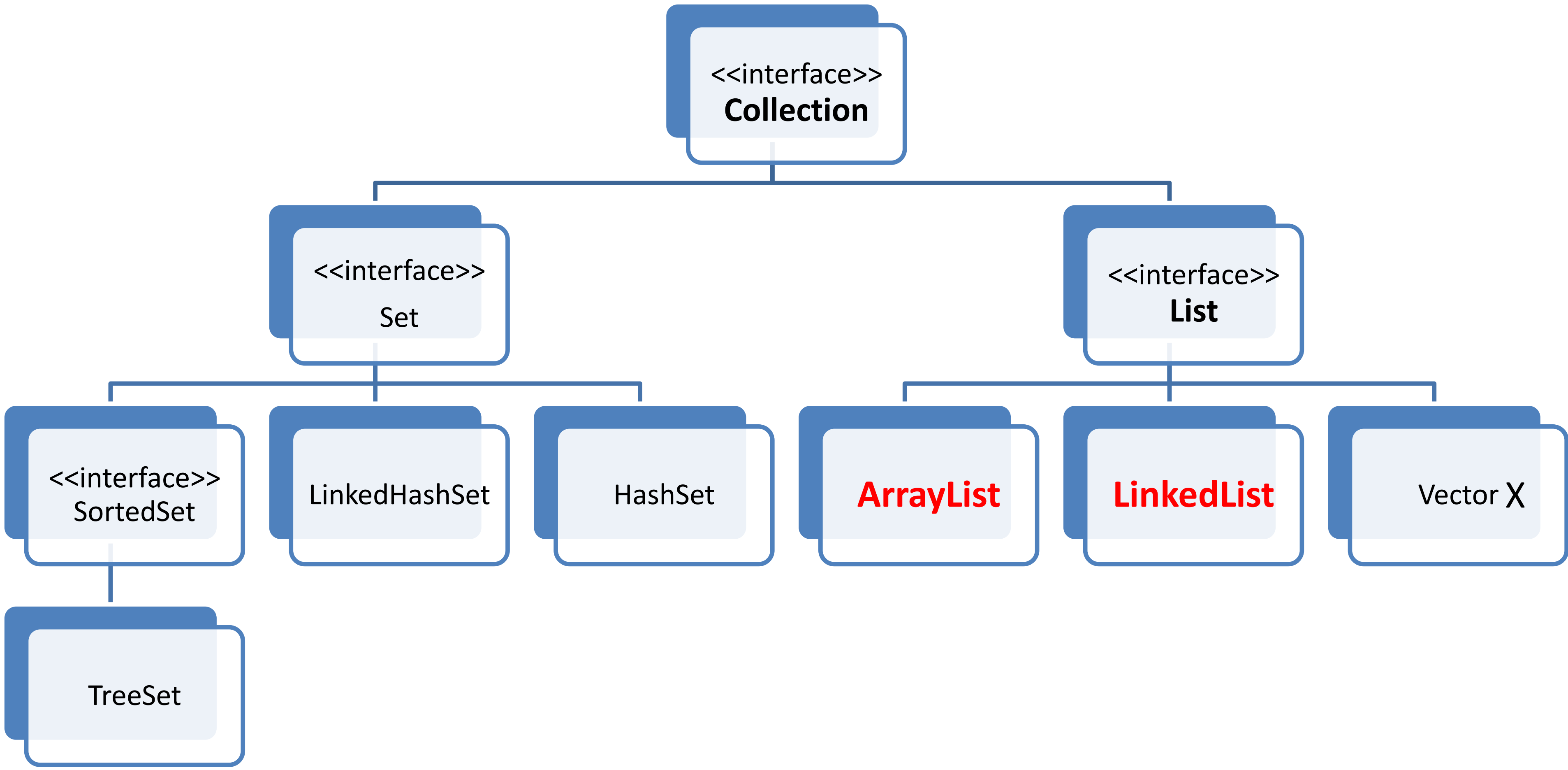
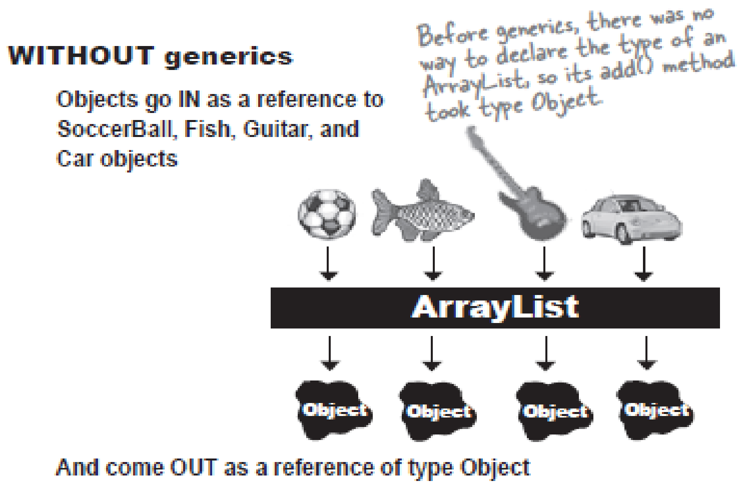
Java collections framework. Generics

