This specification is not final and is subject to change. Use is subject to license terms.



# Unnamed Classes and Instance `main` Methods (Preview)

***Changes to the Java® Language Specification • Version 21-internal-adhoc.gbierman.20230524***



This document describes changes to the Java Language Specification ⇗ to support *Unnamed Classes and Instance `main` Methods*, which is a preview feature of Java SE 21. See JEP 445 ⇗ for an overview of the feature.

A companion document describes the changes needed to the Java Virtual Machine Specification ⇗ to support Unnamed Classes and Instance `main` Methods.

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:*

*2023-05-24: Misc editorial changes.*

*2023-05-19: 12.1.4 Change to the definition of a candidate `main` method, reflecting change in the JEP.*

*2023-05-15:*

- *Added reference to companion JVMS document.*
- *12.1.4 Improvements to explanation of candidate `main` methods.*

*2023-05-02: First draft released*

# Chapter 6: Names

## 6.7 Fully Qualified Names and Canonical Names

Every primitive type, named package, named top level class, and top level interface has a *fully qualified name*:

- The fully qualified name of a primitive type is the keyword for that primitive type, namely `byte`, `short`, `char`, `int`, `long`, `float`, `double`, or `boolean`.
- The fully qualified name of a named package that is not a subpackage of a named package is its simple name.
- The fully qualified name of a named package that is a subpackage of another named package consists of the fully qualified name of the containing package, followed by ".", followed by the simple (member) name of the subpackage.
- The fully qualified name of a named top level class or top level interface that is declared in an unnamed package is the simple name of the class or interface.
- The fully qualified name of a top level class or top level interface that is declared in a named package consists of the fully qualified name of the package, followed by ".", followed by the simple name of the class or interface.

Each member class, member interface, and array type *may* have a fully qualified name:

- A member class or member interface *M* of another class or interface *C* has a fully qualified name if and only if *C* has a fully qualified name.

  In that case, the fully qualified name of *M* consists of the fully qualified name of *C*, followed by ".", followed by the simple name of *M*.

- An array type has a fully qualified name if and only if its element type has a fully qualified name.

  In that case, the fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "`[]`".

A local class, local interface, or anonymous class does not have a fully qualified name.

Every primitive type, named package, named top level class, and top level interface has a *canonical name*:

- For every primitive type, named package, top level class, and top level interface, the canonical name is the same as the fully qualified name.

Each member class, member interface, and array type *may* have a canonical name:

- A member class or member interface *M* declared in another class or interface *C* has a canonical name if and only if *C* has a canonical name.

  In that case, the canonical name of *M* consists of the canonical name of *C*, followed by ".", followed by the simple name of *M*.

- An array type has a canonical name if and only if its component type has a canonical name.

In that case, the canonical name of the array type consists of the canonical name of the component type of the array type followed by "`[]`".

A local class, local interface, or anonymous class does not have a canonical name.

**Example 6.7-1. Fully Qualified Names**

- *The fully qualified name of the type `long` is "`long`".*
- *The fully qualified name of the package `java.lang` is "`java.lang`" because it is subpackage `lang` of package `java`.*
- *The fully qualified name of the class `Object`, which is defined in the package `java.lang`, is "`java.lang.Object`".*
- *The fully qualified name of the interface `Enumeration`, which is defined in the package `java.util`, is "`java.util.Enumeration`".*
- *The fully qualified name of the type "array of `double`" is "`double[]`".*
- *The fully qualified name of the type "array of array of array of array of `String`" is "`java.lang.String[][][][]`".*

*In the code:*

```
package points;
class Point    { int x, y; }
class PointVec { Point[] vec; }
```

*the fully qualified name of the type `Point` is "`points.Point`"; the fully qualified name of the type `PointVec` is "`points.PointVec`"; and the fully qualified name of the type of the field `vec` of class `PointVec` is "`points.Point[]`".*

**Example 6.7-2. Fully Qualified Names v. Canonical Name**

*The difference between a fully qualified name and a canonical name can be seen in code such as:*

```
package p;
class O1 { class I {} }
class O2 extends O1 {}
```

*Both `p.O1.I` and `p.O2.I` are fully qualified names that denote the member class `I`, but only `p.O1.I` is its canonical name.*

# Chapter 7: Packages and Modules

## 7.3 Compilation Units

*CompilationUnit* is the goal symbol (2.1 ⇗) for the syntactic grammar (2.3 ⇗) of Java programs. It is defined by the following production:

