Java Enhancement Proposals Simplify Teaching Java to Students (Part 2)

When we think about teaching Java to new learners, there's good news and bad news. First, the bad news. Java has a reputation for being verbose and difficult to learn. What a shame!

Java is the leading language for long-lasting, server-side applications because it's great at scaling *up.* But Java's presence in education has been declining because Java hasn't been good at scaling *down.* That's unfortunate. Every expert starts out as a beginner.

But here's the good news: Several initiatives over the past few years have made Java easier to learn.

In a September 2022 design note titled '[Paving the on-ramp]', Java Language Architect Brian Goetz discussed reducing the efforts of learning Java for beginners while keeping some fundamental tenets in mind:

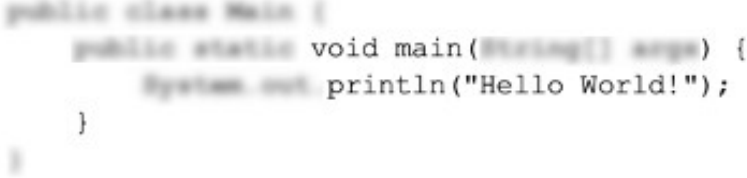
- Do not introduce a separate "beginners' dialect" of Java.
- Do not introduce a special tooling workflow for beginners.
- Changes must be a *natural*, consistent evolution of the Java language and tooling.
- Create a series of independent JEPs covering the language, existing tools, and new tools.

Nowadays, a student can start learning Java with one shell command and only three lines of code. In some settings, only one line of code suffices. This article describes language enhancements that eliminate boilerplate code in small programs and specialized programming environments that simplify Java program development.

Java Language Enhancements

Some of the challenges in teaching Java to students and more junior technical professionals include Java's verbose "Hello World!" program, having access to machines that can have Java installed (Chromebooks, for example), and older curricula that don't incorporate more modern Java features.

Figure 1 shows what we want beginners to notice when they start learning Java:



- Figure 1 -

Java's introductory tutorials typically start with disclaimers about `class`, `public`, `static`, `System`, and other aspects of a "Hello World!" program. A student's first exposure to Java is like clicking a license agreement's Accept button. Most people say, “The license agreement is too long and complicated, so I'll just click the Accept button and hope for the best.” In the same way, Java instructors will often tell students to “Ignore those parts of the program. We'll explain them later.”

One of Java's underlying principles is the notion that every function belongs inside of a class. Even `main` – a program's starting point – is a method within a class. And you don't automatically have an instance of that class, so the main method has to be static.

The problem with this is that beginners need time to learn about object-oriented programming. What's second nature for experienced Java developers is mystifying for novices. Beginners don't understand how classes work until they write several programs in a language like Java. We can explain the word `static` several times in several different ways, but students don't understand `static` until they feel compelled to use it.

In his ‘Paving the on-ramp‘ design note, Brian Goetz suggested ways to trim the boilerplate code from Java's simplest programs. As always, the prime directive was to avoid breaking the language's well-established rules. So having the `main` method live outside of a class wasn't part of the plan. Instead, the core idea was to add an *implicitly declared class* feature to Java. From Java's early days, the unnamed package has been implicitly declared, so the new implicitly declared class idea wasn't a shock to the system.

In September 2025, Goetz's ideas came to fruition with the full implementation of JEP 511: *Module Import Declarations* and JEP 512: *Compact Source Files and Instance Main Methods* in Java.[1]

To see how these features work, we look at Example 1.

```
void main() {
    IO.println("Hello World!");
}
```

- Example 1 -

1 Six earlier JEPs that were proposed and developed in OpenJDK supported this effort:
JEP 445— Unnamed Classes and Instance Main Methods(Preview)
JEP 463— Implicitly Declared Classes and Instance Main Methods (Second Preview)
JEP 476— Module Import Declarations (Preview)
JEP 477— Implicitly Declared Classes and Instance Main Methods (Third Preview)
JEP 494— Module Import Declarations (Second Preview)
JEP 495— Simple Source Files and Instance Main Methods (Fourth Preview)

