List of Slides

1 Title
2 **Chapter 21:** Collections
3 Chapter aims
4 **Section 2:** Example:Reversing a text file
5 Aim
6 Reversing a text file
7 Collections API
8 Collections API: Lists
9 Collections API: Lists: `List` interface
12 Collections API: Lists: `ArrayList`
14 Reversing a text file
18 Reversing a text file
19 Trying it
20 Coursework: Sorting election leaflets
21 **Section 3:** Example:Sorting a text file using an `ArrayList`
22 Aim

23 Sorting a text file using an `ArrayList`
24 The `SortList` class?
26 The `SortList` class?
27 Collections API: `Collections` class
30 The `SortList` class?
31 The `Sort` class
35 Coursework: Sorting election leaflets, with `compareTo()`
36 **Section 4:** Example:Prime numbers
37 Aim
38 Prime numbers
39 Collections API: Sets
40 Collections API: Sets: `Set` interface
43 Prime numbers
44 Design: Storing data
45 Design: Storing data: hash table
48 Standard API: `Object`: `hashCode()`
50 Collections API: Sets: `HashSet`
52 Standard API: `Integer`: as a box for `int`: works with collections

53 Prime numbers
55 Prime numbers
56 Prime numbers
57 Trying it
58 Trying it
59 Trying it
60 Trying it
61 Coursework: Finding duplicate voters
62 **Section 5:** Example:Sorting a text file using a `TreeSet`
63 Aim
64 Sorting a text file using a `TreeSet`
65 Design: Storing data: ordered binary tree
68 Collections API: Sets: `TreeSet`
69 Sorting a text file using a `TreeSet`
70 Collections API: `Iterator` interface
73 Collections API: Lists: `List` interface: `iterator()`
75 Sorting a text file using a `TreeSet`
76 Collections API: Sets: `Set` interface: `iterator()`

77 Collections API: Sets: `TreeSet: iterator()`
79 Design: Sorting a list: tree sort
80 Sorting a text file using a `TreeSet`
84 Trying it
85 Coursework: Sorting election leaflets, using a `TreeSet`
86 **Section 6:** Summary of lists and sets
87 Aim
88 Summary of lists and sets
89 Collections API: `Collection` interface
95 Collections API: Lists: `List` interface: extends `Collection`
96 Collections API: Sets: `Set` interface: extends `Collection`
97 Summary of lists and sets
98 Collections API: Lists: `add(index)` and `remove(index)`
100 Design: Storing data: linked list
102 Collections API: Lists: `LinkedList`
103 Summary of lists and sets
104 Summary of lists and sets
105 Summary of lists and sets

106 **Section 7:** Example:Word frequency count
107 Aim
108 Word frequency count
109 Collections API: Maps
111 Word frequency count
112 The `WordWithFrequency` class
113 The `WordWithFrequency` class
114 Collections API: Maps: `Map` interface
118 Collections API: Maps: `TreeMap`
120 The `WordFrequencyMap` class
121 The `WordFrequencyMap` class
122 The `WordFrequencyMap` class
123 Statement: for-each loop: on collections
125 The `WordFrequencyMap` class
126 The `WordFrequencyMap` class
127 The `WordFrequency` class
131 The `WordFrequency` class
132 Trying it

133 **Section 8:** Example:Word frequency count sorted by frequency
134 Aim
135 The `WordWithFrequency` class
137 The `WordWithFrequency` class
138 The `WordWithFrequency` class
139 Standard API: `Object`: `hashCode()`: making a good definition
140 The `WordWithFrequency` class
141 The `WordWithFrequency` class
142 The `WordFrequencyMap` class
143 Collections API: Maps: `HashMap`
145 The `WordFrequencyMap` class
147 The `WordFrequencyMap` class
148 Collections API: `Collection` interface: constructor taking a `Collection`
149 The `WordFrequencyMap` class
150 The `WordFrequency` class
151 Trying it
152 Coursework: Finding duplicate voters, using a `HashMap`
153 **Section 9:** Collections of collections

154 Aim
155 Collections of collections
156 Coursework: Finding duplicate voters, using a `HashMap` of `LinkedList`s
157 Concepts covered in this chapter

Title

Java Just in Time

John Latham

March 4, 2019

Chapter 21

Collections

Chapter aims

- Need to handle **collection**s of **object**s quite common
  - previously seen **array** used to store things in **index**ed **list**.
- Here explore Java's **collections framework**
  - group of **class**es and **interface**s
  - provide various mechanisms for storing collections
  - more convenient to use than arrays.
- E.g. `ArrayList`
  - essentially wrapped up array with automatic **array extension**.
- We look at `List`s, `Set`s and `Map`s
  - specified as **interface**s
  - **implement**ed by `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, `TreeMap` and `HashMap`

Section 2

Example:
Reversing a text file

Aim

AIM: To introduce the Java **collections framework**, and in particular the idea of **list collection**s, the `List` **interface** and the `ArrayList` **class**.

Reversing a text file

- Program reads lines of **text file**
  - outputs in reverse order to second text file.
- E.g. file of examination results in ascending order of merit
  - want them in descending order.
- Read **data** line by line
  - store it all
  - output lines in reverse order.
- Could use **array**
  - but don't know in advance how many lines
  - use `ArrayList` rather than **array extension**.

Collections API

- Common need to store **collection**s of **data**
  - Java **API** provides **collections framework**.
- Group of **class**es and **interface**s designed to store collections
  - in various different ways.
- Typically allow elements to be added without worrying about memory allocation
  - automatically grow big enough.

Collections API: Lists

- One kind of **collection** in **collections framework**
  - **list collection**
- Collections of **data** which are **list**s or sequences.
  - Duplicate elements are permitted
  - elements stored in some order
  - each occurs at particular **list index**, starting at zero.
- Lists similar to **array**s.

Collections API: Lists: `List` interface

- The **interface** `java.util.List` part of **collections framework**
  - specifies **instance methods** needed to support **list collection**.

Method definitions in interface `List` (some of them).

Method	Return	Arguments	Description
`size`	**`int`**		Returns the size of this `List`, that is, the number of elements in it.
`add`	**`boolean`**	`Object`	Appends the given `Object` to the end of the `List`. Returns **`true`**.

Collections API: Lists: `List` interface

Method definitions in interface `List` (some of them).

Method	Return	Arguments	Description
`get`	`Object`	**`int`**	Returns the `Object` at the specified **list index**, which must be legal (`0 <= index < size()`) to avoid an `IndexOutOfBoundsException`.
`set`	`Object`	**`int`**, `Object`	Overwrites an existing element with a new one: i.e. it replaces the `Object` at the given **`int`** list index with the given other `Object`. Returns the original `Object`. The index must be legal to avoid an `IndexOutOfBoundsException`.

Collections API: Lists: `List` interface

- Since Java 5.0, `List` is **generic interface**
  - one **type parameter** – **type** of **object**s that can be stored.
- When use **parameterized type** of `List`
  - all occurrences of `Object` in above table replaced by **type argument**.

Collections API: Lists: `ArrayList`

- The **class** `java.util.ArrayList` is part of **collections framework**
  - one **implement**ation of **list collection**
  - **implement**s `java.util.List` **interface**.
- Kind of **list** implemented using **private instance variable**
  - **array** of **type** `java.lang.Object[]`.
  - Array grown automatically
    - by **array extension**.

Collections API: Lists: `ArrayList`

- Since Java 5.0, `ArrayList`, and other classes in collections framework are **generic classes**

```
public class ArrayList<E> implements List<E>
{ ... }
```

- The **type parameter** is **type** of **objects** in list.

Reversing a text file

```
001: import java.io.BufferedReader;
002: import java.io.FileReader;
003: import java.io.FileWriter;
004: import java.io.IOException;
005: import java.io.PrintWriter;
006: import java.util.ArrayList;
007: import java.util.List;
008:
009: // Program to read lines of a file, line by line, and write them in reverse
010: // order to another. Input file is the first argument, output is the second.
011: public class Reverse
012: {
```

Reversing a text file

```
013:   public static void main(String[] args)
014:   {
015:     BufferedReader input = null;
016:     PrintWriter output = null;
017:     try
018:     {
019:       if (args.length != 2)
020:         throw new IllegalArgumentException
021:                     ("There must be exactly two arguments: infile outfile");
022:
023:       input = new BufferedReader(new FileReader(args[0]));
024:       output = new PrintWriter(new FileWriter(args[1]));
025:
026:       // The List for storing the lines.
027:       List<String> lineList = new ArrayList<String>();
028:
```

Reversing a text file

```
029:      // Read the lines into lineList.
030:      String currentLine;
031:      while ((currentLine = input.readLine()) != null)
032:        lineList.add(currentLine);
033:
034:      // Now output them in reverse.
035:      for (int index = lineList.size() - 1; index >= 0; index--)
036:        output.println(lineList.get(index));
037:    } // try
038:    catch (Exception exception)
039:    {
040:      System.err.println(exception);
041:    } // catch
```

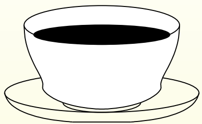
Reversing a text file

```
042:       finally
043:       {
044:         try { if (input != null) input.close(); }
045:         catch (IOException exception)
046:           { System.err.println("Could not close input " + exception); }
047:         if (output != null)
048:         {
049:           output.close();
050:           if (output.checkError())
051:             System.err.println("Something went wrong with the output");
052:         } // if
053:       } // finally
054:     } // main
055:
056: } // class Reverse
```

THE UNIVERSITY of MANCHESTER

Reversing a text file

- Note **type** of `lineList` **variable**
  - **interface** is a **type**
  - also any kind of `List` would work.