*CompilationUnit:*
*OrdinaryCompilationUnit*
*UnnamedClassCompilationUnit*
*ModularCompilationUnit*

*OrdinaryCompilationUnit:*
[*PackageDeclaration*] {*ImportDeclaration*} {*TopLevelClassOrInterfaceDeclaration*}

*UnnamedClassCompilationUnit:*

*{ImportDeclaration} {ClassMemberDeclarationNoMethod} MethodDeclaration*
*{ClassMemberDeclaration}*

*ClassMemberDeclarationNoMethod:*
*FieldDeclaration*
*ClassDeclaration*
*InterfaceDeclaration*
*;*

*ModularCompilationUnit:*
*{ImportDeclaration} ModuleDeclaration*

An *ordinary compilation unit* consists of three parts, each of which is optional:

- A `package` declaration (7.4), giving the fully qualified name (6.7) of the package to which the compilation unit belongs.

  A compilation unit that has no `package` declaration is part of an unnamed package (7.4.2).
- `import` declarations (7.5) that allow classes and interface from other packages, and `static` members of classes and interfaces, to be referred to using their simple names.
- Top level declarations of classes and interfaces (7.6).

An *unnamed class compilation unit* consists of:

- zero or more `import` declarations that allow classes and interface from other packages, and `static` members of classes and interfaces, to be referred to using their simple names.
- The declarations of the members (8.2) of the implicitly declared top level class, at least one of which is a method declaration (8.4).

  *This means that the following compilation unit is unambiguously an ordinary compilation unit:*

  ```
  import p.*;
  class Test { ... }
  ```

  *whereas the following is unambiguously an unnamed class compilation unit:*

  ```
  import p.*;
  static void main(){ ... }
  class Test { ... }
  ```

An unnamed class compilation unit implicitly declares a class that satisfies the following properties:

- It is always a top level class (7.6).
- It is always an *unnamed* class (it has no canonical or fully qualified name (6.7)).
- It is never `abstract` (8.1.1.1).
- It is always `final` (8.1.1.2).
- It is always a member of an unnamed package (7.4.2) and has package access.
- Its direct superclass type is always `Object` (8.1.4).
- It never has any direct superinterface types (8.1.5).
- The body of the class contains every *ClassMemberDeclaration* (these are declarations of

fields (8.3), methods (8.4), member classes (8.5), and member interfaces (9.1.1.3)) from the unnamed class compilation unit. It is not possible for an unnamed class compilation unit to declare an instance initializer (8.6), static initializer (8.7), or constructor (8.8).

- It has an implicitly declared default constructor (8.8.9).

All members of this class, including any implicitly declared members, are subject to the usual rules for member declarations in a class.

It is a compile-time error if this class does not declare a candidate `main` method (12.1.4).

*Note that an unnamed package may have multiple unnamed classes as members.*

A *modular compilation unit* consists of a `module` declaration (7.7), optionally preceded by `import` declarations. The `import` declarations allow classes and interfaces from packages in this module and other modules, as well as `static` members of classes and interfaces, to be referred to using their simple names within the `module` declaration.

Every compilation unit implicitly imports every `public` class or interface declared in the predefined package `java.lang`, as if the declaration `import java.lang.*;` appeared at the beginning of each compilation unit immediately after any `package` declaration. As a result, the names of all those classes and interfaces are available as simple names in every compilation unit.

The host system determines which compilation units are *observable*, except for the compilation units in the predefined package `java` and its subpackages `lang` and `io`, which are all always observable.

Each observable compilation unit may be *associated* with a module, as follows:

- The host system may determine that an observable ordinary compilation unit is associated with a module chosen by the host system, except for (i) the ordinary compilation units in the predefined package `java` and its subpackages `lang` and `io`, which are all associated with the `java.base` module, and (ii) any ordinary compilation unit in an unnamed package, which is associated with a module as specified in 7.4.2.
- The host system must determine that an observable modular compilation unit is associated with the module declared by the modular compilation unit.

*The observability of a compilation unit influences the observability of its package (7.4.3), while the association of an observable compilation unit with a module influences the observability of that module (7.7.6).*

When compiling the modular and ordinary compilation units associated with a module *M*, the host system must respect the dependences specified in *M*'s declaration. Specifically, the host system must limit the ordinary compilation units that would otherwise be observable, to only those that are *visible to M*. The ordinary compilation units that are visible to *M* are the observable ordinary compilation units associated with the modules that are *read by M*. The modules read by *M* are given by the result of *resolution*, as described in the `java.lang.module` package specification, with *M* as the only root module. The host system must perform resolution to determine the modules read by *M*; it is a compile-time error if resolution fails for any of the reasons described in the `java.lang.module` package specification.

