Return Values, and the Early Exit

A void method does something; a method with a return type answers something -- and judging is answering, which is exactly what makes return values useful for testers.

Automation Foundations · ashishautomationlabs.store

A `void` method does something. A method with a return type answers something -- and that is what makes methods genuinely powerful for testers, because judging is answering:

```
public class M2 {
    static boolean isPass(int actual, int expected) {
        return actual == expected;
    }
    static String verdict(boolean passed) {
        return passed ? "PASS" : "FAIL";
    }
    public static void main(String[] args) {
        boolean r1 = isPass(200, 200);
        boolean r2 = isPass(500, 200);
        System.out.println("Check 1: " + verdict(r1));
        System.out.println("Check 2: " + verdict(r2));
    }
}
```

```
Check 1: PASS
Check 2: FAIL
```

Two things matter here. First, the return type on the left of the name must match what `return` actually hands back -- `boolean` for `isPass`, `String` for `verdict`. Get them out of step and the compiler refuses, the strict reviewer from Chapter 1 doing its job. Second, `isPass` takes two parameters separated by a comma, and the order matters absolutely: `isPass(200, 200)` fills `actual` with the first and `expected` with the second. Swap two same-typed arguments by accident and nothing complains -- the method just quietly answers a different question. Worth remembering the next time a test starts reporting backwards.

return does two things at once

It hands back a value, and it ends the method immediately. The second part is easy to forget and easy to use well:

```
static String describe(String env) {
    if (env.equals("prod")) {
        return "PRODUCTION - be careful";
    }
    return "Non-production: " + env;
}
```

```
PRODUCTION - be careful
Non-production: uat
```

When `env` is `"prod"`, the first `return` fires and the method is finished -- the line below it never runs. This is an **early return**: a clean way to handle a special case first, then get on with the normal path. A `void` method can use a bare `return;` for the same purpose -- leave now, hand back nothing.

> *Predict, given* `static int add(int a, int b) { return a + b; }` *-- what is the return type and method name, which are the parameters in* `add(3, 4)`*, and what happens to any code written after a* `return` *in the same block?*

>

> *Answer -- return type* `int`*, name* `add`*.* `a` *and* `b` *are the parameters;* `3` *and* `4` *are the arguments. Code after a* `return` *never runs, because* `return` *ends the method immediately.*

A method can hand back the right answer every time and still not change anything you expected it to. Whether it can is a separate question -- and it depends on what, exactly, got passed in.