

## Generic methods

Function `count`, to the right, returns the number of times item occurs in array `b`. It should work if `b` is an Integer array, a Boolean array, a JFrame array—the type of the array elements shouldn't matter. But the array elements and argument item must have the same type.

This is accomplished by making `count` a *generic function*, by placing *type parameter* `T` within “<>” just before the return type. `<T>` is a declaration of type parameter `T`, and `T` then appears in three other places.

A call on `count` does not explicitly give a type argument for `T`. Instead, `T` is inferred from the types of the arguments of the call. Here are two calls on `count`, each followed by the value it returns:

```
count(5, new Integer[] {5, 3, 5, 2})      2
count("b", new String[] {"bc", "b", "b", "b"})  3
```

```
/** Return the number of times item occurs in b.
 * Precondition: item is not null. */
public static <T> int count(T item, T[] b) {
    int n= 0;
    for (T e : b) {
        if (item.equals(e)) n= n+1;
    }
    return n;
}
```

### Creating a Pair with elements the same

Consider class `Pair` to the right. We write function `twoOf(v)` to return a `Pair` that has `v` in both its elements. Thus,

```
twoOf(v)    and    new Pair(v, v)
```

do the same thing.

The return type of `twoOf(v)` should be

```
Pair<T, T>
```

where `T` is the type of `v`. Because `T` has to occur in at least two places, this requires a generic method, which we write like this:

```
/** Return a pair (v, v). */
public static <T> Pair<T, T> twoOf(T v) {
    return new Pair<>(v, v);
}
```

The occurrence of `<T>` before the return type (and after keyword `static`) marks the function as generic, with type parameter `T`.

```
/** An instance contains an ordered pair. */
public class Pair<E, F> {
    public E first;    // First element
    public F second;  // Second element

    /** Constructor: a null pair */
    public Pair() {}

    /** Constructor: a pair (e, f) */
    public Pair(E e, F f) {
        first= e;
        second= f;
    }

    /** return a representation of this pair. */
    public @Override String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

Again, a call does not explicitly give a type argument for `T`. Instead, `T` is inferred from the arguments of the call. Below are two examples. Each call produces a `Pair` object; to the right of the call is what its `toString` function produces. The second call shows that `twoOf(v)` is most useful when the argument of a call is long—the argument has to be written only once.

```
twoOf(5)                                its toString produces "(5, 5)"
twoOf(new Pair<>("this is not 6", 5))    its toString produces "((this is not 6, 5), (this is not 6, 5))"
```

### A method with two type parameters

We write a static function to tell whether its two `Pair` parameters have equal first and second elements. Two type parameters are needed, `E` is used for the first element and `F` for the second.

```
/** Return true iff the fields of p1 equal the fields of p2. */
public static <E, F> boolean equals(Pair<E, F> p1, Pair<E, F> p2) {
    return p1.first.equals(p2.first) && p1.second.equals(p2.second);
}
```