Section 10

Collections & Generics

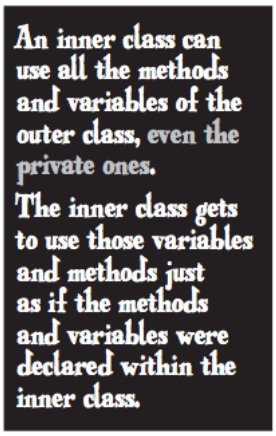
Learning Outcomes

- After this lecture you should be able to:
  - Create and use Inner classes
  - Describe and use a number of data structures available through the API
  - Use the Collections API
  - Differentiate between generic classes & generic methods
  - Implement the Comparable interface
  - Implement the Comparator interface
  - Discuss object equality
  - Override equals() & hashCode() methods
  - ***Reading*** and ***studying*** recommended text is essential to improve your understanding of the above

Inner Classes

- Classes can be nested inside one another

```
class OuterClass {
    class InnerClass {
        void doSomething(){
        }
    } // end inner class
} // end outer class
```

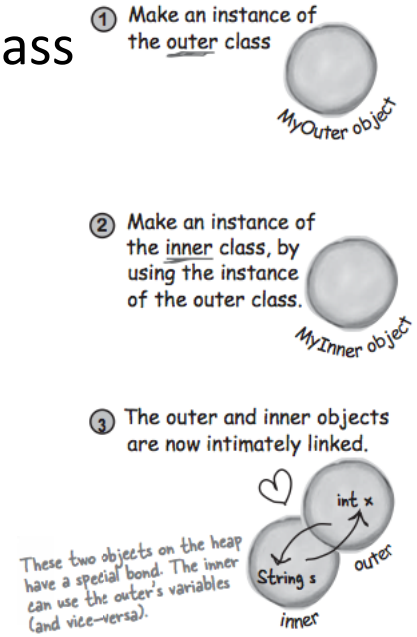


Inner Class & Outer Class Variable

```
class OuterClass {
    private int x;
    class InnerClass {
        void doSomething(){
            x = 12;
        }
    } // end inner class
} // end outer class
```

Inner Class

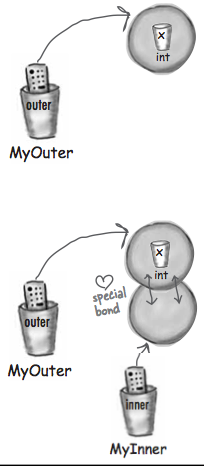
- An inner class instance must be tied to an outer class instance
- *Instance* of inner class accesses something in an *instance* of the outer class
- Note – an *inner* object must be tied to a specific *outer* object on the heap.



Instance of an Inner Class

- Instantiate an inner class from code within the outer class, the instance of the outer class is the one the inner object will bond with.

```
class MyOuter  {
   private int x;                          // The outer class has a private instance variable 'x'
   MyInner inner = new MyInner();          // Make an instance of the inner class
   public void doStuff() {
      inner.go();                          // call a method on the inner class
    }

   class MyInner {
      void go() {
         x = 42;                           // The method in the inner class uses the outer class instance variable 'x', as if 'x' belonged to the inner class.
       }
    } // close inner class

} // close outer class
```



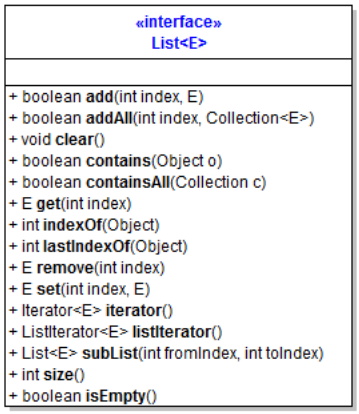
Why Use Inner Classes

- It is a way of logically grouping classes that are only used in one place:
  - If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- It increases encapsulation:
  - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- It can lead to more readable and maintainable code:
  - Nesting small classes within top-level classes places the code closer to where it is used.

Data Structures

- Java allows you to collect and manipulate your data without having to write your own sort algorithm.
- LinkedList<E> allows for constant-time insertions or removals *using iterators*, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but finding a position in the list takes time proportional to the size of the list.
- ArrayList<E>, on the other hand, allow fast random read access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap.