*The readability relation is reflexive, so M reads itself, and thus all of the modular and ordinary compilation units associated with M are visible to M.*

*The modules read by M drive the packages that are uniquely visible to M (7.4.3), which in turn drives*

*both the top level packages in scope and the meaning of package names for code in the modular and ordinary compilation units associated with M (6.3 ⇗, 6.5.3 ⇗, 6.5.5 ⇗).*

*The rules above ensure that package and type names used in annotations in a modular compilation unit (in particular, annotations applied to the module declaration) are interpreted as if they appeared in an ordinary compilation unit associated with the module.*

Classes and interfaces declared in different ordinary compilation units can refer to each other, circularly. A Java compiler must arrange to compile all such classes and interfaces at the same time.

# Chapter 8: Classes

A class declaration defines a new class and describes how it is implemented (8.1).

A *top level class* (7.6 ⇗) is a class declared directly in a compilation unit.

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

Some kinds of nested class are an *inner class* (8.1.3), which is a class that can refer to enclosing class instances, local variables, and type variables.

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

A *record class* (8.10 ⇗) is a class declared with abbreviated syntax that defines a simple aggregate of values.

For very small programs and casual development, a top level class can be *unnamed*, that is, it has no canonical or fully qualified name (6.7). An unnamed class is never declared explicitly, but rather is declared implicitly by an unnamed class compilation unit (7.3). An unnamed top level class can be the initial class of the program (12.1.4) but cannot be referred to by a name from any source code, including its own.

This chapter discusses the common semantics of all classes. Details that are specific to particular kinds of classes are discussed in the sections dedicated to these constructs.

A class may be declared `public` (8.1.1 ⇗) so it can be referred to from code in any package of its module and potentially from code in other modules.

A class may be declared `abstract` (8.1.1.1 ⇗), and must be declared `abstract` if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. The degree to which a class can be extended can be controlled explicitly (8.1.1.2 ⇗): it may be declared `sealed` to limit its subclasses, or it may be declared `final` to ensure no subclasses. Each class except `Object` is an extension of (that is, a subclass of) a single existing class (8.1.4 ⇗) and may implement interfaces (8.1.5 ⇗).

A class may be *generic* (8.1.2 ⇗), that is, its declaration may introduce type variables whose bindings differ among different instances of the class.

Class declarations may be decorated with annotations (9.7 ⇗) just like any other kind of declaration.

The body of a class declares members (fields, methods, classes, and interfaces), instance and static initializers, and constructors (8.1.7 ⇗). The scope (6.3 ⇗) of a member (8.2 ⇗) is the entire body of the declaration of the class to which the member belongs. Field, method, member class,

member interface, and constructor declarations may include the access modifiers `public`, `protected`, or `private` (6.6 ⇗). The members of a class include both declared and inherited members (8.2 ⇗). Newly declared fields can hide fields declared in a superclass or superinterface. Newly declared member classes and member interfaces can hide member classes and member interfaces declared in a superclass or superinterface. Newly declared methods can hide, implement, or override methods declared in a superclass or superinterface.

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

Member class declarations (8.5 ⇗) describe nested classes that are members of the surrounding class. Member classes may be `static`, in which case they have no access to the instance variables of the surrounding class; or they may be inner classes.

Member interface declarations (8.5 ⇗) describe nested interfaces that are members of the surrounding class.

Method declarations (8.4 ⇗) describe code that may be invoked by method invocation expressions (15.12 ⇗). A class method is invoked relative to the class; an instance method is invoked with respect to some particular object that is an instance of a class. A method whose declaration does not indicate how it is implemented must be declared `abstract`. A method may be declared `final` (8.4.3.3 ⇗), in which case it cannot be hidden or overridden. A method may be implemented by platform-dependent `native` code (8.4.3.4 ⇗). A `synchronized` method (8.4.3.6 ⇗) automatically locks an object before executing its body and automatically unlocks the object on return, as if by use of a `synchronized` statement (14.19 ⇗), thus allowing its activities to be synchronized with those of other threads (17 ⇗).

Method names may be overloaded (8.4.9 ⇗).

