16 collections and generics

Data structures

> Sheesh... and all this time I could have just let Java put things in alphabetical order? Third grade really sucks. We never learn anything useful...

Sorting is a snap in Java. You have all the tools for collecting and manipulating your data without having to write your own sort algorithms (unless you're reading this right now sitting in your Computer Science 101 class, in which case, trust us—you are SO going to be writing sort code while the rest of us just call a method in the Java API). The Java Collections Framework has a data structure that should work for virtually anything you'll ever need to do. Want to keep a list that you can easily keep adding to? Want to find something by name? Want to create a list that automatically takes out all the duplicates? Sort your co-workers by the number of times they've stabbed you in the back? Sort your pets by number of tricks learned? It's all here...

Tracking song popularity on your jukebox

Congratulations on your new job—managing the automated jukebox system at Lou's Diner. There's no Java inside the jukebox itself, but each time someone plays a song, the song data is appended to a simple text file.
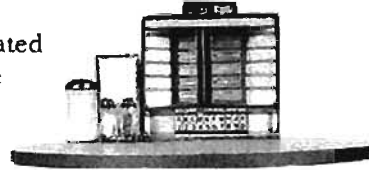
Your job is to manage the data to track song popularity, generate reports, and manipulate the playlists. You're not writing the entire app—some of the other software developer/ waiters are involved as well, but you're responsible for managing and sorting the data inside the Java app. And since Lou has a thing against databases, this is strictly an in-memory data collection. All you get is the file the jukebox keeps adding to. Your job is to take it from there.

You've already figured out how to read and parse the file, and so far you've been storing the data in an ArrayList.

SongList.txt

```
Pink Moon/Nick Drake
Somersault/Zero 7
Shiva Moon/Prem Joshua
Circles/BT
Deep Channel/Afro Celts
Passenger/Headmix
Listen/Tahiti 80
```

This is the file the jukebox device writes. Your code must read the file, then manipulate the song data.

Challenge #1
Sort the songs in alphabetical order

You have a list of songs in a file, where each line represents one song, and the title and artist are separated with a forward slash. So it should be simple to parse the line, and put all the songs in an ArrayList.

Your boss cares only about the song titles, so for now you can simply make a list that just has the song titles.

But you can see that the list is not in alphabetical order... what can you do?

You know that with an ArrayList, the elements are kept in the order in which they were inserted into the list, so putting them in an ArrayList won't take care of alphabetizing them, unless... maybe there's a sort() method in the ArrayList class?

Here's what you have so far, without the sort:

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}
```

We'll store the song titles in an ArrayList of Strings

The method that starts loading the file and then prints the contents of the songList ArrayList

Nothing special here... just read the file and call the addSong() method for each line.

The addSong method works just like the Quiz-Card in the I/O chapter—you break the line (that has both the title and artist) into two pieces (tokens) using the split() method.

We only want the song title, so add only the first token to the SongList (the ArrayList).

```
File Edit Window Help Dance
%java Jukebox1
[Pink Moon, Somersault,
Shiva Moon, Circles,
Deep Channel, Passenger,
Listen]
```

The songList prints out with the songs in the order in which they were added to the ArrayList (which is the same order the songs are in within the original text file).

This is definitely NOT alphabetical!

But the ArrayList class does NOT have a sort() method!

When you look in ArrayList, there doesn't seem to be any method related to sorting. Walking up the inheritance hierarchy didn't help either—it's clear that *you can't call a sort method on the ArrayList.*

Handwritten note on screenshot:

ArrayList has a lot of methods, but there's nothing here that looks like it would sort...

I do see a collection class called TreeSet... and the docs say that it keeps your data sorted. I wonder if I should be using a TreeSet instead of an ArrayList...

ArrayList is <u>not</u> the only collection

Although ArrayList is the one you'll use most often, there are others for special occasions. Some of the key collection classes include:

Don't worry about trying to learn these other ones right now. We'll go into more details a little later

➤ **TreeSet**
Keeps the elements sorted and prevents duplicates.

➤ **HashMap**
Let's you store and access elements as name/value pairs.

➤ **LinkedList**
Designed to give better performance when you insert or delete elements from the middle of the collection. (In practice, an ArrayList is still usually what you want.)

➤ **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.

You *could* use a TreeSet...
Or you could use the <u>Collections.sort()</u> method

If you put all the Strings (the song titles) into a **TreeSet** instead of
an ArrayList, the Strings would automatically land in the right place,
alphabetically sorted. Whenever you printed the list, the elements would
always come out in alphabetical order.

And that's great when you need a *set* (we'll
talk about sets in a few minutes) or when
you know that the list must *always* stay
sorted alphabetically.

On the other hand, if you don't need the
list to stay sorted, 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*. With ArrayList, inserts can
be blindingly fast because the new element
just goes in at the end.

java.util.Collections
public static void **copy**(List destination, List source)
public static List **emptyList**()
public static void **fill**(List listToFill, Object objToFillItWith)
public static int **frequency**(Collection c, Object o)
public static void **reverse**(List list)
public static void **rotate**(List list, int distance)
public static void **shuffle**(List list)
public static void **sort**(List list)
public static boole~~ ~~ AU/list Object oldVal, Object newVal)
// many more met~~ ~~

Hmmm... there IS a sort() method
in the Collections class. It takes
a List, and since ArrayList
implements the List interface,
ArrayList IS-A List. Thanks
to polymorphism, you can pass an
ArrayList to a method declared
to take List.

Q: But you CAN add something to an
ArrayList at a specific index instead of just at
the end—there's an overloaded add() method
that takes an int along with the element to add.
So wouldn't it be slower than inserting at the end?

A: Yes, 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.
But most of the time you use ArrayLists, you won't need to put
something at a specific index.

Q: I see there's a LinkedList class, so wouldn't *that* be better for
doing inserts somewhere in the middle? At least if I remember my Data
Structures class from college...

A: Yes, good spot. The LinkedList *can* be quicker when you insert or
remove something from the middle, but for most applications, the difference
between middle inserts into a LinkedList and ArrayList is usually not enough
to care about unless you're dealing with a *huge* number of elements. We'll
look more at LinkedList in a few minutes.

Note: this is NOT the real Collections
class API; we simplified it here by leaving
out the generic type information (which
you'll see in a few pages).

Adding **Collections.sort()** to the Jukebox code

```
import java.util.*;
import java.io.*;

public class Jukebox1 {

    ArrayList<String> songList = new ArrayList<String>();

    public static void main(String[] args) {
        new Jukebox1().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }

    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        songList.add(tokens[0]);
    }
}
```

The Collections.sort() method sorts a list of Strings alphabetically.

Call the static Collections sort() method, then print the list again. The second print out is in alphabetical order!

```
File Edit Window Help Chill
%java Jukebox1

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

Before calling sort().

After calling sort().

But now you need Song objects, not Just simple Strings.

Now your boss wants actual Song class instances in the list, not just Strings, so that each Song can have more data. The new jukebox device outputs more information, so this time the file will have *four* pieces (tokens) instead of just two.

The Song class is really simple, with only one interesting feature—the overridden toString() method. Remember, the toString() method is defined in class Object, so every class in Java inherits the method. And since the toString() method is called on an object when it's printed (System.out.println(anObject)), you should override it to print something more readable than the default unique identifier code. When you print a list, the toString() method will be called on each object.

SongListMore.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
```

The new song file holds four attributes instead of just two. And we want ALL of them in our list, so we need to make a Song class with instance variables for all four song attributes.

```
class Song {
    String title;          Four instance variables for the
    String artist;         four song attributes in the file.
    String rating;
    String bpm;

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;        The variables are all set in
        rating = r;        the constructor when the
        bpm = b;           new Song is created.
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }
                           The getter methods for
                           the four attributes.
    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {    ←  We override toString(), because when you do a System.
        return title;                out.println(aSongObject), we want to see the title.
    }                                When you do a System.out.println(aListOfSongs), it calls
}                                    the toString() method of EACH element in the list.
```

Changing the Jukebox code to use Songs instead of Strings

Your code changes only a little—the file I/O code is the same, and the parsing is the same (String.split()), except this time there will be *four* tokens for each song/line, and all four will be used to create a new Song object. And of course the ArrayList will be of type <Song> instead of <String>.

```java
import java.util.*;
import java.io.*;

public class Jukebox3 {

    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");

        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

Change to an ArrayList of Song objects instead of String.

Create a new Song object using the four tokens (which means the four pieces of info in the song file for this line), then add the Song to the list.

Collections.sort()
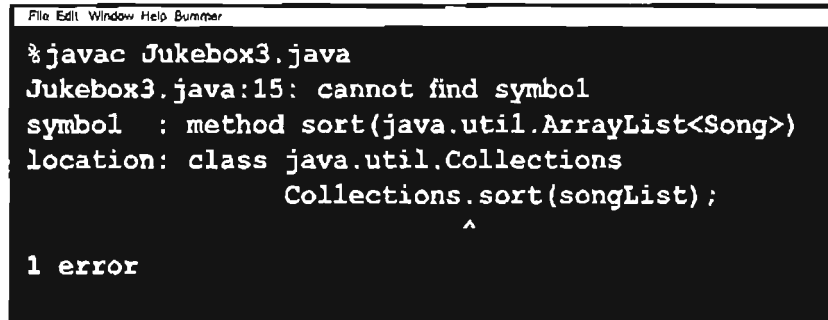
It won't compile!

Something's wrong... the Collections class clearly shows there's a sort() method, that takes a List.