Lecture 14

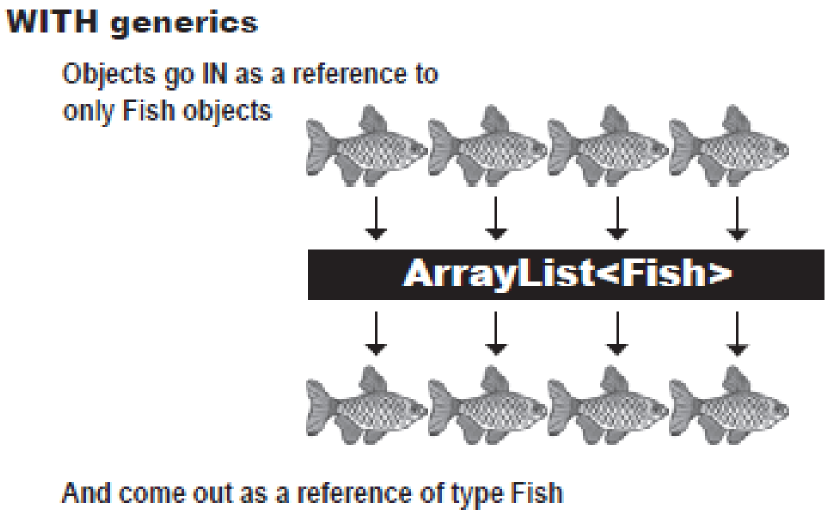
ArrayList is not the only ***Collection***



Java collections: without generics


WITHOUT generics
Objects go IN as a reference to SoccerBall, Fish, Guitar, and Car objects
Before generics, there was no way to declare the type of an ArrayList, so its add() method took type Object.
ArrayList
Object
Object
Object
Object
And come OUT as a reference of type Object

Java collections: with generics


WITH generics
Objects go IN as a reference to only Fish objects
ArrayList<Fish>
And come out as a reference of type Fish

Populating collection with elements

- In the constructor

Collection **myCollection** = new **Collection** (**anotherCollection**)

- Using static method of Collections class

Collections.**addAll** (**myCollection**, **anotherCollection**)

Examples: Arrays.AsList

```
public class AddingGroups {
        public static void main(String[] args) {
                Collection<Integer> collection = new
                        ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
                Integer[] moreInts = { 6, 7, 8, 9, 10 };
                collection.addAll (Arrays.asList(moreInts));
        }
}
```

Arrays.asList cannot be resized

```
public class AddingGroups {
public static void main(String[] args) {
        // Produces a list "backed by" an array:
        List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
        list.set(1, 99);                    // OK -- modify an element
        // list.add(5);                     // Runtime error because the
                                     // underlying array cannot be resized.
         }
}
```

Java generics syntax

To define an **ArrayList** intended to hold **Apple** objects, you say **ArrayList <Apple**> instead of just **ArrayList**.

- Virtually all code which uses generics is a collection-related code
- With generics you create type-safe containers where problems are caught at compile-time instead of runtime

Where to find generics declarations

- Class declaration

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

- Method declaration: return type and argument types

public boolean add(E o){}

- Now if you define

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

- E is replaced with String everywhere in the code of ArrayList. That is if the code for ArrayList would become:

public class ArrayListOfStrings {

public boolean add(String o){}

- The gain is that the general code of ArrayList class adjusts itself to a Type.

