Collections types

Collections Class in Theory

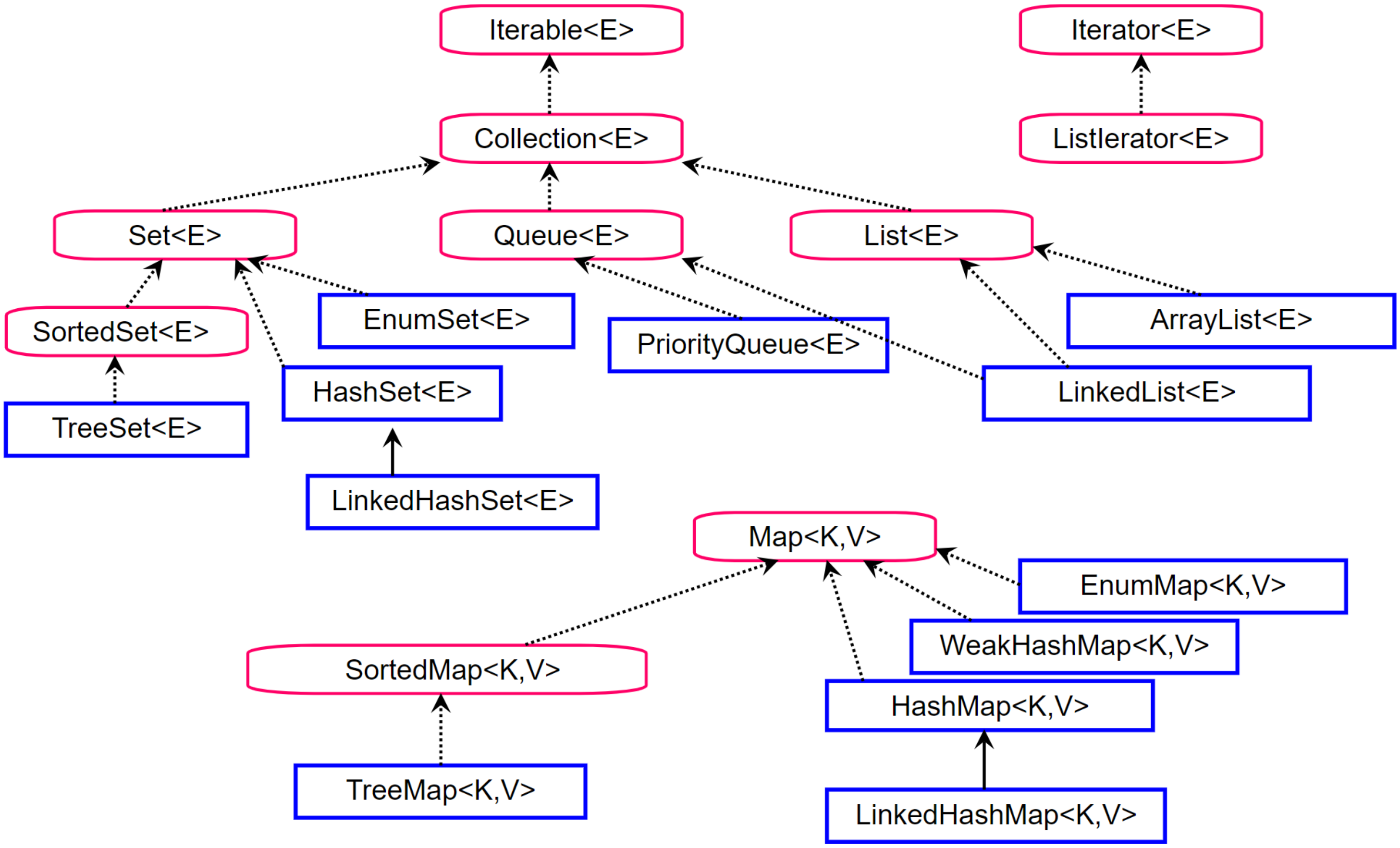
- prebuilt data structure(s) that handle ANY custom object we (create) give it
    - why write the same data structure that other people use over and over
- data structures covered today
    - Linked Lists
    - Arrays of Objects (not simple data type arrays)
- data structures covered later
    - Queues
    - Sets
    - Maps
- each data structure has it's pros and cons
- import java.util.LinkedList

Collections Class details

- The class is a huge help to experienced programmer that know what some data structures are.
    - why we cover AFTER Linked Lists and Stacks/Queues
- the Collections class is a SUPER class, so it itself can do many options to the lower data structures it creates
- all functions and sub-classes (as of 1.5) ARE NOW GENERIC
    - does not matter the object, will work with it

The Collection Class and SubClasses

```
      Queue <Integer> crap  = new Queue<Integer>();  // didn't work!!
      Queue <String> crap2  = new Queue<String>();  // didn't work!!

// ONly Lined lIsts and Priority Lists
```



from https://slideplayer.com/slide/9085017/

The bad side of Collections

- only works with NON-simple data types
  - Integer // int != Integer
  - Double // double != Double
- ***ANY*** CREATED DATA TYPES (like NODE)
  - ***THAT'S WHY GENERIC!!! WORKS WITHOUT A LOT CHANGES!!***
- have to "downcast" to type cast when retrieving objects for the data structures
- have to redo (add) a NEW compareTo that works with general Objects

The Collections version of compareTo

- collections deals with "Objects"
  - Object is a **very** general complex data type
- Old compareTo is used for comparing two items
- COLLECTIONS compareTo is used for sorting MASSIVE AMOUNTS

Overloading the CompareTo (numeric) operator

To Use between same objects

```
public int compareTo(Employee x)
{
        if (this.age == x.age)
        { return 0; }
        else if (this.age < x.age)
        { return -1; }
        else // (this.age == x.age)
        { return 1; }
}
```

To use for Collections

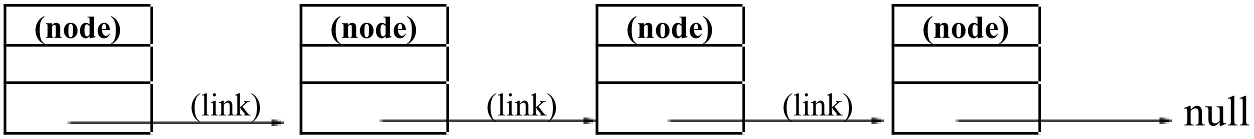
```
public int compareTo(Object that)
{
        Employee x = (Employee) that; // casting

        if (this.age == x.age)
        { return 0; }
        else if (this.age < x.age)
        { return -1; }
        else // (this.age == x.age)
        { return 1; }
}
```

