

JEP 463: Implicitly Declared Classes and Instance Main Methods (Second Preview)

Author	Ron Pressler & Jim Laskey
Owner	Jim Laskey
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	22
Component	specification / language
Discussion	amber dash dev at openjdk dot org
Effort	5
Duration	5
Relates to	JEP 445: Unnamed Classes and Instance Main Methods (Preview) JEP 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
Reviewed by	Gavin Bierman
Endorsed by	Brian Goetz
Created	2023/08/30 18:07
Updated	2024/01/30 14:40
Issue	8315398

Summary

Evolve the Java programming language so that students can write their first programs without needing to understand language features designed for large programs. Far from using a separate dialect of the language, students can write streamlined declarations for single-class programs and then seamlessly expand their programs to use more advanced features as their skills grow. This is a [preview language](#) feature.

History

JEP 445 proposed *Unnamed Classes and Instance main Methods*, which previewed in JDK 21. Feedback has suggested that the feature should preview for a second time in JDK 22 with the following significant changes and, therefore, a revised title.

- The ideas of allowing classes to be unnamed, and of allowing a source file without an enclosing class declaration to implicitly declare an unnamed class, was primarily a specification device to ensure that the class could not be used by other classes. However, this has proven to be a distraction. We have adopted a simpler approach: A source file without an enclosing class declaration is said to implicitly declare a class with a name chosen by the host system. Such implicitly declared classes behave like normal top-level classes and require no additional tooling, library, or runtime support.
 - The procedure for selecting a main method to invoke was too complicated, taking into account both whether the method had a parameter and whether it was a static method or an instance method. In this second preview we propose to simplify the selection process to two steps: If there is a candidate main method with a `String[]` parameter then we invoke that method; otherwise we invoke a candidate main method with no parameters. There is no ambiguity here because a class cannot declare a static method and an instance method of the same name and signature.
- Goals**
- Offer a smooth on-ramp to Java programming so that instructors can introduce concepts in a gradual manner.
 - Help students to write basic programs in a concise manner and grow their code gracefully as their skills grow.
 - Reduce the ceremony of writing simple programs such as scripts and command-line utilities.
 - Do not introduce a separate beginners' dialect of the Java language.
 - Do not introduce a separate beginners' toolchain; student programs should be compiled and run with the same tools that compile and run any Java program.

Motivation

The Java programming language excels for large, complex applications developed and maintained over many years by large teams. It has rich features for data hiding, reuse, access control, namespace management, and modularity which allow components to be cleanly composed while being developed and maintained independently. With these features, components can expose well-defined interfaces for their interaction with other components while hiding internal implementation details so as to permit the independent evolution of each. Indeed, the object-oriented paradigm itself is designed for plugging together pieces that interact through well-defined protocols and abstract away implementation details. This composition of large components is called *programming in the large*. The language also offers many constructs useful for *programming in the small*—everything that is internal to a component. In recent years, we have enhanced both its programming-in-the-large capabilities with [modules](#) and its programming-in-the-small capabilities with [data-oriented programming](#).

The Java programming language is also, however, intended to be a first language. When programmers first start out they do not write large programs. In a team — they write small programs, alone. They have no need for encapsulation and namespaces, useful to separately evolve components written by different people. When teaching programming, instructors start with the basic programming-in-the-small concepts of variables, control flow, and subroutines. At that stage there is no need for the programming-in-the-large concepts of classes, packages, and modules. Making the language more welcoming to newcomers is in the interest of Java veterans but they, too, may find it pleasurable to write simple programs more concisely, without any programming-in-the-large scaffolding.

Consider the classic *Hello, World!* program that is often used as the first program for Java students:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- There is too much clutter here — too much code, too many concepts, too many constructs — for what the program does.
- The class declaration and the mandatory public access modifier are programming-in-the-large constructs. They are useful when encapsulating a code unit with a well-defined interface to external components, but pointless in this little example.
 - The `String[] args` parameter also exists to interface the code with an external component, in this case the operating system's shell. It is mysterious and unhelpful here, especially since it is not used in simple programs like `HelloWorld`.
 - The static modifier is part of the language's class-and-object model. For the novice, static is not just mysterious but harmful: To add more methods or fields that main can call and use the student must either declare them all as static — thereby propagating an idiom which is neither common nor a good habit — or else confront the difference between static and instance members and learn how to instantiate an object.

The new programmer encounters these concepts at the worst possible time, before they learn about variables and control flow, and when they cannot appreciate the utility of programming-in-the-large constructs for keeping a large program well organized. Instructors often offer the admonition, "don't worry about that, you'll understand it later." This is unsatisfying to them and their students alike, and leaves students with the enduring impression that the language is complicated.