ArrayList is-a List, because ArrayList implements the List interface, so... it *should* work.

But it doesn't!

The compiler says it can't find a sort method that takes an ArrayList<Song>, so maybe it doesn't like an ArrayList of Song objects? It didn't mind an ArrayList<String>, so what's the important difference between Song and String? What's the difference that's making the compiler fail?

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol  : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                Collections.sort(songList);
                ^
1 error
```

And of course you probably already asked yourself, "What would it be sorting *on?*" How would the sort method even *know* what made one Song greater or less than another Song? Obviously if you want the song's *title* to be the value that determines how the songs are sorted, you'll need some way to tell the sort method that it needs to use the title and not, say, the beats per minute.

We'll get into all that a few pages from now, but first, let's find out why the compiler won't even let us pass a Song ArrayList to the sort() method.

> WTF? I have no idea how to read the method declaration on this. It says that sort() takes a List<T>, but what is T? And what is that big thing before the return type?

The sort() method declaration

Collections (Java 2 Platform SE 5.0)

file:///Users/kathy/Public/docs/api/index.html — Q▾ Google

Method Detail

sort

public static <T extends Comparable<? super T>> void sort(List<T> list)

Sorts the specified list into ascending order, according to the *natural ordering* of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

From the API docs (looking up the java.util.Collections class, and scrolling to the sort() method), it looks like the sort() method is declared... *strangely.* Or at least different from anything we've seen so far.

That's because 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. So it looks like we'll have to learn how to interpret the documentation before we can figure out why we were able to sort String objects in an ArrayList, but not an ArrayList of Song objects.

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
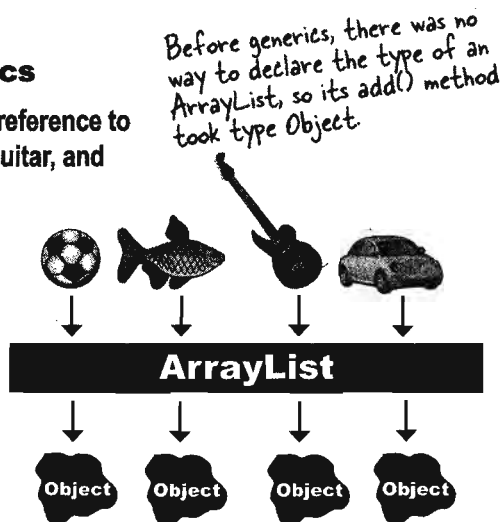
And come OUT as a reference of type Object

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
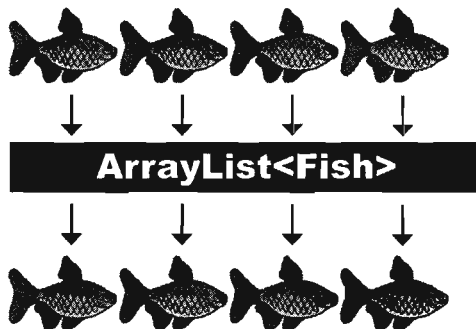
And come out as a reference of type Fish

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.

Learning generics

Of the dozens of things you could learn about generics, there are really only three that matter to most programmers:

● **Creating instances of generified _classes_ (like ArrayList)**

When you make an ArrayList, you have to tell it the type of objects you'll allow in the list, just as you do with plain old arrays.

```
new ArrayList<Song>()
```

● **Declaring and assigning _variables_ of generic types**

How does polymorphism really work with generic types? If you have an ArrayList<Animal> reference variable, can you assign an ArrayList<Dog> to it? What about a List<Animal> reference? Can you assign an ArrayList<Animal> to it? You'll see...

```
List<Song> songList =
        new ArrayList<Song>()
```

● **Declaring (and invoking) _methods_ that take generic types**

If you have a method that takes as a parameter, say, an ArrayList of Animal objects, what does that really mean? Can you also pass it an ArrayList of Dog objects? We'll look at some subtle and tricky polymorphism issues that are very different from the way you write methods that take plain old arrays.

(This is actually the same point as #2, but that shows you how important we think it is.)

```
void foo(List<Song> list)

x.foo(songList)
```

Q: But don't I also need to learn how to create my OWN generic classes? What if I want to make a class type that lets people instantiating the class decide the type of things that class will use?

A: You probably won't do much of that. Think about it—the API designers made an entire library of collections classes covering most of the data structures you'd need, and virtually the only type of classes that really need to be generic are collection classes. In other words, classes designed to hold other elements, and you want programmers using it to specify what type those elements are when they declare and instantiate the collection class.

Yes, it is possible that you might want to _create_ generic classes, but that's the exception, so we won't cover it here. (But you'll figure it out from the things we _do_ cover, anyway.)

Using generic CLASSES

Since ArrayList is our most-used generified type, we'll start by looking at its documentation. They two key areas to look at in a generified class are:

1) The *class* declaration

3) The *method* declarations that let you add elements

Think of "E" as a stand-in for "the type of element you want this collection to hold and return." (E is for Element.)

Understanding ArrayList documentation
(Or, what's the true meaning of "E"?)

The "E" is a placeholder for the REAL type you use when you declare and create an ArrayList

ArrayList is a subclass of AbstractList, so whatever type you specify for the ArrayList is automatically used for the type of the AbstractList.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>  ... {

        public boolean add(E o)
```

Here's the important part! Whatever "E" is determines what kind of things you're allowed to add to the ArrayList.

The type (the value of <E>) becomes the type of the List interface as well.

```
        // more code
}
```

The "E" represents the type used to create an instance of ArrayList. 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.
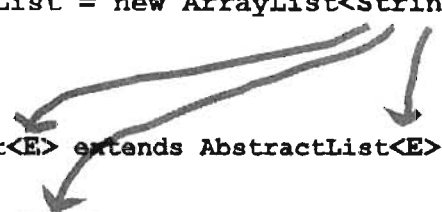
So, new ArrayList<Song> means that "E" becomes "Song", in any method or variable declaration that uses "E".

Using type parameters with ArrayList

THIS code:

```
ArrayList<String> thisList = new ArrayList<String>
```

Means ArrayList:

```
    public class ArrayList<E> extends AbstractList<E> ... {

        public boolean add(E o)
        // more code
    }
```

Is treated by the compiler as:

```
    public class ArrayList<String> extends AbstractList<String>... {

        public boolean add(String o)
        // more code
    }
```

In other words, the "E" is replaced by the *real* type (also called the *type parameter*) that you use when you create the ArrayList. And that's why the add() method for ArrayList won't let you add anything except objects of a reference type that's compatible with the type of "E". So if you make an ArrayList<**String**>, the add() method suddenly becomes **add(String** o). If you make the ArrayList of type **Dog**, suddenly the add() method becomes **add(Dog** o).

Q: Is "E" the only thing you can put there? Because the docs for sort used "T"...

A: You can use anything that's a legal Java identifier. That means anything that you could use for a method or variable name will work as a type parameter. But the convention is to use a single letter (so that's what you should use), and 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".

.

Using generic <u>METHODS</u>

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:

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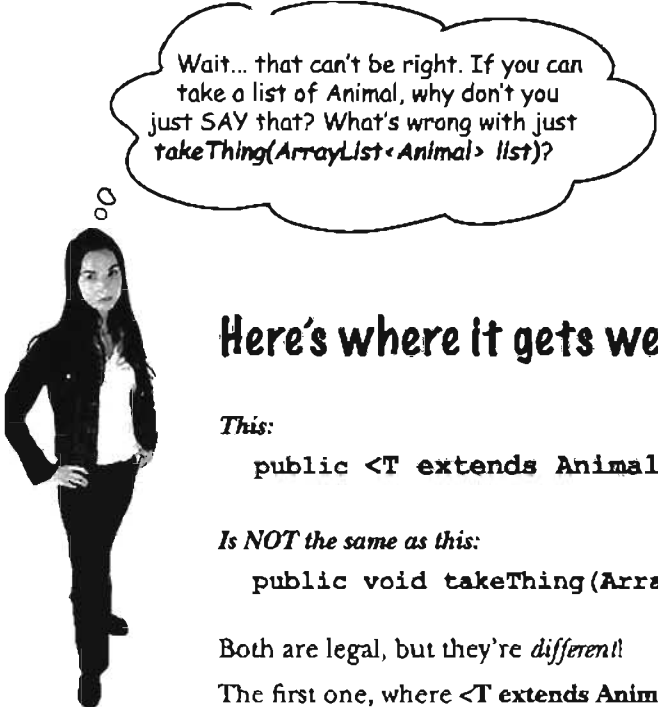
When you declare a type parameter for the class, you can simply use that type any place that you'd use a *real* class or interface type. The type declared in the method argument is essentially replaced with the type you use when you instantiate the class.

● **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...

This still doesn't explain why the sort method failed on an ArrayList of Songs but worked for an ArrayList of Strings...

Remember where we were...

```
File Edit Window Help Bummer
%javac Jukebox3.java
Jukebox3.java:15: cannot find symbol
symbol   : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
                 Collections.sort(songList);
                             ^
1 error
```

```java
import java.util.*;
import java.io.*;

public class Jukebox3 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox3().go();
    }
    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
    }
    void getSongs() {
        try {
            File file = new File("SongList.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line =  null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

This is where it breaks! It worked fine when passed in an ArrayList<String>, but as soon as we tried to sort an ArrayList<Song>, it failed.

Revisiting the sort() method

So here we are, trying to read the sort() method docs to find
out why it was OK to sort a list of Strings, but not a
list of Song objects. And it looks like the answer is...

**The sort() method can take only lists
of Comparable objects.**

**Song is NOT a subtype of
Comparable, so you cannot sort()
the list of Songs.**

At least not yet...