Overloading the CompareTo (String) operator

Function

```
public int compareTo(Employee x)    // comparTo is comparing EMployees
{ return this.getlastName().compareTo(x.getlastName()); } // compareTO is comparing Strings
```

The Linked List Data Structure



- Collections will handle:
    - all Objects
        - using Generics
    - all links NAMED “next”
- must
    - import java.util.LinkedList;
    - create the BASE OBJECT for the linked list

Example (incomplete BASE object)

```
public class Employee
{
        private String first;
        private String last;
        private int age;

        // create other useful methods
}
```

Example Driver

```
LinkedList <Employee> TAMU = new LinkedList <Employee>();

Employee adjunct = new Employee("Prof. L", "Lupoli", 30);
Employee dean = new Employee("Jack", "McLaughlin", 90);
Employee professor = new Employee("Peter", "Joyce", 60);

TAMU.add(adjunct);
TAMU.add(dean);
TAMU.add(professor);
```

Linked List Constructor Summary
LinkedList()   Constructs an empty list.
LinkedList(Collection<? extends E> c)   Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Linked List Method Summary	
boolean	**add**(E o) Appends the specified element to the ***end*** of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**addFirst**(E o) Inserts the given element at the ***beginning*** of this list.
void	**addLast**(E o) Appends the given element to the ***end*** of this list.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this `LinkedList`.
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
E	**element**() Retrieves, but does not remove, the head (first element) of this list.
(any data type the List is made up of) ***E***	**get**(int index) Returns the element at the specified position in this list.
E	**getFirst**() Returns the first element in this list.
E	**getLast**() Returns the last element in this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

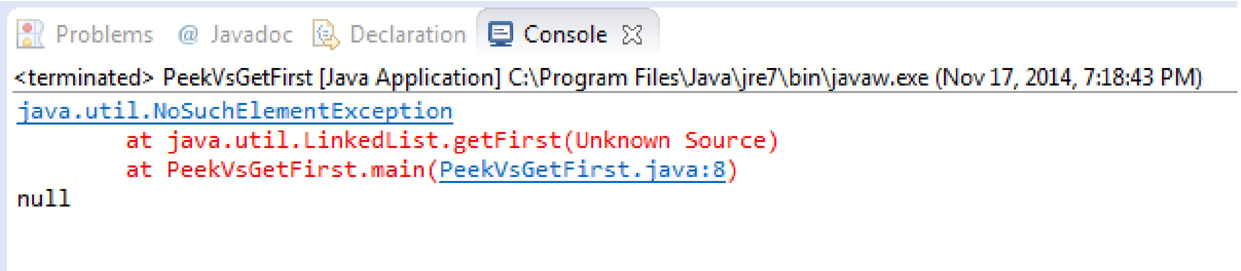
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
ListIterator<E>	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
boolean	**offer**(E o) Adds the specified element as the tail (last element) of this list.
E	**peek**() Retrieves, but does not remove, the head (first element) of this list.
E	**poll**() Retrieves and removes the head (first element) of this list.
E	**remove**() Retrieves and removes the ***head*** (first element) of this list.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this list.
E	**removeFirst**() Removes and returns the first element from this list.
E	**removeLast**() Removes and returns the last element from this list.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.

Peek vs. Get First

- thanks to Corbin Crockett F'14
- the difference between the two is getFirst can throw an exception

getFirst throwing an exception

```
import java.util.LinkedList;
import java.util.NoSuchElementException;
public class PeekVsGetFirst {

    public static void main(String [] args){
        LinkedList<Car> cars = new LinkedList<Car>();
        try{
            cars.getFirst();
        }
        catch(NoSuchElementException e){
            e.printStackTrace();
        }

        System.out.println(cars.peekFirst());
    }
}
```



```
<terminated> PeekVsGetFirst [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Nov 17, 2014, 7:18:43 PM)
java.util.NoSuchElementException
	at java.util.LinkedList.getFirst(Unknown Source)
	at PeekVsGetFirst.main(PeekVsGetFirst.java:8)
null
```

1. Create a Java Projected, download the Driver using a Linked List set of Cars here
2. Create a base object “Car” that contains the data members’ nickname, license, and MPG. The class then needs:
    a. constructors
    b. setters/getters
    c. toString
    d. equals
    e. compareTo **(skip , will get after we make Car comparable)**
    f. remember, it can ALLL be done for you!! (Don’t remember, here!!)
3. Remember the enhanced for loop in C++?? Look for it in the given driver. What are we UNABLE to do to the LinkedList using that type of for loop?
4. Looking at the auto-generated equals method (of using Eclipse), for two cars to be equal, what exactly has to match?

Answer$_b$:

The Array List Data structure

- ArrayList is creating an array of objects (INDEXCARDS in this example)

INDEXCARD
name
tele
zip

  - we give it the BASE object
  - ArrayList<OBJECT> **anyname** = new ArrayList<OBJECT>();
- uses an iterator to traverse the array
- must import
  - java.util.ArrayList;

0	1	2	3	4	5	6	7	8
name	name	Name	name	name	name	name	name	name
tele	tele	Tele	tele	tele	tele	tele	tele	tele
zip	zip	Zip	zip	zip	zip	zip	zip	zip

Example Object (incomplete) for the ArrayList

```
class INDEXCARD
{
        private String name;
        private String tele;
        private int zip;

        INDEXCARD(String x, String t, int z) { name = x;  tele = t zip = z; }

        public String getName() { return name; }
        public String getTele() { return tele; }
        public int getZip() { return zip; }

        // rest of complete class profile methods
        public String toString() { return (name + “ “ + tele + “ “ + zip); }
}
```

Use of ArrayList in the Driver

```
public static void main(String args[])
{
ArrayList<INDEXCARD> greetings = new ArrayList<INDEXCARD>();
INDEXCARD x = new INDEXCARD("Prof. Lupoli", “1800SUPERMAN”, 21117);

greetings.add(x); // inserts “x” into the array list
```

Draw what the ArrayList would look like at this moment: **Answer$_b$:**

Difference between size & capacity

- the **size** is the number of elements in the list
- the **capacity** is how many elements the list can potentially accommodate without reallocating its internal structures
- so we would need to add default values in order to start messing with them individually

```
// size is a parameter in this function

ArrayList <T> table = new ArrayList<T>();

while(table.size() < size) { table.add(new BSTree()); }
```

Commonly used ArrayList(Vector) Functions

- there are more, you can check Java’s website for info

ArrayList Constructor Summary

`ArrayList()`

Constructs an empty list with an initial capacity of ten.

```
ArrayList<TEACHER> Staff_Roster = new ArrayList<TEACHER>();
```

`ArrayList(Collection<? extends E> c)`

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`ArrayList(int initialCapacity)`

Constructs an empty list with the specified initial capacity.

