatible with *T* (5.2 ⇗).
- Every `case` constant associated with the switch block that is a constant expression is assignment compatible with *T*, and *T* is one of `char`, `byte`, `short`, `int`, `Character`, `Byte`,

`Short`, `Integer`, or `String`.

- Every pattern *p* associated with the switch block, *p* is applicable for type *T* (14.30.3 ⇗).

Switch blocks are not designed to work with the types `boolean`, `long`, `float`, and `double`. The selector expression of a `switch` statement or `switch` expression can not have one of these types.

The switch block of a `switch` statement or a `switch` expression must be switch compatible with the type of the selector expression, or a compile-time error occurs.

A switch label in a switch block is said to be *dominated* if for every value that it applies to, one of the preceding switch labels would also apply. It is a compile-time error if any switch label in a switch block is dominated. The rules for determining whether a switch label is dominated are as follows:

- A `case` label with a `case` pattern *q* is dominated if there is a preceding unguarded `case` label in the switch block with a `case` pattern *p*, and *p* dominates *q* (14.30.3 ⇗).

  The definition of one pattern dominating another pattern is based on types. For example, the type pattern Object o dominates the type pattern String s, and so the following results in a compile-time error:

  ```
  Object obj = ...
  switch (obj) {
      case Object o ->
          System.out.println("An object");
      case String s   ->                 // Error - dominated case label
          System.out.println("A string");
  }
  ```

  A guarded `case` label with a `case` pattern is dominated by a `case` label with the same pattern but without the guard. For example, the following results in a compile-time error:

  ```
  String str = ...;
  switch (str) {
      case String s ->
          System.out.println("A string");
      case String s when s.length() == 2 ->  // Error - dominated case label
          System.out.println("Two character string");
      ...
  }
  ```

  On the other hand, a guarded `case` label with a `case` pattern is not considered to dominate an unguarded `case` label with the same `case` pattern. This allows the following common pattern programming style:

  ```
  Integer j = ...;
  switch (j) {
      case Integer i when i <= 0 ->
          System.out.println("Less than or equal to zero");
      case Integer i ->
          System.out.println("An integer");
  }
  ```

  The only exception is where the guard is a constant expression that has the value `true`, for example:

  ```
  Integer j = ...;
  switch (j) {
      case Integer i when true ->            // Allowed but why write this?
          System.out.println("An integer");
      case Integer i ->                     // Error - dominated case label
          System.out.println("An integer");
  }
  ```

A case label with multiple patterns is dominated if any one of these patterns is dominated by a pattern that appears as a case pattern in a preceding unguarded case label; for example `Integer _` is dominated by `Number _` on a preceding unguarded case:

```
Object o = ...
switch (o) {
    case Number _ ->
        System.out.println("A Number");
    case Integer _, String _ ->              // Error - dominated case pattern: `Integer _`
        System.out.println("An Integer or a String");
}
```

- A `case` label with a `case` constant *c* is dominated if one of the following holds:
  - *c* is a constant expression of a primitive type *S*, and there is a preceding `case` label in the switch block with a `case` pattern *p*, where *p* is unconditional for the wrapper class of *S*.
  - *c* is either a constant expression of a reference type *T*, and there is a preceding `case` label in the switch block with a `case` pattern *p*, where *p* is unconditional for the type *T*.
  - *c* is an enum constant of type *T*, and there is a preceding `case` label in the switch block with a `case` pattern *p*, where *p* is unconditional for the type *T*.

  For example, a `case` label with an `Integer` type pattern dominates a `case` label with an integer literal:

  ```
  Integer j = ...;
  switch (j) {
      case Integer i ->
          System.out.println("An integer");
      case 42 ->                               // Error - dominated!
          System.out.println("42!");
  }
  ```

  Analysis of guards—undecidable in general—is not attempted. For example, the following results in a compile-time error, even though the first switch label does not match if the value of the selector expression is `42`:

  ```
  Integer j = ...;
  switch (j) {
      case Integer i when i != 42 ->
          System.out.println("An integer that isn't 42");
      case 42 ->                                   // Error - dominated!
          System.out.println("42!");
  }
  ```

  Any `case` labels with `case` constants should appear before those with `case` patterns; for example:

  ```
  Integer j = ...;
  switch (j) {
      case 42 ->
          System.out.println("42");
      case Integer i when i < 50 ->
          System.out.println("An integer less than 50");
      case Integer i  ->
          System.out.println("An integer");
  }
  ```

- A `case` label with a `case` pattern is dominated if there is a preceding `default` label in the switch block.
- A `case` label with a `null` case constant is dominated if there is a preceding `default` label in the switch block.

If used, a `default` label should come last in a `switch` block.

For historical reasons, a `default` label may appear before `case` labels that do not have a `null` `case` constant or a `case` pattern.

```
int i = ...;
switch(i) {
    default ->
        System.out.println("Some other integer");
    case 42 -> // allowed
        System.out.println("42");
}
```

This style is discouraged in new code.