Declaring your own generic methods

- Declare in a class and use in a method

public class ClassName<E> extends …{

public boolean Add(E o)

- Declare only in a method

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

This method takes an array list of anything which is an Animal

- Not the same as:

public void takeThing(ArrayList<Animal> list)

This method takes only array list of Animal objects

Sorting songs, which do not implement Comparable – no generics

```
System.out.println("Sorted songs:");
Collections.sort(songs);
System.out.println(songs);
```

Run-time exception

```
Exception in thread "main" java.lang.ClassCastException: sorting.Song cannot be cast to
java.lang.Comparable
at java.util.ComparableTimSort.countRunAndMakeAscending(Unknown Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.Arrays.sort(Unknown Source)
at java.util.Collections.sort(Unknown Source)
at sorting.SortingSongs.main(SortingSongs.java:16)
```

Sorting <Song> ArrayList with Song objects which do not implement Comparable

This time compiler will detect that Song is not Comparable

Compile-time exception

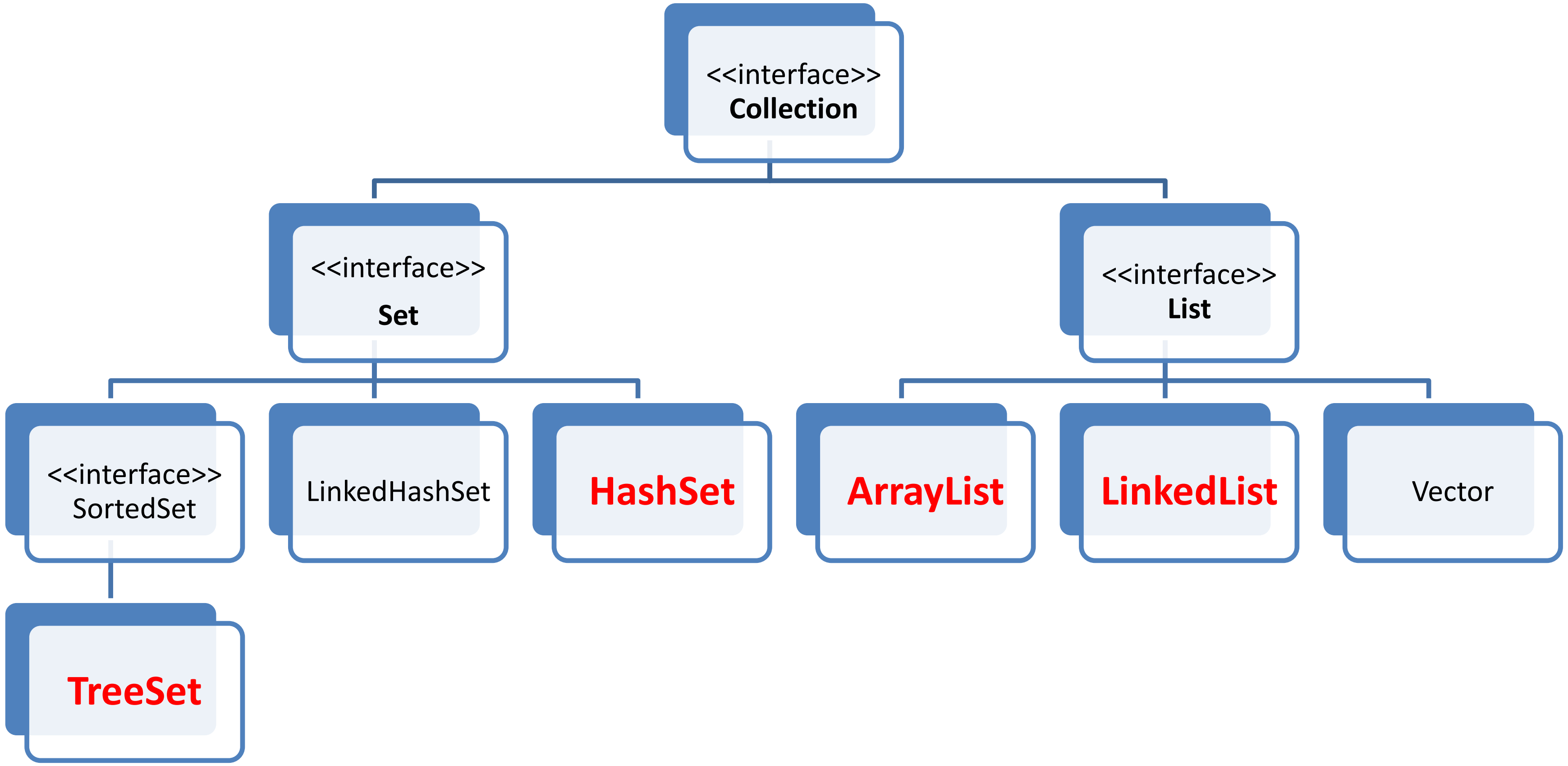
```
%javac SortingSongs.java
SortingSongs.java:15: cannot find symbol
symbol : method sort(java.util.ArrayList<Song>)
location: class java.util.Collections
Collections.sort(songList);
^
1 error
```