```
Collections (Java 2 Platform SE 5.0)
file:///Users/kathy/Public/docs/api/index.html
Q▼ Google

Method Detail

sort

public static <T extends Comparable<? super T>> void sort(List<T> list)
    Sorts the specified list into ascending order, according to the natural ordering of its elements. All
    elements in the list must implement the Comparable interface. Furthermore, all elements in the list
    must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException
    for any elements e1 and e2 in the list).
```

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This says "Whatever 'T' is must
be of type Comparable."

(Ignore this part for now. But
if you can't, it just means
that the type parameter for
Comparable must be of type T
or one of T's supertypes.)

You can pass in only a List (or
subtype of list, like ArrayList)
that uses a parameterized type
that "extends Comparable".

> Um... I just checked the docs for
> String, and String doesn't EXTEND
> Comparable--it IMPLEMENTS it.
> **Comparable** is an **Interface**. So it's nonsense
> to say <T *extends* Comparable>.

```
public final class String extends Object implements Serializable,
                                    Comparable<String>, CharSequence
```

In generics, "extends" means "extends or implements"

The Java engineers had to give you a way to put a constraint on a parameterized type, so that you can restrict it to, say, only subclasses of Animal. But you also need to constrain a type to allow only classes that implement a particular interface. So here's a situation where we need one kind of syntax to work for both situations—inheritance and implementation. In other words, that works for both *extends* and *implements*.

And the winning word was... *extends*. But it really means "is-a", and works regardless of whether the type on the right is an interface or a class.

In generics, the keyword "extends" really means "is-a", and works for BOTH classes and interfaces.

Comparable is an interface, so this REALLY reads, "T must be a type that implements the Comparable interface".

↓

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

↱

It doesn't matter whether the thing on the right is a class or interface... you still say "extends".

Q: Why didn't they just make a new keyword, "is"?

A: Adding a new keyword to the language is a REALLY big deal because it risks breaking Java code you wrote in an earlier version. Think about it—you might be using a variable "is" (which we do use in this book to represent input streams). And since you're not allowed to use keywords as identifiers in your code, that means any earlier code that used the keyword *before* it was a reserved word, would break. So whenever there's a chance for the Sun engineers to reuse an existing keyword, as they did here with "extends", they'll usually choose that. But sometimes they don't have a choice...

A few (very few) new keywords *have* been added to the language, such as **assert** in Java 1.4 and **enum** in Java 5.0 (we look at enum in the appendix). And this does break people's code, however you sometimes have the option of compiling and running a *newer* version of Java so that it behaves as though it were an older one. You do this by passing a special flag to the compiler or JVM at the command-line, that says, "Yeah, yeah, I KNOW this is Java 1.4, but please pretend it's really 1.3, because I'm using a variable in my code named *assert* that I wrote back when you guys said it would OK!#$%".

(To see if you have a flag available, type *javac* (for the compiler) or *java* (for the JVM) at the command-line, without anything else after it, and you should see a list of available options. You'll learn more about these flags in the chapter on deployment.)

Finally we know what's wrong...
The Song class needs to implement Comparable

We can pass the ArrayList<Song> to the sort() method only if the Song class implements Comparable, since that's the way the sort() method was declared. A quick check of the API docs shows the Comparable interface is really simple, with only one method to implement:

Java.lang.Comparable

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

And the method documentation for compareTo() says

```
Returns:
a negative integer, zero, or a
positive integer as this object
is less than, equal to, or greater
than the specified object.
```

It looks like the compareTo() method will be called on one Song object, passing that Song a reference to a different Song. The Song running the compareTo() method has to figure out if the Song it was passed should be sorted higher, lower, or the same in the list.

Your big job now is to decide what makes one song greater than another, and then implement the compareTo() method to reflect that. A negative number (any negative number) means the Song you were passed is greater than the Song running the method. Returning a positive number says that the Song running the method is greater than the Song passed to the compareTo() method. Returning zero means the Songs are equal (at least for the purpose of sorting... it doesn't necessarily mean they're the same object). You might, for example, have two Songs with the same title.

(Which brings up a whole different can of worms we'll look at later...)

The big question is: what makes *one* song less than, equal to, or greater than *another* song?

You can't implement the Comparable interface until you make that decision.

✏ Sharpen your pencil

Write in your idea and pseudo code (or better, REAL code) for implementing the compareTo() method in a way that will sort() the Song objects by title.

Hint: If you're on the right track, it should take less than 3 lines of code!

The new, improved, comparable Song class

We decided we want to sort by title, so we implement the compareTo()
method to compare the title of the Song passed to the method against
the title of the song on which the compareTo() method was invoked.
In other words, the song running the method has to decide how its
title compares to the title of the method parameter.

Hmmm... we know that the String class must know about alphabetical
order, because the sort() method worked on a list of Strings. We know
String has a compareTo() method, so why not just call it? That way, we
can simply let one title String compare itself to another, and we don't
have to write the comparing/alphabetizing algorithm!

*Usually these match...we're specifying the type that
the implementing class can be compared against*

```
class Song implements Comparable<Song> {
    String title;
    String artist;
    String rating;
    String bpm;

    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}
```

*This means that Song objects can be compared to
other Song objects, for the purpose of sorting.*

*The sort() method sends a Song to compareTo()
to see how that Song compares to the Song on
which the method was invoked.*

*←— Simple! We just pass the work
on to the title String objects,
since we know Strings have a
compareTo() method.*

*This time it worked. It prints the list, then calls sort
which puts the Songs in alphabetical order by title.*

```
File Edit Window Help Ambient

%java Jukebox3

[Pink Moon, Somersault, Shiva Moon, Circles, Deep
Channel, Passenger, Listen]

[Circles, Deep Channel, Listen, Passenger, Pink
Moon, Shiva Moon, Somersault]
```

We can sort the list, but...

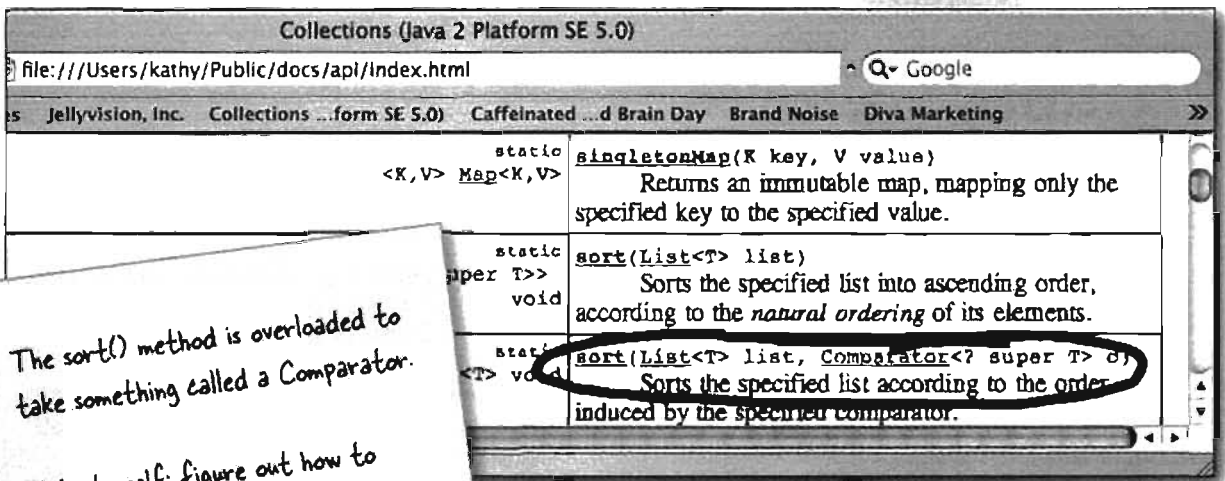There's a new problem—Lou wants two different views of the song list, one by song title and one by artist!

But when you make a collection element comparable (by having it implement Comparable), you get only one chance to implement the compareTo() method. So what can you do?

The horrible way would be to use a flag variable in the Song class, and then do an *if* test in compareTo() and give a different result depending on whether the flag is set to use title or artist for the comparison.

But that's an awful and brittle solution, and there's something much better. Something built into the API for just this purpose—when you want to sort the same thing in more than one way.

Look at the Collections class API again. There's a second sort() method—and it takes a Comparator.

> That's not good enough. Sometimes I want it to sort by artist instead of title.

static <K,V> Map<K,V>	singletonMap(K key, V value) Returns an immutable map, mapping only the specified key to the specified value.
static ...per T>> void	sort(List<T> list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
static <T> v...	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the order induced by the specified comparator.

Collections (Java 2 Platform SE 5.0)
file:///Users/kathy/Public/docs/api/index.html — Q▾ Google
Jellyvision, Inc.   Collections ...form SE 5.0)   Caffeinated ...d Brain Day   Brand Noise   Diva Marketing

The sort() method is overloaded to take something called a Comparator.

Note to self: figure out how to get/make a Comparator that can compare and order the songs by artist instead of title...

Using a custom Comparator

An element in a list can compare *itself* to another of its own type in only one way, using its compareTo() method. But a Comparator is external to the element type you're comparing—it's a separate class. So you can make as many of these as you like! Want to compare songs by artist? Make an ArtistComparator. Sort by beats per minute? Make a BPMComparator.

Then all you need to do is call the overloaded sort() method that takes the List and the Comparator that will help the sort() method put things in order.

The sort() method that takes a Comparator will use the Comparator instead of the element's own compareTo() method, when it puts the elements in order. In other words, if your sort() method gets a Comparator, it won't even *call* the compareTo() method of the elements in the list. The sort() method will instead invoke the compare() method on the Comparator.

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.

java.util.Comparator

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

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.

Q: So does this mean that if you have a class that doesn't implement Comparable, and you don't have the source code, you could still put the things in order by creating a Comparator?

A: That's right. The other option (if it's possible) would be to subclass the element and make the subclass implement Comparable.

Q: But why doesn't *every* class implement Comparable?

A: Do you really believe that *everything* can be ordered? If you have element types that just don't lend themselves to any kind of natural ordering, then you'd be misleading other programmers if you implement Comparable. And you aren't taking a huge risk by not implementing Comparable, since a programmer can compare anything in any way that he chooses using his own custom Comparator.

Updating the Jukebox to use a Comparator

We did three new things in this code:

1) Created an inner class that implements Comparator (and thus the *compare()* method that does the work previously done by *compareTo()*).

2) Made an instance of the Comparator inner class.

3) Called the overloaded sort() method, giving it both the song list and the instance of the Comparator inner class.

Note: we also updated the Song class toString() method to print both the song title and the artist. (It prints *title: artist* regardless of how the list is sorted.)

