List Range Views

…

Range-View Operations

subList(int fromIndex, int toIndex)

- returns a List view of portion of this list
- w/ indices in `[fromIndex..toIndex[`
- This half-open range mirrors the typical for loop; e.g.,

```
for (int i = fromIndex; i < toIndex; i++) { ... }
```

Please note

- returned List is backed up by the original List
- so changes in the former are reflected in the latter

What are Range-View Operations good for?

Eliminate need for explicit range operations

- Any operation that expects a List can be used as a range operation
- Any polymorphic algorithm that operates on a List works with subLists
- How? → Just pass subList view instead of whole List
  → `myMethod(someList.subList(fromIndex, toIndex));`

So…

How would you…

Examples

```
// Remove a range of elements from a List?
list.subList(fromIndex, toIndex).clear();

// Search for an element in a range?
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

These idioms return the index of the found element in the subList, not the index in the backing List

Example – Dealing a handsize of cards from a deck

```
java Deal 4 5
[8 of hearts, jack of spades, 3 of spades, 4 of spades, king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts, queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds, 9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts, ace of hearts]
```

Method to deal a hand from a deck

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();

    List<E> handView = deck.subList(deckSize - n, deckSize);

    // copy selected elements
    List<E> hand = new ArrayList<E>(handView);

    // removes selected elements from deck
    handView.clear();
    return hand;
}
```

Let's now make a deck then use our dealHand(...) method

```
public class Deal {
    public static void main(String[] args) {

        if (args.length < 2) {
            System.out.println("Usage: Deal hands cards");
            return;
        }

        // Parse CLI arguments
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
```

Preparing the deck

```
// Make a normal 52-card deck
String[] suit = new String[] {
        "spades", "hearts",
        "diamonds", "clubs"
};

String[] rank = new String[] {
        "ace", "2", "3", "4",
        "5", "6", "7", "8", "9", "10",
        "jack", "queen", "king"
};

List<String> deck = new ArrayList<String>();

for (int i = 0; i < suit.length; i++)
     for (int j = 0; j < rank.length; j++)
          deck.add(rank[j] + " of " + suit[i]);
```

Last but not least; shuffle & deal!

```
        // Shuffle the deck
        Collections.shuffle(deck);

        if (numHands * cardsPerHand > deck.size()) {
              System.out.println("Not enough cards.");
              return;
        }

        // use dealHand method…
        for (int i = 0; i < numHands; i++)
              System.out.println(dealHand(deck, cardsPerHand));

} // end of class Deal
```

Be careful when using subList(…)

- Semantics of returned List = undefined if elements are added to, or removed from, the backing List in any way other than via the returned List

- It is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently)

Be careful when using subList(…)

ADVICE = Use subList only as a transient object

- i.e. to perform one or a sequence of range operations on the backing List

- The longer you use it → the greater the probability to compromise it
  E.g. by modifying the backing List directly
  E.g. by modifying it through another subList object

Objects Ordering

…

How to sort a Collection?

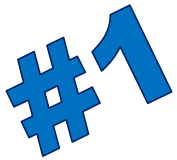
A List may be sorted as follows;

```
Collections.sort(myList);
```

Default orders that are used make sense;

List consists of Objects of class…	sorted into
String	alphabetical order
Date	chronological order
…	…

How to make a Collection Sortable?



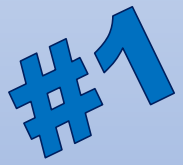
Implement the Comparable Interface

- We sort with → Collections.sort(list)
- Elements of which do not implement Comparable, will throw a ClassCastException

Provide a Comparator object

- We sort with → Collections.sort(list, comparator)
- Throws ClassCastException if elements cannot be compared to one another using the comparator

Objects Ordering



Implementing the Comparable Interface

Classes implementing the Comparable Interface

Class	Natural Ordering
Byte	Signed numerical
Character	Unsigned numerical
Long	Signed numerical
Integer	Signed numerical
Short	Signed numerical
Double	Signed numerical
Float	Signed numerical
BigInteger	Signed numerical
BigDecimal	Signed numerical
Boolean	Boolean.FALSE < Boolean.TRUE
File	System-dependent lexicographic on path name
String	Lexicographic
Date	Chronological
CollationKey	Locale-specific lexicographic

Writing your own Comparable Objects

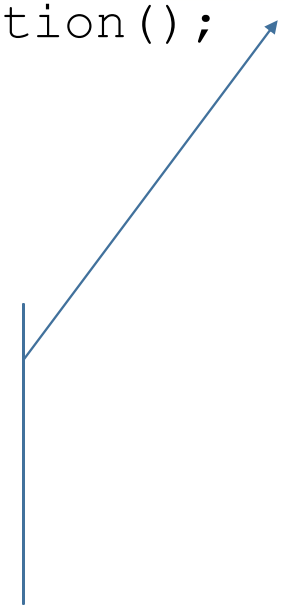
```
public class Name implements Comparable<Name> {

    private final String firstName, lastName;


    public String firstName() { return firstName; }
    public String lastName()  { return lastName;  }
```

- Name objects are immutable
- Recommended for objects that will be used as elements in Sets or as keys in Maps
- These collections will break if you modify their elements or keys while they're in the collection

Writing your own Comparable Objects

```
public Name(String firstName, String lastName) {
      if (firstName == null || lastName == null)
            throw new NullPointerException();
      this.firstName = firstName;
      this.lastName = lastName;
}
```