The motivation for this JEP is not merely to reduce ceremony. We aim to help programmers that are new to the Java language, or to programming in general, learn the language in a manner that introduces concepts in the right order: Start with the fundamental programming-in-the-small concepts, then proceed to advanced programming-in-the-large concepts when they are actually beneficial and can be more easily grasped.

We propose to do this not by changing the structure of the Java language — code is still enclosed in methods, which are enclosed in classes, which are enclosed in packages, which are enclosed in modules — but by hiding these details until they are useful in larger programs. We offer an on-ramp, a gradual incline that gracefully merges onto the highway. When students move on to larger programs they need not discard what they learned in the early stages, but rather they see how it all fits within the larger picture.

The changes we offer here are just one step in making the Java language easier to learn. They do not even address all the speed bumps in the *Hello, World!* program: The beginner may still be puzzled by the mysterious `System.out.println` incantation, and still needs to import basic utility classes for essential functionality even in first-week programs. We may address these pains in a future JEP.

Description

First, we enhance the protocol by which Java programs are launched to allow *instance main methods*. Such methods are not static, need not be public, and need not have a `String[]` parameter. Then we can simplify the *Hello, World!* program to:

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

Second, we allow a [compilation unit](#), i.e., a source file, to *implicitly declare* a class:

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

This is [preview language feature](#), disabled by default

To try the examples below in JDK 22 you must enable preview features as follows:

- Compile the program with `javac --release 22 --enable-preview Main.java` and run it with `java --enable-preview Main`; or,
- When using the [source code launcher](#), run the program with `java --source 22 --enable-preview Main.java`

The launch protocol

New programmers just want to write and run a computer program. The *Java Language Specification*, however, focuses on defining the core Java unit of the class and the basic compilation unit, namely a source file comprised of a package declaration, followed by some `import` declarations, followed by one or more class declarations. All it has to say about a Java *program* is this:

The Java Virtual Machine starts execution by invoking the method `main` of some specified class or interface, passing it a single argument which is an array of strings.

The JLS further says:

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 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`.

The actions of choosing the class containing the `main` method, assembling its dependencies in the form of a module path or a class path (or both), loading the class, initializing it, and invoking the `main` method with its arguments constitute the *launch protocol*. In the JDK it is implemented by the *launcher*, i.e., the Java executable.

A flexible launch protocol

We enhance the launch protocol to offer more flexibility in the declaration of a program's entry point and, in particular, to allow *instance main* methods, as follows:

- Allow the main method of a launched class to have public, protected, or default (i.e., package) access.
- If the launched class contains a main method with a `String[]` parameter then choose that method.
- Otherwise, if the class contains a main method with no parameters then choose that method.
- In either case, if the chosen method is static then simply invoke it.
- Otherwise, the chosen method is an instance method and the launched class must have a zero-parameter, non-private constructor (i.e., of public, protected, or package access). Invoke that constructor and then invoke the main method of the resulting object. If there is no such constructor then report an error and terminate.
- If there is no suitable main method then report an error and terminate.

These changes allow us to write *Hello, World!* with no access modifiers, no static modifiers, and no `String[]` parameter, so the introduction of these constructs can be postponed until they are needed:

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

Implicitly declared classes

In the Java language, every class resides in a package and every package resides in a module. These namespacing and encapsulation constructs apply to all code, but small programs that do not need them can omit them. A program that does not need class namespaces can omit the package statement, making its classes implicit members of the unnamed package; classes in the unnamed package cannot be referenced explicitly by classes in named packages. A program that does not need to encapsulate its packages can omit the module declaration, making its packages implicit members of the unnamed module; packages in the unnamed module cannot be referenced explicitly by packages in named modules.

Before classes serve their main purpose as templates for the construction of objects, they serve only as namespaces for methods and fields. We should not require students to confront the concept of classes before they are comfortable with the more basic building blocks of variables, control flow, and subroutines, before they embark on learning object orientation, and when they are still writing simple, single-file programs. Even though every method resides in a class, we can stop requiring explicit class declarations for code that does not need it — just as we do not require explicit package or module declarations for code that does not need them.

Henceforth, if the Java compiler encounters a source file with a method that is not enclosed in a class declaration then it will consider that method, any other such methods, and any unenclosed fields and any classes in the file to form the body of an *implicitly declared* top-level class.