This program has a main method without having the words `class`, `public`, `static`, or `String`. It calls `println` with no reference to `System.out`. Here's what's happening under the hood:

- A *compact source file* has a main method that's not inside an explicit class declaration. If you name the file `HelloAgain.java` and issue the `javac HelloAgain.java` command, you get a new file named `HelloAgain.class`. In other words, the file name determines the name of the implicitly declared class.
- With the enhancements in Java 25, `main` need not be declared static. That is, `main` can be an instance method. In addition, `main` need not be declared public. Boilerplate code is melting away!
- The `main` method no longer needs to have a `String[] args` parameter.
- The `java.lang` package now contains a class named `IO`. This class contains the following methods:

  - `public static void println()`
  - `public static void println(Object obj)`
  - `public static void print(Object obj)`
  - `public static String readln()`
  - `public static String readln(String prompt)`

  Since classes in `java.lang` are imported by default, the programmer can call any of these methods without an explicit import declaration. The only requirement is to use the `IO.` prefix at the start of each call.

As an added benefit, you can mix and match these handy new features. Examples 2, 3, and 4 are complete Java programs.

```
class HelloAgain {

    void main(String[] args) {
        IO.println("Hello World!");
    }
}
```

- Example 2 -

```
import static java.lang.System.out;

void main() {
    out.println("Hello World!");
    write("Goodbye!");
```

```
}

void write(String output) {
    IO.println(output);
}
```

- Example 3 -

```
void main() {
    MyInnerClass.write("Goodbye!");
}

class MyInnerClass {

    static void write(String output) {
        IO.println(output);
    }
}
```

- Example 4 -

Java 25 contains some module-level import features that make life easier for new programmers. Example 5 imports all the packages in `java.desktop` in one fell swoop.

```
import module java.desktop;

void main() {

    // From javax.swing:
    var frame = new JFrame();

    // From java.awt:
    var button = new Button("Submit");

    frame.add(button);
    frame.setSize(200, 200);
    frame.setVisible(true);
}
```

- Example 5 -

Even better, all public top-level classes and interfaces of the packages in the `java.base` module are implicitly imported into compact source files. For a look at this feature, see Example 6.

```
void main() throws IOException {

    // From java.util.Random:
```

```
        var number = new Random().nextInt();

        // From java.io:
        try (var fWriter = new FileWriter("data");
              var pWriter = new PrintWriter(fWriter)) {

            pWriter.print(number);
        }
    }
```

- Example 6 -

In the United States, the College Board offers Advanced Placement exams in many subjects for college-bound students with above-average preparation. The Board offers two Computer Science exams.

- The Computer Science Principles (AP CSP) exam is language-agnostic.
- The Computer Science A (AP CSA) exam is Java-specific.

Representatives from Oracle's Java in Education team are working with the Board to update the CSA exam to include the latest Java language features.

Specialized Programming Environments

The most popular Java IDEs are built for large projects. For example, to run a "Hello World!" program from scratch in IntelliJ IDEA, you start by creating a project, choosing between Java, Kotlin, Groovy and many other platforms, naming the project, specifying the project's location, and then deciding on the build system and the JDK version. The Project tool window displays a hierarchy of folders and files in which `main` is six levels deep. Running `main` involves finding the correct icon or selecting from a menu with at least 18 items. For a seasoned developer, the steps are trivial. But, for a new Java student, the procedure is overwhelming. The versatility of an IDE like IntelliJ IDEA is overkill for a new programmer.

Java 9, released in 2017, delivered JEP 222: *jshell: The Java Shell (Read-Eval-Print Loop)*. With a Read-Eval-Print Loop (REPL), the programmer types a single line of code and presses Enter. The REPL executes the code immediately. If the code is an expression, the REPL displays the expression's value. Then the programmer follows with another line of code.

