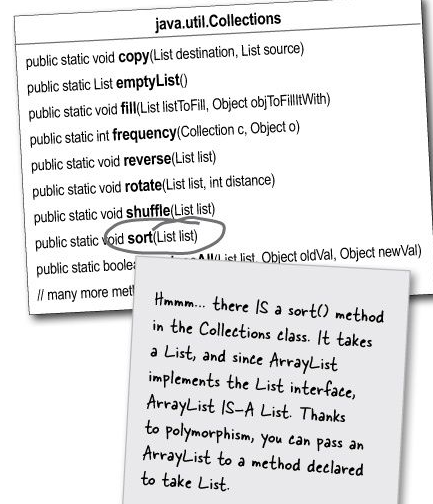
**B”H**

**Chapter 16**

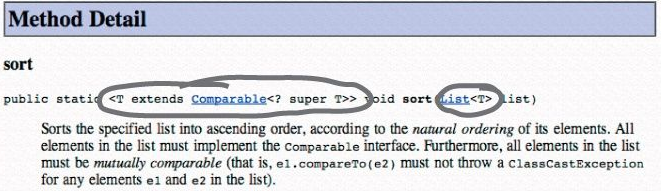
**Collections and Generics: Data structures**

* Although **ArrayList** is the one you’ll use most often, there are others.
* Some of the key collection classes include:
  + **TreeSet:**
    - Keeps the elements sorted and prevents duplicates.
    - TreeSet might be more expensive than you need — every time you insert into a TreeSet, the TreeSet has to take the time to figure out where in the tree the new element must go.
  + **HashMap:** Lets you store and access elements as name/value pairs.
  + **LinkedList:** Makes it easy to create structures like stacks or queues.
  + **HashSet:** Prevents duplicates in the collection, and given an element, can find that element in the collection quickly.
  + **LinkedHashMap:** Like a regular HashMap, except it can remember the order in which elements (name/ value pairs) were inserted, or it can be configured to remember the order in which elements were last accessed.

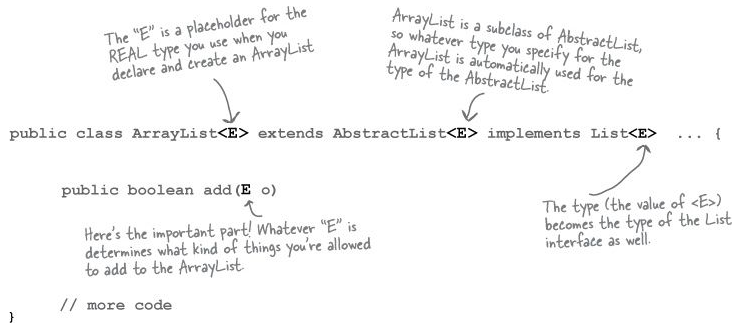


* It’s slower to insert something in an ArrayList somewhere other than at the end. So using the overloaded add( index, element) method doesn’t work as quickly as calling the add(element) — which puts the added element at the end. Most of the time you use ArrayLists, you won’t need to put something at a specific index.
* See Jukebox1.java

