Module Import Declarations (Second Preview)

Changes to the Java® Language Specification • Version 24-internal-adhoc.gbierman.20241024

This document describes changes to the Java Language Specification ⇗ to support *Module Import Declarations*, which is a preview feature of Java SE 24. See JEP 494 ⇗ for an overview of the feature.

A companion document describes the changes needed to the Java Virtual Machine Specification ⇗ to support Module Import Declarations.

The preview feature *Simple Source Files and Instance `main` Methods* proposed by JEP 495 ⇗ depends on this feature.

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

Changelog:

2024-10: First draft of second preview. Changes from first preview:

- *To allow module declarations to declare a transitive dependence on the `java.base` module (7.7.1, 1.5).*
- *Allow a type-import-on-demand declaration to shadow a module import declaration (6.4.1).*

Chapter 1: Introduction

1.5 Preview Features

A *preview feature* is:

- a new feature of the Java programming language ("preview language feature"), or
- a new module, package, class, interface, field, method, constructor, or enum constant in the `java.*` or `javax.*` namespace ("preview API")

that is fully specified, fully implemented, and yet impermanent. It is available in implementations of a given release of the Java SE Platform to provoke developer feedback based on real world use; this may lead to it becoming permanent in a future release of the Java SE Platform.

Implementations must disable, at both compile time and run time, the preview features defined by a given release of the Java SE Platform, unless the user indicates via the host system, at both compile time and run time, that preview features are to be enabled.

The preview features defined by a given release of the Java SE Platform are enumerated in the Java SE Platform Specification for that release. The preview features are specified as follows:

- Preview language features are specified in standalone documents that indicate changes ("diffs") to *The Java® Language Specification* for that release. The specifications of preview language features are incorporated into *The Java® Language Specification* by reference, and made a part thereof, if and only if preview features are enabled at compile time.

  Java SE 24 defines one preview language feature: *Module Import Declarations*. The standalone documents which specify this preview feature are available in the same download bundle as this PDF of *The Java® Language Specification*.

  The preview feature *Module Import Declarations* allows module declarations to declare a transitive dependence on the `java.base` module. The declaration of the `java.se` module has accordingly been extended with a transitive dependence on the `java.base` as part of this preview feature. This updated declaration of the `java.se` module is considered to be *participating* in the preview feature. This means that the `java.se` module can be both compiled and used at run time without indication from the user that preview features are to be enabled.

- Preview APIs are specified within the Java SE API Specification for that release.

The rules for use of preview language features are as follows:

- If preview features are disabled, then any source code reference to a preview language feature, or to a class or interface declared using a preview language feature, causes a compile-time error.
- If preview features are enabled, then any source code reference to a class or interface declared using a preview language feature causes a *preview warning*, unless one of the following is true:
  - The reference appears in a declaration that is annotated to suppress preview warnings (9.6.4.5 ⇗).

    - The reference appears in an import declaration (7.5).

  When preview features are enabled, Java compilers are strongly encouraged to give a non-suppressible warning for every source code reference to a preview language feature. Details of this warning are beyond the scope of this specification, but the intent should be to alert programmers to the possibility of code being affected by future changes to preview language features.

Some preview APIs are described as *reflective* by the Java SE Platform Specification, principally in the `java.lang.reflect`, `java.lang.invoke`, and `javax.lang.model` packages. The rule for use of reflective preview APIs is as follows:

- Whether preview features are enabled or disabled, any source code reference to a reflective preview API element causes a preview warning, unless one of the following is true:
    - The declaration where the reference appears is within the same module as the declaration of the reflective preview API element.
    - The reference appears in a declaration that is annotated to suppress preview warnings.
    - The reference appears in an import declaration.

All preview APIs not described as reflective in the Java SE Platform Specification are *normal*. The rules for use of normal preview APIs are as follows:

- If preview features are disabled, then any source code reference to a normal preview API element causes a compile-time error, unless:
    - The declaration where the reference appears is within the same module as the declaration of the normal preview API element.
- If preview features are enabled, then any source code reference to a normal preview API element causes a preview warning, unless one of the following is true:
    - The declaration where the reference appears is within the same module as the declaration of the normal preview API element.
    - The reference appears in a declaration that is annotated to suppress preview warnings.
    - The reference appears in an import declaration.