Instance initializers (8.6 ⇗) are blocks of executable code that may be used to help initialize an instance when it is created (15.9 ⇗).

Static initializers (8.7 ⇗) are blocks of executable code that may be used to help initialize a class.

Constructors (8.8 ⇗) are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded (8.8.8 ⇗).

## 8.1 Class Declarations

A *class declaration* specifies a class.

There are three kinds of class declarations: *normal class declarations*, *enum declarations* (8.9 ⇗), and *record declarations* (8.10 ⇗).

*ClassDeclaration:*
*NormalClassDeclaration*
*EnumDeclaration*
*RecordDeclaration*

*NormalClassDeclaration:*
{*ClassModifier*} `class` *TypeIdentifier* [*TypeParameters*]
[*ClassExtends*] [*ClassImplements*] [*ClassPermits*] *ClassBody*

A class is also implicitly declared by an unnamed class compilation unit (7.3), a class instance creation expression (15.9.5 ⇗) and an enum constant that ends with a class body (8.9.1 ⇗).

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

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

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

# Chapter 9: Interfaces

An interface declaration defines a new interface that can be implemented by one or more classes. Programs can use interfaces to provide a common supertype for otherwise unrelated classes, and to make it unnecessary for related classes to share a common `abstract` superclass.

Interfaces have no instance variables, and typically declare one or more `abstract` methods; otherwise unrelated classes can implement an interface by providing implementations for its `abstract` methods. Interfaces may not be directly instantiated.

A *top level interface* (7.6 ⇗) is an interface declared directly in a compilation unit.

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

An *annotation interface* (9.6 ⇗) is an interface declared with distinct syntax, intended to be implemented by reflective representations of *annotations* (9.7 ⇗).

This chapter discusses the common semantics of all interfaces. Details that are specific to particular kinds of interfaces are discussed in the sections dedicated to these constructs.

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

A class may be declared to *directly implement* one or more interfaces (8.1.5 ⇗), meaning that any instance of the class implements all the `abstract` methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct superclasses and direct superinterfaces do. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing a superclass.

Unlike a class, an interface cannot be declared `final`. However, an interface may be declared `sealed` (9.1.1.4 ⇗) to limit its subclasses and subinterfaces.

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

*Note that, unlike classes, it is not possible to declare, even implicitly, an unnamed top level interface (7.3).*

# Chapter 12: Execution

This chapter specifies activities that occur during execution of a program. It is organized around the life cycle of the Java Virtual Machine and of the classes, interfaces, and objects that form a program.

The Java Virtual Machine starts up by loading a specified class or interface, then invoking ~~the~~ a method `main` in this specified class or interface. Section 12.1 outlines the loading, linking, and initialization steps involved in executing `main`, as an introduction to the concepts in this chapter. Further sections specify the details of loading (12.2 ⇗), linking (12.3 ⇗), and initialization (12.4 ⇗).

The chapter continues with a specification of the procedures for creation of new class instances (12.5 ⇗); and finalization of class instances (12.6 ⇗). It concludes by describing the unloading of classes (12.7 ⇗) and the procedure followed when a program exits (12.8 ⇗).

## 12.1 Java Virtual Machine Startup

The Java Virtual Machine starts execution by invoking ~~the~~ a method `main` of some specified class or interface. If this `main` method has a formal parameter, it is passed ~~passing it~~ a single argument which is an array of strings. ~~In the examples in this specification, this first class is typically called `Test`.~~

The precise semantics of Java Virtual Machine startup are given in Chapter 5 of *The Java Virtual Machine Specification, Java SE 21 Edition*. Here we present an overview of the process from the viewpoint of the Java programming language.

The manner in which the initial class or interface is specified to the Java Virtual Machine is beyond the scope of this specification, but it is typical, in host environments that use command lines, for the fully qualified name of the initial class or interface to be specified as a command line argument and for following command line arguments to be used as strings to be provided as the argument to the method `main`. If the original compilation unit was an unnamed class compilation unit (7.3), then the name of the file that contained the compilation unit is typically used to specify the name of the initial class or interface.