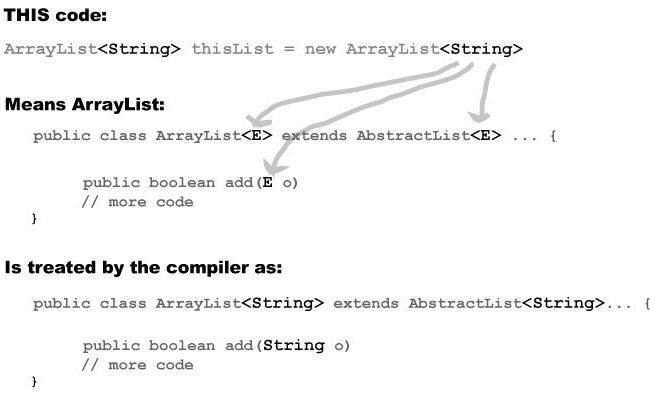
**Generics**



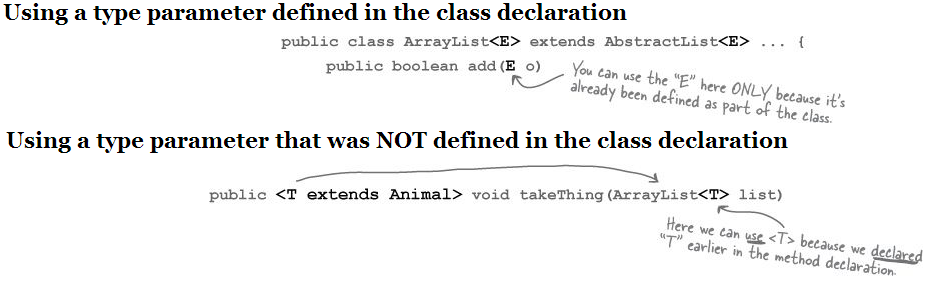
* Look up at the sort() method in the java.util.Collections class.
  + The sort() method (along with other things in the whole collection framework in Java) makes heavy use of generics.
  + Anytime you see something with angle brackets in Java source code or documentation, it means generics — a feature added to Java 5.0.
* Generics means more type-safety
* Virtually all of the code you write that deals with generics will be collection-related code.
* 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, there was no way to declare the type of an ArrayList, so its add() method took type Object.
* Now with generics, you can put only Fish objects in the ArrayList<Fish>, so the objects come out as Fish references.
* Regarding generics, there are really only three things that matter to most programmers:
  + 1. **Creating instances of generified classes (like ArrayList).** Such as: **new ArrayList<Song>()**
    2. **Declaring and assigning variables of generic types.** How does polymorphism really work with generic types?
       - If you have an **ArrayList<Animal>** reference variable, you **cannot** assign an **ArrayList<Dog>** to it.
       - If you have a method that takes as a parameter **ArrayList<Animal>** you cannot pass it an **ArrayList<Dog>**
       - However this **is** possible: **List<Song> songList = new ArrayList<Song>()**

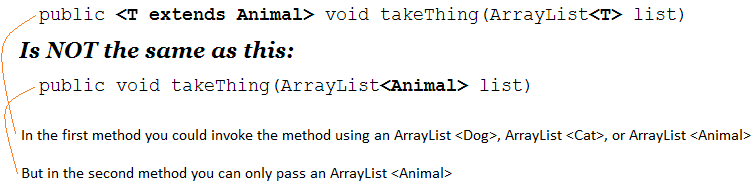


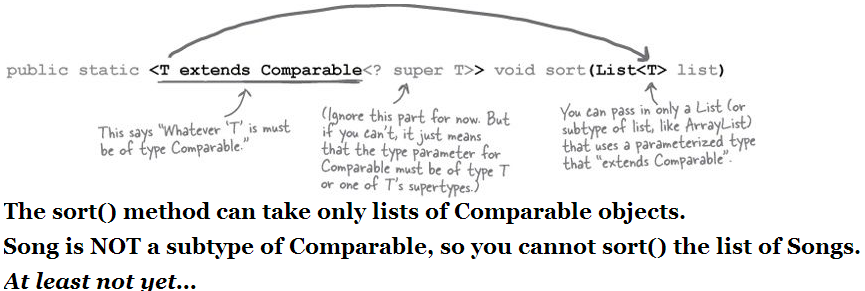
* When you see an “E” in the ArrayList documentation, you can do a mental find/ replace to exchange it for whatever < type > you use to instantiate ArrayList.
* In other words, the “E” is replaced by the real type (also called the type parameter) that you use when you create the ArrayList.



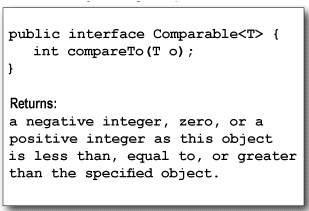
* You don’t need to use **E**. You can use anything. But the convention is to use a single letter. A further convention is to use “T” unless you’re specifically writing a collection class, where you’d use “E” to represent the “type of the Element the collection will hold”.
* A generic class means that the class declaration includes a type parameter. A generic method means that the method declaration uses a type parameter in its signature.