Chapter 3: Lexical Structure

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

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) may be used in certain 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 `module`, when recognized as a terminal in a *SingleModuleImportDeclaration* (7.5.5) or a *ModuleDeclaration* (7.7 ⇗).
   - For `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 `n o n - s e a l e d c l a s s` 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 one of the following entities into a program:

- A module, declared in a `module` declaration (7.7 ⇗)
- A package, declared in a `package` declaration (7.4 ⇗)
- An imported class or interface, declared in a single-type-import declaration, ~~or~~ a type-import-on-demand declaration, or a single-module-import declaration (7.5.1 ⇗, 7.5.2 ⇗, 7.5.5)
- 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

The rest of the section is unchanged.

6.3 Scope of a Declaration

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

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

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

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

The declaration of a subpackage is never in scope.

The package `java` is always in scope.

The scope of a class or interface imported by a single-type-import declaration (7.5.1), ~~or~~ a type-import-on-demand declaration (7.5.2), or a single-module-import declaration (7.5.5) is the module declaration (7.7) and all the class and interface declarations (8.1, 9.1) of the compilation unit in which the `import` declaration appears, as well as any annotations on the module declaration or package declaration of the compilation unit.

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

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

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

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

The scope of a class's type parameter (8.1.2) is the type parameter section of the class declaration, and the type parameter section of any superclass type or superinterface type of the class declaration, and the class body. If the class is a record class (8.10), then the scope of the type parameter additionally includes the header of the record declaration (8.10.1).

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

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

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

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

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

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

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

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

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

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

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

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

The rest of the section is unchanged.

6.4 Shadowing and Obscuring

6.4.1 Shadowing

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

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

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

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

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

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

A package declaration never shadows any other declaration.

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

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

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

A single-module-import declaration never causes any other declaration to be shadowed.

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

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

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

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

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

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

The rest of the section is unchanged.

6.5 Determining the Meaning of a Name

6.5.1 Syntactic Classification of a Name According to Context

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

- In a `requires` directive in a module declaration (7.7.1)
- To the right of `to` in an `exports` or `opens` directive in a module declaration (7.7.2 ⇗)
- To the right of `module` in a single-module-import declaration (7.5.5)

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

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

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

- To name a class or interface:
  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 ⇗)
- As the *Identifier* or dotted *Identifier* sequence that constitutes any *ReferenceType* (including a *ReferenceType* to the left of the brackets in an array type, or to the left of the < in a parameterized type, or in a non-wildcard type argument of a parameterized type, or in an `extends` or `super` clause of a wildcard type argument of a parameterized type) in the 17 contexts where types are used (4.11 ⇗):
  1. In an `extends` or `implements` clause of a class declaration (8.1.4 ⇗, 8.1.5 ⇗)
  2. In an `extends` clause of an interface declaration (9.1.3 ⇗)
  3. The return type of a method (8.4.5 ⇗, 9.4 ⇗), including the type of an element of an annotation interface (9.6.1 ⇗)
  4. In the `throws` clause of a method or constructor (8.4.6 ⇗, 8.8.5 ⇗, 9.4 ⇗)
  5. In an `extends` clause of a type parameter declaration of a generic class, interface, method, or constructor (8.1.2 ⇗, 9.1.2 ⇗, 8.4.4 ⇗, 8.8.4 ⇗)
  6. The type in a field declaration of a class or interface (8.3 ⇗, 9.3 ⇗)
  7. The type in a formal parameter declaration of a method, constructor, or lambda expression (8.4.1 ⇗, 8.8.1 ⇗, 9.4 ⇗, 15.27.1 ⇗)

8. The type of the receiver parameter of a method (8.4 ⇗)
9. The type in a local variable declaration in either a statement (14.4.2 ⇗, 14.14.1 ⇗, 14.14.2 ⇗, 14.20.3 ⇗) or a pattern (14.30.1 ⇗)
10. A type in an exception parameter declaration (14.20 ⇗)
11. The type in a record component declaration of a record class (8.10.1 ⇗)
12. In an explicit type argument list to an explicit constructor invocation statement or class instance creation expression or method invocation expression (8.8.7.1 ⇗, 15.9 ⇗, 15.12 ⇗)
13. In an unqualified class instance creation expression, either as the class type to be instantiated (15.9 ⇗) or as the direct superclass or direct superinterface of an anonymous class to be instantiated (15.9.5 ⇗)
14. The element type in an array creation expression (15.10.1 ⇗)
15. The type in the cast operator of a cast expression (15.16 ⇗)
16. The type that follows the `instanceof` relational operator (15.20.2 ⇗)
17. In a method reference expression (15.13 ⇗), as the reference type to search for a member method or as the class type or array type to construct.