```
import java.util.*;
import java.io.*;

public class Jukebox5 {
    ArrayList<Song> songList = new ArrayList<Song>();
    public static void main(String[] args) {
        new Jukebox5().go();
    }

    class ArtistCompare implements Comparator<Song> {
        public int compare(Song one, Song two) {
            return one.getArtist().compareTo(two.getArtist());
        }
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);

        ArtistCompare  artistCompare = new ArtistCompare();
        Collections.sort(songList, artistCompare);

        System.out.println(songList);
    }

    void getSongs() {
        // I/O code here
    }

    void addSong(String lineToParse) {
        // parse line and add to song list
    }
}
```

Create a new inner class that implements Comparator (note that its type parameter matches the type we're going to compare—in this case Song objects.)

This becomes a String (the artist)

We're letting the String variables (for artist) do the actual comparison, since Strings already know how to alphabetize themselves.

Make an instance of the Comparator inner class.

Invoke sort(), passing it the list and a reference to the new custom Comparator object

Note: we've made sort-by-title the default sort, by keeping the compareTo() method in Song use the titles. But another way to design this would be to implement both the title sorting and artist sorting as inner Comparator classes, and not have Song implement Comparable at all. That means we'd always use the two-arg version of Collections.sort().

collections exercise

```
import _____;

public class SortMountains {

   LinkedList_____ mtn = new LinkedList_____();

   class NameCompare _____ {
      public int compare(Mountain one, Mountain two) {

         return _____;
      }
   }
   class HeightCompare _____ {
      public int compare(Mountain one, Mountain two) {

         return (_____);
      }
   }
   public static void main(String [] args) {
      new SortMountain().go();
   }
   public void go() {
      mtn.add(new Mountain("Longs", 14255));
      mtn.add(new Mountain("Elbert", 14433));
      mtn.add(new Mountain("Maroon", 14156));
      mtn.add(new Mountain("Castle", 14265));

      System.out.println("as entered:\n" + mtn);
      NameCompare nc = new NameCompare();

      _____;
      System.out.println("by name:\n" + mtn);
      HeightCompare hc = new HeightCompare();

      _____;
      System.out.println("by height:\n" + mtn);
   }
}

class Mountain {

   _____;

   _____;

   _____ {

      _____;

      _____;
   }

   _____ {

      _____;
   }
}
```
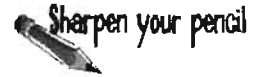
Reverse Engineer

Assume this code exists in a single file. Your job is to fill in the blanks so the the program will create the output shown.

Note: answers are at the end of the chapter.

Output:

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```

Sharpen your pencil

Fill-in-the-blanks

For each of the questions below, fill in the blank with one of the words from the "possible answers" list, to correctly answer the question. Answers are at the end of the chapter.

Possible Answers:

Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in myArrayList implement? _____

2. What method must the class of the objects stored in myArrayList implement? _____

3. Can the class of the objects stored in myArrayList implement both Comparator AND Comparable? _____

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in myArrayList implement Comparable? _____

5. Can the class of the objects stored in myArrayList implement Comparator? _____

6. Must the class of the objects stored in myArrayList implement Comparable? _____

7. Must the class of the objects stored in myArrayList implement Comparator? _____

8. What must the class of the myCompare object implement? _____

9. What method must the class of the myCompare object implement? _____

Uh-oh. The sorting all works, but now we have duplicates...

The sorting works great, now we know how to sort on both *title* (using the Song object's compareTo() method) and *artist* (using the Comparator's compare() method). But there's a new problem we didn't notice with a test sample of the jukebox text file—*the sorted list contains duplicates*.

It appears that the diner jukebox just keeps writing to the file regardless of whether the same song has already been played (and thus written) to the text file. The SongListMore.txt jukebox text file is a complete record of every song that was played, and might contain the same song multiple times.

```
File Edit Window Help TooManyNotes

%java Jukebox4


[Pink Moon: Nick Drake, Somersault: Zero 7, Shiva Moon: Prem
Joshua, Circles: BT, Deep Channel: Afro Celts, Passenger:
Headmix, Listen: Tahiti 80, Listen: Tahiti 80, Listen: Tahiti
80, Circles: BT]


[Circles: BT, Circles: BT, Deep Channel: Afro Celts, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua,
Somersault: Zero 7]


[Deep Channel: Afro Celts, Circles: BT, Circles: BT, Passenger:
Headmix, Pink Moon: Nick Drake, Shiva Moon: Prem Joshua, Listen:
Tahiti 80, Listen: Tahiti 80, Listen: Tahiti 80, Somersault:
Zero 7]
```

Before sorting.

After sorting using the Song's own compareTo() method (sort by title).

After sorting using the ArtistCompare Comparator (sort by artist name).

SongListMore.txt