ArrayList Method Summary

boolean `add(E o)`

Appends the specified element to the ***end*** of this list.

```
TEACHER t1 = new TEACHER(“Prof.”, “Lupoli”, 30); // constructor created for TEACHER
Staff_Roster.add(t1); // create the object first, then place into the ArrayList
```

void `add(int index, E element)`

Inserts the specified element at the specified position in this list.

```
same as above, with an index
```

boolean `addAll(Collection<? extends E> c)`

Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.

```
TEACHER AACC1 = new TEACHER(“Janice”, “Gilbert”, 30);
TEACHER AACC2 = new TEACHER(“Bud”, “Brengle”, 54);
TEACHER AACC3 = new TEACHER(“Karen”, “Hommel”, 30);
TEACHER AACC4 = new TEACHER(“Mary Jane”, “Blasi”, 45);
AACC_Staff_Roster.add(AACC1); // AACC_Staff_Roster already established (code not shown)
AACC_Staff_Roster.add(AACC2);
AACC_Staff_Roster.add(AACC3);
AACC_Staff_Roster.add(AACC4);
Staff_Roster.addAll(AACC_Staff_Roster); // adds all AACC teachers to Staff Roster
```

boolean `addAll(int index, Collection<? extends E> c)`

Inserts all of the elements in the specified Collection into this list, starting at the specified position.

```
same as above, with an index
```

void `clear()`

Removes all of the elements from this list.

```
Staff_Roster.clear();
```

Object `clone()`

Returns a shallow copy of this `ArrayList` instance.

```
ArrayList Staff_Roster2 = (ArrayList) Staff_Roster.clone();
// creates a copy of the original ArrayList
```

boolean	**contains**(Object elem) Returns true if this list contains the specified element.

TEACHER t1 = new TEACHER(“Prof.”, “Lupoli”, 30);
if(Staff_Roster.contains(t1))	

E	**get**(int index) Returns the element at the specified position in this list.

System.out.println(Staff_Roster.get(i)); // USES OVERLOADED TOSTRING!!! (if created)
// prints the object’s data in a line // MORE ON THIS LATER!!!	

int	**indexOf**(Object elem) ***Searches*** for the first occurrence of the given argument, testing for equality using the equals method.

TEACHER t1 = new TEACHER(“Prof.”, “Lupoli”, 30); *// MUST CREATE TARGET NODE TO MATCH!!!*
Staff_Roster.indexOf(t1)
OR // MORE ON THIS LATER!!!
Staff_Roster.indexOf(new TEACHER(“Prof.”, “Lupoli”, 30)) // SUGGESTED
RETURNS a -1 IF NOTHING IS FOUND!!!	

boolean	**isEmpty**() Tests if this list has no elements.
while(!Staff_Roster.isEmpty())	
int	**lastIndexOf**(Object elem) Returns the index of the last occurrence of the specified object in this list.

TEACHER t2 = new TEACHER(“Jack”, “McLaughlin”, 70);
Staff_Roster.lastIndexOf(t2);	

E	**remove**(int index) Removes the element at the specified position in this list.
Staff_Roster.remove(0); // removes object at index 0	
boolean	**remove**(Object o) Removes a **single** instance of the specified element from this list, if it is present (optional operation). **Searches through the array to find the object.**

TEACHER t2 = new TEACHER(“Jack”, “McLaughlin”, 70);
Staff_Roster.remove(t2);
OR *// MUST CREATE TARGET NODE TO MATCH!!!*
Staff_Roster.remove(new TEACHER(“Jack”, “McLaughlin”, 70)); // SUGGESTED	

E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.

TEACHER t2 = new TEACHER(“Jack”, “McLaughlin”, 70);
Staff_Roster.set(2, t2);	

int	**size**() Returns the number of elements in this list.
for(int i = 0; i < Staff_Roster.size(); i++)	

“get” function from the Method Summary

INDEXCARD
name
tele
zip

- retrieves a literal OBJECT back at that index
    - this is a problem
    - because in our case the Object is REALLY an INDEXCARD

“Getting” to gather info

```
ArrayList <INDEXCARD> TAMU  = new ArrayList <INDEXCARD>();
INDEXCARD newProfessor = new INDEXCARD("Prof. L", "1800SUPERMAN", 21117);
INDEXCARD dean = new INDEXCARD("Jack", "1800THEDEAN", 26751);
INDEXCARD oldProfessor = new INDEXCARD("Peter", "1900THEPROF", 28178);

TAMU.add(newProfessor);
TAMU.add(dean);
TAMU.add(oldProfessor);

//Complex Method Calling
String target_name = "Prof. L";

if(target_name.equals(TAMU.get(0).getname())) // had to go down several layers
{}                                             // just to make sure we got a String

//Simple Method Calling

String newguy; // will get from retrieval
String target_name = "Prof. L";

newguy = TAMU.get(0).getname();
```

```
if(target_name.equals(newguy))
```

ArrayList's indexOf method

- The indexOf requires the use of the OVERLOADED equals method
  - ***WHICH YOU SHOULD ALREADY HAVE!!!***
  - uses complete class profile
- used for other classes in Collections as well
- ***returns a literal OBJECT back so you can FIND an object in an ArrayList***

0	1	2	3	4	5	6	7	8
Prof. L	Jack	Peter	**name**	**name**	**name**	**name**	**name**	**name**
…	**…**	**…**	**tele**	**tele**	**tele**	**tele**	**tele**	**tele**
21117	26751	28178	**zip**	**zip**	**zip**	**zip**	**zip**	**zip**