- A switch label is dominated if there is a preceding `case null, default` label in the switch block.

  If used, a `case null, default` label always comes last in a `switch` block.

- A `default` label is dominated if there is a preceding unguarded `case` label in the switch block with a `case` pattern *p* where *p* is unconditional for the type of the selector expression (14.30.3 ⇗).
- A `case null, default` label is dominated if there is a preceding unguarded `case` label in the switch block with a `case` pattern *p* where *p* is unconditional for the type of the selector expression (14.30.3 ⇗).

  A `case` label with a `case` pattern that is unconditional for the type of the selector expression will, as the name suggests, match every value and so behaves like a `default` label. A switch block can not have more than one switch label that acts like a `default`.

It is a compile-time error if in a switch block there is a `case` label with `case` patterns $p_1,\ldots,p_n$ ($n > 1$) where one of the patterns p_i ($1 \le i < n$) dominates another of the patterns p_j ($i < j \le n$).

It is a compile-time error if, in a switch block that consists of switch labeled statement groups, a statement is labeled with a `case` pattern that declares one or more pattern variables, and either:

- An immediately preceding statement in the switch block can complete normally (14.22 ⇗), or
- The statement is labeled with more than one switch label.

The first condition prevents a statement group from "falling through" to another statement group without initializing pattern variables. For example, were a statement labeled by `case Integer i` reachable from the preceding statement group, the pattern variable `i` would not have been initialized:

```
Object o = "Hello";
switch (o) {
    case String s:
        System.out.println("String: " + s );  // No break!
    case Integer i:
        System.out.println(i + 1);            // Error! Can be reached
                                              // without matching the
                                              // pattern `Integer i`
    default:
}
```

Switch blocks consisting of switch label statement groups allow multiple labels to apply to a statement group. The second condition prevents a statement group from being executed based on one label without initializing the pattern variables of another label. For example:

```
Object o = "Hello World";
switch (o) {
    case String s:
    case Integer i:
```

```
        System.out.println(i + 1);  // Error! Can be reached
                                    // without matching the
                                    // pattern `Integer i`
    default:
}

Object obj = null;
switch (obj) {
    case null:
    case String s:
        System.out.println(s);      // Error! Can be reached
                                    // without matching the
                                    // pattern `String s`
    default:
}
```

Both of these conditions apply only when the `case` *pattern declares pattern variables. The following examples, in contrast, are unproblematic:*

```
record R() {}
record S() {}

Object o = "Hello World";
switch (o) {
    case String s:
        System.out.println(s);        // No break!
    case R():
        System.out.println("It's either an R or a string");
        break;
    default:
}

Object ob = new R();
switch (ob) {
    case R():
    case S():                         // Multiple case labels!
        System.out.println("Either R or an S");
        break;
    default:
}

Object obj = null;
switch (obj) {
    case null:
    case R():                         // Multiple case labels!
        System.out.println("Either null or an R");
        break;
    default:
}
```

14.11.1.2 Determining which Switch Label Applies at Run-Time

Both the execution of a `switch` statement (14.11.3 ⇗) and the evaluation of a `switch` expression (15.28.2 ⇗) need to determine if a switch label associated with the switch block *applies* to the value of the selector expression. This proceeds as follows:

1. If the value is the null reference, then a `case` label with a `null` `case` constant applies.
2. If the value is not the null reference, then we determine the first (if any) `case` label in the switch block that applies to the value as follows:
    - A `case` label with a non-null `case` constant *c* applies to a value of type `Character`, `Byte`, `Short`, or `Integer`, if the value is first subjected to unboxing conversion (5.1.8 ⇗)

and the constant *c* is equal to the unboxed value.

Any unboxing conversion must complete normally as the value being unboxed is guaranteed not to be the null reference.

Equality is defined in terms of the `==` operator (15.21 ⇗).

- A `case` label with a non-null `case` constant *c* applies to a value that is not of type `Character`, `Byte`, `Short`, or `Integer`, if the constant *c* is equal to the value.

  Equality is defined in terms of the `==` operator unless the value is a `String`, in which case equality is defined in terms of the `equals` method of class `String`.

- ~~Determining that a `case` label with a `case` pattern *p* applies to a value proceeds first by checking if the value matches the pattern *p* (14.30.2).~~

  Determining that a `case` label with `case` patterns $p_1,\ldots,p_n$ ($n \geq 1$) *applies* to a value proceeds by determining the first (if any) `case` pattern p_i ($1 \leq i \leq n$) that applies.

  If the process of determining which `case` pattern applies completes abruptly, then the process of determining which switch label applies completes abruptly for the same reason.

  Determining that a `case` pattern *p* applies to a value proceeds first by checking if the value matches the pattern *p* (14.30.2).

  If pattern matching completes abruptly then the process of determining which ~~switch label~~`case` pattern applies completes abruptly for the same reason.

  If pattern matching succeeds and the `case` ~~label~~pattern is unguarded then this `case` ~~label~~pattern applies.