Java collections: Two primary categories

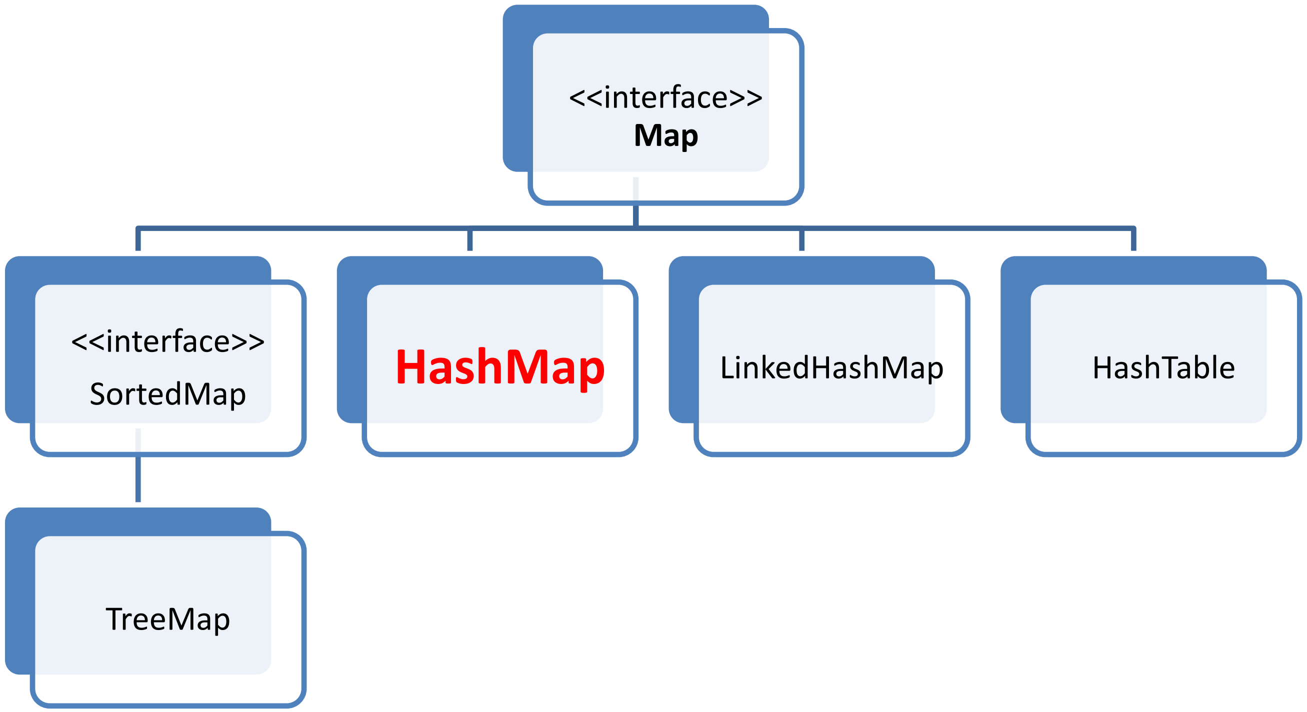
The distinction is based on the number of items that are held in each "slot" in the container.