> *For example, in a UNIX implementation, the command line:*
>
> ```
> java Test reboot Bob Dot Enzo
> ```
>
> *will typically start a Java Virtual Machine by invoking method `main` of class `Test` (a class in an unnamed package), passing it an argument array containing the four strings `"reboot"`, `"Bob"`, `"Dot"`, and `"Enzo"`.*

*Whereas if the file `HelloWorld.java` contained the following unnamed class compilation unit:*

```
void main() {
    System.out.println("Hello, World!");
}
```

*which has been compiled, then the command line:*

```
java HelloWorld
```

*will typically start a Java Virtual Machine by invoking the `main` method of the implicitly declared unnamed class (7.3) producing the output:*

```
Hello, World!
```

We now outline the steps the Java Virtual Machine may take to execute ~~`Test`~~ the initial class or interface, as an example of the loading, linking, and initialization processes that are described further in later sections.

### 12.1.1 Load the Initial Class or Interface ~~`Test`~~

The initial attempt to execute ~~the~~ a method `main` of the initial class or interface ~~`Test`~~ discovers that ~~the class `Test`~~ it is not loaded - that is, that the Java Virtual Machine does not currently

contain a binary representation for this class or interface. The Java Virtual Machine then uses a class loader to attempt to find such a binary representation. If this process fails, then an error is thrown. This loading process is described further in 12.2 ⇗.

### 12.1.2 Link the Initial Class or Interface ~~Test~~: Verify, Prepare, (Optionally) Resolve

After the class or interface ~~Test~~ is loaded, it must be initialized before a method `main` can be invoked. And ~~Test~~, like all classes and interfaces, it must be linked before it is initialized. Linking involves verification, preparation, and (optionally) resolution. Linking is described further in 12.3 ⇗.

Verification checks that the loaded representation of the class or interface ~~Test~~ is well-formed, with a proper symbol table. Verification also checks that the code that implements the class or interface ~~Test~~ obeys the semantic requirements of the Java programming language and the Java Virtual Machine. If a problem is detected during verification, then an error is thrown. Verification is described further in 12.3.1 ⇗.

Preparation involves allocation of static storage and any data structures that are used internally by the implementation of the Java Virtual Machine, such as method tables. Preparation is described further in 12.3.2 ⇗.

Resolution is the process of checking symbolic references from the class or interface ~~Test~~ to other classes and interfaces, by loading the other classes and interfaces that are mentioned and checking that the references are correct.

The resolution step is optional at the time of initial linkage. An implementation may resolve symbolic references from a class or interface that is being linked very early, even to the point of resolving all symbolic references from the classes and interfaces that are further referenced, recursively. (This resolution may result in errors from these further loading and linking steps.) This implementation choice represents one extreme and is similar to the kind of "static" linkage that has been done for many years in simple implementations of the C language. (In these implementations, a compiled program is typically represented as an "`a.out`" file that contains a fully-linked version of the program, including completely resolved links to library routines used by the program. Copies of these library routines are included in the "`a.out`" file.)

An implementation may instead choose to resolve a symbolic reference only when it is actively used; consistent use of this strategy for all symbolic references would represent the "laziest" form of resolution. In this case, if the class or interface ~~Test~~ had several symbolic references to another class, then the references might be resolved one at a time, as they are used, or perhaps not at all, if these references were never used during execution of the program.

The only requirement on when resolution is performed is that any errors detected during resolution must be thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error. Using the "static" example implementation choice described above, loading and linkage errors could occur before the program is executed if they involved a class or interface mentioned in the initial class or interface ~~Test~~ or any of the further, recursively referenced, classes and interfaces. In a system that implemented the "laziest" resolution, these errors would be thrown only when an incorrect symbolic reference is actively used.

The resolution process is described further in 12.3.3 ⇗.

### 12.1.3 Initialize the Initial Class or Interface ~~Test~~: Execute Initializers

In our continuing example, the Java Virtual Machine is still trying to execute ~~the~~ a method `main` of the initial class or interface~~Test~~. This is permitted only if the class has been initialized (12.4.1 ⇗).

Initialization consists of execution of any class variable initializers and static initializers of the initial class or interface ~~Test~~, in textual order. But before it ~~Test~~ can be initialized, its direct superclass must be initialized, as well as the direct superclass of its direct superclass, and so on, recursively. In the simplest case, the initial class or interface ~~Test~~ has `Object` as its implicit direct superclass; if class `Object` has not yet been initialized, then it must be initialized before the initial class or interface ~~Test~~ is initialized. Class `Object` has no superclass, so the recursion terminates here.

If the initial class or interface ~~Test~~ has another class `Super` as its superclass, then `Super` must be initialized before the initial class or interface~~Test~~. This requires loading, verifying, and preparing `Super` if this has not already been done and, depending on the implementation, may also involve resolving the symbolic references from `Super` and so on, recursively.