Coffee time: Why does the `add()` **instance method** of `List` always **return** `true`? You can find the answer to this by looking at the **API** on-line documentation for `List` and observe that it **extends** the `Collection` **interface**.

Trying it

Console Input / Output

```
$ cat input.txt
Bear,Rupert     13.7%
Smith,James     51.5%
Brown,Margaret  68.2%
Jones,Stephen   87.9%
Jackson,Helen   100%
$ java Reverse input.txt output.txt
$ cat output.txt
Jackson,Helen   100%
Jones,Stephen   87.9%
Brown,Margaret  68.2%
Smith,James     51.5%
Bear,Rupert     13.7%
$ java Reverse /dev/null output.txt
$ ls -l output.txt
-rw-------  1 jtl jtl 0 Jul 01 19:12 output.txt
$ _
```

Run

Coursework: Sorting election leaflets

(Summary only)

Write a program to **sort** election information leaflets into delivery order.

Section 3

Example:
Sorting a text file using an `ArrayList`

Aim

AIM: To reinforce the use of `ArrayList`, in particular, showing uses of the `set()` **instance method** of a `List`. We also note that an **array** can be created from a `List`, and vice versa. Finally, we look at the `Collections` **class** and observe that it has a `sort()` **generic method**.

Sorting a text file using an `ArrayList`

- Revisit program to **sort** lines of **text file**
  - previously used **array** with **array extension**
  - here use `ArrayList`.
- Could develop separate **class** to sort any `List` of `Comparable` items
  - as we nearly did for `Comparable` (`Sortable`) array. . .

The `SortList` class?

```
001: import java.util.List;
002:
003: // Provides a class method for sorting a List of any Comparable objects.
004: public class SortList
005: {
006:   public static <ListType extends Comparable<ListType>>
007:          void sort(List<ListType> list)
008:   {
009:     // Each pass of the sort reduces unsortedLength by one.
010:     int unsortedLength = list.size();
011:     // If no change is made on a pass, the main loop can stop.
012:     boolean changedOnThisPass;
013:     do
014:     {
015:       changedOnThisPass = false;
```

The SortList class?

```
016:         for (int pairLeftIndex = 0;
017:              pairLeftIndex < unsortedLength - 1; pairLeftIndex++)
018:         {
019:           if (list.get(pairLeftIndex).compareTo(list.get(pairLeftIndex + 1)) > 0)
020:           {
021:             ListType thatWasAtPairLeftIndex = list.get(pairLeftIndex);
022:             list.set(pairLeftIndex, list.get(pairLeftIndex + 1));
023:             list.set(pairLeftIndex + 1, thatWasAtPairLeftIndex);
024:             changedOnThisPass = true;
025:           } // if
026:         } // for
027:         unsortedLength--;
028:       } while (changedOnThisPass);
029:     } // sort
030:
031: } // class SortList
```

The `SortList` class?

- Could do above, or another way:
  - turn `List` into **array**
  - sort with `Arrays.sort()`
  - turn back into `List`.
  - Java **API** contains **method**s for such conversions.
- But don't even need do that! ...

Collections API: `Collections` class

- `java.util.Collections` provides **class methods** to perform manipulations of **collection**s.
- One called `sort`
  - takes `List` of `Object`s
  - **sort**s into **natural ordering**.
  - Items in `List` must all be **type** `java.lang.Comparable`
  - and be **mutually comparable**.
  - or **exception throw**n.
- Uses **merge sort**
  - far more efficient than **bubble sort** (but less simple).

Collections API: `Collections` class

- At Java 5.0 many methods in `Collections` became **generic methods**.
- `Collections.sort()` has single **type parameter**
  - **type** of items in given `List`
  - must be `Comparable` with themselves.
- Would expect heading:

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

Collections API: `Collections` class

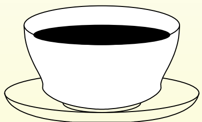
- In fact heading is:

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

- `<? super T>` means
  - "any type that is `T` or a **superclass** (or **superinterface**) of it".
- I.e. any **type argument** must **implement** `Comparable` with itself
  - or superclass of itself.
- Many **type parameter**s in **API** expressed like that
  - leads to more flexibility and convenience.

The SortList class?

Coffee time: *Warning – this one is subtle stuff.* If a class, `A`, implements `Comparable<A>`, and a class `B` **extends** `A`, then `B` also implements `Comparable<A>`. But, does it **implement** `Comparable<B>` as well? You might think it does, because any `B` can be compared with any other `B`, via the **instance method** defined in `A`. However, perhaps surprisingly, Java regards that it does not: `Comparable<B>` is not 'implied' from `Comparable<A>` in the same way that `int compareTo(B other)` would not **override** `int compareTo(A other)` – it would be an **overloaded method** instead. This has surprising implications. The code `<T extends Comparable<? super T>>` gets around this problem.



The Sort class

```
001: import java.io.BufferedReader;
002: import java.io.FileReader;
003: import java.io.FileWriter;
004: import java.io.IOException;
005: import java.io.PrintWriter;
006: import java.util.ArrayList;
007: import java.util.Collections;
008: import java.util.List;
009:
010: // Program to sort lines of a file, line by line, and write to another.
011: // Input file is the first argument, output is the second.
012: public class Sort
013: {
```

The Sort class

```
014:     public static void main(String[] args)
015:     {
016:       BufferedReader input = null;
017:       PrintWriter output = null;
018:       try
019:       {
020:         if (args.length != 2)
021:           throw new IllegalArgumentException
022:                       ("There must be exactly two arguments: infile outfile");
023:
024:         input = new BufferedReader(new FileReader(args[0]));
025:         output = new PrintWriter(new FileWriter(args[1]));
026:
027:         // The List for storing the lines.
028:         List<String> lineList = new ArrayList<String>();
029:
```

The `Sort` class

```
030:         // Read the lines into lineList.
031:         String currentLine;
032:         while ((currentLine = input.readLine()) != null)
033:           lineList.add(currentLine);
034:
035:         // Sort lineList.
036:         Collections.sort(lineList);
037:
038:         // Now output them.
039:         for (int index = 0; index < lineList.size(); index++)
040:           output.println(lineList.get(index));
041:       } // try
042:       catch (Exception exception)
043:       {
044:         System.err.println(exception);
045:       } // catch
```

The Sort class

```
046:      finally
047:      {
048:        try { if (input != null) input.close(); }
049:        catch (IOException exception)
050:          { System.err.println("Could not close input " + exception); }
051:        if (output != null)
052:        {
053:          output.close();
054:          if (output.checkError())
055:            System.err.println("Something went wrong with the output");
056:        } // if
057:      } // finally
058:    } // main
059:
060: } // class Sort
```

Coursework: Sorting election leaflets, with `compareTo()`

(Summary only)

Write a program to **sort** election information leaflets into delivery order, using a `compareTo()` **instance method**.

Section 4

Example: Prime numbers

Aim

AIM: To introduce the idea of **set collections**, the `Set` **interface** and the `HashSet` **class**. For this we explore **hash tables** and meet `hashCode()` from `Object`. We also see that the class `Integer` **implements** `Comparable<Integer>`.

Prime numbers

- A **prime number** is positive **integer** which can be divided without remainder by only itself and one.
  - their pursuit and understanding has been holy grail for many mathematicians.
- Program outputs all prime numbers **less than or equal** to given **command line argument**.
- Simple and fast approach
  - maintain **set** of all multiples of prime numbers found so far.
  - Consider all numbers from two up to given maximum.
  - If number is not multiple of prime number previously found
    - print it
    - add all multiples, up to maximum, to set.
- Based on Sieve of Eratosthenes.

Collections API: Sets

- Another kind of **collection** in **collections framework**
  - **set collection**.
- Collections of **data** which are **set**s
  - adding element already present has no effect
  - order added *not* preserved.
- To determine if `Object`s are equivalent
  - uses `equals()` **instance method** of elements.

THE UNIVERSITY of MANCHESTER

Collections API: Sets: `Set` interface

- The **interface** `java.util.Set` part of **collections framework**
  - specifies **instance method**s needed to support **set collection**.
- Including. . .

Collections API: Sets: `Set` interface

Method definitions in interface `Set` (some of them).			
Method	**Return**	**Arguments**	**Description**
`size`	**`int`**		Returns the size of this `Set`, that is, the number of elements in it.
`add`	**`boolean`**	`Object`	Inserts the given `Object` into the `Set`, unless an **equivalent** one is already present. Returns **`true`** if it gets added, **`false`** otherwise.
`contains`	**`boolean`**	`Object`	Return **`true`** if the `Set` contains an `Object` which is equivalent to the given one, **`false`** otherwise.

Collections API: Sets: `Set` interface

- Since Java 5.0 `Set` is **generic interface**
  - **type parameter** is **type** of **object**s that can be stored.
- When use **parameterized type** of `Set` rather than **raw type**
  - all the occurrences of `Object` in above table replaced by **type argument**.

Prime numbers

- We use `HashSet`
  - implementation based on **hash table**.