- The **Collection** only holds one item in each slot.
- The **Map** holds two objects, a *key* and an associated *value*, in each slot.

The most useful containers implementing ***Collection*** interface



The most useful containers implementing Map interface



Implementations of Collection Interface

- **Lists: ArrayList** and **LinkedList.** Hold elements in the same order in which they are inserted. The difference is the underlying implementation.
- **Sets:**
  - **HashSet** holds one of each identical item. The storage order is not important (often, you only care whether something is a member of the **Set**, not the order in which it appears) .
  - **TreeSet** keeps the objects in ascending comparison order, **LinkedHashSet** keeps the objects in the order in which they were added.

Program to an interface

List<Apple> apples = new ArrayList<Apple>();

- You make an object of a concrete class, upcast it to the corresponding interface, and then use the interface throughout the rest of your code.

Example

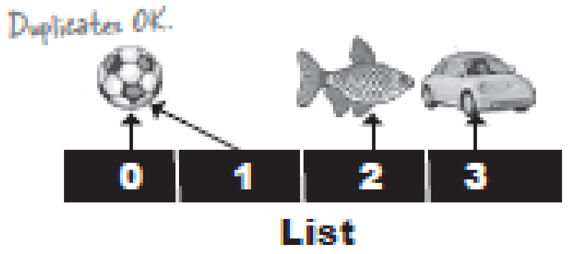
```
public class SimpleCollection {
        public static void main(String[] args) {
                Collection <Integer> c = new ArrayList<Integer>();
                for(int i = 0; i < 10; i++)
                        c.add(i); // Autoboxing: adding Integer
                for(Integer i : c)
                        System.out.print(i + ", ");
        }
} /* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

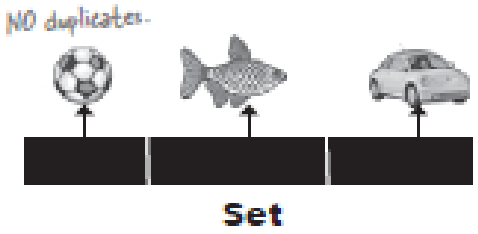
- Since this example only uses **Collection** methods, any object of a class inherited from **Collection** would work
- All **Collections** can be traversed using the foreach syntax

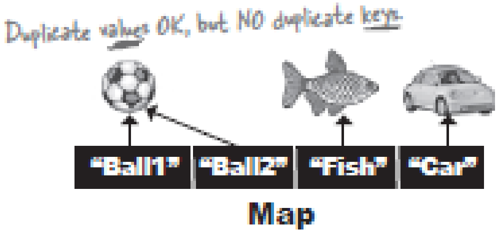
List

- Resizes itself: a modifiable sequence

New problem with songs: duplicates







Using HashSet to remove duplicates

```
List <Song> songs=reader.songList;
System.out.println(songs);