The extraction of a TypeName from the identifiers of a ReferenceType in the 17 contexts above is intended to apply recursively to all sub-terms of the ReferenceType, such as its element type and any type arguments.

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

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

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

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

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

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

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

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

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

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

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

- *The name of a field, parameter, or local variable may be used as an expression (15.14.1 ⇗).*
- *The name of a method may appear in an expression only as part of a method invocation expression (15.12 ⇗).*
- *The name of a class or interface may appear in an expression only as part of a class literal (15.8.2 ⇗), a qualified `this` expression (15.8.4 ⇗), a class instance creation expression (15.9 ⇗), an array creation expression (15.10.1 ⇗), a cast expression (15.16 ⇗), an `instanceof` expression (15.20.2 ⇗), an enum constant (8.9 ⇗), or as part of a qualified name for a field or method.*
- *The name of a package may appear in an expression only as part of a qualified name for a class or interface.*

Chapter 7: Packages and Modules

7.5 Import Declarations

An *import declaration* allows a named class, interface, or `static` member to be referred to by a simple name (6.2 ⇗) that consists of a single identifier.

Without the use of an appropriate import declaration, a reference to a class or interface declared in another package, or a reference to a `static` member of another class or interface, would typically need to use a fully qualified name (6.7 ⇗).

ImportDeclaration:
  SingleTypeImportDeclaration
  TypeImportOnDemandDeclaration
  SingleStaticImportDeclaration
  StaticImportOnDemandDeclaration
  SingleModuleImportDeclaration

- A single-type-import declaration (7.5.1 ⇗) imports a single named class or interface, by mentioning its canonical name (6.7 ⇗).
- A type-import-on-demand declaration (7.5.2 ⇗) imports all the accessible classes and interfaces of a named package, class, or interface as needed, by mentioning the canonical name of the package, class, or interface.
- A single-static-import declaration (7.5.3 ⇗) imports all accessible `static` members with a given name from a class or interface, by giving its canonical name.

- A static-import-on-demand declaration (7.5.4 ⇗) imports all accessible `static` members of a named class or interface as needed, by mentioning the canonical name of the class or interface.
- A single-module-import declaration (7.5.5) imports all the accessible classes and interfaces of the packages exported by a given module, as needed.

The scope and shadowing of a class, interface, or member imported by these declarations is specified in 6.3 and 6.4 ⇗.

An `import` declaration makes classes, interfaces, or members available by their simple names only within the compilation unit that actually contains the `import` declaration. The scope of the class(es), interface(s), or member(s) introduced by an `import` declaration specifically does not include other compilation units in the same package, other `import` declarations in the current compilation unit, or a `package` declaration in the current compilation unit (except for the annotations of a `package` declaration).

7.5.5 Single-Module-Import Declarations

A *single-module-import declaration* allows all `public` top level classes and interfaces of the packages exported by a named module to be imported as needed.

SingleModuleImportDeclaration:
`import` `module` *ModuleName* `;`

A single-module-import declaration `import module M;` imports, on demand, all the `public` top level classes and interfaces in the following packages:

1. The packages exported by the module `M` to the current module.
2. The packages exported by the modules that are read by the current module due to reading the module `M`. This allows a program to use the API of a module, which might refer to classes and interfaces from other modules, without having to import all those other modules.

It is a compile-time error if the module *ModuleName* is not read by the current module (7.3 ⇗).

The modules read by the current module are given by the result of resolution, as described in the `java.lang.module` package specification (7.3 ⇗).

Two or more single-module-import declarations in the same compilation unit may name the same module. All but one of these declarations are considered redundant; the effect is as if that module was imported only once.

A single-module-import declaration can be used in any source file. It is not required for the source file to be part of a module. For example, modules `java.base` and `java.sql` are part of the standard Java runtime, so they can be imported by programs which are not themselves developed as modules.

It is sometimes useful to import a module that does not export any packages. This is because the module may transitively require other modules that do export packages. For example, the `java.se` module does not export any packages, but it requires a number of modules transitively, so the effect of the single-module-import declaration `import module java.se;` is to import the packages which are exported by those modules (and so on, recursively).