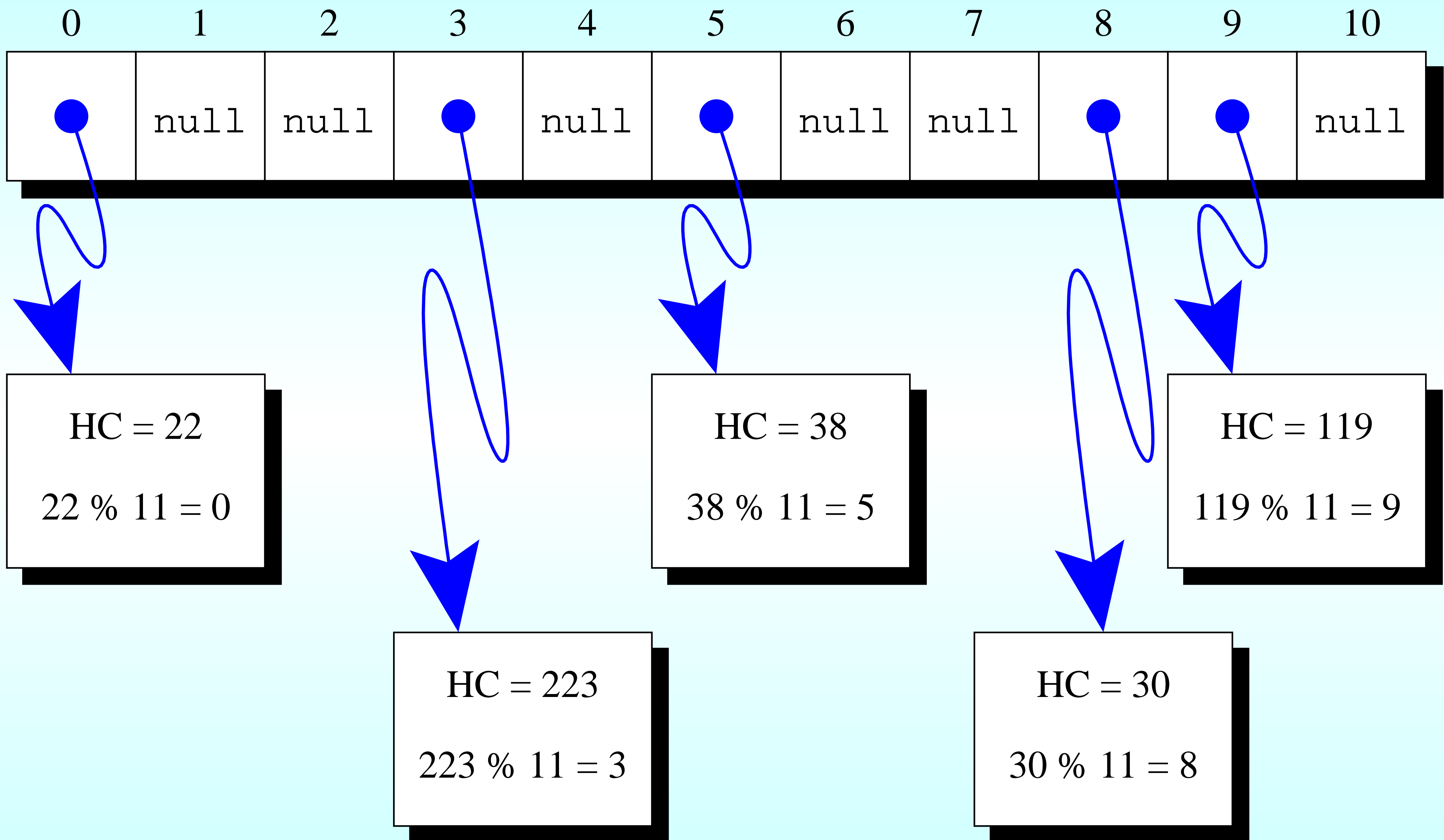
Design: Storing data

- Collections of **data** need stored in **computer memory** at **run time**
  - placed in **data structure**.
- E.g. obvious example: **array**.
- Common requirement to find data using some kind of search **algorithm**
  - e.g. **linear search**
  - **binary search**
- May need data sorted in particular order
  - **sort algorithm**
    - e.g. **bubble sort**.

Design: Storing data: hash table

- A **hash table** is **data structure**
  - stores **data** so can be retrieved quickly.
- Uses **array**
  - **array index** based on **hash code** provided by each item.
- Data items which are **equivalent** *must* have same hash code
  - and if not equivalent try to have different hash codes.
- To insert item
  - take hash code
  - divide by size of array, take remainder
  - place item at that array index.
- To find item, compute array index and check array.

Design: Storing data: hash table



Design: Storing data: hash table

- May get clashes
  - two items not equivalent but same array index
  - various strategies for coping
    * e.g. find next available free slot – leads to partial **linear search**.
- For best efficiency must minimize clash occurrence
  - make size of array **prime number**
  - **design** hash **function**
    so tend to get different hash codes for non-equivalent items.

Standard API: `Object: hashCode()`

- Every object has **instance method** `hashCode`
  - defined in `java.lang.Object`
  - designed to help classes that use **hash table**
    * e.g. `java.util.HashSet`.
- Definition in `Object` gives distinct **objects** distinct **hash code**
  - (usually) based on memory address of **reference**.
- Classes that **override** `equals()`
  - should also override `hashCode()`
    * so **equivalent** objects get same hash code
    * but non-equivalent tend to have different one.
- So will work properly if need to be used as elements of
  - `HashSet`, etc..

Standard API: `Object`: `hashCode()`

```
MyClass v1 = new MyClass(...);
MyClass v2 = new MyClass(...);

if (v1.equals(v2) && v1.hashCode() != v2.hashCode())
  System.out.println("Your hash tables will not work!");
else if (! v1.equals(v2) && v1.hashCode() == v2.hashCode())
  System.out.println("Your hash tables may operate slowly.");
```

Collections API: Sets: `HashSet`

- `java.util.HashSet` part of **collections framework**
  - one implementation of **set collection**.
- It **implement**s `java.util.Set` **interface**.
- Uses **hash table**
  - **hash code**s obtained from `hashCode()` **instance method** of elements.
- To work, any **object**s which are **equivalent**
  - *must* have same hash code
    - otherwise multiple copies of equivalent items will be allowed!
- To be efficient non-equivalent objects
  should tend to have different hash codes.

Collections API: Sets: `HashSet`

- Since Java 5.0 `HashSet` is **generic class**
  - **type parameter** is **type** of **object**s that can be stored.

```
public class HashSet<E> implements Set<E>
{ ... }
```

- `java.lang.Integer` **implement**s `java.lang.Comparable<Integer>`
  - provides `compareTo()`
  - **override**s `equals()`
  - `hashCode()`

  so that `Integer` **object**s behave properly as
  - `Comparable`s
  - in **hash table**s, etc..

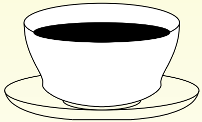
Prime numbers

```
001: import java.util.HashSet;
002: import java.util.Set;
003:
004: // List all the prime numbers less than or equal to the command line argument.
005: // (Warning: this program does not catch RuntimeExceptions.)
006: public class Primes
007: {
008:   public static void main(String[] args)
009:   {
010:     // The maximum number we need to consider.
011:     int maxPossiblePrime = Integer.parseInt(args[0]);
012:
013:     // The set of all multiples of prime numbers found so far.
014:     // These are therefore not prime numbers.
015:     Set<Integer> multiplesOfPrimesFound = new HashSet<Integer>();
016:
```

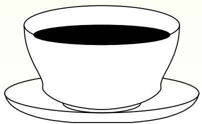
Prime numbers

```
017:     // Consider every number from 2 up to maximum,
018:     // it is a possible prime, output and count it if it is.
019:     int noOfPrimesFoundSoFar = 0;
020:     for (int possiblePrimeNumber = 2;
021:          possiblePrimeNumber <= maxPossiblePrime; possiblePrimeNumber++)
022:       if (! multiplesOfPrimesFound.contains(possiblePrimeNumber))
023:       {
024:         // possiblePrimeNumber really is a prime number.
025:         noOfPrimesFoundSoFar++;
026:
   System.out.println(noOfPrimesFoundSoFar + " : " + possiblePrimeNumber);
027:         // Now add multiples of possiblePrimeNumber to multiplesOfPrimesFound.
028:         for (int primeMultiple = possiblePrimeNumber * 2;
029:              primeMultiple <= maxPossiblePrime;
030:              primeMultiple += possiblePrimeNumber)
031:           multiplesOfPrimesFound.add(primeMultiple);
032:       } // if
033:   } // main
034:
035: } // class Primes
```

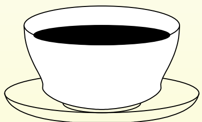
Prime numbers



Coffee time: Did you notice the *two* places where **autoboxing** wraps an **`int`** inside an `Integer`?

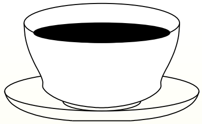


Coffee time: What do you imagine is the **hash code** for an `Integer` object containing the number n?



Coffee time: Suppose the implementers of the `Integer` **class** had forgotten to **override** `hashCode()`, so that every `Integer` object had a *unique* hash code. What would be the effect of our `Primes` program?

Prime numbers



Coffee time: Find all the places where we previously wrote an `equals()` **instance method** and devise a suitable `hashCode()` instance method to go with each one.