  If pattern matching succeeds and the `case` ~~label~~pattern is guarded, then the guard is evaluated. If the result is of type `Boolean`, it is subjected to unboxing conversion (5.1.8 ⇗).

  If evaluation of the guard or the subsequent unboxing conversion (if any) completes abruptly for some reason, the process of determining which ~~switch label~~pattern applies completes abruptly for the same reason.

  Otherwise, if the resulting value is `true` then the `case` ~~label~~pattern applies.

- A `case null, default` label applies to every value

3. If the value is not the null reference, and no `case` label applies according to the rules of step 2, then if a `default` label is associated with the switch block, that label applies.

A single `case` label can contain several `case` constants. The label applies to the value of the selector expression if any one of its constants is equal to the value of the selector expression. For example, in the following code, the `case` label matches if the enum variable `day` is either one of the enum constants shown:

```
switch (day) {
    ...
    case SATURDAY, SUNDAY :
        System.out.println("It's the weekend!");
        break;
    ...
}
```

If a `case` label with a `case` pattern applies, then this is because the process of pattern matching the value against the pattern has succeeded (14.30.2). If a value successfully matches a pattern then the process of pattern matching initializes any pattern variables declared by the pattern.

For historical reasons, a `default` label is only considered after all `case` labels have failed to match, even if some of those labels appear after the `default` label. However, subsequent labels may only make use of

non-`null` case constants (14.11.1), and as a matter of style, programmers are encouraged to place their `default` labels last.

In C and C++ the body of a `switch` statement can be a statement and statements with `case` labels do not have to be immediately contained by that statement. Consider the simple loop:

```
for (i = 0; i < n; ++i) foo();
```

where `n` is known to be positive. A trick known as Duff's device can be used in C or C++ to unroll the loop, but this is not valid code in the Java programming language:

```
int q = (n+7)/8;
switch (n%8) {
    case 0: do { foo();    // Great C hack, Tom,
    case 7:      foo();    // but it's not valid here.
    case 6:      foo();
    case 5:      foo();
    case 4:      foo();
    case 3:      foo();
    case 2:      foo();
    case 1:      foo();
            } while (--q > 0);
}
```

Fortunately, this trick does not seem to be widely known or used. Moreover, it is less needed nowadays; this sort of code transformation is properly in the province of state-of-the-art optimizing compilers.

14.14 The `for` Statement

14.14.2 The enhanced `for` statement

The enhanced `for` statement has the form:

EnhancedForStatement:
`for` `(` *LocalVariableDeclaration* `:` *Expression* `)`
Statement

EnhancedForStatementNoShortIf:
`for` `(` *LocalVariableDeclaration* `:` *Expression* `)`
StatementNoShortIf

The following productions from 4.3, 8.3, 8.4.1, and 14.4 are shown here for convenience:

LocalVariableDeclaration:
{VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier:
Annotation
`final`

LocalVariableType:
UnannType
`var`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]
_

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

The type of the *Expression* must be an array type (10.1 ⇗) or a subtype of the raw type `Iterable`, or a compile-time error occurs.

The header of the enhanced `for` statement either declares a local variable whose name is the identifier given by *VariableDeclaratorId*, or declares an unnamed local variable (6.1).. When the enhanced `for` statement is executed, the local variable is initialized, on each iteration of the loop, to successive elements of the `Iterable` or the array produced by the expression.

The rules for a local variable declared in the header of an enhanced `for` statement are specified in 14.4, disregarding any rules in that section which apply when the *LocalVariableType* is `var`. In addition, all of the following must be true, or a compile-time error occurs:

- The *VariableDeclaratorList* consists of a single *VariableDeclarator*.
- The *VariableDeclarator* has no initializer.
- The *VariableDeclaratorId* has no bracket pairs if the *LocalVariableType* is `var`.

The scope and shadowing of a local variable declared in the header of an enhanced `for` statement is specified in 6.3 ⇗ and 6.4 ⇗.

References to the local variable from a nested class or interface, or a lambda expression, are restricted, as specified in 6.5.6.1 ⇗.

The type *T* of the local variable declared in the header of the enhanced `for` statement is determined as follows:

- If the *LocalVariableType* is *UnannType*, and no bracket pairs appear in *UnannType* or *VariableDeclaratorId*, then *T* is the type denoted by *UnannType*.
- If the *LocalVariableType* is *UnannType*, and bracket pairs appear in *UnannType* or *VariableDeclaratorId*, then *T* is specified by 10.2 ⇗.
- If the *LocalVariableType* is `var`, then let *R* be derived from the type of the *Expression*, as follows:
  - If the *Expression* has an array type, then *R* is the component type of the array type.
  - Otherwise, if the *Expression* has a type that is a subtype of `Iterable<`*X*`>`, for some type *X*, then *R* is *X*.
  - Otherwise, the *Expression* has a type that is a subtype of the raw type `Iterable`, and *R* is `Object`.

  T is the upward projection of *R* with respect to all synthetic type variables mentioned by *R* (4.10.5 ⇗).