List Interface



- The **List** is the base interface for all list types, and the **ArrayList** and **LinkedList** classes are two common List's implementations.

ArrayList<E>

- **ArrayList**:
  - An implementation that stores elements in a backing array.
  - The array's size will be automatically expanded if there isn't enough room when adding new elements into the list.
  - It's possible to set the default size by specifying an initial capacity when creating a new ArrayList.
  - Basically, an ArrayList offers constant time for the following operations: size, isEmpty, get, set, iterator, and listIterator; amortized constant time for the add operation; and linear time for other operations.
  - Therefore, this implementation can be considered if we want fast, random access of the elements.

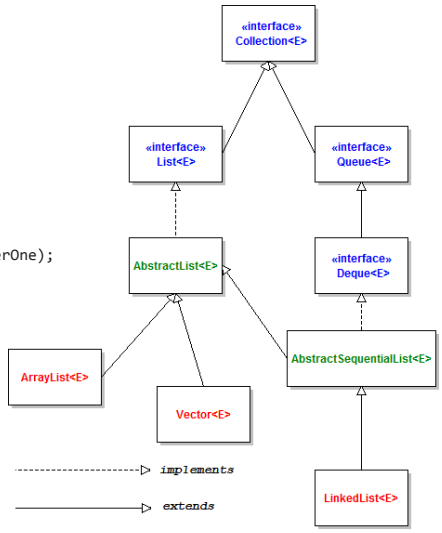
LinkedList<E>

- **LinkedList**:
  - An implementation that stores elements in a doubly-linked list data structure.
  - It offers constant time for adding and removing elements at the end of the list; and linear time for operations at other positions in the list.
  - Therefore, we can consider using a LinkedList if fast adding and removing elements at the end of the list is required.

- Besides ArrayList and LinkedList, Vector class is a legacy collection and later was retrofitted to implement the List interface. Vector is thread-safe, but ArrayList and LinkedList are not.

Inheritance Tree of List Collections

```
List<Integer> listNumbers = new ArrayList<>();
List<String> linkedNames = new LinkedList<>();


List<Integer> listNumbers = new ArrayList<>(1000);


List<Integer> listNumberOne;  // existing collection
List<Integer> listNumberTwo = new ArrayList<>(listNumberOne);



 List<Number> linkedNumbers = new LinkedList<>();

 linkedNumbers.add(new Integer(123));
 linkedNumbers.add(new Float(3.1415));
 linkedNumbers.add(new Double(299.988));
 linkedNumbers.add(new Long(67000));
```



Other Collections

- ArrayList & LinkedLists are not the only collections including:
  - TreeSet
    - Keeps elements sorted and prevents duplicates
  - HashMap
    - Store & access elements as name/pair values
  - HashSet
    - Prevents duplicates in a collection, and given an element, can find that element in the collection quickly
  - LinkedHashMap
    - Similar to a HashMap, except it remembers the order in which elements (name/value pairs) were inserted
  - …

java.util.Collections

- No sort method available for an ArrayList
- Collections.sort() ? `public static void sort(List list)`
  - Takes a List and since ArrayList implements the List interface, ArrayList IS-A List
  - Through Polymorphism, you can pass an ArrayList to a method declared to take a List.

```
import java.util.Collections;
```

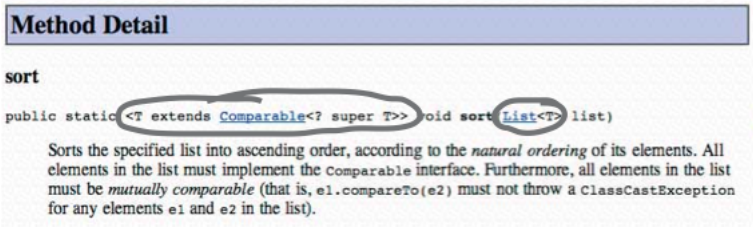
Collections.sort() - Strings

```
List <String> friends = new ArrayList<String>( );


try{
    File file = new File("Friends.txt");
    BufferedReader reader = new BufferedReader(new FileReader(file));
    String line = null;
    while ((line = reader.readLine()) != null){
      friends.add(line);
    }
  } catch(Exception e){
      e.printStackTrace();
  }

    Iterator<String> itr = friends.iterator( );

    while ( itr.hasNext( ) ) {
        String person = itr.next( );
        System.out.println(person);
    }

    Collections.sort(friends);
```

Collections.sort() - Objects

```
List <Person> friends = new ArrayList<Person>( );


friends.add(new Person("Jane Doe", "(098)555-5555" ))
friends.add(new Person("John Doe", "(098)555-5455" ))
friends.add(new Person("Jennifer Doe", "(098)555-5355" ))

// Wont compile
Collections.sort(friends);
```

- won't compile - what is it sorting?



