equalsIgnoreCase, and When Case Matters

Chapter 5 left a defect open: a search for "logintest" missed "LoginTest". Use .equals when case matters and .equalsIgnoreCase when it doesn't -- a real decision.

Automation Foundations · ashishautomationlabs.store

Chapter 5 left a defect open: a list search for `"logintest"` missed `"LoginTest"`, because `.equals` compares text exactly, including capital letters. When you want to ignore case, Java has a method that says so:

```
public class Eq5 {
    public static void main(String[] args) {
        String ran = "LoginTest";
        System.out.println("equals          : " + ran.equals("logintest"));
        System.out.println("equalsIgnoreCase: " + ran.equalsIgnoreCase("logintest"));
    }
}
```

```
equals          : false
equalsIgnoreCase: true
```

Use `.equals` when case matters, and `.equalsIgnoreCase` when it doesn't -- comparing a browser name or an environment label a user might type in any case. Choosing between them is a real decision, not a detail: a password check must be case-exact, while a "yes/no" answer from a user usually shouldn't be.

Which comparison to use

For primitives -- `int`, `double`, `boolean` -- use `==`; it compares the values, and that's always correct. For objects -- `String`, `Integer`, your own types -- use `.equals`; it compares the contents. `==` on objects compares identity, the arrows, which is almost never what a test wants.

Two questions this raises

If `==` is wrong for text, why did my very first test with it pass? Almost certainly you used two literals, and the String pool made them one shared object, so `==` happened to be `true`. It wasn't comparing your text; it was reporting that there was one object. The first value that comes from a file or an API breaks the illusion.

Does this trap apply to numbers too, like `Integer`? Yes, in the same shape. An `int` is a primitive, so `==` compares values and is always fine. But `Integer`, the object form you met with

lists, is a reference type -- so `==` compares arrows there too. Java even keeps a small cache of low `Integer` values, which makes `==` appear to work for small numbers and then fail for large ones. Compare their values, or use `.equals`, and never rely on `==` for the object form.

One more question sits under all of this: will `.equals` work correctly on your own objects later? Only once you teach it how. For library types like `String`, `.equals` already compares contents. For a class you write yourself, the built-in `.equals` starts out comparing identity -- the same `==` behavior -- until you define what "equal" means for your type. The moment you create your own type -- a `TestResult`, a `Defect`, a `User` -- you'll need to decide when two of them count as equal. You teach Java that rule by writing an `equals` method, and a partner called `hashCode`. That's a Chapter 7 topic, once you know how to build a class.