Initialization may thus cause loading, linking, and initialization errors, including such errors involving other classes and interfaces.

The initialization process is described further in 12.4 ⇗.

### 12.1.4 Invoke ~~Test.main~~ a `main` method

Finally, after completion of the initialization for the initial class or interface ~~Test~~ (during which other consequential loading, linking, and initializing may have occurred), ~~the~~ a `main` method ~~main~~ of the initial class or interface ~~Test~~ is invoked.

~~The method `main` must be declared `public`, `static`, and `void`. It must specify a formal parameter (8.4.1 ⇗) whose declared type is array of `String`. Therefore, either of the following declarations is acceptable:~~

```
~~public static void main(String[] args)~~

~~public static void main(String... args)~~
```

A method of the initial class or interface is a *candidate* if it is named `main` and one of the following applies:

- It is a `static` method, declared in the initial class or interface, with a `void` result, with `public`, `protected` or package access, and with either no formal parameters or a single formal parameter whose declared type is an array of `String`.

  *Note that such a `static` method may not be inherited from a superclass of the initial class or interface.*

- It is an instance method, declared in or inherited by the initial class or interface, with a `void` result, with `public`, `protected` or package access, and with either no formal parameters or a single formal parameter whose declared type is an array of `String`; and where, moreover, the initial class or interface is not an inner class.

*Note that a candidate `main` method may have a `throws` clause (8.4.6 ⇗).*

*The permitted signature of a `main` method expanded significantly in Java SE 21. Prior to that, the only variation possible in the signature of `main` was `String[]` versus `String...` for the type of the single formal parameter. In Java SE 21 and above, `main` can have one of twelve possible signatures: six `static` and six non-`static`. This number doubles to 24 if `String[]` is distinguished from `String...` in the type of the single formal parameter.*

*Note that it is not a compile-time error if the initial class or interface counts more than one candidate `main` method among its members.*

*The presence of a `main` method in a class or interface may not be immediately apparent because a non-`static` `main` method may be inherited. For example, a default method in an interface is an instance method (9.4), so may be a candidate when inherited by a class that implements the interface. Development tools are encouraged to highlight when a class or interface has a member `main` method that could serve as the start of the program.*

*A behavioral change was made in Java SE 21, whereby an inherited `static` `main` method is no longer considered a candidate method. Any existing initial class or interface whose only `main` method is both `static` and inherited will need to be refactored to continue to serve as the start of the program.*

A `main` method of the initial class or interface is invoked, as if by application of the following rules:

- If there is a `static` candidate method with a formal parameter then this method is invoked, passing the argument array (12.1).
- Otherwise, if there is a `static` candidate method with no formal parameters then this method is invoked.
- Otherwise, if there is a instance candidate method with a formal parameter, then this method is invoked, passing the argument array, on an instance of the initial class created by using a constructor with no formal parameters and either `public`, `protected`, or package access.
- Otherwise, if there is a instance candidate method with no formal parameters, then this method is invoked on an instance of the initial class created by using a constructor with no formal parameters and either `public`, `protected`, or package access.

*The behavior of an implementation if there is no candidate method to invoke, or if there is no suitable constructor in the initial class when invoking an instance candidate method, is beyond the scope of this specification.*

# Chapter 13: Binary Compatibility

## 13.1 The Form of a Binary

Programs must be compiled either into the `class` file format specified by *The Java Virtual Machine Specification, Java SE 20 Edition*, or into a representation that can be mapped into that format by a class loader written in the Java programming language.

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

1. The class or interface must be named by its *binary name*, which must meet the following constraints:
    - The binary name of a named top level class or interface (7.6) is its canonical name (6.7). The binary name of an unnamed top level class (7.3) is any valid identifier (3.8).

      *In simple implementations of the Java SE Platform, where compilation units are stored in files, the binary name of an unnamed top level class would typically be the name of the file containing the unnamed top level class compilation unit (7.3) minus any extension (such as `.java` or `.jav`).*

    - The binary name of a member class or interface (8.5, 9.5) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by the

simple name of the member.