Method Detail

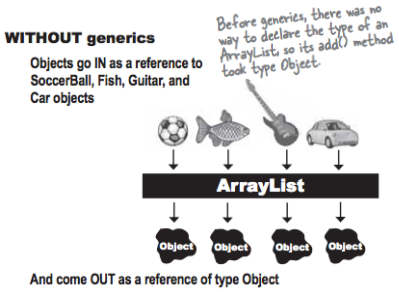
sort

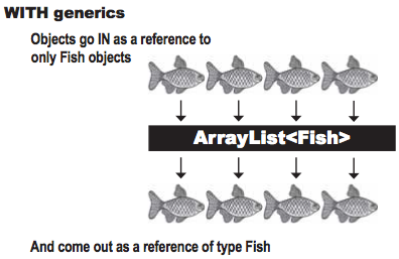
public static <T extends Comparable<? super T>> void sort(List<T> list)

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).

Generics

- sort() method uses generics –
- In Java <> relate to the use of generics
  - With generics you can create type-safe collections where more problems are caught at compile-time instead of runtime.





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

Using Generic Classes

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

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)

    // more code
}
```

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.

Generic Methods

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

```
public class ArrayList<E> extends AbstractList<E> ... {
        public boolean add(E o) // E already defined in the class
```

  - Using a type parameter NOT defined in the class definition

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

  - (T declared earlier in the method declaration - unusual)

Generic Methods

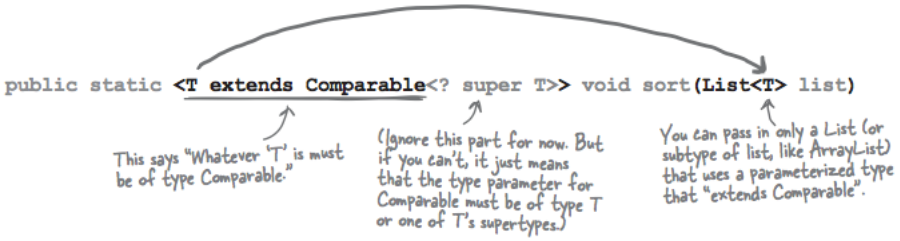
- <T extends Animal> is part of the method declaration, meaning any subclass of Animal is legal

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

                ArrayList<Animal>
                ArrayList<Dog>
                ArrayList<Cat>
```

- Where the method argument is (ArrayList<Animal> list) means that ONLY an ArrayList<Animal> is legal

```
Public void takeSomething(ArrayList<Animal> list)
```

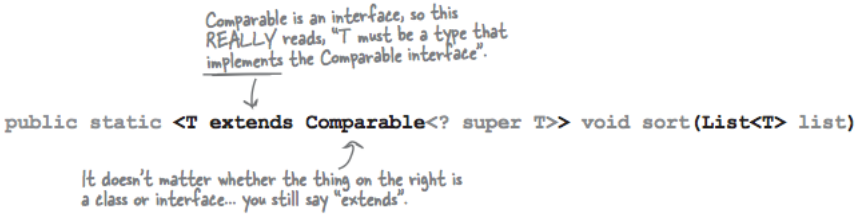
Back to sort()



- The sort method only takes lists of Comparable objects
  - Person is not a subtype of Comparable, therefore it cannot be sorted (as things stand!)

extends in Generics

- In generics extends means extends OR implements
- In generics, the keyword extends really means IS-A and works for both classes & interfaces



Person implements Comparable

- What makes one person less than equal to or greater than another?
- Once you decide this you implement the Comparable interface

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

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

Comparable

- Person class is now comparable

```
class Person implements Comparable<Person>{

…

    public int compareTo(Person p){
        return name.compareTo(p.getName());
    }
```

