static, and One Name for Several Shapes

static means belongs to the class, not any one object -- which is why Counter.total is shared while each object counts only its own. Java also lets several methods share one name, as long as their parameters differ.

Automation Foundations · ashishautomationlabs.store

In Chapter 7 you called methods on objects: `login.isPassed()`, `page.getUrl()`. Those are **instance methods** -- they need an object, because they work with that object's fields. A **static** method belongs to the class itself, so you can call it without creating any object. `Math.abs(...)` from Chapter 3 is static: you never wrote `new Math()`. The difference shows clearly when the same class has both kinds of data:

```
public class M7 {
    public static void main(String[] args) {
        Counter a = new Counter();
        Counter b = new Counter();
        a.record();
        a.record();
        b.record();
        System.out.println("a counted    : " + a.mine);
        System.out.println("b counted    : " + b.mine);
        System.out.println("total (static): " + Counter.total);
    }
}
class Counter {
    int mine = 0;
    static int total = 0;
    void record() {
        mine++;
        total++;
    }
}
```

```
a counted    : 2
b counted    : 1
total (static): 3
```

Each `Counter` object has its own `mine`, exactly as Chapter 7 taught. But there is only ever one `total`, shared by the whole class -- which is why it is read as `Counter.total`, through the class name, not through an object.

The rule for choosing: if a method works with a particular object's data, make it an instance method. If it is a general utility that depends only on what you pass in, make it `static`. A method that formats a defect id, converts milliseconds to seconds, or builds a URL from an environment

name is a good static helper -- exactly how utility classes in real frameworks are written. One trap worth planting now: a static method cannot use an object's fields directly, because there is no object involved. If you try, the compiler refuses -- not an inconvenience, but Java telling you the truth about what the method actually has access to.

One name, several shapes

Java lets two methods share a name as long as their parameter lists differ. This is **overloading**, and you have already used it -- `System.out.println` accepts text, numbers, booleans, and objects, all under one name:

```
static String tag(String id) {
    return "[" + id + "]";
}
static String tag(String id, String sev) {
    return "[" + id + " / " + sev + "]";
}

[DEF-101]
[DEF-101 / HIGH]
```

Java picks the right one by counting and typing the arguments at the call. Use overloading when the methods really do the same job with different information available -- not to bolt unrelated behavior onto a familiar name.

Methods are the unit a framework is assembled from. A Page Object's actions -- `login(user, pass)`, `searchFor(term)` -- are instance methods with parameters, returning either nothing or the next page object. A utility class of static helpers formats dates, builds URLs, and reads config. Your test methods themselves are methods, marked with an annotation like `@Test` so the runner can find them -- the same way the JVM finds `main`. Learning a framework next will not mean learning a new idea, only new method names.

Speaking of the JVM finding `main`: that line is the last unpaid debt this book took on, back in Chapter 1. Every word in it is now something you know.