The precise meaning of the enhanced `for` statement whose header declares a local variable *Identifier* is given by translation into a basic `for` statement, as follows:

- If the type of *Expression* is a subtype of `Iterable`, then the basic `for` statement has this form:

```
for (I #i = Expression.iterator(); #i.hasNext(); ) {
    {VariableModifier} T Identifier = (TargetType) #i.next();
    Statement
}
```

  where:

- If the type of *Expression* is a subtype of `Iterable<X>` for some type argument *X*, then *I* is the type `java.util.Iterator<X>`. Otherwise, *I* is the raw type `java.util.Iterator`.
- `#i` is an automatically generated identifier that is distinct from any other identifiers (automatically generated or otherwise) that are in scope (6.3) at the point where the enhanced `for` statement occurs.
- *{VariableModifier}* is as given in the header of the enhanced `for` statement.
- *T* is the type of the local variable as determined above.
- If *T* is a reference type, then *TargetType* is *T*. Otherwise, *TargetType* is the upper bound of the capture conversion (5.1.10) of the type argument of *I*, or `Object` if *I* is raw.

- Otherwise, the *Expression* necessarily has an array type, *S*`[]`, and the basic `for` statement has this form:

```
S[] #a = Expression;
L1: L2: ... Lm:
for (int #i = 0; #i < #a.length; #i++) {
    {VariableModifier} T Identifier = #a[#i];
    Statement
}
```

where:

- L_1 ... L_m is the (possibly empty) sequence of labels immediately preceding the enhanced `for` statement.
- `#a` and `#i` are automatically generated identifiers that are distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the enhanced `for` statement occurs.
- *{VariableModifier}* is as given in the header of the enhanced `for` statement.
- *T* is the type of the local variable as determined above.

For example, this code:

```
List<? extends Integer> l = ...
for (float i : l) ...
```

will be translated to:

```
for (Iterator<Integer> #i = l.iterator(); #i.hasNext(); ) {
    float #i0 = (Integer)#i.next();
    ...
```

Example 14.14-1. Enhanced `for` And Arrays

The following program, which calculates the sum of an integer array, shows how enhanced `for` works for arrays:

```
int sum(int[] a) {
    int sum = 0;
    for (int i : a) sum += i;
    return sum;
}
```

Example 14.14-2. Enhanced `for` And Unboxing Conversion

The following program combines the enhanced `for` statement with auto-unboxing to translate a histogram into a frequency table:

```
Map<String, Integer> histogram = ...;
double total = 0;
for (int i : histogram.values())
    total += i;
for (Map.Entry<String, Integer> e : histogram.entrySet())
    System.out.println(e.getKey() + " " + e.getValue() / total);
}
```

The precise meaning of an enhanced `for` statement whose header declares an unnamed local variable is given by translation into a basic `for` statement as above but using an unnamed local variable in place of *Identifier*.

14.20 The `try` statement

14.20.3 `try`-with-resources

A `try`-with-resources statement is parameterized with variables (known as *resources*) that are initialized before execution of the `try` block and closed automatically, in the reverse order from which they were initialized, after execution of the `try` block. `catch` clauses and a `finally` clause are often unnecessary when resources are closed automatically.

TryWithResourcesStatement:
`try` *ResourceSpecification Block* [*Catches*] [*Finally*]

ResourceSpecification:
`(` *ResourceList* [`;`] `)`

ResourceList:
Resource {`;` *Resource*}

Resource:
LocalVariableDeclaration
VariableAccess

VariableAccess:
ExpressionName
FieldAccess

The following productions from 4.3, 8.3, 8.4.1, and 14.4 are shown here for convenience:

LocalVariableDeclaration:
{VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier:
Annotation
`final`

LocalVariableType:
UnannType
`var`

VariableDeclaratorList:
VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
VariableDeclaratorId [= VariableInitializer]

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]
_

Dims:

{Annotation} `[ ]` *{{Annotation}* `[ ]}`

VariableInitializer:
Expression
ArrayInitializer

See 8.3 for UnannType.

The *resource specification* denotes the resources of the `try`-with-resources statement, either by declaring local variables with initializer expressions or by referring to existing variables. An existing variable is referred to by an expression name (6.5.6 ⇗) or a field access expression (15.11 ⇗).

The rules for a local variable declared in a resource specification are specified in 14.4. In addition, all of the following must be true, or a compile-time error occurs:

- The *VariableDeclaratorList* consists of a single *VariableDeclarator*.
- The *VariableDeclarator* has an initializer.
- The *VariableDeclaratorId* has no bracket pairs.

The scope and shadowing of a local variable declared in a resource specification is specified in 6.3 ⇗ and 6.4 ⇗.

References to the local variable from a nested class or interface, or a lambda expression, are restricted, as specified in 6.5.6.1 ⇗.

The type of a local variable declared in a resource specification is specified in 14.4.1 ⇗.

The type of a local variable declared in a resource specification, or the type of an existing variable referred to in a resource specification, must be a subtype of `AutoCloseable`, or a compile-time error occurs.

