Writing Your Own Generic Method

A type parameter like <T> lets one method serve any type with no casting at the call. Type erasure is the honest limit: that type information is gone by run time.

Automation Foundations · ashishautomationlabs.store

You can put a type parameter on your own methods and classes. The convention is a single capital letter: `T` for type, `K` and `V` for key and value, `E` for element.

```
static <T> T firstOf(List<T> items) {
    if (items.isEmpty()) {
        return null;
    }
    return items.get(0);
}
class Pair<K, V> {
    private final K key;
    private final V value;
    // ...
}
```

```
first String : LoginTest
first Integer: 200
boxed pair   : env=uat
mixed pair   : timeout=30
```

Read the method signature carefully. The `<T>` before the return type declares the parameter. Then `T` is used as the return type and inside `List<T>`. One method served a `List<String>` and a `List<Integer>`, returning the right type each time with no casting at the call. Without generics you would write this twice, or write it once returning `Object` and cast at every call -- which is the raw-type problem again. This is how a framework writes utility methods that work with any type while staying type-safe.

One honest limitation, and a common interview question. Generic type information exists at compile time and is largely removed afterwards, which is called **type erasure**. At run time a `List<String>` and a `List<Integer>` are both simply a `List`. That is why you cannot ask an object what its type parameter was, and why generics protect you when you compile rather than when you run.

> *Using the code above: what does the* `<T>` *before the return type of* `firstOf` *declare? Why did* `firstOf(codes)` *return an* `Integer` *without a cast? What is type erasure, in one sentence?*

>

> *Answer -- a type parameter, so the method can work with any type. The compiler knew the argument was a* `List<Integer>` *and applied* `T` *accordingly. And generic type information is used at compile time and largely removed afterwards, so a* `List<String>` *is just a* `List` *at run time.*

Generics have been a compile-time safety net this whole chapter. The next problem is a run-time one, and it's the reason this chapter exists at all.