```
Pink Moon/Nick Drake/5/80
Somersault/Zero 7/4/84
Shiva Moon/Prem Joshua/6/120
Circles/BT/5/110
Deep Channel/Afro Celts/4/120
Passenger/Headmix/4/100
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Listen/Tahiti 80/5/90
Circles/BT/5/110
```

The SongListMore text file now has duplicates in it, because the jukebox machine is writing every song played, in order. Somebody decided to play "Listen" three times in a row, followed by "Circles", a song that had been played earlier.

We can't change the way the text file is written because sometimes we're going to need all that information. We have to change the java code.
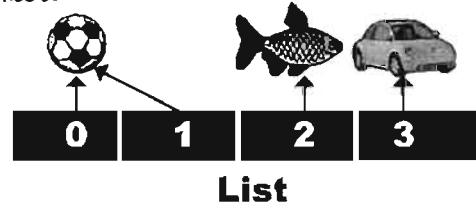
We need a Set instead of a List

From the Collection API, we find three main interfaces, **List**, **Set**, and **Map**. ArrayList is a **List**, but it looks like *Set* is exactly what we need.

Duplicates OK.

▶ **LIST** - when *sequence* matters

Collections that know about *Index position.*

Lists know where something is in the list. You can have more than one element referencing the same object.

List

NO duplicates.

▶ **SET** - when *uniqueness* matters

Collections that *do not allow duplicates.*

Sets know whether something is already in the collection. You can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal—we'll look at what object equality means in a moment).

Set

Duplicate values OK, but NO duplicate keys.
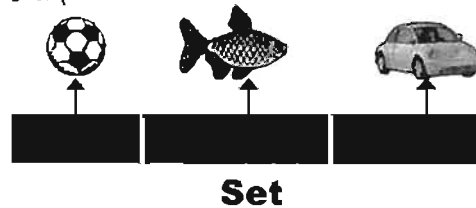
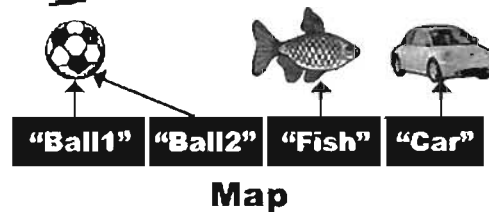▶ **MAP** - when *finding something by key* matters

Collections that use *key-value pairs.*
Maps know the value associated with a given key. You can have two keys that reference the same value, but you cannot have duplicate keys. Although keys are typically String names (so that you can make name/value property lists, for example), a key can be any object.
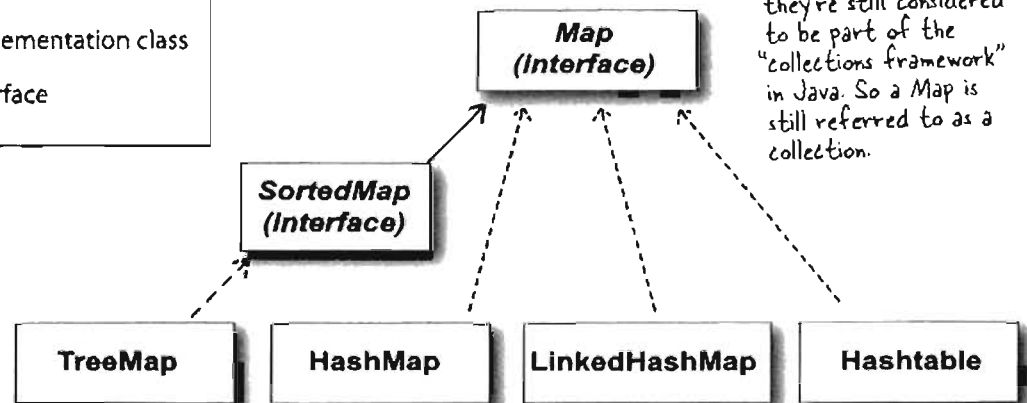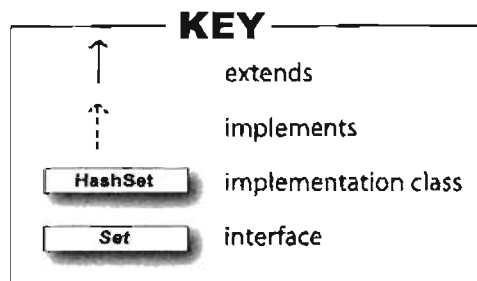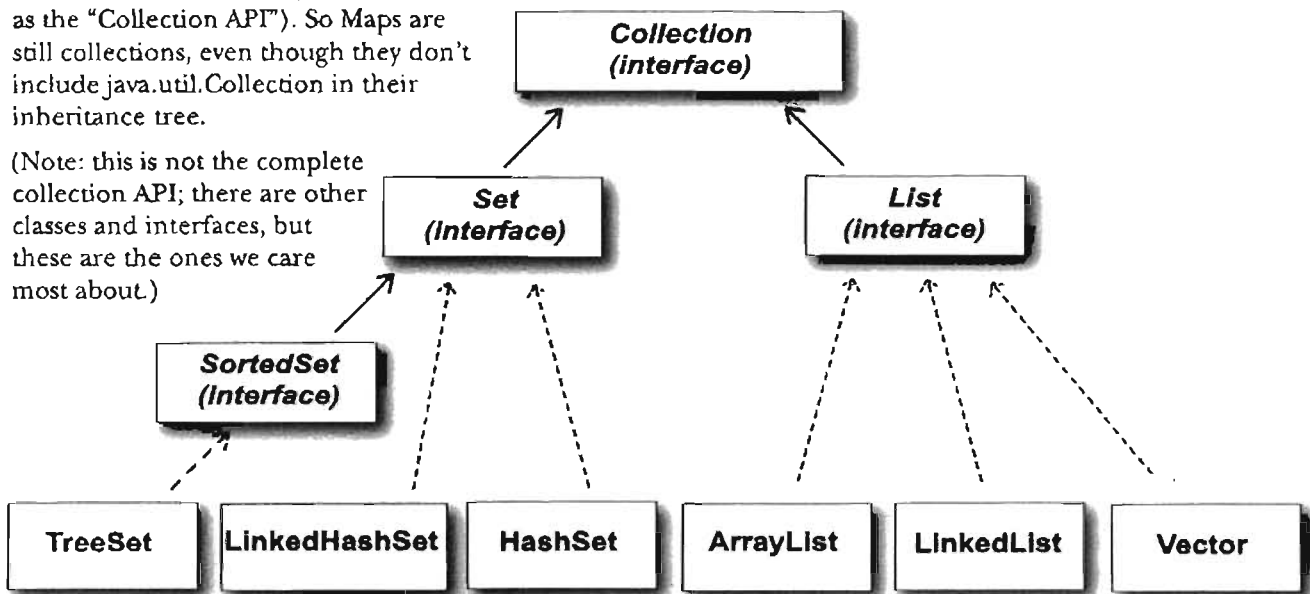
"Ball1" **"Ball2"** **"Fish"** **"Car"**
Map

The Collection API (part of it)

Notice that the Map interface doesn't actually extend the Collection interface, but Map is still considered part of the "Collection Framework" (also known as the "Collection API"). So Maps are still collections, even though they don't include java.util.Collection in their inheritance tree.

(Note: this is not the complete collection API; there are other classes and interfaces, but these are the ones we care most about.)

Collection
(interface)

Set
(Interface)

List
(interface)

SortedSet
(Interface)

| **TreeSet** | **LinkedHashSet** | **HashSet** | | **ArrayList** | **LinkedList** | **Vector** |

KEY

↑ extends

↑ implements

| HashSet | implementation class

| Set | interface

Map
(Interface)

SortedMap
(Interface)

Maps don't extend from java.util.Collection, but they're still considered to be part of the "collections framework" in Java. So a Map is still referred to as a collection.

| **TreeMap** | **HashMap** | **LinkedHashMap** | **Hashtable** |

Using a HashSet instead of ArrayList

We added on to the Jukebox to put the songs in a HashSet. (Note: we left out some of the Jukebox code, but you can copy it from earlier versions. And to make it easier to read the output, we went back to the earlier version of the Song's toString() method, so that it prints only the title instead of title *and* artist.)

```java
import java.util.*;
import java.io.*;

public class Jukebox6 {
    ArrayList<Song> songList = new ArrayList<Song>();
    // main method etc.

    public void go() {
        getSongs();

        System.out.println(songList);

        Collections.sort(songList);

        System.out.println(songList);

        HashSet<Song> songSet = new HashSet<Song>();

        songSet.addAll(songList);

        System.out.println(songSet);
    }

    // getSongs() and addSong() methods
}
```

We didn't change getSongs(), so it still puts the songs in an ArrayList

Here we create a new HashSet parameterized to hold Songs

HashSet has a simple addAll() method that can take another collection and use it to populate the HashSet. It's the same as if we added each song one at a time (except much simpler).

```
File Edit Window Help GetBetterMusic

%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles, Deep Channel,
Passenger, Listen, Listen, Listen, Circles]

[Circles, Circles, Deep Channel, Listen, Listen, Listen,
Passenger, Pink Moon, Shiva Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles, Listen, Deep Channel,
Passenger, Circles, Listen, Somersault]
```

Before sorting the ArrayList

After sorting the ArrayList (by title).

After putting it into a HashSet, and printing the HashSet (we didn't call sort() again).

The Set didn't help!!
We still have all the duplicates!

(And it lost its sort order when we put the list into a HashSet, but we'll worry about that one later...)

What makes two objects equal?

First, we have to ask—what makes two Song references duplicates? They must be considered *equal*. Is it simply two references to the very same object, or is it two separate objects that both have the same *title*?

This brings up a key issue: *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, one object on the heap.

Two references that refer to the same object on the heap are equal. Period. If you call the hashCode() method on both references, you'll get the same result. If you don't override the hashCode() method, the default behavior (remember, you inherited this from class Object) is that each object will get a unique number (most versions of Java assign a hashcode based on the object's memory address on the heap, so no two objects will have the same hashcode).

If you want to know if two *references* are really referring to the same object, use the == operator, which (remember) compares the bits in the variables. If both references point to the same object, the bits will be identical.
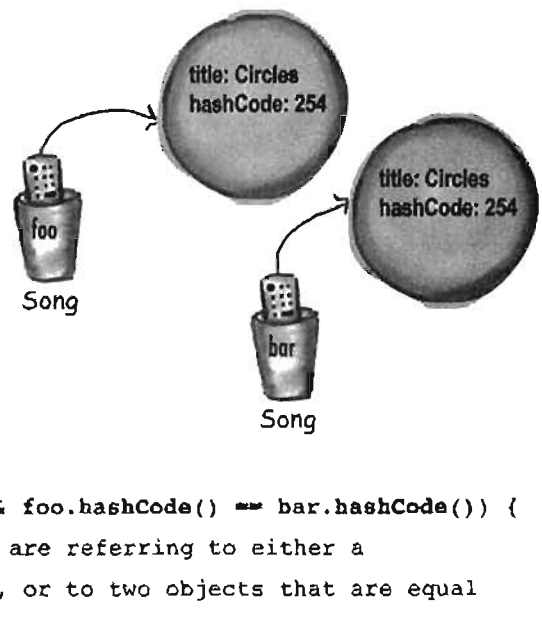


```
if (foo == bar) {
    // both references are referring
    // to the same object on the heap
}
```
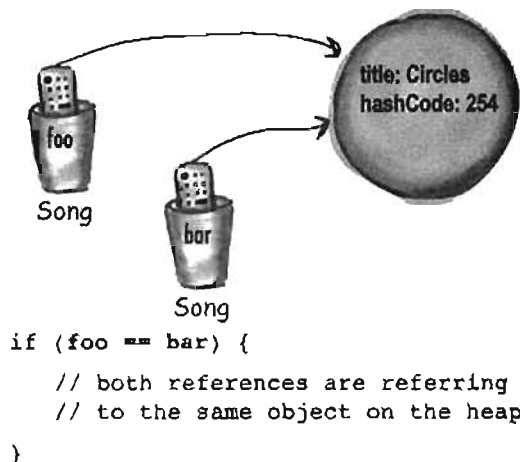
▶ Object equality
Two references, two objects on the heap, but the objects are considered *meaningfully equivalent*

If you want to treat two different Song objects as equal (for example if you decided that two Songs are the same if they have matching *title* variables), you must override *both* the hashCode() and equals() methods inherited from class Object.

As we said above, if you *don't* override hashCode(), the default behavior (from Object) is to give each object a unique hashcode value. So you must override hashCode() to be sure that two equivalent objects return the same hashcode. But you must also override equals() so that if you call it on *either* object, passing in the other object, always returns *true*.


