# The wildcard ?

PTo the right is a simple method to print the elements of an ArrayList. The type of elements is Object so that, ostensibly, elements of any ArrayList can be printed. But the following code doesn't even compile:

```
ArrayList<Integer> c= new ArrayList<>();
c.add(3); c.add(4);
print(c);
```

```
/** Print elements of b, one per line. */
public static void print(ArrayList<Object> b) {
    for (Object e : b)
        System.out.println(e);
}
```

Why? As explained in a pdf file in the JavaHyperText entry for *generics*, ArrayList<Integer> is not a subclass of ArrayList<Object>, so the argument type is not the same as or narrower than the parameter type.

To write this generic print method, use the *wildcard* ?, as shown to the right[1]. The notation ArrayList<?> stands for "an ArrayList of elements of any type". Use this method instead of the first one above, and the code above is syntactically correct, as is also the following:

```
ArrayList<String> d= new ArrayList<>();
d.add("ab"); d.add("cd");
print(c);
```

```
/** Print elements of b, one per line. */
public static void print(ArrayList<?> b) {
    for (Object e : b)
        System.out.println(e);
}
```

**Using more than one wildcard**

Consider class Pair again, as shown to the right. The following method has two wildcards.

```
/** Print p. */
public static void print(Pair<?, ?> p) {
    System.out.println(p);
}
```

These wild cards can be associated with different values, as they do in this code:

```
Pair<Integer, String> p3= new Pair<>(3, "abc");
print(p3);
```

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

/** 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 + ")";
    }
}
```

**Notes**

1. We have just scratched the surface of a rather confusing set of rules for the use of the wildcard. Further items in the JavaHyperText entry for *generics* provide more detail.

2. Do not use a wildcard in a return type (e.g. public static Pair<?, ?> p() {…}) because it forces the user of the method to deal with wildcards.

3. ArrayList<Object> and ArrayList<?> are not the same. You can add *any* object to an ArrayList<Object>. The only thing you can add to an ArrayList<?> is null, because null is the only value that can go into *any* ArrayList.

4. We said in our introduction to generics that Java generics are defined in terms of *type erasure*. Generics provide more type safety, but almost all aspects of generics are erased from the program before it is compiled. You can see evidence of this yourself. Place both print methods that appear above into a Java class. You will get this compile-time error message: *Error: name clash: print(java.util.ArrayList<?>) and print(java.util.ArrayList<java.lang.Object>) have the same erasure*.

[1] The term *wild card* comes from some card games, played with a deck of 52 cards. If a card, say the two of clubs, is a wild card, the player holding it may give it any value they want.