Overloading the "equals" operator

THIS		THAT
Jack	name	Jack
1800THEDEAN	tele	1800THEDEA N
26751	zip	26751

Finding a value in an ArrayList

```
ArrayList <INDEXCARD> TAMU  = new ArrayList <INDEXCARD>();
INDEXCARD newProfessor = new INDEXCARD("Prof. L", "1800SUPERMAN", 21117);
INDEXCARD dean = new INDEXCARD("Jack", "1800THEDEAN", 26751);
INDEXCARD oldProfessor = new INDEXCARD("Peter", "1900THEPROF", 28178);

TAMU.add(newProfessor);
TAMU.add(dean);
TAMU.add(oldProfessor);

// Find Jack!

// This identifies that the item is at least in the ArrayList
if(TAMU.indexOf(new INDEXCARD("Jack", "1800THEDEAN", 26751)) > -1) // But why -1???
```

```
// this returns the index number it WAS found in.
int indexFound = TAMU.indexOf(new INDEXCARD("Jack", "1800THEDEAN", 26751);

if( indexFound > -1) // change something
{}
else // was never found
{}
```

BUT!!! BY DEFAULT, INDEX OF USES THE OLD EQUALS (string) to compare!!!

They aren't STRINGS!!!

Make sure the object class has it's own equals!!!

1. Change the driver we just used to an ArrayList.
2. Using the "get" method to retrieve the value at [0]
3. Print what was retrieved. What was it?
4. Now, comment out "toString". Using the "get" method to retrieve the value at [0]. Again, print what was retrieved. What was it?
5. Using the code above, use index of to find a discoverable Car in the ArrayList. Confirm it was found.
6. Comment out "equals" method in Car. Try again. What happens?

Vectors in Theory

- can be thought of as an arrays that shrinks and grows
- can change while your program is running, called dynamic (ever changing)
- adds NEW items in the next available spot starting from 0
    - increments/decrements itself

One already established in vector

[0]	(only one segment of memory used)
▨	

Added one more data element to the vector

[0]	[1]	(only two segments of memory used)
▨	▨	←

Erased one data element in the vector

[0]	(back to one)
▨	

Vectors for Real

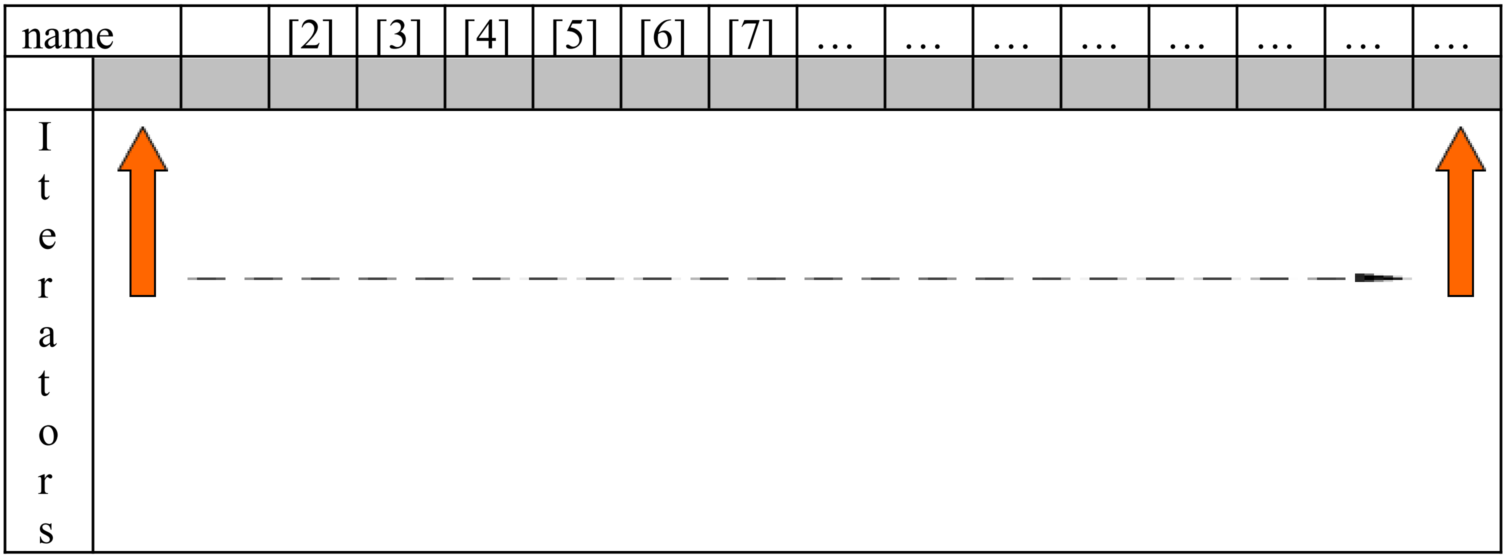
- are almost EXACTLY like ArrayList
    - same methods!!
    - Also inherits Collection features/behaviors
- there are battles online between developers which is faster, better, etc…
- the difference by JAVA is:

ArrayList
Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.) If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.
Vector
The `Vector` class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a `Vector` can grow or shrink as

needed to accommodate adding and removing items after the `Vector` has been created. Unlike the new collection implementations, `Vector` is synchronized.

Iterators in Array/Vector/Linked Lists

- The iterator is a common item used to ***traverse a collection***
  - must
    - import java.util.Iterator;
  - an iterator of the same type as the Collection must be created
  - a Collection must be created
    - Iterator <OBJECT> x = COLLECTION.iterator();
- think of it as CURSOR



name		[2]	[3]	[4]	[5]	[6]	[7]	…	…	…	…	…	…	…	…

Iterators

- The iterator has two important functions
  - hasNext()
    - checks to see if at the end of the Collection
  - next()
    - moves to next element in the Collection

Example of an iterator in use

```
Iterator <Employee> Radford = TAMU.iterator();  // TAMU was the orig. list
```

```
// list of Radford Employees
while (Radford.hasNext())
{
        Employee aPerson = Radford.next(); // gathering and placing into a temp Employee
        System.out.print(aPerson.firstname + " ");
        System.out.println(aPerson.lastname);

        OR

        System.out.println(aPerson); // if we have a toString()
}
```

Editing using Iterators or get()