Example 7 shows a brief command-line session using Java's own JShell REPL:

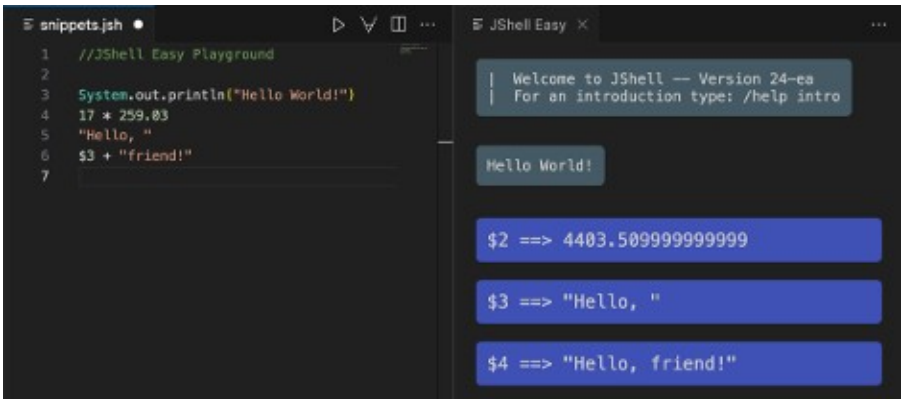
```
$ jshell
|  Welcome to JShell -- Version 26-ea
|  For an introduction type: /help intro

jshell> IO.println("Hello World!")
```

```
Hello World!

jshell> 17 * 259.03
$2 ==> 4403.509999999999

jshell> "Hello, "
$3 ==> "Hello, "

jshell> $3 + "friend!"
$4 ==> "Hello, friend!"

jshell> /exit
|  Goodbye
$
```

- Example 7 -

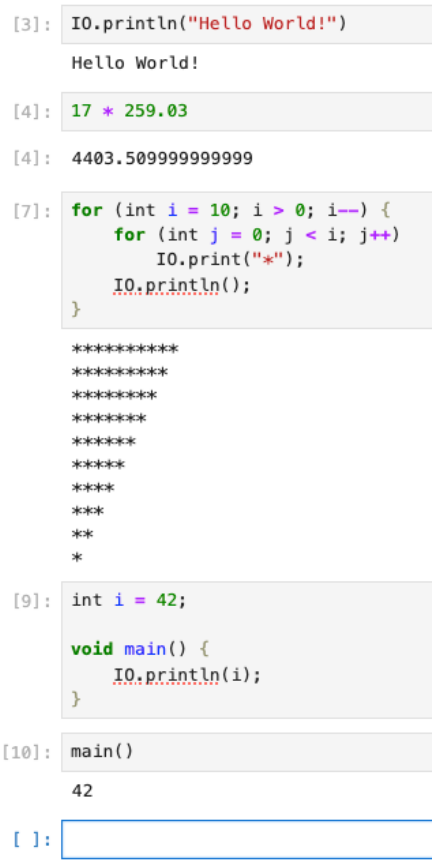
In this session, the "Hello World" snippet is only one line of code. For the most part, semicolons are optional. The environment even accepts simple expressions and displays their values without the need for a `println` call.

In October 2023, Oracle announced a new Java extension for VS Code. The extension includes features such as autocompletion, error highlighting, hover documentation, code formatting, refactoring, and more. By adding Senthilnathan's JShell Easy extension to VS Code, a session takes on a more modern look and feel. (See Figure 2.)