Example 7.5.5-1. Single-Module-Import in Ordinary Compilation Units

Modules allow a set of packages to be grouped together for reuse under a single name, and the exported packages of a module are intended to form a cohesive and coherent API. Single-module-

import declarations allow the developer to import all the packages exported by a module in one go, simplifying the reuse of modular libraries. For example:

```
import module java.xml;
```

causes the simple names of the `public` top level classes and interfaces declared in all packages exported by module `java.xml` to be available within the class and interface declarations of the compilation unit. Thus, the simple name `XPath` refers to the interface `XPath` of the package `javax.xml.xpath` exported by the module `java.xml` in all places in the compilation unit where that class declaration is not shadowed or obscured.

Assume the following compilation unit associated with module `M0`:

```
package q;
import module M1;    // What does this import?
class C { ... }
```

where module `M0` has the following declaration:

```
module M0 { requires M1; }
```

The meaning of the single-module-import declaration `import module M1;` depends on the exports of `M1` and any modules that `M1` requires transitively. Consider as an example:

```
module M1 {
    exports p1;
    exports p2 to M0;
    exports p3 to M3;
    requires transitive M4;
    requires M5;
}

module M3 { ... }

module M4 { exports p10; }

module M5 { exports p11; }
```

The effect of the single-module-import declaration `import module M1;` is then:

1. *Import the `public` top level classes and interfaces from package `p1`, since `M1` exports `p1` to everyone;*
2. *Import the `public` top level classes and interfaces from package `p2`, since `M1` exports `p2` to `M0`, the module with which the compilation unit is associated; and*
3. *Import the `public` top level classes and interfaces from package `p10`, since `M1` requires transitively `M4`, which exports `p10`.*

Nothing from packages `p3` or `p11` is imported by the compilation unit.

Single-module-import declarations may appear in a source file containing only a package declaration. Such files are typically called `package-info.java` and are used as the sole repository for package-level annotations and documentation (7.4.1 ↗).

Example 7.5.5-2. Single-Module-Import in Modular Compilation Units

Import declarations can also appear in a modular compilation unit. The following modular compilation unit uses a single-module-import declaration, allowing the simple name of the interface `Driver` associated with module `java.sql` to be used in the `provides` directive:

```
import module java.sql;
```

```
module com.myDB.core {
    exports ...
    requires transitive java.sql;
    provides Driver with com.myDB.greatDriver;
}
```

It is possible for a modular compilation unit that declares a module `M` to also import the module `M`. In the following example, this means that the simple name of a class `C` can be used in a `uses` directive:

```
import module M;
module M {
    ...
    exports p;
    ...
    uses C;
    ...
}
```

where the package `p` exported by module `M` is declared as follows:

```
package p;
class C { ... }
```

Without the single-module-import declaration, the qualified name of the class `C` would need to be used in the `uses` directive.

Suppose a module declaration as follows:

```
module M2 {
    requires java.se;
    exports p2;
    ...
}
```

where the package `p2` exported by `M2` is declared as follows:

```
package p2;
import module java.xml;
class MyClass {
    ...
}
```

Even though the module `M2` does not directly express a dependency on the module `java.xml`, the import of module `java.xml` is still correct as the resolution process will determine that the module `java.xml` is read by module `M2`.

Example 7.5.5-3. Ambiguous Imports

Clearly importing multiple modules could lead to name ambiguities, for example:

```
import module java.base;
import module java.desktop;

...
List l = ...        // Error - Ambiguous name!
...
```

The module `java.base` exports the package `java.util`, which has a `public List` interface. The module `java.desktop` exports the package `java.awt`, which a `public List` class. Having imported both modules, the use of the simple name `List` is clearly ambiguous and results in a compile-time *error.*

However, just importing a single module can also lead to a name ambiguity, for example:

```
import module java.desktop;

...
Element e = ...      // Error - Ambiguous name!
...
```

The module `java.desktop` exports packages, `javax.swing.text` and `javax.swing.text.html.parser`, which have a `public Element` interface and a `public Element` class, respectively. Thus the use of the simple name `Element` is ambiguous and results in a compile-time error.

A single-type-import declaration can be used to resolve a name ambiguity. The earlier example where the simple name `List` is ambiguous can be resolved as follows:

```
import module java.base;
import module java.desktop;

import java.util.List;   // Resolving the ambiguity of the simple name List

...
List l = ...             // Ok - List is resolved to java.util.List
...
```

7.7 Module Declarations

7.7.1 Dependences

The `requires` directive specifies the name of a module on which the current module has a dependence.