Trying it

Console Input / Output

```
$ java Primes 1000
 (Output shown using multiple columns to save space.)
1 : 2
2 : 3
3 : 5
4 : 7
5 : 11
6 : 13
7 : 17
8 : 19
9 : 23
10 : 29
11 : 31
12 : 37
13 : 41
14 : 43
15 : 47
16 : 53
17 : 59
18 : 61
19 : 67
20 : 71
21 : 73
22 : 79
23 : 83
24 : 89
25 : 97
26 : 101
27 : 103
28 : 107
29 : 109
30 : 113
31 : 127
32 : 131
33 : 137
34 : 139
35 : 149
36 : 151
37 : 157
38 : 163
39 : 167
40 : 173
41 : 179
42 : 181
43 : 191
44 : 193
45 : 197
46 : 199
47 : 211
48 : 223
49 : 227
50 : 229
51 : 233
52 : 239
53 : 241
54 : 251
55 : 257
56 : 263
57 : 269
58 : 271
59 : 277
60 : 281
61 : 283
62 : 293
63 : 307
64 : 311
65 : 313
66 : 317
67 : 331
68 : 337
69 : 347
70 : 349
71 : 353
72 : 359
73 : 367
74 : 373
75 : 379
76 : 383
77 : 389
78 : 397
79 : 401
80 : 409
81 : 419
82 : 421
83 : 431
84 : 433
85 : 439
86 : 443
87 : 449
88 : 457
89 : 461
90 : 463
91 : 467
92 : 479
93 : 487
94 : 491
95 : 499
96 : 503
97 : 509
98 : 521
99 : 523
100 : 541
101 : 547
102 : 557
103 : 563
104 : 569
105 : 571
106 : 577
107 : 587
108 : 593
109 : 599
110 : 601
111 : 607
112 : 613
113 : 617
114 : 619
115 : 631
116 : 641
117 : 643
118 : 647
119 : 653
120 : 659
121 : 661
122 : 673
123 : 677
124 : 683
125 : 691
126 : 701
127 : 709
128 : 719
129 : 727
130 : 733
131 : 739
132 : 743
133 : 751
134 : 757
135 : 761
136 : 769
137 : 773
138 : 787
139 : 797
140 : 809
141 : 811
142 : 821
143 : 823
144 : 827
145 : 829
146 : 839
147 : 853
148 : 857
149 : 859
150 : 863
151 : 877
152 : 881
153 : 883
154 : 887
155 : 907
156 : 911
157 : 919
158 : 929
159 : 937
160 : 941
161 : 947
162 : 953
163 : 967
164 : 971
165 : 977
166 : 983
167 : 991
168 : 997
$ _
```

Run

Trying it

- How fast?

Console Input / Output

```
$ time java Primes 1000000 > primes.txt

real    0m5.175s
user    0m3.800s
sys     0m1.217s
$ cat primes.txt
1 : 2
2 : 3
 (... lines removed to save space.)
78496 : 999961
78497 : 999979
78498 : 999983
$ _
```

Run

Trying it

- But does need lot of space. . . .

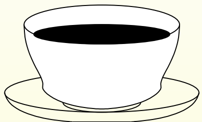
Console Input / Output

```
$ time java Primes 10000000 > primes.txt
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.HashMap.addEntry(HashMap.java:753)
        at java.util.HashMap.put(HashMap.java:385)
        at java.util.HashSet.add(HashSet.java:200)
        at Primes.main(Primes.java:31)

real    0m59.125s
user    0m55.945s
sys     0m1.110s
$ cat primes.txt
1 : 2
2 : 3
$ _
```

Run

Trying it



Coffee time: The `Primes` program has been a suitable introduction to the use of **set collection**s, but actually, there may be a better way to implement the same algorithm. Consider this: the **set** contains all the non-prime numbers up to the maximum, and as the maximum gets bigger, the difference between this set and *all* the numbers up to the maximum, gets proportionally smaller. With this in mind, what even simpler way could we use to implement the set of non-primes?

Coursework: Finding duplicate voters

(Summary only)

Write a program to detect people voting more than once in voting records.

Section 5

Example:
Sorting a text file using a `TreeSet`

Aim

AIM: To introduce the `TreeSet` **class**, for which we explore **ordered binary trees** and **tree sort**. We also meet the `Iterator` **interface**, together with how it is used on a `List` and a `Set`, especially a `TreeSet`.

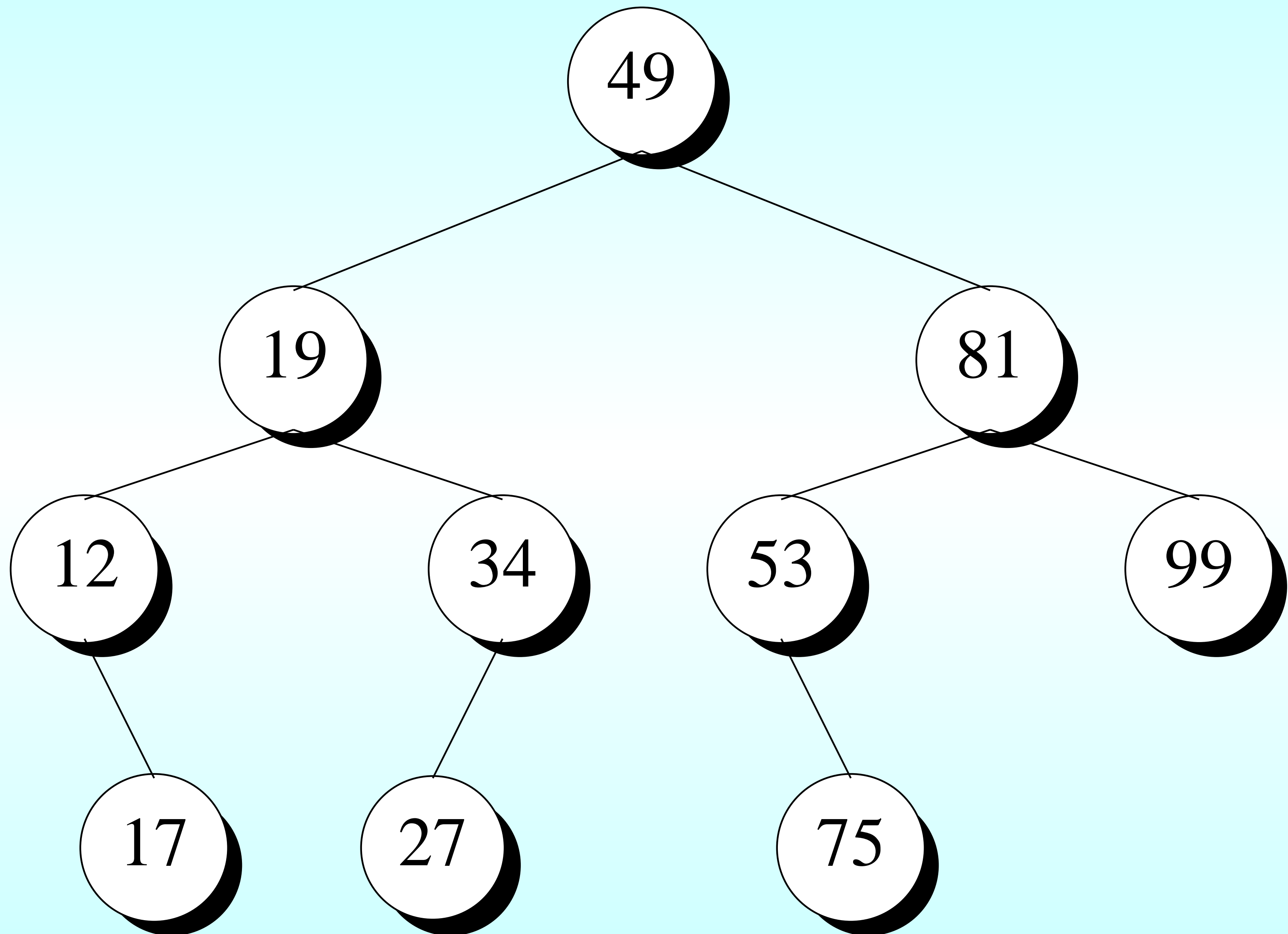
Sorting a text file using a `TreeSet`

- Seen two ways of **sorting text file**
  - using **array**
  - using `ArrayList`
- Here use `TreeSet`
  - causes interesting twist:
    multiple copies of line in input produce only one copy in output.

Design: Storing data: ordered binary tree

- An **ordered binary tree** (**OBT**)
  - **data structure** for quick storage / retrieval of **data**.
- Data stored in tree
  - each branch having possible left subtree
  - and/or right subtree (binary)
  - data kept in some **total order** from left to right across tree.
    * For every item in tree
      all items in left subtree are **less than**
      all items in right subtree are **greater than**.

Design: Storing data: ordered binary tree



Design: Storing data: ordered binary tree

- Do not have to search entire tree to find item
  - start at top
  - if not yet found
    * go left if search item less than item here
    * else right.
- OBT searching similar efficiency to **binary search**
  - (essentially) halve search space each stage as proceed down tree.
- Not as fast as **hash table** with *good* (and quick) **hash code function**
  - but OBT useful when wish to retrieve data in order.

Collections API: Sets: `TreeSet`

- `java.util.TreeSet` part of **collections framework**
  - another implementation of **set collection**
  - **implement**s `java.util.Set` **interface**.
- Uses **ordered binary tree**
  - has to be possible to order elements stored in it.
  - Simplest way: ensure **class** of elements implements `java.lang.Comparable`.
- Since Java 5.0, `TreeSet` is **generic class**
  - **type parameter** is **type** of **object**s that can be stored.

```
public class TreeSet<E> implements Set<E>
{ ... }
```

Sorting a text file using a `TreeSet`

- Program will
  - insert lines into `TreeSet`
  - use `Iterator` to access in order.

Collections API: `Iterator` interface

- The **interface** `java.util.Iterator` part of **collections framework**
  - specifies **instance methods** for
    - accessing elements in **collection**
    - one by one.

Method definitions in interface `Iterator` (some of them).

Method	Return	Arguments	Description
`hasNext`	**`boolean`**		Returns **`true`** if the **iteration** has more elements, **`false`** otherwise.
`next`	`Object`		Returns the next element in the iteration, and moves the iteration on to the element following that one.