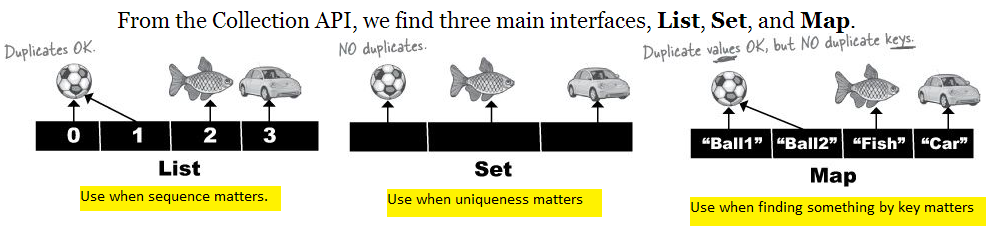
* In generics, “extends” means “extends or implements” – i.e. “extends” really means “is-a”, and works for BOTH classes and interfaces.
* See Jukebox3.java and notice the .split("/") method.



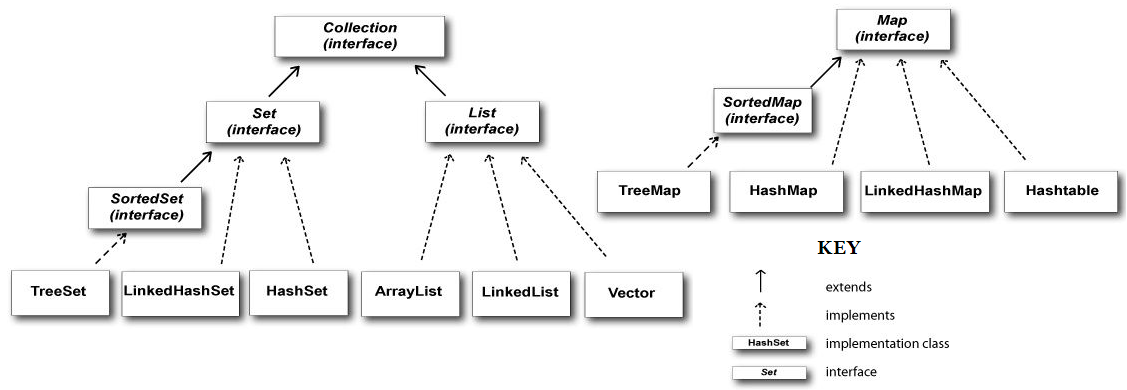
* See the Song class where it implements the Comparable interface (implements the compareTo method)

**Using a custom Comparator**

* There’s a second sort() method — and it takes a Comparator.
* An element in a list can compare itself to another of its own type in only one way, using its compareTo() method.
* If you pass a Comparator to the sort() method, the sort order is determined by the Comparator rather than the element’s own compareTo() method.
* Want to compare songs by artist? Make an ArtistComparator. Sort by beats per minute? Make a BPMComparator.
* So, the rules are:
  + Invoking the one-argument sort(List o) method means the list element’s compareTo() method determines the order. So the elements in the list MUST implement the Comparable interface.
  + Invoking sort(List o, Comparator c) means the list element’s compareTo() method will NOT be called, and the Comparator’s compare() method will be used instead. That means the elements in the list do NOT need to implement the Comparable interface.
* See Jukebox5.java

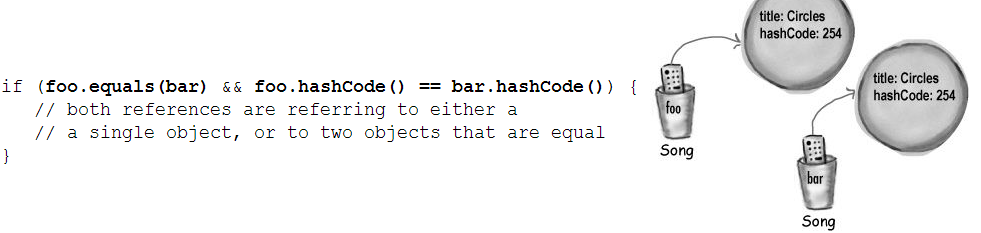


* Note that the Map interface doesn’t actually extend the Collection interface, but Map is still considered part of the “Collection Framework”
* The Collection API (part of it):