```
if (foo.equals(bar) && foo.hashCode() == bar.hashCode()) {
    // both references are referring to either a
    // a single object, or to two objects that are equal
}
```

How a HashSet checks for duplicates: 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 it also compares the object's hashcode to the hashcode of all the other objects in the HashSet, and if there's no matching hashcode, the HashSet assumes that this new object is not a duplicate.

In other words, if the hashcodes are different, the HashSet assumes there's no way the objects can be equal!

So you must override hashCode() to make sure the objects have the same value.

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.
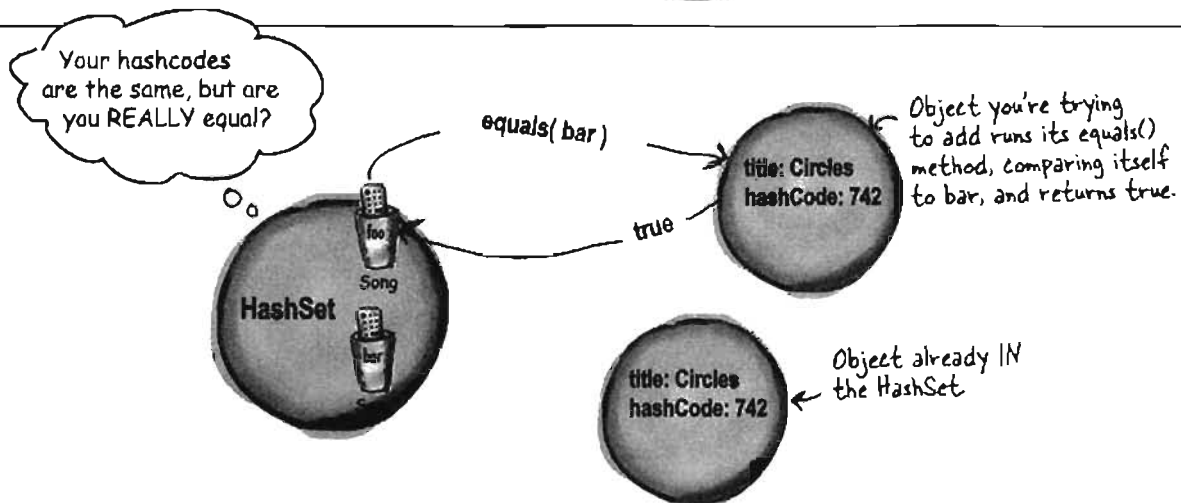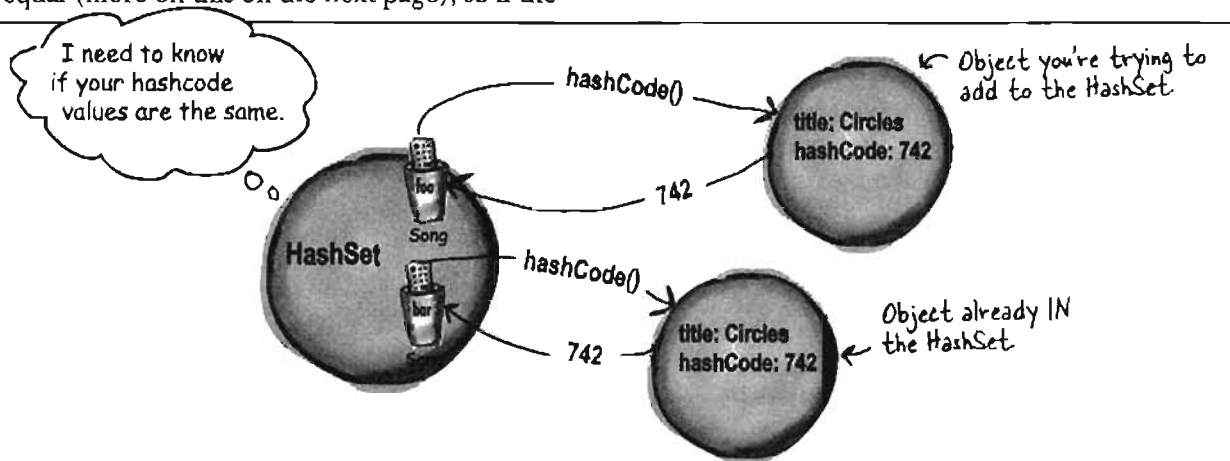
The Song class with overridden hashCode() and equals()

```
class Song implements Comparable<Song>{
    String title;
    String artist;
    String rating;
    String bpm;

    public boolean equals(Object aSong) {
        Song s = (Song) aSong;
        return getTitle().equals(s.getTitle());
    }

    public int hashCode() {
        return title.hashCode()
    }
    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }

    Song(String t, String a, String r, String b) {
        title = t;
        artist = a;
        rating = r;
        bpm = b;
    }

    public String getTitle() {
        return title;
    }

    public String getArtist() {
        return artist;
    }

    public String getRating() {
        return rating;
    }

    public String getBpm() {
        return bpm;
    }

    public String toString() {
        return title;
    }
}
```

The HashSet (or anyone else calling this method) sends it another Song.

The GREAT news is that title is a String, and Strings have an overridden equals() method. So all we have to do is ask one title if it's equal to the other song's title.

Same deal here... the String class has an overridden hashCode() method, so you can just return the result of calling hashCode() on the title. Notice how hashCode() and equals() are using the SAME instance variable.

Now it works! No duplicates when we print out the HashSet. But we didn't call sort() again, and when we put the ArrayList into the HashSet, the HashSet didn't preserve the sort order.

```
File Edit Window Help RebootWindows

%java Jukebox6

[Pink Moon, Somersault, Shiva Moon, Circles,
Deep Channel, Passenger, Listen, Listen,
Listen, Circles]

[Circles, Circles, Deep Channel, Listen,
Listen, Listen, Passenger, Pink Moon, Shiva
Moon, Somersault]

[Pink Moon, Listen, Shiva Moon, Circles,
Deep Channel, Passenger, Somersault]
```

Java Object Law For HashCode()
and equals()

The API docs for class Object state the rules you MUST follow:

- If two objects are equal, they MUST have matching hashcodes.

- If two objects are equal, calling equals() on either object MUST return true. In other words, if (a.equals(b)) then (b.equals(a)).

- If two objects have the same hashcode value, they are NOT required to be equal. But if they're equal, they MUST have the same hashcode value.

- So, if you override equals(), you MUST override hashCode().

- The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.

- The default behavior of equals() is to do an == comparison. In other words, to test whether the two references refer to a single object on the heap. So if you don't override equals() in a class, no two objects can EVER be considered equal since references to two different objects will always contain a different bit pattern.

a.equals(b) must also mean that a.hashCode() == b.hashCode()

But a.hashCode() == b.hashCode() does NOT have to mean a.equals(b)

there are no
Dumb Questions

Q: How come hashcodes can be the same even if objects aren't equal?

A: HashSets use hashcodes to store the elements in a way that makes it much faster to access. If you try to find an object in an ArrayList by giving the ArrayList a copy of the object (as opposed to an index value), the ArrayList has to start searching from the beginning, looking at each element in the list to see if it matches. But a HashSet can find an object much more quickly, because it uses the hashcode as a kind of label on the "bucket" where it stored the element. So if you say, "I want you to find an object in the set that's exactly like this one..." the HashSet gets the hashcode value from the copy of the Song you give it (say, 742), and then the HashSet says, "Oh, I know exactly where the object with hashcode #742 is stored...", and it goes right to the #742 bucket.

This isn't the whole story you get in a computer science class, but it's enough for you to use Hash-Sets effectively. In reality, developing a good hash-code algorithm is the subject of many a PhD thesis, and more than we want to cover in this book.

The point is that 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. And yes, that means that multiple objects would all land in the same bucket in the HashSet (because each bucket represents a single hashcode value), but that's not the end of the world. It might mean that the Hash-Set is just a little less efficient (or that it's filled with an extremely large number of elements), but if the HashSet finds more than one object in the same hashcode bucket, the HashSet will simply use the equals() method to see if there's a perfect match. In other words, hashcode values are some-times 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.

And if we want the set to stay sorted, we've got 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.

```java
import java.util.*;
import java.io.*;
public class Jukebox8 {
    ArrayList<Song> songList = new ArrayList<Song>();
    int val;

    public static void main(String[] args) {
        new Jukebox8().go();
    }

    public void go() {
        getSongs();
        System.out.println(songList);
        Collections.sort(songList);
        System.out.println(songList);
        TreeSet<Song> songSet = new TreeSet<Song>();
        songSet.addAll(songList);
        System.out.println(songSet);
    }

    void getSongs() {
        try {
            File file = new File("SongListMore.txt");
            BufferedReader reader = new BufferedReader(new FileReader(file));
            String line = null;
            while ((line= reader.readLine()) != null) {
                addSong(line);
            }

        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    void addSong(String lineToParse) {
        String[] tokens = lineToParse.split("/");
        Song nextSong = new Song(tokens[0], tokens[1], tokens[2], tokens[3]);
        songList.add(nextSong);
    }
}
```

Instantiate a TreeSet instead of HashSet. Calling the no-arg TreeSet constructor means the set will use the Song object's compareTo() method for the sort.

(We could have passed in a Comparator.)

We can add all the songs from the HashSet using addAll(). (Or we could have added the songs individually using songSet.add() just the way we added songs to the ArrayList.)

What you MUST know about TreeSet...

TreeSet looks easy, but make sure you really understand what you need to do to use it. We thought it was so important that we made it an exercise so you'd *have* to think about it. Do NOT turn the page until you've done this. *We mean it.*

Sharpen your pencil

Look at this code. Read it carefully, then answer the questions below. (Note: there are no syntax errors in this code.)