- there are multiple ways of access a Collection
    - get()
    - iterator and next()
- but the design of the internal custom object needs to allow editing material
    - setter/getters, etc…
- using an iterator
    - next() gives us access to a **specific** element in the Collection
    - that will give us direct access
    - that will give us the ability to change a value ***<u>within</u>*** the collection
- using get()
    - get() returns an object referencing the original one

Iterator Version #1 (no editing)

```
// a collection of BST was set up within an ArrayList named table
// the table itself was full of <BinarySearchTree<Node>> (s)

Iterator <BinarySearchTree<Node>> list = table.iterator();
while (list.hasNext())
{
      Node x = new Node("Lupoli");
      Node y = new Node("Jensen");
      BinarySearchTree retreived = list.next();
      retreived.insert(x);
      retreived.printTree();
      retreived.insert(y);
      retreived.printTree();
      break; // I only wanted to add to ONE element in the entire ArrayList
}
```

Iterator Version #2

```
        Iterator<NODE> y = Java_Class.iterator();
        while (y.hasNext())
        {
                // collect the NODE
                y.next().setFname("Bowen"); // node gave us access to change the value
        }
        y = Java_Class.iterator();
        while (y.hasNext()) // just to display the list again
        {
                // collect the NODE
                NODE aPerson = y.next();
                System.out.println(aPerson);
        }
```

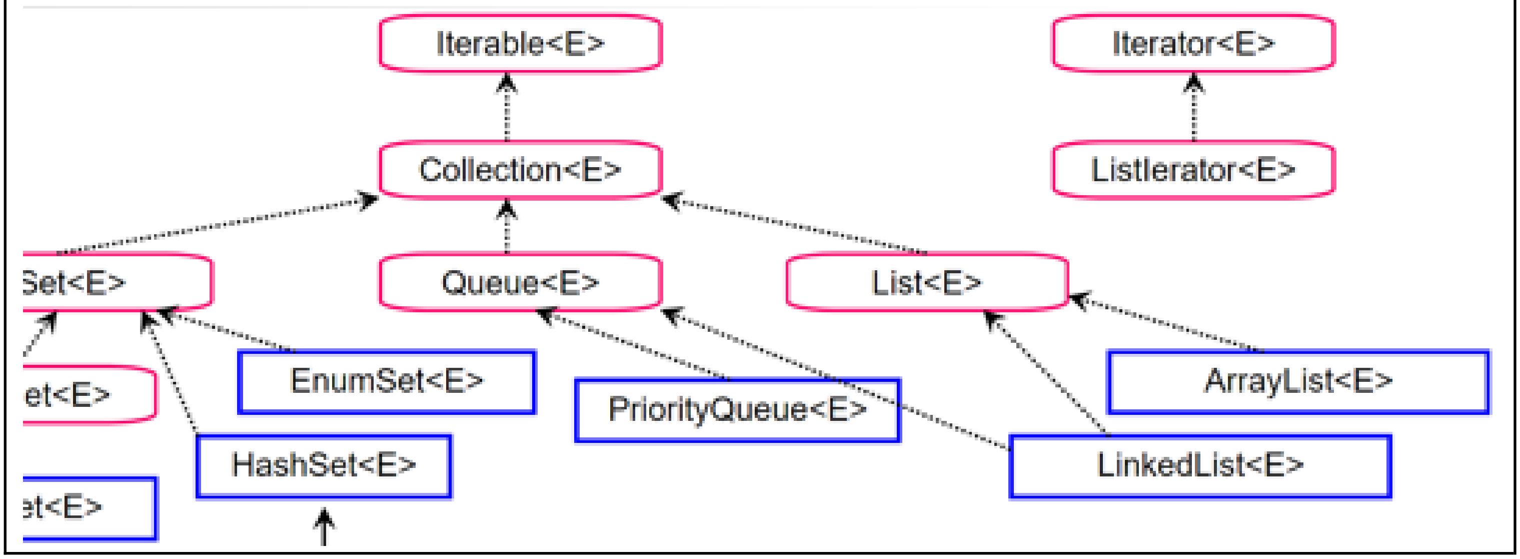
Iterator Version #3

```
for (int i = 0; i < Java_Class.size(); i++)
{ Java_Class.set(i, new NODE("N/A", "N/A", -1, -1, -1)); }

y = Java_Class.iterator();
while (y.hasNext())// just to display the list again
{
        // collect the NODE
        NODE aPerson = y.next();
        System.out.println(aPerson);
}
```

The Collection Class and it's functions

- Collection is the super class of all collections and data structure covered above
- so the functions can be used on the ArrayList, LinkedList, and Queue
- HAS VERY POWERFUL AND USEFUL methods!!
  - sorting!!!

Collections and it's Children



Commonly used Collection Functions

Collections Method Summary	
static `<T> int`	`binarySearch(List<? extends T> list, T key, Comparator<? super T> c)` Searches the specified list for the specified object using the binary search algorithm. <NEEDS TO BE SORTED FIRST>
`static boolean`	`disjoint(Collection<?> c1, Collection<?> c2)` Returns `true` if the two specified collections have no elements in common.
static `<T> List<T>`	`emptyList()` Returns the empty list (immutable).
static `<T> void`	`fill(List<? super T> list, T obj)` Replaces all of the elements of the specified list with the specified element.
`static int`	`frequency(Collection<?> c, Object o)` Returns the number of elements in the specified collection equal to the specified object.
`static int`	`indexOfSubList(List<?> source, List<?> target)` Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
`static int`	`lastIndexOfSubList(List<?> source, List<?> target)` Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static `<T> ArrayList<T>`	`list(Enumeration<T> e)` Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.

static
`omparable<? super T>>`
`T`
	`max(Collection<? extends T> coll)`
	Returns the maximum element of the given collection, according to the *natural ordering* of its elements.

static `<T> T`	`max(Collection<? extends T> coll, Comparator<? super T> comp)` Returns the maximum element of the given collection, according to the order induced by the specified comparator.

static
`omparable<? super T>>`
`T`
	`min(Collection<? extends T> coll)`
	Returns the minimum element of the given collection, according to the *natural ordering* of its elements.

static `<T> T`	`min(Collection<? extends T> coll, Comparator<? super T> comp)` Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static `<T> List<T>`	`nCopies(int n, T o)` Returns an immutable list consisting of `n` copies of the specified object.
static `<T> boolean`	`replaceAll(List<T> list, T oldVal, T newVal)` Replaces all occurrences of one specified value in a list with another.

static void	reverse(List<?> list) Reverses the order of the elements in the specified list.
static <T> Comparator<T>	reverseOrder() Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.
static <T> Comparator<T>	reverseOrder(Comparator<T> cmp) Returns a comparator that imposes the reverse ordering of the specified comparator.
static void	shuffle(List<?> list) Randomly permutes the specified list using a default source of randomness.
static void	shuffle(List<?> list, Random rnd) Randomly permute the specified list using the specified source of randomness.
static Comparable<? super T>> void	sort(List<T> list) Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
static <T> void	sort(List<T> list, Comparator<? super T> c) Sorts the specified list according to the ***order induced by*** the specified comparator.
static void	swap(List<?> list, int i, int j) Swaps the elements at the specified positions in the specified list.