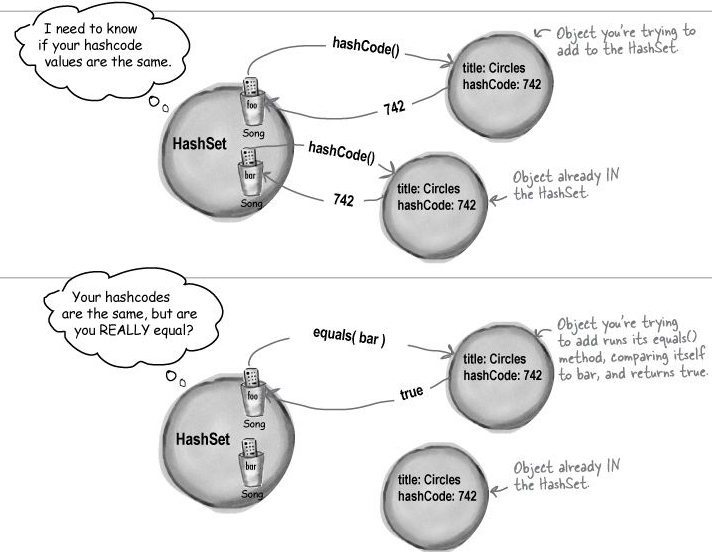


**Reference Equality vs. Object Equality**

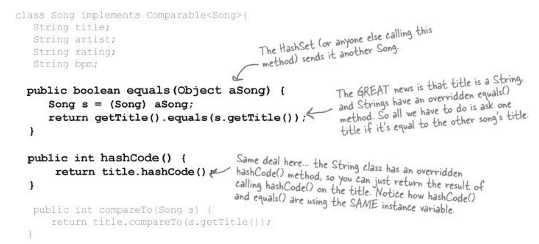
* If two objects foo and bar are equal, foo.equals(bar) must be true, and both foo and bar must return the same value from hashCode().
* For a Set to treat two objects as duplicates, you must override the hashCode() and equals() methods inherited from class Object, so that you can make two different objects be viewed as equal.
* **Reference equality**
  + Two references that point to one object on the heap.
  + If you want to know if two references are really referring to the same object, use the == operator.
* **Object equality**
  + Two references that point to two objects on the heap, but the objects are considered meaningfully equivalent.
  + If you want to treat two different Song objects as equal you must override both the hashCode() and equals() methods inherited from class Object. If you don’t override hashCode(), the default behavior (from Object) is to give each object a unique hashcode value. You must also override equals() so that if you call it on either object, passing in the other object, always returns true.



* A HashSet checks for duplicates by using hashCode() and equals()
  + When you put an object into a Hashset, it uses the object’s hashcode value to determine where to put the object in the Set.
  + But two objects with the same hashCode() might not be equal (more on this on the next page), so if the HashSet finds a matching hashcode for two objects — one you’re inserting and one already in the set — the HashSet will then call one of the object’s equals() methods to see if these hashcode-matched objects really are equal. And if they’re equal, the HashSet knows that the object you’re attempting to add is a duplicate of something in the Set, so the add doesn’t happen. You don’t get an exception, but the HashSet’s add() method returns a boolean to tell you (if you care) whether the new object was added. So if the add() method returns false, you know the new object was a duplicate of something already in the set.



* See Jukebox6.java Song.java – note the following:



**REVIEW OF JAVA OBJECT LAW FOR HASHCODE() AND EQUALS()**

* + If two objects are equal, they MUST have matching hashcodes.
  + If two objects have the same hashcode value, they are NOT required to be equal.
  + So, if you override equals(), you MUST override hashCode().
* Hashcodes can be the same without necessarily guaranteeing that the objects are equal, because the “hashing algorithm” used in the hashCode() method might happen to return the same value for multiple objects. Hashcode values are sometimes used to narrow down the search, but to find the one exact match, the HashSet still has to take all the objects in that one bucket (the bucket for all objects with the same hashcode) and then call equals() on them to see if the object it’s looking for is in that bucket.