It is a compile-time error for a resource specification to declare two local variables with the same name.

Note that a resource specification may contain the declaration of multiple unnamed local variables (6.1).

Resources are `final`, in that:

- A local variable declared in a resource specification is implicitly declared `final` if it is not explicitly declared `final` (4.12.4 ⇗).
- An existing variable referred to in a resource specification must be a `final` or effectively `final` variable that is definitely assigned before the `try`-with-resources statement (16 ⇗), or a compile-time error occurs.

Resources are initialized in left-to-right order. If a resource fails to initialize (that is, its initializer expression throws an exception), then all resources initialized so far by the `try`-with-resources statement are closed. If all resources initialize successfully, the `try` block executes as normal and then all non-null resources of the `try`-with-resources statement are closed.

Resources are closed in the reverse order from that in which they were initialized. A resource is closed only if it initialized to a non-null value. An exception from the closing of one resource does not prevent the closing of other resources. Such an exception is *suppressed* if an exception was thrown previously by an initializer, the `try` block, or the closing of a resource.

A `try`-with-resources statement whose resource specification indicates multiple resources is treated as if it were multiple `try`-with-resources statements, each of which has a resource specification that indicates a single resource. When a `try`-with-resources statement with *n* resources (*n* > 1) is translated, the result is a `try`-with-resources statement with *n*-1 resources. After *n* such translations, there are *n* nested `try-catch-finally` statements, and the overall translation is complete.

14.20.3.1 Basic `try`-with-resources

A `try`-with-resources statement with no `catch` clauses or `finally` clause is called a *basic* `try`-with-resources statement.

If a basic `try`-with-resources statement is of the form:

```
try (VariableAccess ...)
    Block
```

then the resource is first converted to a local variable declaration by the following translation:

```
try (T #r = VariableAccess ...) {
    Block
}
```

`T` is the type of the variable denoted by *VariableAccess* and `#r` is an automatically generated identifier that is distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the `try`-with-resources statement occurs. The `try`-with-resources statement is then translated according to the rest of this section.

The meaning of a basic `try`-with-resources statement of the form:

```
try ({VariableModifier} R Identifier = Expression ...)
    Block
```

is given by the following translation to a local variable declaration and a `try-catch-finally` statement:

```
{
    final {VariableModifierNoFinal} R Identifier = Expression;
    Throwable #primaryExc = null;

    try ResourceSpecification_tail
        Block
    catch (Throwable #t) {
        #primaryExc = #t;
        throw #t;
    } finally {
        if (Identifier != null) {
            if (#primaryExc != null) {
                try {
                    Identifier.close();
                } catch (Throwable #suppressedExc) {
                    #primaryExc.addSuppressed(#suppressedExc);
                }
            } else {
                Identifier.close();
            }
        }
    }
}
```

{VariableModifierNoFinal} is defined as *{VariableModifier}* without `final`, if present.

`#t`, `#primaryExc`, and `#suppressedExc` are automatically generated identifiers that are distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the `try`-with-resources statement occurs.

Alternatively, the meaning of a basic `try`-with-resources statement of the form:

```
try ({VariableModifier} R _ = Expression ...)
    Block
```

is given by the following translation to a local variable declaration and a `try-catch-finally` statement:

```
{
    final {VariableModifierNoFinal} R #i = Expression;
    Throwable #primaryExc = null;

    try ResourceSpecification_tail
        Block
    catch (Throwable #t) {
        #primaryExc = #t;
        throw #t;
    } finally {
        if (#i != null) {
            if (#primaryExc != null) {
                try {
                    #i.close();
                } catch (Throwable #suppressedExc) {
                    #primaryExc.addSuppressed(#suppressedExc);
                }
            } else {
                #i.close();
            }
        }
    }
}
```

{VariableModifierNoFinal} is defined as *{VariableModifier}* without `final`, if present.

`#t`, `#primaryExc`, `#suppressedExc`, and `#i` are automatically generated identifiers that are distinct from any other identifiers (automatically generated or otherwise) that are in scope at the point where the `try`-with-resources statement occurs.

If the resource specification indicates one resource, then *ResourceSpecification_tail* is empty (and the `try-catch-finally` statement is not itself a `try`-with-resources statement).

If the resource specification indicates *n* > 1 resources, then *ResourceSpecification_tail* consists of the 2nd, 3rd, ..., *n*'th resources indicated in the resource specification, in the same order (and the `try-catch-finally` statement is itself a `try`-with-resources statement).

Reachability and definite assignment rules for the basic `try`-with-resources statement are implicitly specified by the ~~translation~~ translations above.

In a basic `try`-with-resources statement that manages a single resource:

- If the initialization of the resource completes abruptly because of a `throw` of a value *V*, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*.
- If the initialization of the resource completes normally, and the `try` block completes abruptly because of a `throw` of a value *V*, then:
    - If the automatic closing of the resource completes normally, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*.
    - If the automatic closing of the resource completes abruptly because of a `throw` of a value *V2*, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*, with *V2* added to the suppressed exception list of *V*.