Such an implicitly declared class (or *implicit class* for short) is always a member of the unnamed package. It is also final and does not implement any interface nor extend any class other than `Object`. An implicit class cannot be referenced by name, so there can be no [method references](#) to its static methods; the `this` keyword can still be used, however, and so can method references to instance methods.

The code of an implicit class cannot refer to the implicit class by name, so instances of an implicit class cannot be constructed directly. Such a class is useful only as a standalone program or as an entry point to a program. Therefore, an implicit class must have a `main` method that can be launched as described above. This requirement is enforced by the Java compiler.

An implicit class resides in the unnamed package, and the unnamed package resides in the unnamed module. While there can be only one unnamed package (barring multiple class loaders) and only one unnamed module, there can be multiple implicit classes in the unnamed module. Every implicit class contains a `main` method and so represents a program, thus multiple such classes in the unnamed package represent multiple programs.

An implicit class is almost exactly like an explicitly declared class. Its members can have the same modifiers (e.g., `private` and `static`) and the modifiers have the same defaults (e.g., package access and instance membership). One key difference is that while an implicit class has a default zero-parameter constructor, it can have no other constructor.

With these changes we can now write *Hello, World!* as:

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

Top-level members are interpreted as members of the implicit class, so we can also write the program as:

```
String greeting() { return "Hello, World!"; }

void main() {
    System.out.println(greeting());
}
```

or, using a field, as:

```
String greeting = "Hello, World!";

void main() {
    System.out.println(greeting);
}
```

If an implicit class has an instance `main` method rather than a static `main` method then launching it is equivalent to the following, which employs the existing [anonymous class declaration](#) construct:

```
new Object() {
    // the implicit class's body
}.main();
```

A source file named `HelloWorld.java` containing an implicit class can be launched with the source-code launcher, like so:

```
$ java HelloWorld.java
```

The Java compiler will compile that file to the launchable class file `HelloWorld.class`. In this case the compiler chooses `HelloWorld` for the class name as an implementation detail, but that name still cannot be used directly in Java source code.

The `javadoc` tool cannot generate API documentation for an implicit class, as implicit classes do not define any API accessible from other classes, but the fields and methods of an implicit class can generate API documentation.

Growing a program

A *Hello, World!* program written as an implicit class is much more focused on what the program actually does, omitting concepts and constructs it does not need. Even so, all members are interpreted just as they are in an ordinary class. To evolve an implicit class into an ordinary class, all we need to do is wrap its declaration, excluding `import` statements, inside an explicit class declaration.

Eliminating the `main` method altogether may seem like the natural next step, but it would work against the goal of gracefully evolving a first Java program to a larger one and would impose some non-obvious restrictions (see [below](#)). Dropping the `void` modifier would similarly create a distinct Java dialect.

Alternatives

- Use JShell for introductory programming* — A JShell session is not a program but a sequence of code snippets. Declarations typed into `JShell` are implicitly viewed as static members of some unspecified class, with some unspecified access level, and statements execute in a context in which all previous declarations are in scope.

This is convenient for experimentation — which is the primary use case for JShell — but not a good model for learning to write Java programs. Evolving a batch of working declarations in JShell into a real Java program leads to a non-idiomatic style of code because it declares each method, class, and variable as static. JShell is a great tool for exploration and debugging, but it is not the on-ramp programming model we are looking for.

- Interpret code units as static members* — Methods and fields are non-static by default. Interpreting top-level members in an implicit class as static would change the meaning of the code units in such a class — introducing, in effect, a distinct Java dialect. To preserve the meaning of such members when we evolve an implicit class into an ordinary class we would have to add explicit static modifiers. This is not what we want as we scale up from a handful of methods to a simple class. We want to start using classes as classes, not as containers of static members.

- Interpret code units as locals* — We can already declare local variables within methods. Assume that we could also declare local methods, i.e., methods within other methods. Then we could interpret the body of a simple program as the body of a main method, with variables interpreted as local variables rather than fields, and methods interpreted as local methods rather than class members. This would allow us to eschew the `main` method altogether and write top-level statements.

The problem with this approach is that, in the Java language, locals behave differently from fields, and in a more restricted way to boot: Locals can only be accessed from inside lambda bodies or inner classes when they are [effectively final](#). The proposed design allows us to separate locals and fields in the same manner as they have always been. The burden of writing a main method is not onerous, even for new students.

- Introduce package-level methods and fields* — A user experience similar to that shown above could be achieved by allowing package-level methods and fields to be declared in a file without an explicit package or `class` declaration. However, such a feature would have a far wider impact on how Java code is written in general.