**TreeSet**

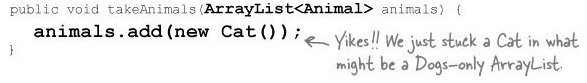
* TreeSet is similar to HashSet in that it prevents duplicates. But it also keeps the list sorted.
* It works just like the sort() method in that if you make a TreeSet using the set’s no-arg constructor, the TreeSet uses each object’s compareTo() method for the sort. But you have the option of passing a Comparator to the TreeSet constructor, to have the TreeSet use that instead.
* The downside to TreeSet is that if you don’t need sorting, you’re still paying for it with a small performance hit. But you’ll probably find that the hit is almost impossible to notice for most apps.
* To use a TreeSet, one of these things must be true:
* The elements in the list must be of a type that implements Comparable
* You use the TreeSet’s overloaded constructor that takes a Comparator
* See BookCompareTest.java which uses both ways

**Maps**

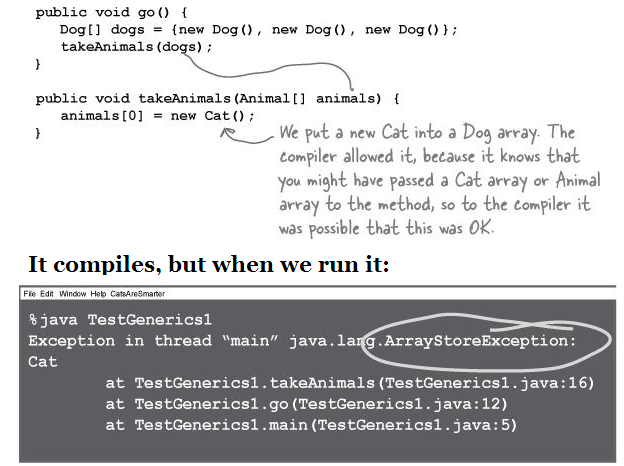
* Each element in a Map is actually TWO objects — a key and a value.
* You can have duplicate values, but NOT duplicate keys.
* When you print a Map, it gives you the key = value, in braces { } instead of the brackets [ ] you see when you print lists and sets.
* See TestMap.java

**Polymorphism with Generics**

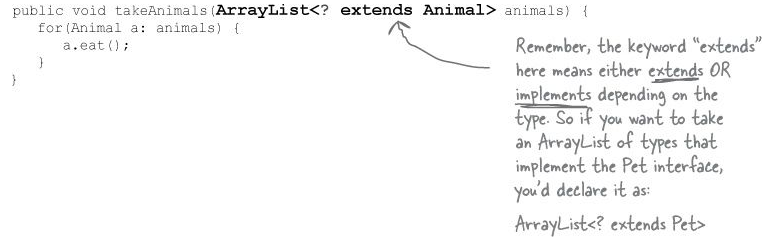
* If you declare a method to take ArrayList<Animal> it can take ONLY an ArrayList<Animal>, not ArrayList<Dog> or ArrayList<Cat>
* Imagine if it did, look what could happen:



* If you passed a Dog ArrayList to the above method, then suddenly you’d end up with a Cat in the Dog list.
* So instead, the compiler just won’t let you take the risk.
* Array types are checked again at runtime, but collection type checks happen only when you compile



* There **IS** a way to create a method argument that can accept an ArrayList of any Animal subtype. The simplest way is to use a wildcard — added to the Java language explicitly for this reason.



* When you use a wildcard in your method argument, the compiler will STOP you from doing anything that could hurt the list referenced by the method parameter.
* You can still invoke methods on the elements in the list, but you cannot add elements to the list.
* I.e. the compiler won’t let you do anything that might be horrible at runtime.
* Alternate syntax for doing the same thing:
  + public <T extends Animal> void takeThing(ArrayList <T> list)
  + public void takeThing(ArrayList <? extends Animal> list)
* The first way can save typing in some cases such as:
  + public <T extends Animal> void takeThing(ArrayList <T> one, ArrayList <T> two)
  + public void takeThing(ArrayList <? extends Animal> one, ArrayList <? extends Animal> two)