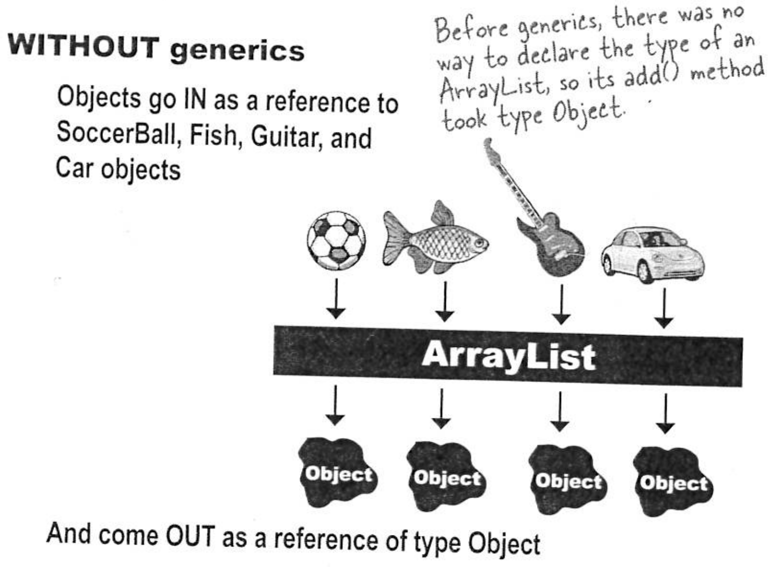
Generics means more type-safety

We'll just say it right here—*virtually all of the code you write that deals with generics will be collection-related code.* Although generics can be used in other ways, the main point of generics is to let you write type-safe collections. In other words, code that makes the compiler stop you from putting a Dog into a list of Ducks.

Before generics (which means before Java 5.0), the compiler could not care less what you put into a collection, because all collection implementations were declared to hold type Object. You could put *anything* in any ArrayList; it was like all ArrayLists were declared as ArrayList<Object>.

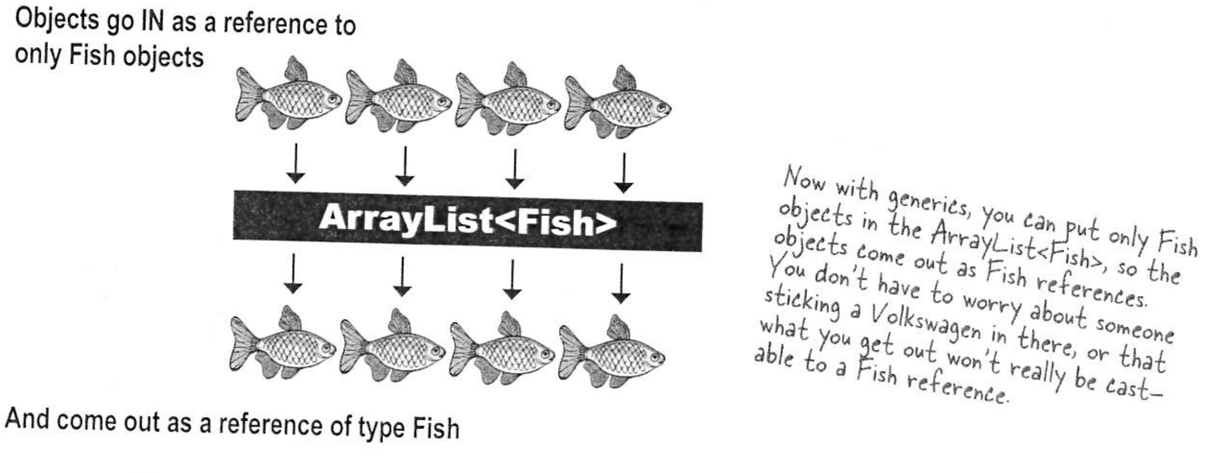
WITHOUT generics


Before generics, there was no way to declare the type of an ArrayList, so its add() method took type Object.
Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects
ArrayList
Object
Object
Object
Object
And come OUT as a reference of type Object

With generics, you can create type-safe collections where more problems are caught at compile-time instead of runtime.

Without generics, the compiler would happily let you put a Pumpkin into an ArrayList that was supposed to hold only Cat objects.

WITH generics


Objects go IN as a reference to only Fish objects
ArrayList<Fish>
And come out as a reference of type Fish
Now with generics, you can put only Fish objects in the ArrayList<Fish>, so the objects come out as Fish references. You don't have to worry about someone sticking a Volkswagen in there, or that what you get out won't really be cast-able to a Fish reference.

Using generic METHODS

A generic *class* means that the *class declaration* includes a type parameter. A generic *method* means that the method declaration uses a type parameter in its signature.

You can use type parameters in a method in several different ways:

① Using a type parameter defined in the class declaration

```
public class ArrayList<E> extends AbstractList<E> ... {
     public boolean add(E o)
```

You can use the "E" here ONLY because it's already been defined as part of the class.

When you declare a type parameter for the class, you can simply use that type any place that you'd use a *real* class or interface type. The type declared in the method argument is essentially replaced with the type you use when you instantiate the class.

② Using a type parameter that was NOT defined in the class declaration

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

Here we can use <T> because we declared "T" earlier in the method declaration.

If the class itself doesn't use a type parameter, you can still specify one for a method, by declaring it in a really unusual (but available) space—*before the return type*. This method says that T can be "any type of Animal".

Wait... that can't be right. If you can take a list of Animal, why don't you just SAY that? What's wrong with just *takeThing(ArrayList<Animal> list)*?

Here's where it gets weird...

This:

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

Is NOT the same as this:

```
public void takeThing(ArrayList<Animal> list)
```

Both are legal, but they're *different*!

The first one, where **<T extends Animal>** is part of the method declaration, means that any ArrayList declared of a type that is Animal, or one of Animal's subtypes (like Dog or Cat), is legal. So you could invoke the top method using an ArrayList<Dog>, ArrayList<Cat>, or ArrayList<Animal>.

But... the one on the bottom, where the method argument is (ArrayList<Animal> list) means that *only* an ArrayList<Animal> is legal. In other words, while the first version takes an ArrayList of any type that is a type of Animal (Animal, Dog, Cat, etc.), the second version takes *only* an ArrayList of type Animal. Not ArrayList<Dog>, or ArrayList<Cat> but only ArrayList<Animal>.

And yes, it does appear to violate the point of polymorphism. but it will become clear when we revisit this in detail at the end of the chapter. For now, remember that we're only looking at this because we're still trying to figure out how to sort() that SongList, and that led us into looking at the API for the sort() method, which had this strange generic type declaration.

For now, all you need to know is that the syntax of the top version is legal, and that it means you can pass in a ArrayList object instantiated as Animal or any Animal subtype.

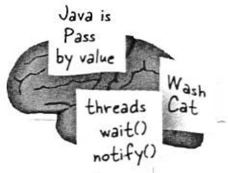
And now back to our sort() method...

A Brain-Friendly Guide

Head First Java™

2nd Edition
Covers Java 5.0

Learn how threads can change your life



Make Java concepts stick to your brain

Fool around in the Java Library

Avoid embarassing OO mistakes

Bend your mind around 42 Java puzzles

Make attractive and useful GUIs

O'REILLY®

Kathy Sierra & Bert Bates