- If the initialization of the resource completes normally, and the `try` block completes normally, and the automatic closing of the resource completes abruptly because of a `throw` of a value *V*, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*.

In a basic `try`-with-resources statement that manages multiple resources:

- If the initialization of a resource completes abruptly because of a `throw` of a value *V*, then:
    - If the automatic closings of all successfully initialized resources (possibly zero) complete normally, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*.
    - If the automatic closings of all successfully initialized resources (possibly zero) complete abruptly because of `throw`s of values *V1*...*Vn*, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*, with any remaining values *V1*...*Vn* added to the suppressed exception list of *V*.
- If the initialization of all resources completes normally, and the `try` block completes abruptly because of a `throw` of a value *V*, then:
    - If the automatic closings of all initialized resources complete normally, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*.
    - If the automatic closings of one or more initialized resources complete abruptly because of `throw`s of values *V1*...*Vn*, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*, with any remaining values *V1*...*Vn* added to the suppressed exception list of *V*.
- If the initialization of every resource completes normally, and the `try` block completes normally, then:
    - If one automatic closing of an initialized resource completes abruptly because of a `throw` of value *V*, and all other automatic closings of initialized resources complete normally, then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V*.
    - If more than one automatic closing of an initialized resource completes abruptly because of `throw`s of values *V1*...*Vn* (where *V1* is the exception from the rightmost resource failing to close and *Vn* is the exception from the leftmost resource failing to close), then the `try`-with-resources statement completes abruptly because of a `throw` of the value *V1*, with any remaining values *V2*...*Vn* added to the suppressed exception list of *V1*.

14.20.3.2 Extended `try`-with-resources

A `try`-with-resources statement with at least one `catch` clause and/or a `finally` clause is called an *extended* `try`-with-resources statement.

The meaning of an extended `try`-with-resources statement:

```
try ResourceSpecification
    Block
[Catches]
[Finally]
```

is given by the following translation to a basic `try`-with-resources statement nested inside a `try-catch` or `try-finally` or `try-catch-finally` statement:

```
try {
    try ResourceSpecification
        Block
}
[Catches]
[Finally]
```

The effect of the translation is to put the resource specification "inside" the `try` statement. This allows a `catch` clause of an extended `try`-with-resources statement to catch an exception due to the automatic initialization or closing of any resource.

Furthermore, all resources will have been closed (or attempted to be closed) by the time the `finally` block is executed, in keeping with the intent of the `finally` keyword.

14.30 Patterns

14.30.1 Kinds of Patterns

A *type pattern* is used to test whether a value is an instance of the type appearing in the pattern. A *record pattern* is used to test whether a value is an instance of a record class type and, if it is, to recursively perform pattern matching on the record component values.

Pattern:
    TypePattern
    RecordPattern

TypePattern:
    LocalVariableDeclaration

RecordPattern:
    ReferenceType ([~~*PatternList*~~ *ComponentPatternList*])

~~*PatternList*~~ **:**
    ~~*Pattern* { , *Pattern* }~~

ComponentPatternList:
    ComponentPattern { , *ComponentPattern* }

ComponentPattern:
    Pattern
    UnnamedPattern

UnnamedPattern:
    _

The following productions from 4.3, 8.3, 8.4.1, and 14.4 are shown here for convenience:

LocalVariableDeclaration:
    {VariableModifier} LocalVariableType VariableDeclaratorList

VariableModifier:
    Annotation
    `final`

LocalVariableType:
    UnannType
    `var`

VariableDeclaratorList:
    VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
    VariableDeclaratorId [= VariableInitializer]

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]
_

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

See 8.3 for UnannType.

A pattern that does not appear as an element in the nested pattern list of a record pattern is called a *top-level* pattern; otherwise it is called a *nested* pattern.

A type pattern declares exactly one pattern variable. This pattern variable may be an *unnamed pattern variable* (denoted by _), otherwise the ~~The~~ *Identifier* in the local variable declaration specifies the name of the pattern variable.

The rules for a pattern variable declared in a type pattern are specified in 14.4. In addition, all of the following must be true, or a compile-time error occurs:

- The *LocalVariableType* in a top-level type pattern denotes a reference type (and furthermore is not `var`).
- The *VariableDeclaratorList* consists of a single *VariableDeclarator*.
- The *VariableDeclarator* has no initializer.
- The *VariableDeclaratorId* has no bracket pairs.

The type of a pattern variable declared in a top-level type pattern is the reference type denoted by *LocalVariableType*.

The type of a pattern variable declared in a nested type pattern is determined as follows:

- If the *LocalVariableType* is *UnannType* then the type of the pattern variable is denoted by *UnannType*
- If the *LocalVariableType* is `var` then the type pattern must be nested in a component pattern list of a record pattern with type *R*. Let *T* be the type of the corresponding component field in *R*. The type of the pattern variable is the upward projection of *T* with respect to all synthetic type variables mentioned by *T*.