Theory of a Comparable/Comparator

- if the data type was simple, a <, >, ==, would simply do
- but how do we compare or order ***OBJECTS BY CERTAIN DATA***
    - by age
    - by name
    - etc…
- we would have to pick a certain part of that object(s) in order to compare
- there are different ways of comparing Strings and integer values
    - Strings uses the compareTo function covered in Strings
    - Integers can use <, >, = =, etc…

Comparables

- introspective – using own custom built compareTo function
    - can be used to compare **both** homogeneous and heterogeneous objects
    - usually the compareTo method's design has some type or natural ordering
        - by name
        - by income

        - etc…
    - shortcomings
        - can only have ***one*** compareTo function
            - can then use Comparators if you need more sorting options

Deciding between comparable & comparator

- designing how you will use either is important

Gameplan with given code

	(string compareTo) String	(<, >, ==, etc...) Int
Comparible → Homogenous Object	Employee Fullname	Employee age
Comparible → Heterogenous	Employ Fullname vs Index Card Fullname	★
Comparator (Homogenous Objects)	Employee Fullname	Employee age

Compara*ble* for a String

Option #1

```
public class Employee implements Comparable<Object> {

    private String first;
    private String last;
    private int age;

    public Employee(String first, String last, int i) {
        this.first = first;
        this.last = last;
        this.age = i;
    }

    public String getFirst() { return first; }
    public String getLast() { return last; }
    public int getAge() { return age; }

    public int compareTo(Object x)
    {
        if(x instanceof Employee)
        {
            Employee e = (Employee) x;
            if(this.getLast().equals(e.getLast()))
                return this.getFirst().compareTo(e.getFirst());
            else
                return this.getLast().compareTo(e.getLast());
        }
        else if (x instanceof IndexCard)
        {
            IndexCard e = (IndexCard) x;
            if(this.getLast().equals(e.getLast()))
                // comparing Employee first to IndexCard first
                return this.getFirst().compareTo(e.getFirst());
            else
                return this.getLast().compareTo(e.getLast());
        }
        else { return -1; } // not a match
    }
    public String toString() {
        return "Employee [first=" + first + ", last=" + last + "]";
    }
}
```

2nd option for compareTo

```
public class Employee implements Comparable<Employee> {

… // same as above

      public int compareTo(Employee e) {
            if(this.getLast().equals(e.getLast()))
                  return this.getFirst().compareTo(e.getFirst());
            else
                  return this.getLast().compareTo(e.getLast());
…
}
```

Using a comparable

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class Driver {

      public static void main(String args[])
      {
            ArrayList<Employee> x = new ArrayList<Employee>();

            Employee adjunct = new Employee("Dan", "Malesko", 30);
            Employee dean = new Employee("Jack", "McLaughlin", 90);
            Employee professor = new Employee("Peter", "Joyce", 60);
            IndexCard lupoli = new IndexCard("Prof", "Lupoli", "1800SUPERMAN", 21117);

            // comparing an Employee to an Employee
            System.out.println(adjunct.compareTo(dean));
            // comparing an Employee to an IndexCard
            System.out.println(adjunct.compareTo(lupoli));

            x.add(adjunct);
            x.add(dean);
            x.add(professor);

            Iterator<Employee> before = x.iterator();
            while (before.hasNext())
            {
                  Employee aPerson = before.next();
                  System.out.print(aPerson.getFirst() + " ");
                  System.out.println(aPerson.getLast());
            }

            Collections.sort(x);  // Comparable Option

            Iterator<Employee> after = x.iterator();
            while (after.hasNext())
            {
                  Employee aPerson = after.next();
                  System.out.print(aPerson.getFirst() + " ");
                  System.out.println(aPerson.getLast());
            }
      }
}
```

```
-2
1
Dan Malesko
Jack McLaughlin
Peter Joyce
----------------------------
Peter Joyce
Dan Malesko
```

Comparators

- uses an interface
  - info on Comparator available methods here
  - no code given, just the function header ***compare***
    - ***basically same as compareTo***
  - requires a class to implement compare
    - usually a very simple class to create
- used to compare HOMOGENOUS objects that the Comparator didn't cover
  - gives you options if you want to sort the Object in another fashion

Compara*tor* Setup Example

Base Object (incomplete)

```
class Employee // notice no implements here!!!
{
        private String firstname, lastname;
        private int age;

        Employee(String f, String l, int a)
        {
                firstname = f;
                lastname = l;
                age = a;
        }

        public String getfirstName() { return firstname; }
        public String getlastName() { return lastname; }
        public int getAge() { return age; }
}
```

Comparator for a String

```
import java.util.*;

//http://leepoint.net/notes-//java/data/collections/comparators.html

class FULLNAME implements Comparator<Employee>
{
// Comparator interface requires defining compare method.
public int compare(Employee a, Employee b)
{
        if(a.getlastName().equals(b.getlastName())
        { return a.getfirstName().compareTo(b.getfirstName()); }
        else { return a.getlastName().compareTo(b.getlastName()); }
}// What values can CompareTo return??
```

Comparator for an Int (Ascending)

```
import java.util.*;

class AGE implements Comparator<Employee>
{
// Comparator interface requires defining compare method.
   public int compare(Employee a, Employee b)
   {       //... Sort the age of the Objects
      if(a.getAge() < b.getAge()) {return -1;}
      else if(a.getAge() > b.getAge()) {return 1;}
      else { return 0; }
   }
}
```

- Compare/compareTo ALWAYS needs three responses
  - -1
  - 0 (identical)
  - 1

Full Example