- The binary name of a local class or interface (14.3 ⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by a non-empty sequence of digits, followed by the simple name of the local class.
- The binary name of an anonymous class (15.9.5 ⇗) consists of the binary name of its immediately enclosing class or interface, followed by `$`, followed by a non-empty sequence of digits.
- The binary name of a type variable declared by a generic class or interface (8.1.2 ⇗, 9.1.2 ⇗) is the binary name of its immediately enclosing class or interface, followed by `$`, followed by the simple name of the type variable.
- The binary name of a type variable declared by a generic method (8.4.4 ⇗) is the binary name of the class or interface declaring the method, followed by `$`, followed by the descriptor of the method (JVMS §4.3.3), followed by `$`, followed by the simple name of the type variable.
- The binary name of a type variable declared by a generic constructor (8.8.4 ⇗) is the binary name of the class declaring the constructor, followed by `$`, followed by the descriptor of the constructor (JVMS §4.3.3), followed by `$`, followed by the simple name of the type variable.

2. A reference to another class or interface must be symbolic, using the binary name of the class or interface.
3. A reference to a field that is a constant variable (4.12.4 ⇗) must be resolved at compile time to the value *V* denoted by the constant variable's initializer.

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

   If such a field is non-`static`, then no reference to the field should be present in the code in a binary file, except in the class containing the field. (It will be a class rather than an interface, since an interface has only `static` fields.) The class should have code to set the field's value to *V* during instance creation (12.5 ⇗).
4. Given a legal expression denoting a field access in a class *C*, referencing a field named *f* that is not a constant variable and is declared in a (possibly distinct) class or interface *D*, we define the *qualifying class or interface of the field reference* as follows:
   - If the expression is referenced by a simple name, then if *f* is a member of the current class or interface, *C*, then let *Q* be *C*. Otherwise, let *Q* be the innermost lexically enclosing class or interface declaration of which *f* is a member. In either case, *Q* is the qualifying class or interface of the reference.
   - If the reference is of the form *TypeName*`.`*f*, where *TypeName* denotes a class or interface, then the class or interface denoted by *TypeName* is the qualifying class or interface of the reference.
   - If the expression is of the form *ExpressionName*`.`*f* or *Primary*`.`*f*, then:
     - If the compile-time type of *ExpressionName* or *Primary* is an intersection type $V_1$ `&` ... `&` $V_n$ (4.9 ⇗), then the qualifying class or interface of the reference is the

erasure (4.6 ⇗) of $V_1$.

- Otherwise, the erasure of the compile-time type of *ExpressionName* or *Primary* is the qualifying class or interface of the reference.

- If the expression is of the form `super.`*f*, then the superclass of *C* is the qualifying class or interface of the reference.
- If the expression is of the form *TypeName*`.super.`*f*, then the superclass of the class denoted by *TypeName* is the qualifying class or interface of the reference.

The reference to *f* must be compiled into a symbolic reference to the qualifying class or interface of the reference, plus the simple name of the field, *f*.

The reference must also include a symbolic reference to the erasure of the declared type of the field, so that the verifier can check that the type is as expected.

5. Given a method invocation expression or a method reference expression in a class or interface *C*, referencing a method named *m* declared (or implicitly declared (9.2 ⇗)) in a (possibly distinct) class or interface *D*, we define the *qualifying class or interface of the method invocation* as follows:

   - If *D* is `Object` then the qualifying class or interface of the method invocation is `Object`.
   - Otherwise:
     - If the method is referenced by a simple name, then if *m* is a member of the current class or interface *C*, let *Q* be *C*; otherwise, let *Q* be the innermost lexically enclosing class or interface declaration of which *m* is a member. In either case, *Q* is the qualifying class or interface of the method invocation.
     - If the expression is of the form *TypeName*`.`*m* or *ReferenceType*`::`*m*, then the class or interface denoted by *TypeName*, or the erasure of *ReferenceType*, is the qualifying class or interface of the method invocation.
     - If the expression is of the form *ExpressionName*`.`*m* or *Primary*`.`*m* or *ExpressionName*`::`*m* or *Primary*`::`*m*, then:
       - If the compile-time type of *ExpressionName* or *Primary* is an intersection type $V_1$ `&` ... `&` $V_n$, then the qualifying class or interface of the method invocation is the erasure of $V_1$.
       - Otherwise, the erasure of the compile-time type of *ExpressionName* or *Primary* is the qualifying class or interface of the method invocation.
     - If the expression is of the form `super.`*m* or `super::`*m*, then the superclass of *C* is the qualifying class or interface of the method invocation.
     - If the expression is of the form *TypeName*`.super.`*m* or *TypeName*`.super::`*m*, then if *TypeName* denotes a class *X*, the superclass of *X* is the qualifying class or interface of the method invocation; if *TypeName* denotes an interface *X*, *X* is the qualifying class or interface of the method invocation.