  Consider the following record declaration:

  ```
  record R<T>(ArrayList<T> r){}
  ```

  Given the pattern `R<String>(var r)`*, the type of the pattern variable* `r` *is thus* `ArrayList<String>`*.*

A type pattern is said to be *null-matching* if it appears as an element in a pattern list of a record pattern with type *R*, the corresponding record component of *R* has type *U*, and the type pattern is unconditional for the type *U* (14.30.3 ⇗).

Note that this compile-time property of type patterns is used in the process of pattern matching (14.30.2), so it is associated with the type pattern for use at run time.

A record pattern consists of a *ReferenceType* and a nested component pattern list. If *ReferenceType* is not a record class type (8.10) then a compile-time error occurs.

If the *ReferenceType* is a raw type, then the type of the record pattern is inferred, as described in [18.5.5]. It is a compile-time error if no type can be inferred for the record pattern.

Otherwise, the type of the record pattern is *ReferenceType*.

The length of the record pattern's nested component pattern list must be the same as the length of the record component list in the declaration of the record class named by *ReferenceType*; otherwise a compile-time error occurs.

Currently, there is no support for variable arity record patterns. This may be supported in future versions of the Java Programming Language.

A record pattern declares the pattern variables, if any, that are declared by the patterns in the nested component pattern list.

An unnamed pattern is a special pattern that can only appear as an element in a component pattern list of a record pattern with type *R*. Let *T* be the type of the corresponding component field in *R*. An unnamed pattern does not declare any pattern variables and its type is defined to be the upward projection of *T* with respect to all synthetic type variables mentioned by *T*. An unnamed pattern is always null-matching.

14.30.2 Pattern Matching

Pattern matching is the process of testing a value against a pattern at run time. Pattern matching is distinct from statement execution (14.1 ⇗) and expression evaluation (15.1 ⇗). If a value successfully matches a pattern, then the process of pattern matching will initialize all the pattern variables declared by the pattern, if any.

The process of pattern matching may involve expression evaluation or statement execution. Accordingly, pattern matching is said to *complete abruptly* if evaluation of a expression or execution of a statement completes abruptly. An abrupt completion always has an associated reason, which is always a `throw` with a given value. Pattern matching is said to *complete normally* if it does not complete abruptly.

The rules for determining whether a value matches a pattern, and for initializing pattern variables, are as follows:

- The null reference *matches* a ~~type pattern~~ null-matching pattern ~~if the type pattern is null-matching~~ (14.30.1); and *does not match* otherwise.

  If the null reference matches, then the pattern variable, if any, declared by the ~~type~~ null-matching pattern is initialized to the null reference.

  If the null reference does not match, then the pattern variable, if any, declared by the ~~type~~ null-matching pattern is not initialized.

- A value *v* that is not the null reference *matches* a type pattern of type *T* if *v* can be cast to *T* without raising a `ClassCastException`; and *does not match* otherwise.

  If *v* matches, then the pattern variable declared by the type pattern is initialized to *v*.

  If *v* does not match, then the pattern variable declared by the type pattern is not initialized.

- The null reference *does not match* a record pattern.

  In this case, any pattern variables declared by the record pattern are not initialized.

- A value *v* that is not the null reference *matches* a record pattern with type *R* and nested component pattern list *L* if (i) *v* can be cast to *R* without raising a `ClassCastException`; and (ii) each record component of *v* matches the corresponding pattern in *L*; and *does not match* otherwise.

  Each record component of *v* is determined by invoking the accessor method of *v* corresponding to that component. If execution of the invocation of the accessor method completes abruptly for reason *S*, then pattern matching completes abruptly by throwing a `MatchException` with cause *S*.

  Any pattern variable declared in a pattern appearing in the nested component pattern list is initialized only if all the patterns in the list match.

Chapter 15: Expressions

15.27 Lambda Expressions

15.27.1 Lambda Parameters

The formal parameters of a lambda expression, if any, are specified by either a parenthesized list of comma-separated parameter specifiers or a parenthesized list of comma-separated identifiers. In a list of parameter specifiers, each parameter specifier consists of optional modifiers, then a type (or `var`), then an identifier that specifies the name of the parameter. In a list of identifiers, each identifier specifies the name of the parameter.

If a lambda expression has no formal parameters, then an empty pair of parentheses appears before the `->` and the lambda body.

If a lambda expression has exactly one formal parameter, and the parameter is specified by an identifier instead of a parameter specifier, then the parentheses around the identifier may be elided.