```
ArrayList<Employee> x = new ArrayList<Employee>();

Employee adjunct = new Employee("Prof. L", "Lupoli", 30);
Employee dean = new Employee("Jack", "McLaughlin", 90);
Employee professor = new Employee("Peter", "Joyce", 60);

x.add(adjunct);
x.add(dean);
x.add(professor);
```

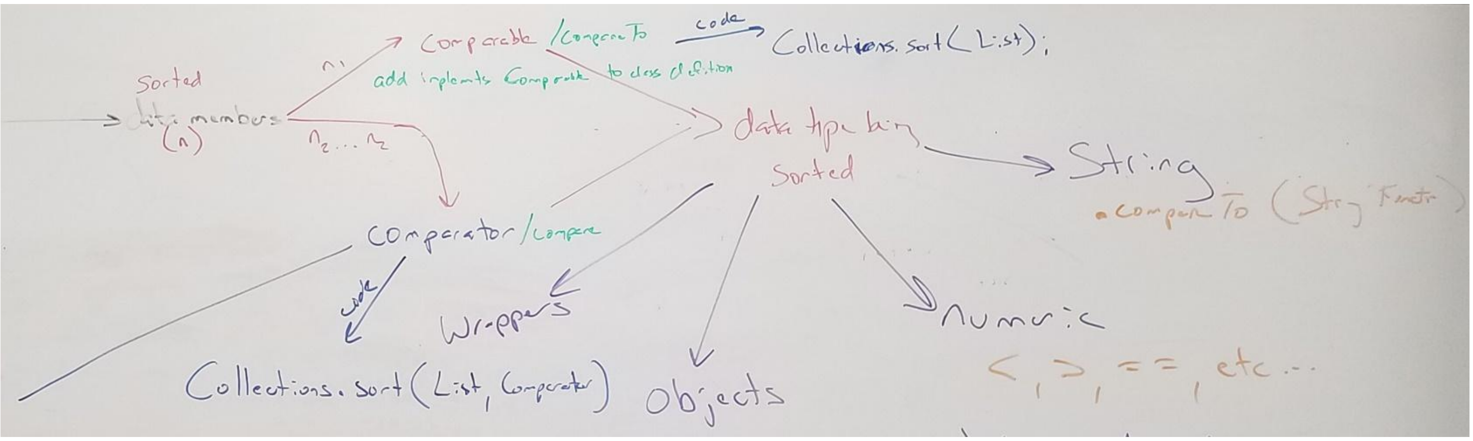
0	1	2
Jack	Peter	Prof. L
…	…	…

```
Iterator<Employee> y = x.iterator();
while (y.hasNext())
{
        Employee aPerson = y.next();
        System.out.print(aPerson.getfirstName() + " ");
        System.out.println(aPerson.getfirstName());
}

// Comparator Option
// comparable for LASTNAME/FULLNAME of Employee
FULLNAME fullname = new FULLNAME();
Collections.sort(x, fullname);
```

0	1	2
Prof. L	Jack	Peter
Lupoli	**McL**	**Joyce**

```
y = x.iterator();
while (y.hasNext())
{
        Employee aPerson = y.next();
        System.out.print(aPerson.getfirstName() + " ");
        System.out.println(aPerson.getfirstName());
}
AGE age = new AGE();
Collections.sort(x, age);
```

Gameplan for Comparable/Comparator Setup



Goal: Sort Animals contained an ArrayList. I want to be able to sort by name and weight (separately).

1. Create Animal class, complete class profile please. The data members are:
    a. Name (String)
    b. Species (String)
    c. Weight (float)
    d. Limbs (don't ask, int)
    e. Eat (Boolean, 0=carnivore, 1=herb…)
2. Create a driver, with an ArrayList and place animals within
3. Confirm your comparing strategy with your instructor before moving on
4. Create the code to compare by both name and weight **(not together)**. Please confirm with output.

    Three strategies that are used:

a. 2 comparators
b. 1 comparable, 1 comparator
c. 2 internal sorting functions

5. ***If you plan*** on a Comparator, name it:

    AnimalZZZComparator.java (ZZZ being the data member about to be sorted)

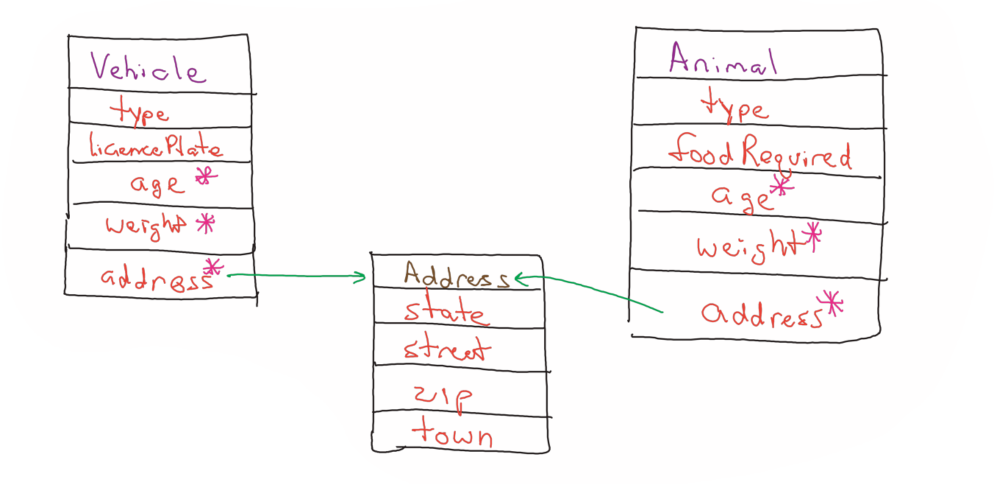
With the Car class, create a compara***tor*** that will order cars by MPG. Then:

1. Create at least 3 instances of Cars
2. Display the list **nicely** BEFORE you sort it
3. Display the list **nicely** AFTER you sort it

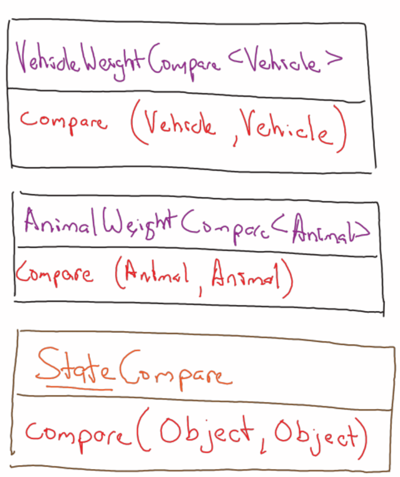
Sorting heterogenous objects

- usually share some kind of common data member
- in the scenario below, Vehicle, Animal and Address are normal objects
    - o all contain complete class profiles with compareTo, equals, etc…
    - o both Vehicle and Animal use Address
- 3 separate comparators used
    - o again, separate files each
    - o Vehicle to Vehicle weight (VehicleWeightCompare)
    - o Animal to Animal weight (AnimalWeightCompare)
    - o Object to Object state (StateCompare)

Complete Classes



Comparators Used


VehicleWeightCompare <Vehicle>
compare (Vehicle, Vehicle)
AnimalWeightCompare<Animal>
Compare (Animal, Animal)
StateCompare
compare(Object, Object)

(exercise next page)

1. Download the zipped code from below:

http://faculty.cse.tamu.edu/slupoli/notes/Java/code/Comparators/HeterogenousComparatorExample.zip

2. Using the driver in code downloaded, draw what the "list" would look like.
3. What should it look like sorted?
4. How would we be able to sort the same list by weight (Vehicle or Animal)?

Using a Collection as a parameter