Collections API: `Iterator` interface

- When **new** `Iterator` **object** obtained from collection
  - `hasNext()` will **return** **`true`**, unless collection is empty.
- First call to `next()` gets first element from iteration if is one
  - second call gets second, and so on.
- Sooner or later `hasNext()` will return **`false`**
  - because `next()` been called as many times as are elements.
- Typically use `hasNext()` to control **loop**
  - `next()` inside loop.

Collections API: `Iterator` interface

- All **list collection**s and **set collection**s have instance method `iterator()`
  - returns **object**, **instance** of some **class** that **implement**s `Iterator`.
- Supports iteration through elements of collection
  - order depends on kind of collection.
- Since Java 5.0, `Iterator` is **generic interface**
  - **type parameter** is **type** of objects stored in collection.
- I.e. if collection was given **type argument**
  - `next()` returns object of that type.

Collections API: Lists: `List` interface: `iterator()`

- The **instance method** `iterator()` specified in **interface** `java.util.List`
  - **return**s **object** that **implement**s `java.util.Iterator`
    - supports **iteration** of elements in ascending order of **list index**.
- E.g. print elements of `List`:

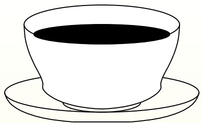
```
public static <ListType> void printList(List<ListType> list)
{
  Iterator<ListType> iterator = list.iterator();
  while (iterator.hasNext())
  {
    ListType item = iterator.next();
    System.out.println(item);
  } // while
} // printList
```

Collections API: Lists: `List` interface: `iterator()`

- For `ArrayList` this way of scanning just as efficient as using list index of each element.
- For some kinds of `List`s accessing by index not efficient
  - but scanning using `Iterator` always will be
    - because designed for that purpose.
- Rule of thumb:
  - whenever need to scan through elements of **list** in
    - arbitrary order
    - or from first to last

    use `Iterator` rather than indices.

Sorting a text file using a `TreeSet`



Coffee time: Identify all the places in this chapter before this point, where we used indices to scan through the elements of a `List`, and devise the changes needed to make them use an `Iterator` instead.

Collections API: Sets: `Set` interface: `iterator()`

- The **instance method** `iterator()` specified in **interface** `java.util.Set`
  - **return**s **object** that **implement**s `java.util.Iterator`
  - supports **iteration** of elements:
    - order depends on kind of **set**
    - may be arbitrary order.

Collections API: Sets: `TreeSet: iterator()`

- The `iterator()` **instance method** of `java.util.TreeSet`
  - **returns object** that **implements** `java.util.Iterator`
    * supports **iteration** of elements in order they appear in tree, from left to right.
- With simplest use of `TreeSet`
  - get **natural ordering** of elements.

Collections API: Sets: `TreeSet: iterator()`

- Rule of thumb: `java.util.HashSet` should be used in preference to `TreeSet`
  - when not desired to obtain values from **set collection** in specific order.
- If little or no **hash code** clashing
  - `HashSet` operates in nearly constant time per addition and membership test
    * `TreeSet` operates in time proportional to logarithm of size of **set**.

Design: Sorting a list: tree sort

- Another **algorithm** for **sort**ing – **tree sort**
  - items from **list** inserted into **ordered binary tree**
  - tree scanned from left to right.
- If **data** to be sorted has no duplicates
  (or desired to exclude multiple elements in result)
  - tree sort can be achieved in Java using **instance** of `java.util.TreeSet`
    - `iterator()` produces `Iterator` giving access to elements
      in order from smallest to largest.
- Duplicate items removed because **set** has no duplicates.

Sorting a text file using a TreeSet

```
001: import java.io.BufferedReader;
002: import java.io.FileReader;
003: import java.io.FileWriter;
004: import java.io.IOException;
005: import java.io.PrintWriter;
006: import java.util.Iterator;
007: import java.util.TreeSet;
008:
009: // Program to sort lines of a file, line by line, and write to another.
010: // Input file is the first argument, output is the second.
011: // Duplicate lines are removed.
012: public class Sort
013: {
```

Sorting a text file using a `TreeSet`

```
014:   public static void main(String[] args)
015:   {
016:     BufferedReader input = null;
017:     PrintWriter output = null;
018:     try
019:     {
020:       if (args.length != 2)
021:         throw new IllegalArgumentException
022:                     ("There must be exactly two arguments: infile outfile");
023:
024:       input = new BufferedReader(new FileReader(args[0]));
025:       output = new PrintWriter(new FileWriter(args[1]));
026:
027:       // The Set for storing the lines: TreeSet so it has an ordered Iterator.
028:       TreeSet<String> lineSet = new TreeSet<String>();
029:
```

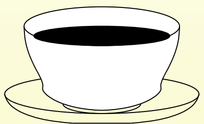
THE UNIVERSITY of MANCHESTER

Sorting a text file using a `TreeSet`

```
030:       // Read the lines into lineSet.
031:       String currentLine;
032:       while ((currentLine = input.readLine()) != null)
033:         lineSet.add(currentLine);
034:
035:       // Now output them in natural ordering.
036:       Iterator<String> iterator = lineSet.iterator();
037:       while (iterator.hasNext())
038:         output.println(iterator.next());
039:     } // try
040:     catch (Exception exception)
041:     {
042:       System.err.println(exception);
043:     } // catch
```

Sorting a text file using a `TreeSet`

```
044:      finally
045:      {
046:        try { if (input != null) input.close(); }
047:        catch (IOException exception)
048:          { System.err.println("Could not close input " + exception); }
049:        if (output != null)
050:        {
051:          output.close();
052:          if (output.checkError())
053:            System.err.println("Something went wrong with the output");
054:        } // if
055:      } // finally
056:    } // main
057:
058: } // class Sort
```



Coffee time: What do you think our `Sort` program would do, if we used a `HashSet` instead of a `TreeSet`?

Trying it

- Program **sort**s input and removes duplicate lines.

Console Input / Output

```
$ cat input.txt
Smith,James     87.9%
Jackson,Helen   100%
Jones,Stephen   51.5%
Jackson,Helen   100%
$ java Sort input.txt output.txt
$ cat output.txt
Jackson,Helen   100%
Jones,Stephen   51.5%
Smith,James     87.9%
$ _
```

Run

Coursework: Sorting election leaflets, using a `TreeSet`

(Summary only)

Write a program to **sort** election information leaflets into delivery order, using a `TreeSet`.

Section 6

Summary of lists and sets

Aim

AIM: To summarize the **collections framework** explored so far, and introduce the `Collection` **interface** and the `LinkedList` **class**, for which we explore **linked lists**. We also revisit `List`.

Summary of lists and sets

- So far met
  - **interface** `List`
    - with `ArrayList` **implement**ation
  - interface `Set`
    - implemented by `HashSet` and `TreeSet`.
- Also common **type**
  of which all **list collection**s and **set collection**s are members.

Collections API: `Collection` interface

- The **interface** `java.util.Collection` part of **collections framework**
  - specifies **instance methods** to support **collection**
    - such as **list collection** / **set collection**.

Method definitions in interface `Collection` (some of them).

Method	Return	Arguments	Description
`size`	**`int`**		Returns the size of this `Collection`, that is, the number of elements in it.

Collections API: `Collection` interface

Method definitions in interface `Collection` (some of them).

Method	Return	Arguments	Description
`add`	**`boolean`**	`Object`	Ensures that this `Collection` contains the given `Object`, or an **equivalent** one if appropriate. It **returns** **`true`** if the `Collection` was modified, **`false`** otherwise. For example, a `List` always appends the element on the end and returns **`true`**, whereas a `Set` will do nothing if it already contains an equivalent element.

Collections API: `Collection` interface

Method definitions in interface `Collection` (some of them).

Method	Return	Arguments	Description
`remove`	**`boolean`**	`Object`	Removes one element equivalent to the given `Object`, and returns **`true`** if the `Collection` was changed (i.e. there was at least one element matching the given one).
`addAll`	**`boolean`**	`Collection`	Adds all the elements of the given `Collection` to this one, and returns **`true`** if this collection was changed. (E.g. the given collection could be empty, or this one could be a `Set` and already contain the elements.)

Collections API: `Collection` interface

Method definitions in interface `Collection` (some of them).

Method	Return	Arguments	Description
`removeAll`	**`boolean`**	`Collection`	Removes all the elements of the given `Collection` from this one, and returns **`true`** if this collection was changed.
`retainAll`	**`boolean`**	`Collection`	Removes all elements of this collection which are *not* contained in the given `Collection`, and returns **`true`** if this collection was changed.

Collections API: `Collection` interface

Method definitions in interface `Collection` (some of them).

Method	Return	Arguments	Description
`contains`	**`boolean`**	`Object`	Returns **`true`** if the `Collection` contains at least one `Object` which is equivalent to the given one, **`false`** otherwise.
`containsAll`	**`boolean`**	`Collection`	Returns **`true`** if this `Collection` contains at least one equivalent `Object` for each element in the given collection, **`false`** otherwise.

Collections API: `Collection` interface

Method definitions in interface `Collection` (some of them).

Method	Return	Arguments	Description
`iterator`	`Iterator`		Returns an **object** that **implements** `java.util.Iterator`, giving access to all the elements of the `Collection`. The order depends on the kind of collection.

- Since Java 5.0, `Collection` is **generic interface**
  - **type parameter** represents **type** of **object**s that can be stored.
- When use **parameterized type** of `Collection`
  - all occurrences of `Object` above replaced by **type argument**.