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

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

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

6. Given a class instance creation expression (15.9 ⇗) or an explicit constructor invocation statement (8.8.7.1 ⇗) or a method reference expression of the form *ClassType* `::` `new` (15.13 ⇗) in a class or interface *C*, referencing a constructor *m* declared in a (possibly distinct) class or interface *D*, we define the *qualifying class of the constructor invocation* as follows:
   - If the expression is of the form `new` *D* `(...)` or *ExpressionName* `.new` *D* `(...)` or *Primary* `.new` *D* `(...)` or *D* `::` `new`, then the qualifying class of the constructor invocation is *D*.
   - If the expression is of the form `new` *D* `(...){...}` or *ExpressionName* `.new` *D* `(...){...}` or *Primary* `.new` *D* `(...){...}`, then the qualifying class of the constructor invocation is the anonymous class declared by the expression.
   - If the expression is of the form `super(...)` or *ExpressionName* `.super(...)` or *Primary* `.super(...)`, then the qualifying class of the constructor invocation is the direct superclass of *C*.
   - If the expression is of the form `this(...)`, then the qualifying class of the constructor invocation is *C*.

   A reference to a constructor must be resolved at compile time to a symbolic reference to the qualifying class of the constructor invocation, plus the declared signature of the constructor (8.8.2 ⇗). The signature of a constructor must include both:
   - The number of parameters of the constructor
   - A symbolic reference to the type of each formal parameter

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

1. If it is a class and is not `Object`, then a symbolic reference to the direct superclass of this class.
2. A symbolic reference to each direct superinterface, if any.
3. A specification of each field declared in the class or interface, given as the simple name of the field and a symbolic reference to the erasure of the type of the field.
4. If it is a class, then the erased signature of each constructor, as described above.
5. For each method declared in the class or interface (excluding, for an interface, its implicitly declared methods (9.2 ⇗)), its erased signature and return type, as described above.
6. The code needed to implement the class or interface:
   - For an interface, code for the field initializers and the implementation of each method with a block body (9.4.3 ⇗).
   - For a class, code for the field initializers, the instance and static initializers, the implementation of each method with a block body (8.4.7 ⇗), and the implementation

of each constructor.

7. Every class or interface must contain sufficient information to recover its canonical name (6.7).
8. Every member class or interface must have sufficient information to recover its source-level access modifier (6.6 ⇗).
9. Every nested class or interface must have a symbolic reference to its immediately enclosing class or interface (8.1.3).
10. Every class or interface must contain symbolic references to all of its member classes and interfaces (8.5 ⇗, 9.5 ⇗), and to all other nested classes and interfaces declared within its body.
11. A construct emitted by a Java compiler must be marked as *synthetic* if it does not correspond to a construct declared explicitly or implicitly in source code, unless the emitted construct is a class initialization method (JVMS §2.9), or it corresponds to an unnamed class implicitly declared in an unnamed class compilation unit (7.3).
12. A construct emitted by a Java compiler must be marked as *mandated* if it corresponds to a formal parameter declared implicitly in source code (8.8.1 ⇗, 8.8.9 ⇗, 8.8.3 ⇗, 15.9.5.1 ⇗).

*The following formal parameters are declared implicitly in source code:*

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

*For reference, the following constructs are declared implicitly in source code, but are not marked as mandated because only formal parameters and modules can be so marked in a `class` file (JVMS §4.7.24, JVMS §4.7.25):*

- *Default constructors of normal and enum classes (8.8.9 ⇗, 8.9.2 ⇗)*
- *Canonical constructors of record classes (8.10.4 ⇗)*
- *Anonymous constructors (15.9.5.1 ⇗)*
- *The `values` and `valueOf` methods of enum classes (8.9.3 ⇗)*
- *Certain `public` fields of enum classes (8.9.3 ⇗)*
- *Certain `private` fields and `public` methods of record classes (8.10.3 ⇗)*
- *Certain `public` methods of interfaces (9.2 ⇗)*
- *Container annotations (9.7.5 ⇗)*

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

- A specification of the name of the module, given as a symbolic reference to the name indicated after `module`. Also, the specification must include whether the module is normal or open (7.7 ⇗).

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

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

---

*Copyright © 1993, 2023, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.*
*All rights reserved. Use is subject to license terms and the documentation redistribution policy.*
***DRAFT 21-internal-adhoc.gbierman.20230524***