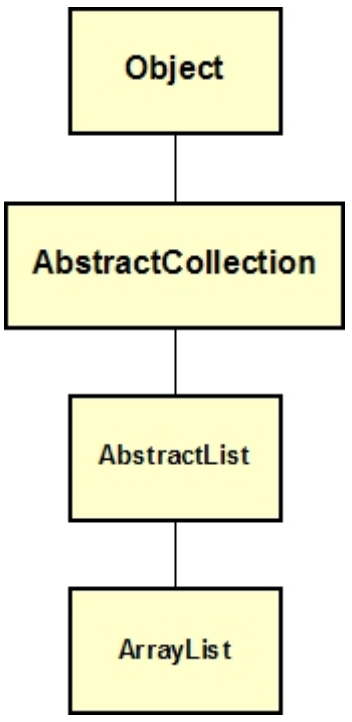
Chapter 3: ArrayList and Its Hierarchy

ArrayList and Its Hierarchy

In Lesson 3, we used an ArrayList, and I've mentioned Java collections a couple of times. In the rest of this lesson, we'll tie them together and see how they relate in Java's class hierarchy. Then we'll build our own smaller version of the hierarchy to see how it's done.

To get started, let's look up ArrayList in the API. We'll look at two items there to understand its inheritance. First, we'll look at the classes it inherits from:



ArrayList class hierarchy

In addition to Object, it inherits from two other classes: AbstractList and AbstractCollection. While we're looking at what ArrayList inherits from them, we'll see what an *abstract class* is, too.

Second, we'll look at its declaration:

```
public class ArrayList<E>

extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, Serializable
```

Besides extending AbstractList, ArrayList also *implements* four *interfaces*: List<E>, RandomAccess, Cloneable, and Serializable. We'll focus on List<E> since it contains the properties we're interested in today. The others work the same way, so we'll ignore them in this lesson in the interest of time and space. In the next chapter, we'll talk about what the <E> means in these declarations, too.

ArrayList's Class Inheritance

We already know that ArrayList inherits from AbstractList and AbstractCollection. Their declarations in the API both contain a keyword we haven't seen before: *abstract*. When we look at their methods, we see that each of them has at least one method with that keyword. Any class with an abstract method is an *abstract class*.

The abstract keyword in a method declaration tells Java that we're going to declare this method, but we're not going to implement it yet. In other words, we're going to tell Java what the method's name is, what its parameter list looks like, and what type of value it will return, but nothing else. That means the method won't work!

Since the method won't work, Java won't let us create objects from an abstract class. Every class we have created until now has been a *concrete* class, one that Java can create objects from. ArrayList is a concrete class, so we can create objects from it. What makes it different from AbstractList or AbstractCollection? And why would we want to create a class that can't be *instantiated* (can't create objects)?

Let's take a step back and talk in more general terms. Lists are something we use every day. We use shopping lists, to-do lists, guest lists, and other lists all the time. Java defines a list as an *ordered collection*. That just means a list is a bunch of things in a specific order, usually the order that we enter them.

In Java, there are several ways to create lists like that. (What those ways are is beyond the scope of this course. They use data structures we don't have time to cover.) When the creators of the Java class library set up the List mechanism, they didn't want to force programmers to use only one type of list. They also didn't want programmers who use lists to have to duplicate a lot of code to have different types of lists.

So they set up the AbstractList class. This class implements list methods that don't have to be changed for different list types. The class doesn't implement list methods that do have to change—instead, it leaves these methods abstract. Then, when someone makes a concrete list class (like ArrayList), it inherits all the common methods without having to rewrite them. It also inherits the abstract method with its name, parameter list, and return type. Before the class can be a concrete class, it must implement any abstract methods it inherits.

Getting back to specifics, AbstractList contains one abstract method: get(). Since ArrayList inherits it, the author of ArrayList had to write a get() method that does exactly what the author of AbstractList said it should do: get an item from the list. Since the author of ArrayList knows what structure it uses, he or she wrote an efficient get() method for that structure and gave us a concrete class that we can use for our lists.

ArrayList also inherits one abstract method from AbstractCollection: size(). ArrayList implements that method, too.

Now, if you'll bear with me through one more topic, we'll put all this to use.

Suppose I know what I want certain groups of classes (like lists) to do. Let's take it one step further than a class with one, or even a few, abstract methods. I know exactly what methods I want this group of classes to implement and what those methods should do. But since the different classes will implement them *all* differently, I won't define *any* of them yet; every one of them will be abstract.

Java has a special name for a completely abstract class. It's called an *interface*. The name comes from the meaning of the word *interface*, which describes how two entities interact. An interface in Java defines the way any concrete class that inherits from it will interact with other classes. Java interfaces are defined a little bit differently than classes are, and we'll look at the differences in just a minute. First, I want to reiterate the main idea of an interface: It defines a set of abstract methods that any class inheriting from it must implement.

We've been discussing two Java interfaces in this lesson without recognizing them as such. They are the *List* and *Collection* interfaces. Going back to the Java API, the AbstractCollection class *implements* the Collection interface. Likewise, the AbstractList class implements the List interface. Notice, too, that the List interface inherits from the Collection interface. That means that any concrete List class (like ArrayList) must implement all the methods defined in *both* interfaces.

Now let's bring it down to where "the rubber meets the road." What does this all mean in practice? If I want to create a concrete list class in Java that inherits from the List interface, there is already a group of defined methods that I have to implement in my class. The advantage is that anyone who uses my class can exchange it for any other concrete List class and use all the same method calls.

For example, Java has another List class named LinkedList. It is a very different type of list from ArrayList. For some processes, ArrayList is much faster. For other processes, LinkedList is faster. So I might want to use each of them in different situations. Since they both inherit from the List interface, they define the same methods to do the same list actions, even though those same methods are implemented differently in each case. But I can write a program using an ArrayList, then change it to use a LinkedList without changing any other code because they both implement all the List methods. Pretty convenient, huh?