Collections API: Lists: `List` interface: extends `Collection`

- The **interface** `java.util.List` is **extension** of `java.util.Collection`.

```
public interface List<E> extends Collection<E>
{
  ...
} // interface List
```

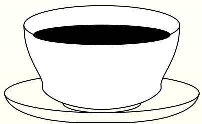
Collections API: Sets: `Set` interface: extends `Collection`

- The **interface** `java.util.Set` is **extension** of `java.util.Collection`.

```
public interface Set<E> extends Collection<E>
{
  ...
} // interface Set
```

Summary of lists and sets

- So **instance** of `ArrayList<T>`
  - is an `ArrayList<T>`
  - is a `List<T>`
  - is a `Collection<T>`.



Coffee time: Consider the instance methods `addAll()`, `removeAll()` and `retainAll()` as they apply to `Set`s. What is the relationship between these and the notions of **set union**, **set intersection** and **set difference**?

- Lists also have instance methods not specified in `Collection`
  - based on use of **list index**
  - already seen `get()` and `set()`.

Collections API: Lists: `add(index)` and `remove(index)`

- `java.util.List` specifies **instance methods** for adding / removing elements at particular **list index**
  - in addition to those defined in `java.util.Collection`
    - for adding element (at the end)
    - or removing element **equivalent** to given one.

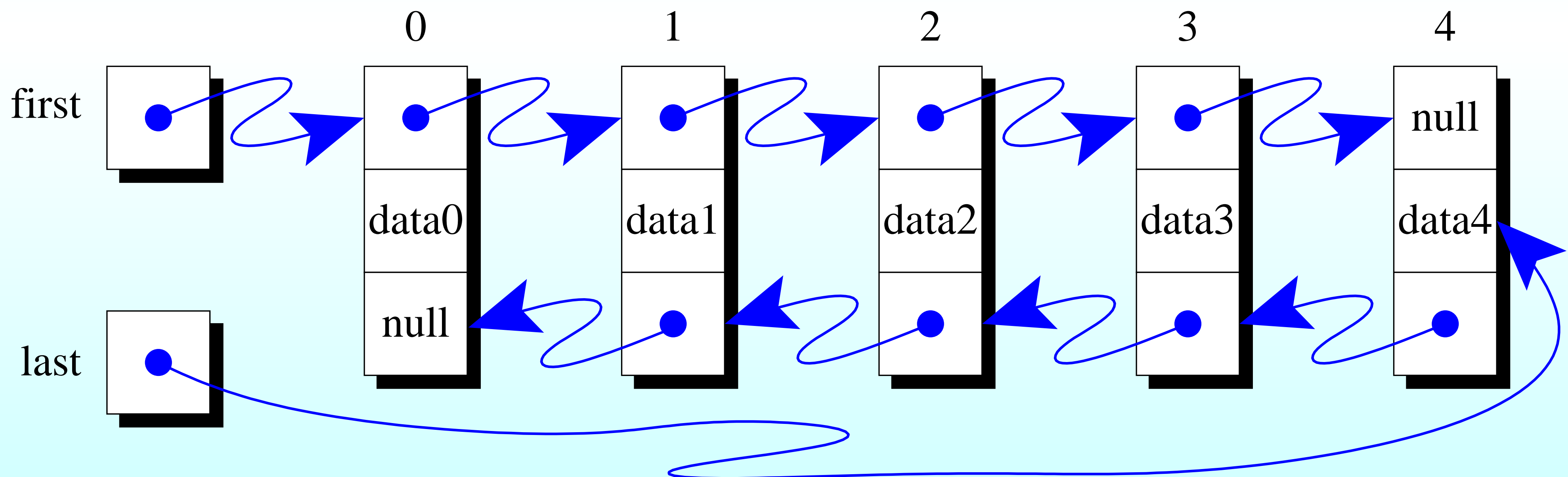
Collections API: Lists: `add(index)` and `remove(index)`

Method definitions in interface `List` (some more of them).

Method	Return	Arguments	Description
`add`		**`int`**, `Object`	Inserts the given `Object` at the specified list index, shifting any elements after that position up by one place. To avoid an `IndexOutOfBoundsException`, the index must be legal (`0 <= index <= size()`).
`remove`	`Object`	**`int`**	Removes the element at the given list index, shifting elements after that position down by one place. To avoid an `IndexOutOfBoundsException`, the index must be legal (`0 <= index < size()`).

Design: Storing data: linked list

- A **linked list** is **data structure**
  - holds **data** in chain of link **objects**
    - each containing (**reference** to) one data element
    - and reference to next link object.
- A **doubly linked list** has links in both directions.



Design: Storing data: linked list

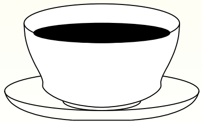
- To access element at particular **list index**
  - chain must be followed from front, counting links until index reached
  - or from back if nearer.
- Not efficient if many **random access**es of elements needed.
- Can be more efficient than **array**
  - e.g. adding at back without needing **array extension**
  - adding / removing at front / middle
    without need to shuffle elements.

Collections API: Lists: `LinkedList`

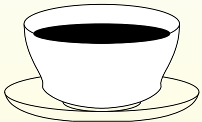
- `java.util.LinkedList` part of **collections framework**
  - another implementation of **list collection**
  - **implements** `java.util.List` **interface**
  - uses **doubly linked list**.
- Since Java 5.0, `LinkedList`, is **generic class**
  - **type parameter** is **type** of **objects** that can be stored.

```
public class LinkedList<E> implements List<E>
{ ... }
```

Summary of lists and sets

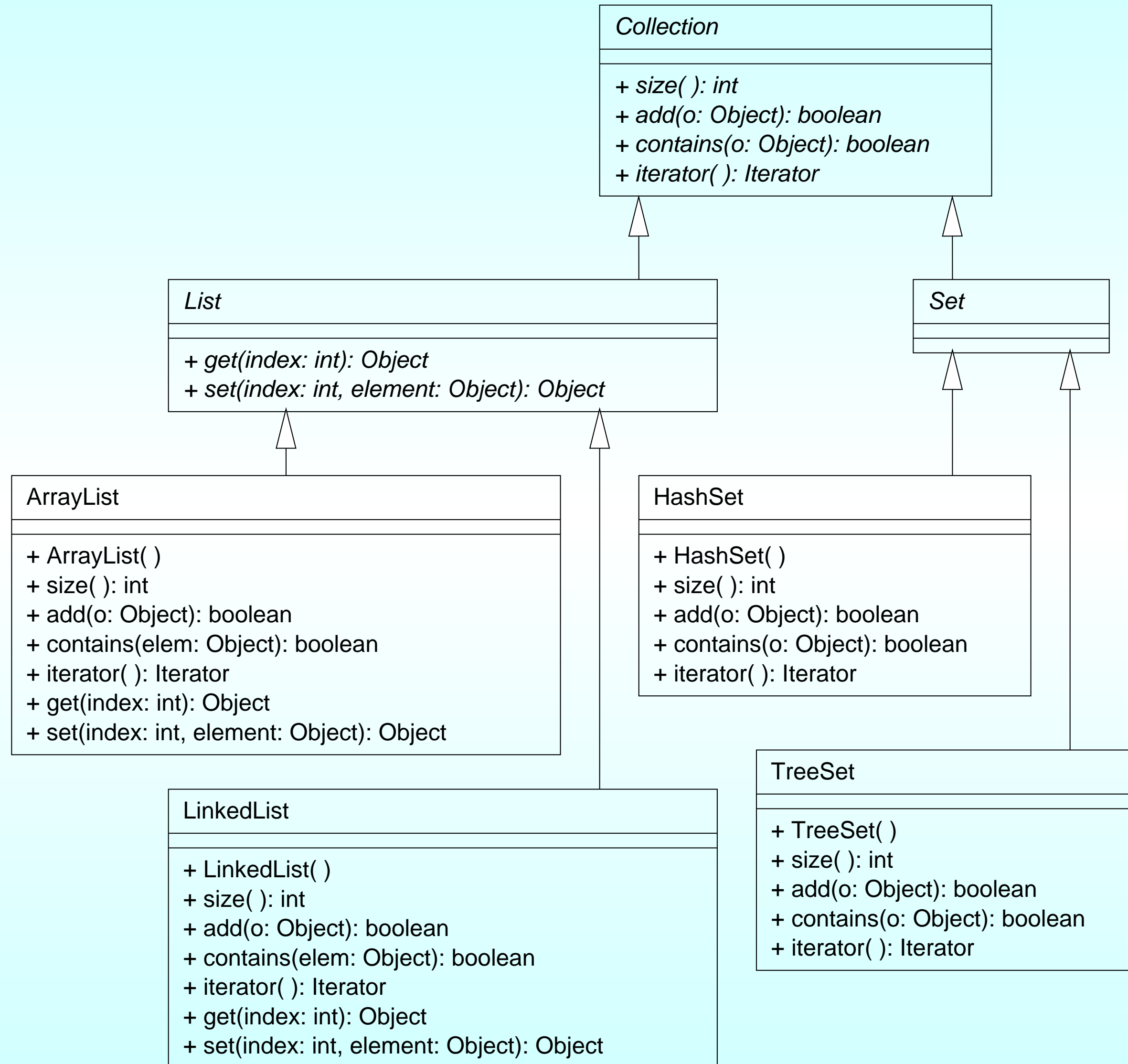


Coffee time: Could we have used a `LinkedList` for our `Reverse` program from Section 11 on page 14? How about if we had added each line at the front of the **list**, using `add(0)`? Would we still use a **list index** for printing the result?