Set <Song> songHashSet = new HashSet<Song>();
songHashSet.addAll(songs);
System.out.println("Songs with NO duplicates:");
System.out.println(songHashSet);
```

Did not remove duplicates!

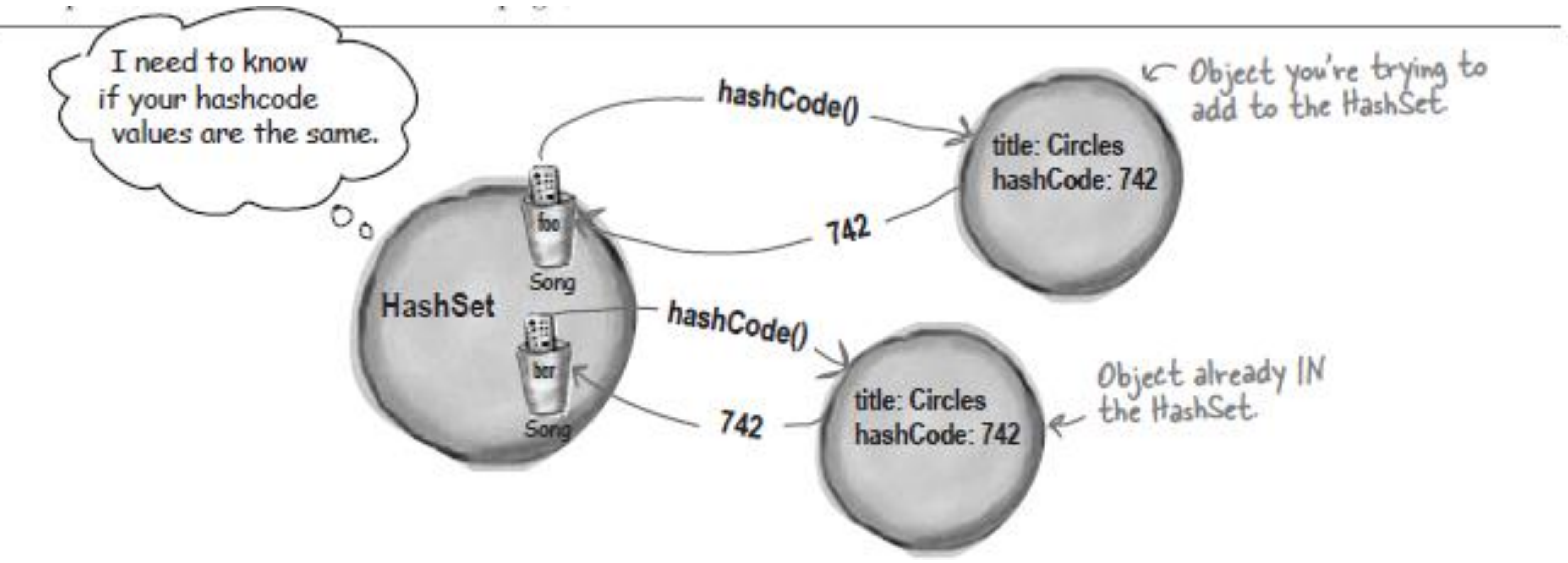
What makes two songs duplicates?

- Reference equality?
- Object equality?

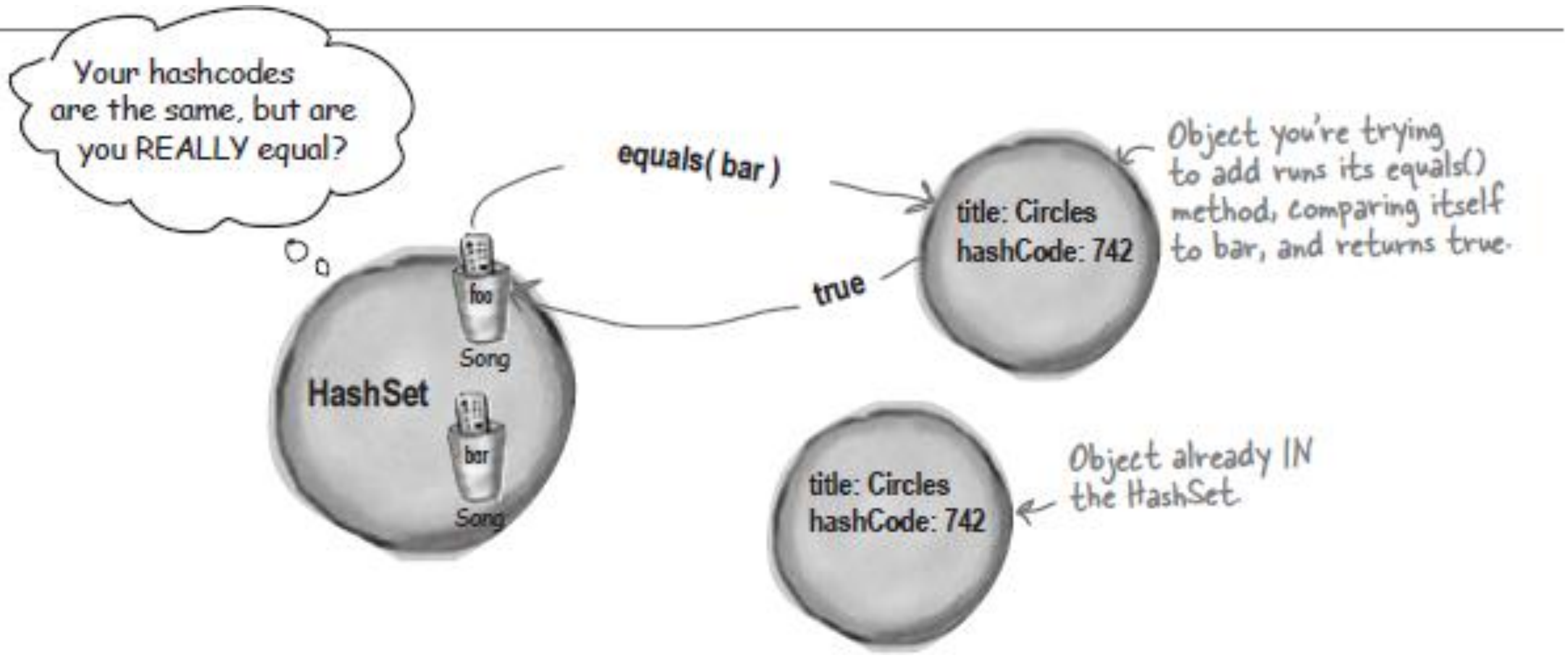
Hashing according to HashCode

- Set is *a hash table*
- It puts elements into some slot of an underlying array, according to the value of hash code
- If two non-equal elements have the same hash code, they are chained starting from a corresponding slot

How HashSet checks for duplicates: hashCode()


I need to know if your hashcode values are the same.
hashCode()
Object you're trying to add to the HashSet
title: Circles
hashCode: 742
742
foo
Song
HashSet
hashCode()
bar
Song
742
title: Circles
hashCode: 742
Object already IN the HashSet.

How HashSet checks for duplicates: equals()


Your hashcodes are the same, but are you REALLY equal?
equals(bar)
true
foo
Song
HashSet
bar
Song
title: Circles
hashCode: 742
Object you're trying to add runs its equals() method, comparing itself to bar, and returns true.
title: Circles
hashCode: 742
Object already IN the HashSet

Default hashCode() and equals(): inherited from Object class

- hashCode: unique Integer for each object on the heap
- equals: compares using ==, compares if two variables reference the same object

The Song class with overridden HashCode and equals