- In the tester class

```
// compiles
Collections.sort(friends);
```

Comparator

- Collection element is comparable (implemented Comparable), but you only get one chance to implement the compareTo() method.
- Sort() method is overloaded to take a comparator

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

- In tester class

```
// Inner class implements Comparator - Note type parameter
class PersonCompare implements Comparator<Person>{
    public int compare(Person one, Person two){
        return one.getPhoneNumber().compareTo(two.getPhoneNumber());
    }
}
// PersonCompare instance
PersonCompare personCompare = new PersonCompare();
Collections.sort(friends, personCompare);
```

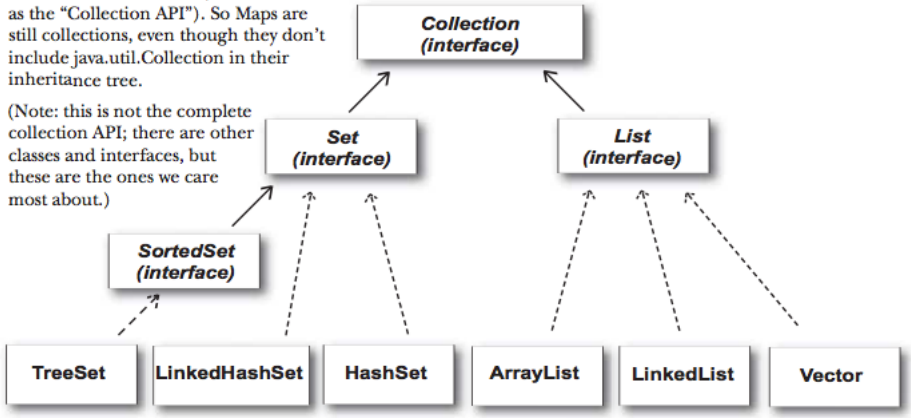
Comparator

- Comparator is external to the element type you are comparing – it's a separate class
- You can write as many as you like
- Call the overloaded sort() method that takes the List and the Comparator that will help the sort() method put things in order.
- 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.

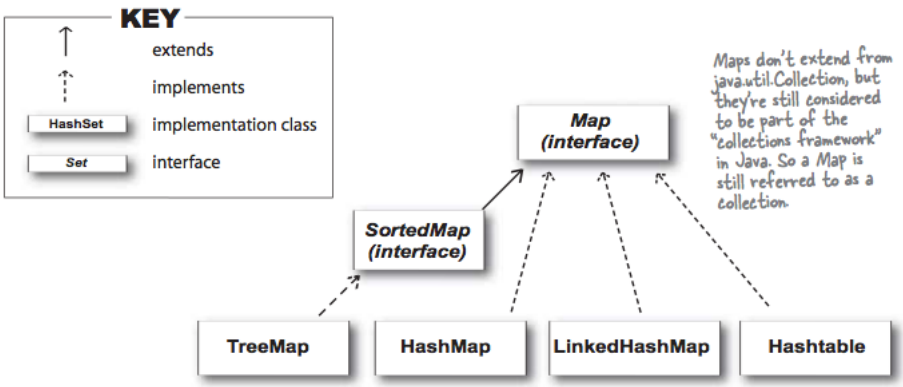
Collections API (part of)

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.)



Maps



HashSet

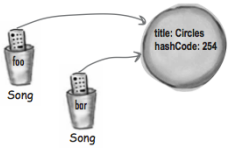
- addAll() method can take another Collection and use it to populate the HashSet
- In tester class

```
HashSet<Person> friendSet = new HashSet<Person>();
friendSet.addAll(friends);
```

Object Equality

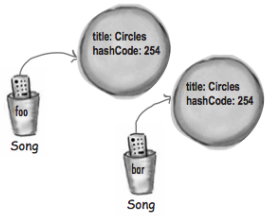
- Reference Equality
  - two references, one object on the heap

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.




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

Object Equality

- In both cases you may want to override the hashCode() method inherited from class Object.
- With Object equality you may also need to override the equals() method
- HashSet uses both hashCode() & equals to check for duplicates.
- In Person class

```
public boolean equals(Object aPerson){
    Person p = (Person) aPerson;
    return getName().equals(p.getName()));
}

public int hashCode(){
    return name.hashCode();
}
```