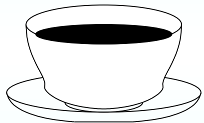


Coffee time: Add the following instance methods to a (copy of?) this diagram. `remove()`, `addAll()`, `removeAll()`, `retainAll()`, `containsAll()`, `add(index)` and `remove(index)`.

Summary of lists and sets



Summary of lists and sets



Coffee time: Why do you think that `Collection`, `List` and `Set` are interfaces, rather than **abstract classes**?

Section 7

Example: Word frequency count

Aim

AIM: To introduce the idea of **map**s, the `Map` **interface** and the `TreeMap` **class**. In particular we observe that a `TreeMap` makes it easy to obtain the values from the map in **key** order. We also see that the **for-each loop** can be used with **collection**s.

Word frequency count

- Read **text file**
  - produce alphabetically **sort**ed list of words on **standard output**
  - each with number of occurrences.
- Use **map**.

Collections API: Maps

- Another kind of **collection** in **collections framework**
  - **map**.
- Could view **array**s and **list collection**s as functions from **key** to corresponding element
  - key is **array index** / **list index**.
- Maps more general
  - key can be any **type** of **object**.
  - For every key in map there is associated value.
  - Two different keys may map on to same value
    - but each possible key maps on to at most one value.

Collections API: Maps

- Another view: **set** of pairs
  - containing key and value
  - keys unique within particular map
  - values may be duplicated.
- Map is **many-to-one association**
  - i.e. **function**.

Word frequency count

- Program will
  - Separate input into words
  - build map from word on to
    - pair containing word and frequency found so far.

The WordWithFrequency class

```
001: // A pairing of a word with its frequency count so far.
002: public class WordWithFrequency
003: {
004:   // The word, occurrences of which are being counted.
005:   private final String word;
006:
007:   // The frequency count of this word so far.
008:   private int frequencySoFar;
009:
010:
011:   // Create a pairing with the given word, and frequency of one.
012:   public WordWithFrequency(String requiredWord)
013:   {
014:     word = requiredWord;
015:     frequencySoFar = 1;
016:   } // WordWithFrequency
```

The WordWithFrequency class

```
019:   // Count another occurrence of this word.
020:   public void incrementFrequency()
021:   {
022:     frequencySoFar++;
023:   } // incrementFrequency
024:
025:
026:   // A String showing the word and its frequency.
027:   @Override
028:   public String toString()
029:   {
030:     return word + " " + frequencySoFar;
031:   } // toString
032:
033: } // class WordWithFrequency
```

Collections API: Maps: `Map` interface

- The **interface** `java.util.Map` part of **collections framework**
  - specifies **instance methods** needed to support **map**.

Method definitions in interface `Map` (some of them).

Method	Return	Arguments	Description
`put`	`Object`	`Object, Object`	Takes a **key** and a value, and adds that association to the map. If the map previously contained a mapping for this key (or an **equivalent** one), the old value is replaced with the new one. Returns the **null reference**, if this is a new key, or **returns** the old value otherwise.

THE UNIVERSITY of MANCHESTER

Collections API: Maps: `Map` interface

Method definitions in interface `Map` (some of them).

Method	Return	Arguments	Description
`get`	`Object`	`Object`	Takes a key and returns the value associated with it, or the null reference if the map does not contain a mapping with a key which equivalent to the given one.

Collections API: Maps: `Map` interface

Method definitions in interface `Map` (some of them).

Method	Return	Arguments	Description
`values`	`Collection`		Returns a `Collection` of the values (not keys) in the map. The `iterator()` **instance method** of the resulting `Collection` may support iterating through the values in a particular order, or not, depending on the kind of `Map`.
`keySet`	`Set`		Returns a `Set` of the keys (not values) in the map.

Collections API: Maps: `Map` interface

- Since Java 5.0, `Map` is **generic interface**
  - *two* **type parameter**s
    - **type** of **object**s used as keys
    - **type** of **object**s stored as values.
- When use **parameterized type** of `Map`
  - all occurrences of `Object` in above table replaced by corresponding **type argument**.

Collections API: Maps: `TreeMap`

- `java.util.TreeMap` part of **collections framework**
  - implementation of **map**
  - **implement**s `java.util.Map` **interface**.
- Uses **ordered binary tree**
  - has to be possible to order **key**s.
  - Simplest way: ensure **class** of keys implements `java.lang.Comparable`.
- `values()` gives `Collection`
  - `iterator()` of this gives **object**
    * **implement**s `java.util.Iterator`
    * supports **iteration** over values of map in key order.

Collections API: Maps: `TreeMap`

- Since Java 5.0, `TreeMap` is **generic class**
  - **type parameter**s are **type** of **object**s used as keys and values.

```
public class TreeMap<K, V> implements Map<K, V>
{ ... }
```

The WordFrequencyMap class

```
001: import java.util.Collection;
002: import java.util.TreeMap;
003:
004: // A map from word to WordWithFrequency.
005: public class WordFrequencyMap
006: {
007:   // The map uses a TreeMap, so that we can obtain the values in natural
008:   // ordering of the keys. I.e., in order by word.
009:   private final TreeMap<String, WordWithFrequency>
010:     wordMappedToWordWithFrequency = new TreeMap<String, WordWithFrequency>();
011:
012:
013:   // Empty constructor, nothing needs doing.
014:   public WordFrequencyMap()
015:   {
016:   } // WordFrequencyMap
```

The WordFrequencyMap class

```
019:  // Count an occurrence of the given word by either incrementing the
020:  // frequency of an existing WordWithFrequency or creating a new one if
021:  // this is the first occurrence of the word.
022:  public void countWord(String word)
023:  {
024:    WordWithFrequency wordWithFrequency
025:      = wordMappedToWordWithFrequency.get(word);
026:    if (wordWithFrequency != null)
027:      wordWithFrequency.incrementFrequency();
028:    else
029:    {
030:      wordWithFrequency = new WordWithFrequency(word);
031:      wordMappedToWordWithFrequency.put(word, wordWithFrequency);
032:    } // else
033:  } // countWord
```

The `WordFrequencyMap` class

- `toString()` exploits fact that
  - `values()` of `TreeMap` yields `Collection`
    * with `Iterator` that presents elements in **key** order.
- I.e. `Iterator` goes through values
  in **lexicographic order** of words used as keys.
- Use **for-each loop** rather than explicitly creating `Iterator`.

THE UNIVERSITY of MANCHESTER

Statement: for-each loop: on collections

- The **enhanced for statement**
  - introduced in Java 5.0
  - more commonly called **for-each loop**.
- Can be used with **collection**s as well as **array**s.
- E.g. Wish to process each element of some `Collection`:

```
Collection<T> c = ...


Iterator<T> i = c.iterator();
while (i.hasNext())
  ... Statement with one use of i.next().
```

Statement: for-each loop: on collections

- Could use for-each loop:

```
Collection<T> c = ...

for (T e : c)
  ... Statement using e.
```

- Shorthand for:

```
Collection<T> c = ...

for (Iterator<T> i = c.iterator(); i.hasNext(); )
{
  T e = i.next();
  ... Statement using e.
} // for
```

- For-each loop suitable if processing all elements using one loop.

The WordFrequencyMap class

```
036:   // Show the words and frequencies in word order.
037:   @Override
038:   public String toString()
039:   {
040:     // Obtain the WordWithFrequency values in word iterable order.
041:     Collection<WordWithFrequency> wordWithFrequencyValues
042:       = wordMappedToWordWithFrequency.values();
043:
044:     String result = "";
045:     for (WordWithFrequency wordWithFrequency : wordWithFrequencyValues)
046:       result += String.format("%s%n", wordWithFrequency);
047:
048:     return result;
049:   } // toString
050:
051: } // class WordFrequencyMap
```

The WordFrequencyMap class

- For-each loop shorter than long way of writing it.

```
Iterator<WordWithFrequency> iterator = wordWithFrequencyValues.iterator();
while (iterator.hasNext())
  result += String.format("%s%n", iterator.next());
```

- Could have made even shorter.

```
for (WordWithFrequency wordWithFrequency
       : wordMappedToWordWithFrequency.values())
  result += String.format("%s%n", wordWithFrequency);
```

The `WordFrequency` class

- The **main method** reads input one **character** at time
  - builds into groups.
    - either sequence of letters and/or apostrophe
    - or sequence of non-letters.

```
001: import java.io.FileReader;
002: import java.io.IOException;
003:
004: // Read a text document from the file named by the first argument,
005: // and report frequency count of each word on standard output.
006: public class WordFrequency
007: {
008:   public static void main(String[] args)
009:   {
```

The WordFrequency class

```
010:    // We see the data as a character stream.
011:    FileReader input = null;
012:    try
013:    {
014:      if (args.length != 1)
015:        throw new IllegalArgumentException
016:                    ("There must be exactly one argument: input-file");
017:
018:      input = new FileReader(args[0]);
019:
020:      // A store of all the words found so far.
021:      WordFrequencyMap wordFrequencyMap = new WordFrequencyMap();
022:
023:      // Remember whether we are reading a word or characters between words.
024:      boolean currentGroupIsAWord = false;
025:
026:      // The group of characters we are currently reading.
027:      String currentGroup = "";
028:
```

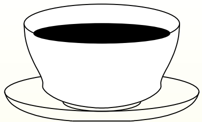
The WordFrequency class

```
029:        int currentCharAsInt;
030:        while ((currentCharAsInt = input.read()) != -1)
031:        {
032:          char currentChar = (char)currentCharAsInt;
033:
034:          // We change group if the kind of the current character
035:          // is not the same as the kind of the current group.
036:          if ( (Character.isLetter(currentChar) || currentChar == '\'')
037:               != currentGroupIsAWord )
038:          {
039:            // We are starting a new group.
040:            if (currentGroupIsAWord)
041:              wordFrequencyMap.countWord(currentGroup.toLowerCase());
042:            currentGroup = "";
043:            currentGroupIsAWord = !currentGroupIsAWord;
044:          } // if
045:          // Whether new or old group, add the current character to it.
046:          currentGroup += currentChar;
047:        } // while
048:
```