LambdaParameters:
`(` [*LambdaParameterList*] `)`
Identifier
_

LambdaParameterList:
LambdaParameter {`,` *LambdaParameter*}
~~*Identifier* {~~~~`,`~~ ~~*Identifier*~~~~}~~
IdentifierOrUnderscore {`,` *IdentifierOrUnderscore*}

LambdaParameter:
{*VariableModifier*} *LambdaParameterType* *VariableDeclaratorId*
VariableArityParameter

LambdaParameterType:
UnannType
`var`

IdentifierOrUnderscore:
Identifier
_

The following productions from 8.4.1, 8.3, and 4.3 are shown here for convenience:

VariableArityParameter:
{*VariableModifier*} *UnannType* {*Annotation*} `...` *Identifier*

VariableModifier:
Annotation
`final`

~~*VariableDeclaratorId:*~~
~~*Identifier [Dims]*~~

VariableDeclaratorId:
Identifier [Dims]
_

Dims:
{*Annotation*} `[]` {{*Annotation*} `[]`}

A formal parameter of a lambda expression may be declared `final`, or annotated, only if specified by a parameter specifier. If a formal parameter is specified by an identifier instead, then the formal parameter is not `final` and has no annotations.

A formal parameter of a lambda expression may be a *variable arity parameter*, indicated by an ellipsis following the type in a parameter specifier. At most one variable arity parameter is

permitted for a lambda expression. It is a compile-time error if a variable arity parameter appears anywhere in the list of parameter specifiers except the last position.

Each formal parameter of a lambda expression has either an *inferred type* or a *declared type*:

- If a formal parameter is specified either by a parameter specifier that uses `var`, or by an identifier instead of a parameter specifier, then the formal parameter has an inferred type. The type is inferred from the functional interface type targeted by the lambda expression (15.27.3 ⇗).
- If a formal parameter is specified by a parameter specifier that does not use `var`, then the formal parameter has a declared type. The declared type is determined as follows:
  - If the formal parameter is not a variable arity parameter, then the declared type is denoted by *UnannType* if no bracket pairs appear in *UnannType* and *VariableDeclaratorId*, and specified by 10.2 ⇗ otherwise.
  - If the formal parameter is a variable arity parameter, then the declared type is an array type specified by 10.2 ⇗.

No distinction is made between the following lambda parameter lists:

```
(int... x) `->` BODY
(int[] x) `->` BODY
```

Either can be used, whether the functional interface's `abstract` method is fixed arity or variable arity. (This is consistent with the rules for method overriding.) Since lambda expressions are never directly invoked, using `int...` for the formal parameter where the functional interface uses `int[]` can have no impact on the surrounding program. In a lambda body, a variable arity parameter is treated just like an array-typed parameter.

A lambda expression where all the formal parameters have declared types is said to be *explicitly typed*. A lambda expression where all the formal parameters have inferred types is said to be *implicitly typed*. A lambda expression with no formal parameters is explicitly typed.

If a lambda expression is implicitly typed, then its lambda body is interpreted according to the context in which it appears. Specifically, the types of expressions in the body, and the checked exceptions thrown by the body, and the type correctness of code in the body all depend on the types inferred for the formal parameters. This implies that inference of formal parameter types must occur "before" attempting to type-check the lambda body.

It is a compile-time error if a lambda expression declares a formal parameter with a declared type *and* a formal parameter with an inferred type.

This rule prevents a mix of inferred and declared types in the formal parameters, such as `(x, int y) -> BODY` or `(var x, int y) -> BODY`. Note that if all the formal parameters have inferred types, the grammar prevents a mix of identifiers and `var` parameter specifiers, such as `(x, var y) -> BODY` or `(var x, y) -> BODY`.

The rules concerning annotation modifiers for a formal parameter declaration are specified in 9.7.4 ⇗ and 9.7.5 ⇗.

It is a compile-time error if `final` appears more than once as a modifier for a formal parameter declaration.

It is a compile-time error if the *LambdaParameterType* of a formal parameter is `var` and the *VariableDeclaratorId* of the same formal parameter has one or more bracket pairs.

The scope and shadowing of a formal parameter is specified in 6.3 ⇗ and 6.4 ⇗.

References to a formal parameter from a nested class or interface, or a nested lambda expression, are restricted, as specified in 6.5.6.1 ⇗.

It is a compile-time error for a lambda expression to declare two formal parameters with the same name. (That is, their declarations mention the same *Identifier*.)

~~*In Java SE 8, the use of _ as the name of a lambda parameter was forbidden, and its use discouraged as the name for other kinds of variable (4.12.3 ⇗). As of Java SE 9, _ is a keyword (3.9 ⇗) so it cannot be used as a variable name in any context.*~~

It is a compile-time error if a formal parameter that is declared `final` is assigned to within the body of the lambda expression.

When the lambda expression is invoked (via a method invocation expression (15.12 ⇗)), the values of the actual argument expressions initialize newly created parameter variables, each of the declared or inferred type, before execution of the lambda body. The *Identifier* that appears in the *LambdaParameter* or directly in the *LambdaParameterList* or *LambdaParameters* may be used as a simple name in the lambda body to refer to the formal parameter.

A *LambdaParameter* denoted by the _ (underscore) character is a special formal parameter of a lambda expression that can appear as an element in a lambda parameter list. Such a formal parameter does not declare a local variable and has an inferred type.

DRAFT 22-internal-adhoc.abimpoudis.20231128