- The constructor checks its arguments for null
- ensures that all Name objects are well formed so that none of the other methods will ever throw a NullPointerException



DIY Comparables

```
public int hashCode() {
      return 31*firstName.hashCode() + lastName.hashCode();
}
```

- hashCode(…) redefined
- Essential for any class that redefines equals(…)
- Rules = equals(…) objects → equal hash codes

```
public String toString() {
      return firstName + " " + lastName;
}
```

- toString(…) redefined
- collections' toString(…) depend on toString(…) of their elements, keys, and values

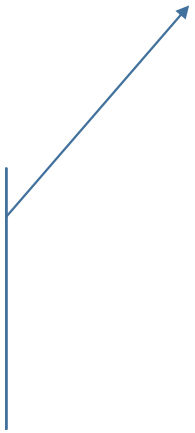
DIY Comparables

```
public boolean equals(Object o) {
      if (!(o instanceof Name))
            return false;
      Name n = (Name) o;
      return     n.firstName.equals(firstName)
                 && n.lastName.equals(lastName);
}
```

- returns false if the specified object is null or of an inappropriate type
- required by the general contracts of this method

DIY Comparables

```
public int compareTo(Name n) {
      int lastCmp = lastName.compareTo(n.lastName);
      return (lastCmp != 0
                  ? lastCmp
                  : firstName.compareTo(n.firstName));
}
```



- throws ClassCastException at runtime if specified object may not be compared to this
- required by the general contracts of this method

Let's use this class

```
public class NameSort {
    public static void main(String[] args) {
        Name nameArray[] = {
            new Name("John", "Smith"),
            new Name("Karl", "Ng"),
            new Name("Jeff", "Smith"),
            new Name("Tom", "Rich")
        };

        List<Name> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

```
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

Objects Ordering

Using Comparators

Comparing w/ Comparators

What if you want to sort objects…

- …in an order other than their natural ordering?
- …that don't implement Comparable?

Introducing… the Comparator

- It is an object that encapsulates an ordering
- Features single compare(…) method that returns <0 / 0 / >0

  IF either arguments has an inappropriate type
  THEN throws ClassCastException

Example

```
public class Employee implements Comparable<Employee> {
    public Name name()      { ... }
    public int number()     { ... }
    public Date hireDate()  { ... }
        ...
}
```

Example

```
public class EmpSort {
     static final Comparator<Employee> SENIORITY_ORDER
          = new Comparator<Employee>() {
            public int compare(Employee e1, Employee e2) {
                return e2.hireDate().compareTo(e1.hireDate());
            }
          };

    // Employee database
    static final Collection<Employee> employees = ... ;

    public static void main(String[] args) {
        List<Employee> e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

Problems with this comparator

Take another look at our implementation so far…

```
SENIORITY_ORDER = new Comparator<Employee>() {
     public int compare(Employee e1, Employee e2) {
          return e2.hireDate().compareTo(e1.hireDate());
     }
};
```

E1.equals(e2) returns true
does not mean that
compare(e1,e2) returns 0

Fixing our Example

```
static final Comparator<Employee> SENIORITY_ORDER
      = new Comparator<Employee>() {
          public int compare(Employee e1, Employee e2) {
           int dateCmp = e2.hireDate().compareTo(e1.hireDate());
           if (dateCmp != 0)
             return dateCmp;

           return (e1.number() < e2.number()
                     ? -1
                     :(e1.number() == e2.number()
                           ? 0
                           : 1));
      }
};
```

Algorithms

What are Java Framework Algorithms?

- Polymorphic algorithms = pieces of reusable functionality
- All = static methods from the Collections class whose 1st argument is the collection on which the operation is to be performed
- Generally operate on List instances, but a few of them operate on arbitrary Collection instances

Overview of List Algorithms

Name	Description
sort	sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
shuffle	randomly permutes the elements in a List
reverse	reverses the order of the elements in a List
rotate	rotates all the elements in a List by a specified distance
swap	swaps the elements at specified positions in a List

Overview of List Algorithms

Name	Description
replaceAll	replaces all occurrences of one specified value with another
fill	overwrites every element in a List with the specified value
copy	copies the source List into the destination List
binarySearch	searches for an element in an ordered List using the binary search algorithm; aka dichotomic search algorithm
indexOfSubList	returns the index of the first sublist (in first parameter List) that is equal to the second parameter List
lastIndexOfSubList	as above but returns the index of the last sublist

Overview of Collection-Level Algorithms

Name	Description
addAll	• adds all the specified elements to a Collection • elements to be added may be specified individually or as an array
frequency	counts the number of times the specified element occurs in the specified collection
disjoint	determines whether two Collections are disjoint; i.e., whether they contain no elements in common
min	Self-explanatory
max	Self-explanatory

Let's take a closer look at Collections.sort(…)

sort(…) uses a slightly optimized merge sort algorithm;

Fast

- n log(n) time, substantially faster on nearly sorted lists
- Empirically shown to be as fast as a highly optimized quicksort
- quicksort is generally faster but isn't stable and doesn't guarantee n log(n) performance

Stable

- It doesn't reorder equal elements
- Important if you sort the same list repeatedly on different attributes

Example – Sorting Based on Comparable

We rely on String elements being Comparable

```
public class Sort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

```
java Sort i walk the line

[i, line, the, walk]
```

Example – Sorting based on Comparator

```
// Make a List of all anagram groups above size threshold
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);

// Sort anagram groups according to size
Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
       return o2.size() - o1.size();
    }}
);
```

See details in map tutorial
For now, focus on comparator only