```
void printList(ArrayList<Employee> x)
{
        Iterator <Employee> student = x.iterator();

        while (student.hasNext())
        {
                System.out.println(student.next());
        }
}
```

Polymorphism in Lists

- Lists

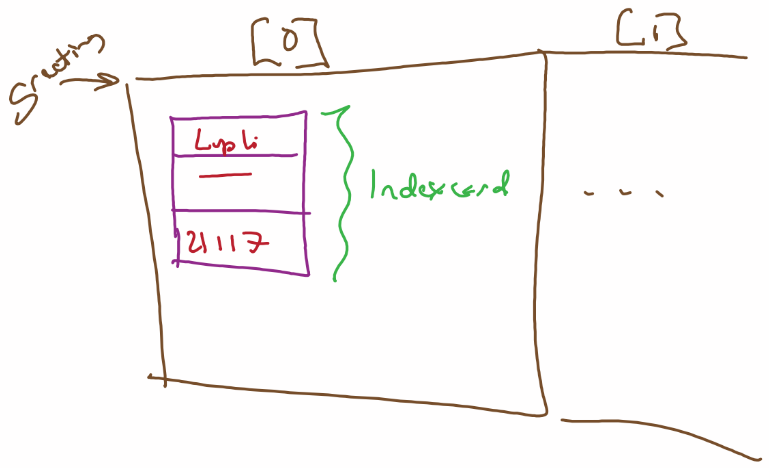
- ArrayList
- Vector
- LinkedList
- Stack (covered later)

Polymorphism Example for Lists

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Stack;
import java.util.Vector;


public class test
{

      public static void main(String args[])
      {
            List <Employee> listA = new ArrayList();
            List <Employee> listB = new LinkedList();
            List <Employee> listC = new Vector();
            List <Employee> listD = new Stack();
      }

}
```

Answer Section

What does an ArrayList of IndexCards Look like?



Creating a Car

```java
public class Car {

        private String nickname;
        private String license;
        private int MPG;


        public String getNickname() { return nickname; }
        public void setNickname(String nickname) { this.nickname = nickname; }
        public String getLicense() { return license; }
        public void setLicense(String license) { this.license = license; }
        public int getMPG() { return MPG; }
        public void setMPG(int mPG) { MPG = mPG; }

        @Override
        public boolean equals(Object obj) {
                if (this == obj) { return true; }
                if (obj == null) { return false; }
                if (getClass() != obj.getClass()) { return false; }
                Car other = (Car) obj;
                if (MPG != other.MPG) { return false; }
                if (license == null)
                {
                        if (other.license != null) { return false; }
                }
                else if (!license.equals(other.license)) {  return false; }
                if (nickname == null) {
                        if (other.nickname != null)
                                return false;
                } else if (!nickname.equals(other.nickname))
                        return false;
                return true;
        }

        @Override
        public String toString() {
                return "Car [nickname=" + nickname + ", license=" + license + ", MPG="
                                + MPG + "]";
        }


}
```

```
import java.util.Comparator;

public class MPG implements Comparator<Car> {

    @Override
    public int compare(Car o1, Car o2) {
        if(o1.getMPG() < o2.getMPG())  { return -1; }
        else if(o1.getMPG() > o2.getMPG()) { return 1; }
        return 0;
    }

}
```

Car and Comparable

```
public class Car implements Comparable <Car>{
    private String nickname;
    private String license;
    private int mpg;

    Car(String n, String l, int m){
        nickname = n;
        license = l;
        mpg = m;
    }
    public String print(){ return nickname + " " + license + " " + mpg; }
    public int compareTo(Car c){
        if (this.mpg > c.mpg){ return 1; }
        else if (this.mpg < c.mpg){ return -1; }
        else { return 0; }
    }

}
```

```
//Benjamin M. Yankowski (Spidey)
//Nathaniel Dorr

import java.util.Collections;
import java.util.LinkedList;

public class NewJunkDriver {

    public static void main(String[] args) {

        LinkedList<Car> list = new LinkedList<Car>();

        list.add(new Car("Camery", "111III01", 25));
        list.add(new Car("Porche", "2FAST4U", 3));
        list.add(new Car("Bike", "[INSERT CENSORSHIP]", 9000000));

        for(int n=0; n < list.size(); n++){
            System.out.println(list.get(n).print());
        }
        Collections.sort(list);
        for(int n=0; n < list.size(); n++){
            System.out.println(list.get(n).print());
        }
    }

}
```

Sources:

Comparable vs. Comparator
http://stackoverflow.com/questions/1440134/java-what-is-the-difference-between-implementing-comparable-and-comparator

Parameterized Generic Classes
http://javahowto.blogspot.com/2008/06/java-generics-examples-parameterized.html
http://docs.oracle.com/javase/tutorial/java/generics/types.html