Your First Program Is an Automated Check

Skip Hello World. Your first Java program compares expected vs actual and prints PASS or FAIL -- the same shape as the last step of every test case you already write.

Automation Foundations · ashishautomationlabs.store

Tradition says a first program prints "Hello, World!" You are not here for tradition. You are here to automate testing.

So your first program is a real -- if small -- automated check: it compares an expected result against an actual result and announces PASS or FAIL. It is the last step of TC-1042, written again in Java.

Build the check

In Eclipse, right-click the `src` folder -> **New -> Class** -> name it `FirstCheck` -> **Finish**. Then type the following exactly -- every brace, every semicolon, every capital letter:

```
public class FirstCheck {
    public static void main(String[] args) {
        String expected = "Welcome, Priya!";
        String actual = "Welcome, Priya!";
        System.out.println("Expected: " + expected);
        System.out.println("Actual : " + actual);
        if (expected.equals(actual)) {
            System.out.println("Result : PASS");
        } else {
            System.out.println("Result : FAIL");
        }
    }
}
```

Before you run it -- you knew this was coming -- **predict the output**. Write the prediction down. Predicted first, actual second; any mismatch is investigated like the defect it is.

Run it: right-click inside the file -> **Run As -> Java Application** (or the green Run button). The actual output:

```
Expected: Welcome, Priya!
Actual : Welcome, Priya!
Result : PASS
```

If your prediction matched: your first test case just executed itself. If Eclipse refused and showed red underlines instead: even better. Jump to the next lesson, fix the program from the error

message, and come back. Either way, you have completed the basic loop of the job: write a check, run it, read the result.

What every line means

A program you can run but cannot explain is a risk -- you know what happens when someone executes steps they do not understand. Walk through the lines.

`public class FirstCheck {` -- In Java, all code lives inside a class. For now, think of a class as a labeled container for related code -- the way a test suite is a labeled container for related test cases. Two rules with immediate effects: the class name must exactly match the file name (`FirstCheck` lives in `FirstCheck.java`), and by convention class names start with a capital letter, with each new word capitalized: `FirstCheck`, `LoginPage`, `TestDataReader`.

`public static void main(String[] args) {` -- The entry point. The JVM starts executing here. Every runnable Java program has exactly this method, spelled exactly this way. What do `static`, `void`, and `String[] args` each mean in full? That answer needs ideas that arrive later in this series -- for today, type it exactly and know that it means "start here."

`String expected = "Welcome, Priya!";` -- A variable: a named piece of stored data. `String` is its type (text), `expected` is its name, and the value follows the `=`. Later lessons go deep on variables; today you only need this: the line stores that text under the name `expected`.

`System.out.println("Expected: " + expected);` -- Prints one line to the console. The `+` joins fixed text with whatever the variable holds. Printing is your first observation tool -- the automation equivalent of a log so you can see what actually happened.

`if (expected.equals(actual))` -- The decision. Read it aloud: _if expected equals actual_ -- the verification step of every test case you have ever written. One early warning: `.equals(...)` is how you compare text in Java. There is a look-alike (`==`) that seems to work and then fails when it matters most. That trap gets its own lesson later.

Braces `{ }` and semicolons `;` -- Braces group instructions into blocks. Semicolons end statements -- a full stop at the end of each instruction. The compiler is completely strict about both.

You wrote a check, ran it, and can explain every line -- or honestly mark `main` as "used now, explained later." The next lesson is the most important skill in Chapter 1: break the program on purpose, and read the compiler's messages like defect reports.