```
import java.util.*;

public class TestTree {
    public static void main (String[] args) {
        new TestTree().go();
    }

    public void go() {
        Book b1 = new Book("How Cats Work");
        Book b2 = new Book("Remix your Body");
        Book b3 = new Book("Finding Emo");

        TreeSet<Book> tree = new TreeSet<Book>();
        tree.add(b1);
        tree.add(b2);
        tree.add(b3);
        System.out.println(tree);
    }
}

class Book {
    String title;
    public Book(String t) {
        title = t;
    }
}
```

1). What is the result when you compile this code?

_____

2). If it compiles, what is the result when you run the TestTree class?

_____

3). If there is a problem (either compile-time or runtime) with this code, how would you fix it?

_____

TreeSet elements MUST be comparable

TreeSet can't read the programmer's mind to figure out how the object's should be sorted. You have to tell the TreeSet *how*.

To use a TreeSet, one of these things must be true:

▶ **The elements in the list must be of a type that implements *Comparable***

The Book class on the previous page didn't implement Comparable, so it wouldn't work at runtime. Think about it, the poor TreeSet's sole purpose in life is to keep your elements sorted, and once again—it had no idea how to sort Book objects! It doesn't fail at compile-time, because the TreeSet add() method doesn't take a Comparable type,The TreeSet add() method takes whatever type you used when you created the TreeSet. In other words, if you say new TreeSet<Book>() the add() method is essentially add(Book). And there's no requirement that the Book class implement Comparable! But it fails at runtime when you add the second element to the set. That's the first time the set tries to call one of the object's compareTo() methods and... can't.

```
class Book implements Comparable {
    String title;
    public Book(String t) {
        title = t;
    }
    public int compareTo(Object b) {
        Book book = (Book) b;
        return (title.compareTo(book.title));
    }
}
```

OR

▶ **You use the TreeSet's overloaded constructor that takes a *Comparator***

TreeSet works a lot like the sort() method—you have a choice of using the element's compareTo() method, assuming the element type implemented the Comparable interface, OR you can use a custom Comparator that knows how to sort the elements in the set. To use a custom Comparator, you call the TreeSet constructor that takes a Comparator.

```
public class BookCompare implements Comparator<Book> {
    public int compare(Book one, Book two) {
        return (one.title.compareTo(two.title));
    }
}

class Test {
    public void go() {
        Book b1 = new Book("How Cats Work");
        Book b2 = new Book("Remix your Body");
        Book b3 = new Book("Finding Emo");
        BookCompare bCompare = new BookCompare();
        TreeSet<Book> tree = new TreeSet<Book>(bCompare);
        tree.add(new Book("How Cats Work"));
        tree.add(new Book("Finding Emo"));
        tree.add(new Book("Remix your Body"));
        System.out.println(tree);
    }
}
```

We've seen Lists and Sets, now we'll use a Map

Lists and Sets are great, but sometimes a Map is the best collection (not Collection with a capital "C"—remember that Maps are part of Java collections but they don't implement the Collection interface).

Imagine you want a collection that acts like a property list, where you give it a name and it gives you back the value associated with that name. Although keys will often be Strings, they can be any Java object (or, through autoboxing, a primitive).

value ⟶

key ⟶ | "Ball1" | "Ball2" | "Ball3" | "Ball4" |

Map
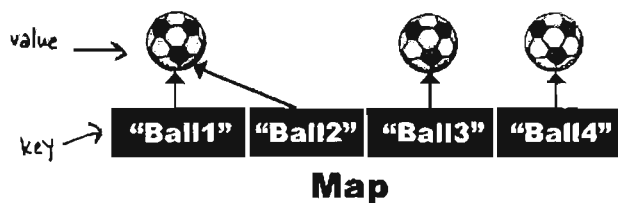
Each element in a Map is actually TWO objects—a _key_ and a _value_. You can have duplicate _values_, but NOT duplicate _keys_.

Map example

```java
import java.util.*;

public class TestMap {

    public static void main(String[] args) {

        HashMap<String, Integer> scores = new HashMap<String, Integer>();

        scores.put("Kathy", 42);
        scores.put("Bert", 343);
        scores.put("Skyler", 420);

        System.out.println(scores);
        System.out.println(scores.get("Bert"));
    }
}
```

HashMap needs TWO type parameters— one for the key and one for the value.

Use put() instead of add(), and now of course it takes two arguments (key, value).

The get() method takes a key, and returns the value (in this case, an Integer).

File Edit Window Help WhoAmI

```
%java TestMap

{Skyler=420, Bert=343, Kathy=42}
343
```

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.

Finally, back to generics

Remember earlier in the chapter we talked about how methods that take arguments with generic types can be... *weird*. And we mean weird in the polymorphic sense. If things start to feel strange here, just keep going—it takes a few pages to really tell the whole story.

We'll start with a reminder on how *array* arguments work, polymorphically, and then look at doing the same thing with generic lists. The code below compiles and runs without errors:

Here's how it works with regular *arrays*:

> **If a method argument is an *array* of Animals, it will also take an array of any Animal subtype.**
>
> **In other words, if a method is declared as:**
>
> **void foo(Animal[] a) { }**
>
> **Assuming Dog extends Animal, you are free to call both:**
>
> **foo(anAnimalArray);**
> **foo(aDogArray);**

```java
import java.util.*;

public class TestGenerics1 {
    public static void main(String[] args) {
        new TestGenerics1().go();
    }

    public void go() {
        Animal[] animals = {new Dog(), new Cat(), new Dog()};
        Dog[] dogs = {new Dog(), new Dog(), new Dog()};
        takeAnimals(animals);
        takeAnimals(dogs);
    }

    public void takeAnimals(Animal[] animals) {
        for(Animal a: animals) {
            a.eat();
        }
    }
}
```

Declare and create an Animal array, that holds both dogs and cats.

Declare and create a Dog array, that holds only Dogs (the compiler won't let you put a Cat in).

Call takeAnimals(), using both array types as arguments...

The crucial point is that the takeAnimals() method can take an Animal[] or a Dog[], since Dog IS-A Animal. Polymorphism in action.

Remember, we can call ONLY the methods declared in type animal, since the animals parameter is of type Animal array, and we didn't do any casting. (What would we cast it to? That array might hold both Dogs and Cats.)

```java
abstract class Animal {
    void eat() {
        System.out.println("animal eating");
    }
}
class Dog extends Animal {
    void bark() { }
}
class Cat extends Animal {
    void meow() { }
}
```

The simplified Animal class hierarchy.

Using polymorphic arguments and generics

So we saw how the whole thing worked with arrays, but will it work the same way when we switch from an array to an ArrayList? Sounds reasonable, doesn't it?

First, let's try it with only the Animal ArrayList. We made just a few changes to the go() method:

Passing in just ArrayList<Animal>

A simple change from Animal[] to ArrayList<Animal>.
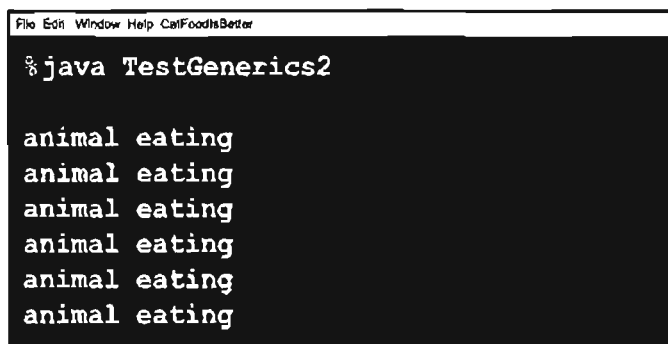
```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat());     ←─── We have to add one at a time since there's no
    animals.add(new Dog());          shortcut syntax like there is for array creation.

    takeAnimals(animals);  ←── This is the same code, except now the "animals"
}                              variable refers to an ArrayList instead of array.
```

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
    a.eat();
    }
}
```

The method now takes an ArrayList instead of an array, but everything else is the same. Remember, that for loop syntax works for both arrays and collections.

Compiles and runs just fine

```
File  Edit  Window  Help  CatFoodIsBetter
%java TestGenerics2

animal eating
animal eating
animal eating
animal eating
animal eating
animal eating
```

But will it work with ArrayList<Dog> ?

Because of polymorphism, the compiler let us pass a Dog array
to a method with an Animal array argument. No problem. And
an ArrayList<Animal> can be passed to a method with an
ArrayList<Animal> argument. So the big question is, will the
ArrayList<Animal> argument accept an ArrayList<Dog>? If it works
with arrays, shouldn't it work here too?