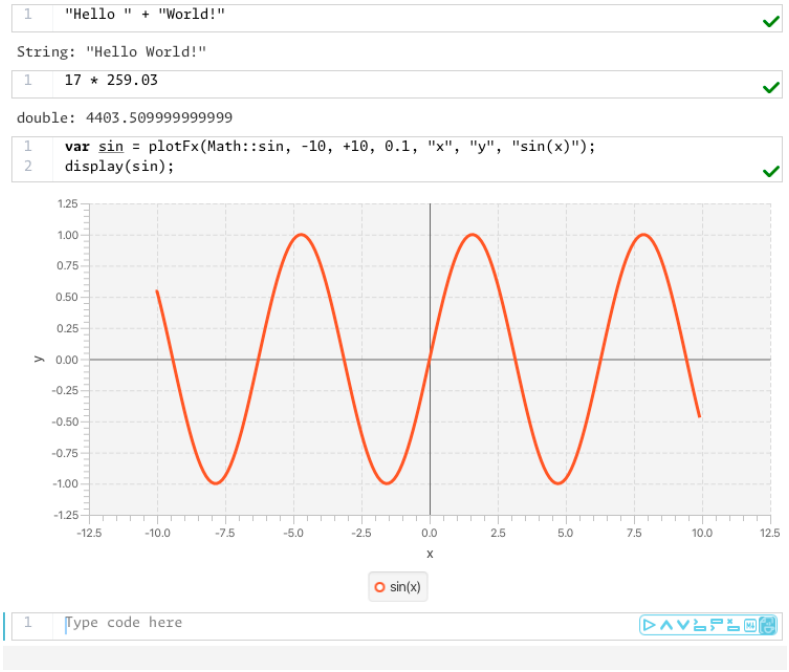
- Figure 2 -

More recently, some independent developers have been creating GUI environments that incorporate JShell's features. Figure 3 shows a session using the JJava kernel for Jupyter notebooks.



- Figure 3 -

Another project, named JTaccuino, provides a JavaFX wrapper around JShell. Figure 4 shows a brief but impressive JTaccuino session.



- Figure 4 -

So much for Read-Eval-Print Loops! What about running complete programs? JEP 330: *Launch Single-File Source-Code Programs* was delivered in Java 11, and JEP 458: *Launch Multi-File Source-Code Programs* became a reality with Java 22. These features make it possible to run Java programs with just one command. When the programmer types

```
java HelloAgain.java
```

the `java` command compiles the source code and runs the code's `main` method, all in one step. When we introduce students to Java and have them run their first programs, we don't have to mention `.class` files and bytecode. We can leave all those details to later discussions. But that's not all! A project named JBang aims to simplify the entire Java development process. To show you what JBang can do, read the code in Example 8.

```
//DEPS org.openjfx:javafx- ↩controls:25.0.1:$
{os.detected.jfxname}
//DEPS org.openjfx:javafx- ↩graphics:25.0.1:$
{os.detected.jfxname}

import module javafx.graphics;
import module javafx.controls;

public class FXDemo extends Application {

    @Override
    public void start(Stage stage) {
        var stackPane = new StackPane();
```

```
            stackPane.getChildren().add(new Button("Hello"));
            var scene = new Scene(stackPane, 200, 180);
            stage.setScene(scene);
            stage.show();
        }

        void main() {
            launch();
        }
}
```

- Example 8 -

(In Example 8, the character ↩ indicates that, despite the way we print it in this article, the source code should *not* have a hard line-break at that point.)

After installing JBang, the programmer runs Example 8 by typing a single command line:

```
jbang FXDemo.java
```

With this command alone, the program runs and displays the frame in Figure 5.

- Figure 5 -

So much happens with only one command! JBang automatically downloads and installs Java if Java isn't already installed. Then JBang downloads the dependencies listed in the file's `DEPS` comments. In Example 8, these dependencies are the modules `javafx.controls` and `javafx.graphics`. JBang solves the tricky problem of managing dependencies with comments at the top of the source file!

And don't ignore the streamlined code in Example 8! JavaFX source files contain their share of boilerplate code, but the new features in Java 25 help reduce some clutter. Example 8 uses module import declarations to scoop up all the required JavaFX classes, while the example's `main` method is brief and to the point.

These enhancements have addressed making Java easier to learn and to teach, but there are some future areas for improvements in the experience, such as downloading and using libraries or choosing and learning a build tool.

Whether you are a new student just learning Java or an experienced expert, these new developments will increase your joy in writing Java programs. We encourage you to share these new features and how you implement them to raise awareness within your community.