A `requires` directive must not appear in the declaration of the `java.base` module, or a compile-time error occurs, because it is the primordial module and has no dependences (8.1.4).

If the declaration of a module does not express a dependence on the `java.base` module, and the module is not itself `java.base`, then the module has an implicitly declared dependence on the `java.base` module.

The `requires` keyword may be followed by the modifier `transitive`. This causes any module which `requires` the current module to have an implicitly declared dependence on the module specified by the `requires transitive` directive.

The `requires` keyword may be followed by the modifier `static`. This specifies that the dependence, while mandatory at compile time, is optional at run time.

If the declaration of a module expresses a dependence on the `java.base` module, and the module is not itself `java.base`, then it is a compile-time error if ~~a~~ the `static` modifier appears after the `requires` keyword.

It is a compile-time error if more than one `requires` directive in a module declaration specifies the same module name.

It is a compile-time error if resolution, as described in the `java.lang.module` package specification, with the current module as the only root module, fails for any of the reasons described in the `java.lang.module` package specification.

For example, if a `requires` directive specifies a module that is not observable, or if the current module directly or indirectly expresses a dependence on itself.

If resolution succeeds, then its result specifies the modules that are read by the current module. The modules read by the current module determine which ordinary compilation units are visible to the current module (7.3 ⇗). The types declared in those ordinary compilation units (and *only* those ordinary compilation units) may be accessible to code in the current module (6.6 ⇗).

The Java SE Platform distinguishes between named modules that are explicitly declared (that is, with a module declaration) and named modules that are implicitly declared (that is, automatic modules). However, the Java programming language does not surface the distinction: `requires` directives refer to named modules without regard for whether they are explicitly declared or implicitly declared.

While automatic modules are convenient for migration, they are unreliable in the sense that their names and exported packages may change when their authors convert them to explicitly declared modules. A Java compiler is encouraged to issue a warning if a `requires` directive refers to an automatic module. An especially strong warning is recommended if the `transitive` modifier appears in the directive.

Example 7.1.1-1. Resolution of `requires transitive` directives

Suppose there are four module declarations as follows:

```
module m.A {
    requires m.B;
}

module m.B {
    requires transitive m.C;
}

module m.C {
    requires transitive m.D;
}

module m.D {
    exports p;
}
```

where the package `p` exported by `m.D` is declared as follows:

```
package p;
public class Point {}
```

and where a package `client` in module `m.A` refers to the `public` type `Point` in the exported package `p`:

```
package client;
import p.Point;
public class Test {
    public static void main(String[] args) {
        System.out.println(new Point());
    }
}
```

The modules may be compiled as follows, assuming that the current directory has one subdirectory per module, named after the module it contains:

```
javac --module-source-path . -d . --module m.D
javac --module-source-path . -d . --module m.C
javac --module-source-path . -d . --module m.B
javac --module-source-path . -d . --module m.A
```

The program `client.Test` may be run as follows:

```
java --module-path . --module m.A/client.Test
```

The reference from code in `m.A` to the exported `public` type `Point` in `m.D` is legal because `m.A` reads `m.D`, and `m.D` exports the package containing `Point`. Resolution determines that `m.A` reads `m.D` as follows:

- *`m.A requires m.B` and therefore reads `m.B`.*
- *Since `m.A` reads `m.B`, and since `m.B requires transitive m.C`, resolution determines that `m.A` reads `m.C`.*
- *Then, since `m.A` reads `m.C`, and since `m.C requires transitive m.D`, resolution determines that `m.A` reads `m.D`.*

In effect, a module may read another module through multiple levels of dependence, in order to support arbitrary amounts of refactoring. Once a module is released for someone to reuse (via `requires`), the module's author has committed to its name and API but is free to refactor its content into other modules which the original module reuses (via `requires transitive`) for the benefit of consumers. In the example above, package `p` may have been exported originally by `m.B` (thus, `m.A requires m.B`) but refactoring has caused some of `m.B`'s content to move into `m.C` and `m.D`. By using a chain of `requires transitive` directives, the family of `m.B`, `m.C`, and `m.D` can preserve access to package `p` for code in `m.A` without forcing any changes to the `requires` directives of `m.A`. Note that package `p` in `m.D` is not "re-exported" by `m.C` and `m.B`; rather, `m.A` is made to read `m.D` directly.