Passing in just ArrayList<Dog>

```
public void go() {
    ArrayList<Animal> animals = new ArrayList<Animal>();
    animals.add(new Dog());
    animals.add(new Cat());
    animals.add(new Dog());
    takeAnimals(animals);   ←— We know this line worked fine.

    ArrayList<Dog> dogs = new ArrayList<Dog>();
    dogs.add(new Dog());
    dogs.add(new Dog());            Make a Dog ArrayList and put a couple dogs in.
    takeAnimals(dogs);   ←— Will this work now that we changed
}                                    from an array to an ArrayList?


public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

When we compile it:

```
%java TestGenerics3

TestGenerics3.java:21: takeAnimals(java.util.
ArrayList<Animal>) in TestGenerics3 cannot be applied to
(java.util.ArrayList<Dog>)
    takeAnimals(dogs);
    ^
1 error
```

It looked so right,
but went so wrong...

> And I'm supposed to be OK with this? That totally screws my animal simulation where the veterinary program takes a list of any type of animal, so that a dog kennel can send a list of dogs, and a cat kennel can send a list of cats... now you're saying I can't do that if I use collections instead of arrays?

What could happen if it *were* allowed...

Imagine the compiler let you get away with that. It let you pass an ArrayList<Dog> to a method declared as:

```
public void takeAnimals(ArrayList<Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

There's nothing in that method that *looks* harmful, right? After all, the whole point of polymorphism is that anything an Animal can do (in this case, the eat() method), a Dog can do as well. So what's the problem with having the method call eat() on each of the Dog references?

Nothing. Nothing at all.

There's nothing wrong with *that* code. But imagine *this* code instead:

```
public void takeAnimals(ArrayList<Animal> animals) {
    animals.add(new Cat()); ←── Yikes!! We just stuck a Cat in what
}                                might be a Dogs-only ArrayList
```

So that's the problem. There's certainly nothing wrong with adding a Cat to an ArrayList<Animal>, and that's the whole point of having an ArrayList of a supertype like Animal—so that you can put all types of animals in a single Animal ArrayList.

But if you passed a Dog ArrayList—one meant to hold ONLY Dogs— to this method that takes an Animal ArrayList, then suddenly you'd end up with a Cat in the Dog list. The compiler knows that if it lets you pass a Dog ArrayList into the method like that, someone could, at runtime, add a Cat to your Dog list. So instead, the compiler just won't let you take the risk.

If you declare a method to take ArrayList<Animal> it can take ONLY an ArrayList<Animal>, not ArrayList<Dog> or ArrayList<Cat>.

Wait a minute... if this is why they won't let you pass a Dog ArrayList into a method that takes an Animal ArrayList—to stop you from possibly putting a Cat in what was actually a Dog list, then why does it work with *arrays*? Don't you have the same problem with arrays? Can't you still add a Cat object to a Dog[] ?

Array types are checked again at <u>runtime</u>, but collection type checks happen only when you <u>compile</u>

Let's say you *do* add a Cat to an array declared as Dog[] (an array that was passed into a method argument declared as Animal[], which is a perfectly legal assignment for arrays).

```
public void go() {
    Dog[] dogs = {new Dog(), new Dog(), new Dog()};
    takeAnimals(dogs);
}

public void takeAnimals(Animal[] animals) {
    animals[0] = new Cat();
}
```

We put a new Cat into a Dog array. The compiler allowed it, because it knows that you might have passed a Cat array or Animal array to the method, so to the compiler it was possible that this was OK.

It compiles, but when we run it:

```
File Edit Window Help CatsAreSmarter
%java TestGenerics1
Exception in thread "main" java.lang.ArrayStoreException:
Cat
        at TestGenerics1.takeAnimals(TestGenerics1.java:16)
        at TestGenerics1.go(TestGenerics1.java:12)
        at TestGenerics1.main(TestGenerics1.java:5)
```

Whew! At least the JVM stopped it.

Wouldn't it be dreamy if there were a way to still use polymorphic collection types as method arguments, so that my veterinary program could take Dog lists and Cat lists? That way I could loop through the lists and call their immunize() method, but it would still have to be safe so that you couldn't add a Cat in to the Dog list. But I guess that's just a fantasy...

Wildcards to the rescue

It looks unusual, but 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.

```
public void takeAnimals(ArrayList<? extends Animal> animals) {
    for(Animal a: animals) {
        a.eat();
    }
}
```

Remember, the keyword "extends" here means either <u>extends</u> OR <u>implements</u> depending on the type. So if you want to take an ArrayList of types that implement the Pet interface, you'd declare it as:

ArrayList<? extends Pet>

So now you're wondering, "What's the *difference?* Don't you have the same problem as before? The method above isn't doing anything dangerous—calling a method any Animal subtype is guaranteed to have—but can't someone still change this to add a Cat to the *animals* list, even though it's really an ArrayList<Dog>? And since it's not checked again at runtime, how is this any different from declaring it without the wildcard?"

And you'd be right for wondering. The answer is NO. When you use the wildcard <?> in your declaration, the compiler won't let you do anything that adds to the list!

> **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.**
>
> **In other words, you can do things *with* the list elements, but you can't put *new* things in the list. So you're safe at runtime, because the compiler won't let you do anything that might be horrible at runtime.**
>
> **So, this is OK inside takeAnimals():**
>
> ```
> for(Animal a : animals) {
>     a.eat();
> }
> ```
>
> **But THIS would not compile:**
>
> ```
> animals.add(new Cat());
> ```

Alternate syntax for doing the same thing

You probably remember that when we looked at the sort() method,
it used a generic type, but with an unusual format where the type
parameter was declared before the return type. It's just a different way
of declaring the type parameter, but the results are the same:

This:

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

Does the same thing as this:

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

there are no Dumb Questions

Q: If they both do the same thing, why would you use
one over the other?

A: It all depends on whether you want to use "T" some-
where else. For example, what if you want the method to
have two arguments—both of which are lists of a type that
extend Animal? In that case, it's more efficient to just declare
the type parameter once:

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

Instead of typing:

```
public void takeThing(ArrayList<? extends Animal> one,
                                  ArrayList<? extends Animal> two)
```

BE the compiler, advanced

Your job is to play compiler and determine which of these statements would compile. But some of this code wasn't covered in the chapter, so you need to work out the answers based on what you DID learn, applying the "rules" to these new situations. In some cases, you might have to guess, but the point is to come up with a reasonable answer based on what you know so far.

(Note: assume that this code is within a legal class and method.)

Compiles?

☐ `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`

☐ `ArrayList<Animal> animals1 = new ArrayList<Dog>();`

☐ `List<Animal> list = new ArrayList<Animal>();`

☐ `ArrayList<Dog> dogs = new ArrayList<Dog>();`

☐ `ArrayList<Animal> animals = dogs;`

☐ `List<Dog> dogList = dogs;`

☐ `ArrayList<Object> objects = new ArrayList<Object>();`

☐ `List<Object> objList = objects;`

☐ `ArrayList<Object> objs = new ArrayList<Dog>();`

```
import java.util.*;

public class SortMountains {

    LinkedList<Mountain> mtn = new LinkedList<Mountain>();

    class NameCompare implements Comparator <Mountain> {
        public int compare(Mountain one, Mountain two) {

            return one.name.compareTo(two.name);

        }
    }
    class HeightCompare implements Comparator <Mountain> {
        public int compare(Mountain one, Mountain two) {

            return (two.height - one.height);

        }
    }
    public static void main(String [] args) {
        new SortMountain().go();
    }
    public void go() {
        mtn.add(new Mountain("Longs", 14255));
        mtn.add(new Mountain("Elbert", 14433));
        mtn.add(new Mountain("Maroon", 14156));
        mtn.add(new Mountain("Castle", 14265));

        System.out.println("as entered:\n" + mtn);
        NameCompare nc = new NameCompare();

        Collections.sort(mtn, nc);
        System.out.println("by name:\n" + mtn);
        HeightCompare hc = new HeightCompare();

        Collections.sort(mtn, hc);
        System.out.println("by height:\n" + mtn);
    }
}

class Mountain {
    String name;
    int height;

    Mountain(String n, int h) {
        name = n;
        height = h;
    }
    public String toString( ) {
        return name + " " + height;
    }
}
```

Did you notice that the height list is in DESCENDING sequence? :)

Output:

```
File  Edit  Window  Help  ThisOne'sForBob
%java SortMountains
as entered:
[Longs 14255, Elbert 14433, Maroon 14156, Castle 14265]
by name:
[Castle 14265, Elbert 14433, Longs 14255, Maroon 14156]
by height:
[Elbert 14433, Castle 14265, Longs 14255, Maroon 14156]
```

fill-in-the-blank solution

Exercise Solution

Possible Answers:
Comparator,

Comparable,

compareTo( ),

compare( ),

yes,

no

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

1. What must the class of the objects stored in myArrayList implement?   **Comparable**

2. What method must the class of the objects stored in myArrayList implement? **compareTo( )**

3. Can the class of the objects stored in myArrayList implement both
   Comparator AND Comparable?   **yes**

Given the following compilable statement:

```
Collections.sort(myArrayList, myCompare);
```

4. Can the class of the objects stored in myArrayList implement Comparable? **yes**

5. Can the class of the objects stored in myArrayList implement Comparator? **yes**

6. Must the class of the objects stored in myArrayList implement Comparable? **no**

7. Must the class of the objects stored in myArrayList implement Comparator? **no**

8. What must the class of the myCompare object implement?   **Comparator**

9. What method must the class of the myCompare object implement?   **compare( )**

BE the compiler solution

Compiles?

☐ `ArrayList<Dog> dogs1 = new ArrayList<Animal>();`

☐ `ArrayList<Animal> animals1 = new ArrayList<Dog>();`

☒ `List<Animal> list = new ArrayList<Animal>();`

☐ `ArrayList<Dog> dogs = new ArrayList<Dog>();`

☐ `ArrayList<Animal> animals = dogs;`

☒ `List<Dog> dogList = dogs;`

☒ `ArrayList<Object> objects = new ArrayList<Object>();`

☒ `List<Object> objList = objects;`

☐ `ArrayList<Object> objs = new ArrayList<Dog>();`