```
//new overridden methods of Object
          public boolean equals(Object aSong)
          {
                    Song s = (Song) aSong;
                    return this.getName().equals(s.getName());
          }

          public int hashCode()
          {
                    return this.getName().hashCode();
          }
```

Laws of Java Object

- The API docs for Object class state that:
  - If two objects are equal they MUST have matching hash codes
  - If 2 objects are equal the equality is reflective

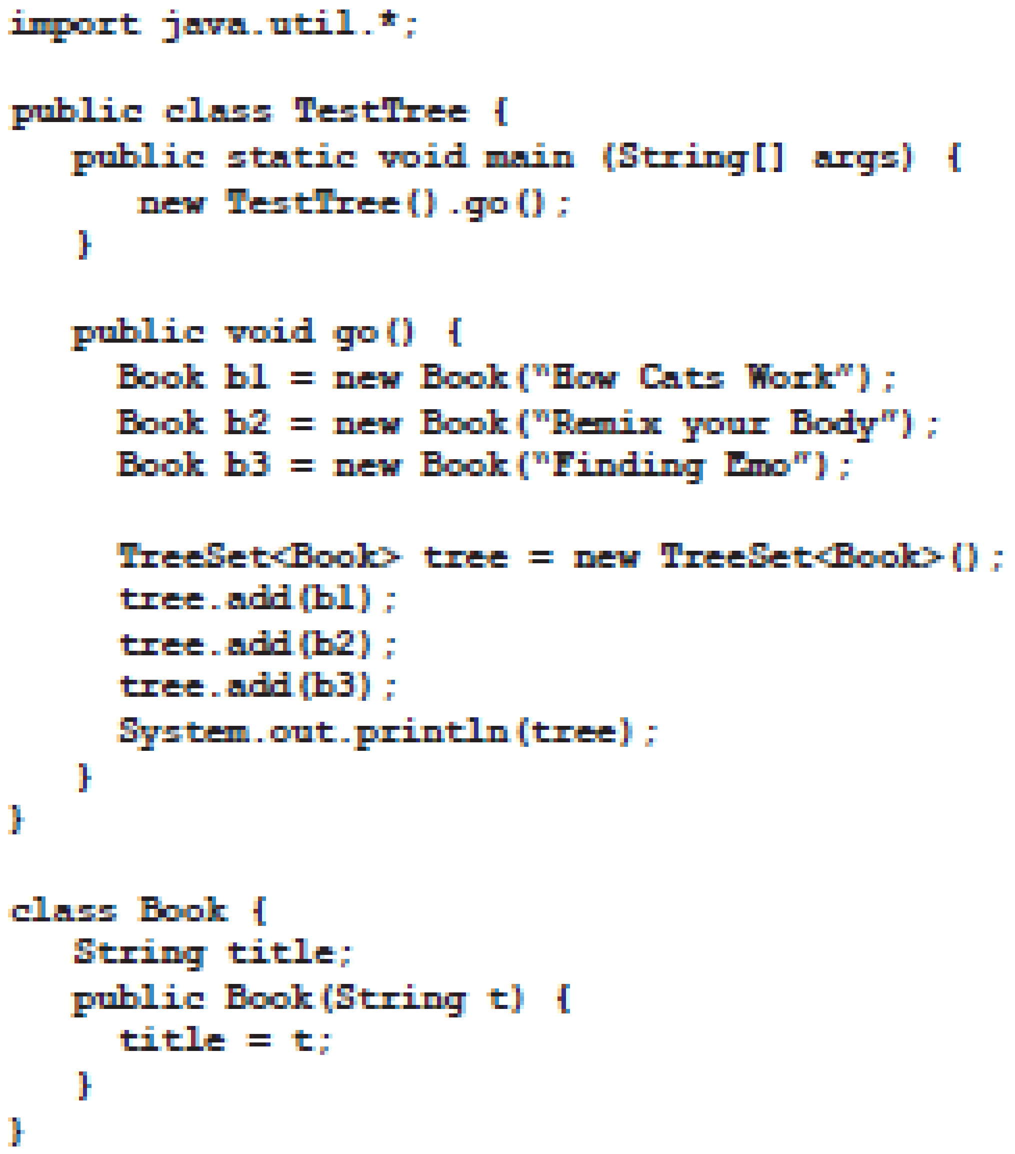
    a.equals(b) implies that ***b.equals(a)***
  - If two objects have the same hashcode they are NOT required to be equal, but if they are equal they MUST to have matching hash codes

Sorted without duplicates

- TreeSet prevents duplicates and also keeps elements sorted

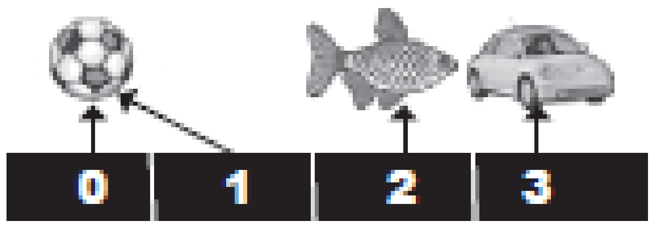
It works like the sort() method of Collections, in that it keeps Comparable objects in natural order, if you use the default TreeSet constructor. It also has a constructor to pass a specific Comparator.

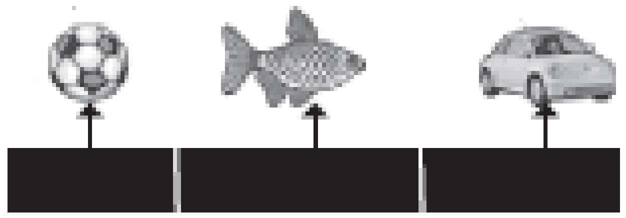
To use TreeSet you MUST…

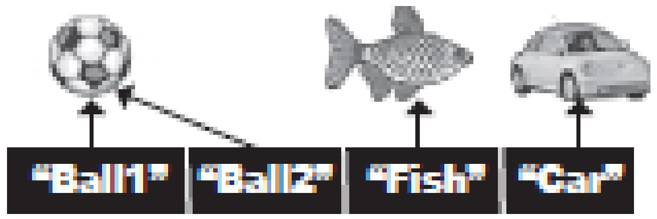


```
import java.util.*;

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;
   }
}
```

Which container to use






"Ball1"
"Ball2"
"Fish"
"Car"