The WordFrequency class

```
049:       // We have a trailing word if the last character was a letter or '.
050:       if (currentGroupIsAWord && ! currentGroup.equals(""))
051:         wordFrequencyMap.countWord(currentGroup.toLowerCase());
052:
053:       // The toString of wordFrequencyMap already has a new line at the end.
054:       System.out.print(wordFrequencyMap);
055:     } // try
056:     catch (Exception exception)
057:     {
058:       System.err.println(exception);
059:     } // catch
060:     finally
061:     {
062:       try { if (input != null) input.close(); }
063:       catch (IOException exception)
064:         { System.err.println("Could not close input " + exception); }
065:     } // finally
066:   } // main
067:
068: } // class WordFrequency
```

The `WordFrequency` class



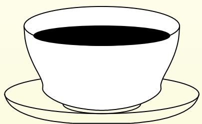
Coffee time: Are you happy with the **condition** of the first **if statement** inside the **while loop**? How would you have written that? Also, for the if statement after the while loop, could we replace the condition with just `currentGroupIsAWord`?

Trying it

Console Input / Output

```
$ java WordFrequency RomeoAndJuliet.txt
 (Output shown using multiple columns to save space.)
'tis 1        be 1          enemy 1      it 1         not 2           romeo 3      thee 1       were 1
a 4           belonging 1   face 1       man 1        o 1             rose 1       thou 1       what's 2
all 1         but 1         foot 1       montague 2   of 1            smell 1      though 1     which 3
and 1         by 1          for 1        my 1         other 3         so 1         thy 3        without 1
any 2         call 1        hand 1       myself 1     owes 1          some 1       thyself 1    would 2
arm 1         call'd 1      he 2         name 6       part 2          sweet 1      title 1
art 1         dear 1        in 1         no 1         perfection 1    take 1       to 1
as 1          doff 1        is 3         nor 5        retain 1        that 4       we 1
$ _
```

Run



Coffee time: Now that you know about `TreeMap`, can you think how we could have a **tree sort** that does not lose duplicate input items?

Section 8

Example: Word frequency count sorted by frequency

Aim

AIM: To introduce the `HashMap` **class**, and the fact that a **collection** can be built to initially contain the same values as some other collection. We also take a look at how we can go about making a good **override** of the `hashCode()` **instance method** of `Object`.

The WordWithFrequency class

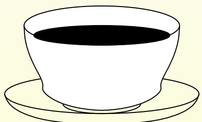
```
001: // A pairing of a word with its frequency count so far.
002: public class WordWithFrequency implements Comparable<WordWithFrequency>
003: {
004:   // The word, occurrences of which are being counted.
005:   private final String word;
006:
007:   // The frequency count of this word so far.
008:   private int frequencySoFar;
009:
010:
011:   // Create a pairing with the given word, and frequency of one.
012:   public WordWithFrequency(String requiredWord)
013:   {
014:     word = requiredWord;
015:     frequencySoFar = 1;
016:   } // WordWithFrequency
017:
018:
```

The WordWithFrequency class

```
019:    // Count another occurrence of this word.
020:    public void incrementFrequency()
021:    {
022:      frequencySoFar++;
023:    } // incrementFrequency
024:
025:
026:    // A String showing the word and its frequency.
027:    @Override
028:    public String toString()
029:    {
030:      return word + " " + frequencySoFar;
031:    } // toString
```

The WordWithFrequency class

```
034:  // Compare this with the given other, returning negative, zero or positive.
035:  // Order first on descending frequency, then on ascending word.
036:  @Override
037:  public int compareTo(WordWithFrequency other)
038:  {
039:    if (frequencySoFar != other.frequencySoFar)
040:      return other.frequencySoFar - frequencySoFar;
041:    else
042:      return word.compareTo(other.word);
043:  } // compareTo
```



Coffee time: How would we change this to make it order by ascending frequency?

The WordWithFrequency class

```
046:   // Return true if and only if the given object is equivalent to this one.
047:   @Override
048:   public boolean equals(Object other)
049:   {
050:     if (other instanceof WordWithFrequency)
051:       return compareTo((WordWithFrequency)other) == 0;
052:     else
053:       return super.equals(other);
054:   } // equals
```

- Also override `hashCode()`
  - even though not strictly needed for this program.

Standard API: `Object`: `hashCode()`: making a good definition

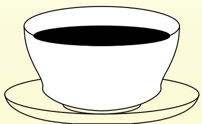
- Classes that **override** `equals()` ought to also override `hashCode()`
  - **return** same value for **equivalent objects**
  - **function** based on
    same **instance variable**s used to define **equivalence** in `equals()`.
- Good **hash code** function should tend to
  give different hash codes for objects that are not equivalent
  - otherwise **hash table**s have too many clashes.
- One way of achieving good spread
  - turn instance variables into numbers
    if not already number – e.g. use their `hashCode()`
  - multiply each by different **prime number**
  - add products.

The WordWithFrequency class

```
057:   // A hash code for this object: equivalent ones have the same hash code.
058:   @Override
059:   public int hashCode()
060:   {
061:     return frequencySoFar * 31 + word.hashCode() * 37;
062:   } // hashCode
063:
064: } // class WordWithFrequency
```

The `WordWithFrequency` class

- Many professional Java programmers make every class have
  - `equals()`, matching `hashCode()`
  - and if class **implements** `Comparable`
    - matching `compareTo()`.
- Even if not intending to need them now
  - in case are needed in future version of program
  - or in another program that reuses class.
- Failing to implement these properly at initial implementation
  - could lead to strange **bugs** at later time.



Coffee time: In some previous examples we had an `equals()`, but no `hashCode()`. Are you tempted to go back and add one in?

The WordFrequencyMap class

- Still have **map** from words onto `WordWithFrequency` **objects**
  - but do not use **natural ordering** of **keys** in `toString()`.
- So (probably) more efficient to use `HashMap` than `TreeMap`.

Collections API: Maps: `HashMap`

- `java.util.HashMap` part of **collections framework**
  - another implementation of **map**
  - **implements** `java.util.Map` **interface**.
- Uses **hash table**
  - each **key** must have appropriate implementation of `hashCode()`
    - for `HashMap` to work correctly.
- `values()` gives `Collection` containing values of map
  - can yield **object implementing** `java.util.Iterator`
    - supports **iteration** over values in no specific order.

Collections API: Maps: `HashMap`

- Rule of thumb: `HashMap` should be used in preference to `java.util.TreeMap`
  - when not desired to obtain values in **key** order.
  - If little or no **hash code** clashing
    - `HashMap` operates in nearly constant time
    - `TreeMap` operates in logarithmic time.
- Since Java 5.0, `HashMap` is **generic class**
  - *two* **type parameter**s for **type** of keys and values.

```
public class HashMap<K, V> implements Map<K, V>
{ ... }
```

The WordFrequencyMap class

```
001: import java.util.Collection;
002: import java.util.HashMap;
003: import java.util.Map;
004: import java.util.TreeSet;
005:
006: // A map from word to WordWithFrequency.
007: public class WordFrequencyMap
008: {
009:   // The map uses a HashMap to efficiently store the WordWithFrequency objects.
010:   private final Map<String, WordWithFrequency>
011:     wordMappedToWordWithFrequency = new HashMap<String, WordWithFrequency>();
012:
013:   // Empty constructor, nothing needs doing.
014:   public WordFrequencyMap()
015:   {
016:   } // WordFrequencyMap
```

The WordFrequencyMap class

```
017:
018:
019:   // Count an occurrence of the given word by either incrementing the
020:   // frequency of an existing WordWithFrequency or creating a new one if
021:   // this is the first occurrence of the word.
022:   public void countWord(String word)
023:   {
024:     WordWithFrequency wordWithFrequency
025:       = wordMappedToWordWithFrequency.get(word);
026:     if (wordWithFrequency != null)
027:       wordWithFrequency.incrementFrequency();
028:     else
029:     {
030:       wordWithFrequency = new WordWithFrequency(word);
031:       wordMappedToWordWithFrequency.put(word, wordWithFrequency);
032:     } // else
033:   } // countWord
```

The `WordFrequencyMap` class

- For `toString()`
  - build **new** `TreeSet` containing *values*
  - iterate through in natural ordering of values
  - i.e. use `compareTo()` from `WordWithFrequency`
    - values covered in descending order of frequency.

Collections API: `Collection` interface: constructor taking a `Collection`

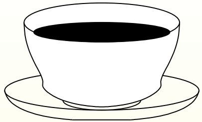
- **API** documentation for `java.util.Collection` **interface** states
  - any **class** which **implement**s it should provide two **constructor method**s
    - one with no **method argument**s builds empty `Collection`
    - other takes existing `Collection` and builds **new** one containing same elements.
- No way for this to be enforced in Java
  - interfaces cannot specify constructor methods!
- Arguably is deficiency in use of interfaces as means of contractual obligation.
- All standard implementations do satisfy requirement.

The WordFrequencyMap class

```
036:   // Show the words and frequencies in frequency order.
037:   @Override
038:   public String toString()
039:   {
040:     // Obtain the WordWithFrequency values in an unpredictable order,
041:     // and put them into a TreeSet so we can extract them in frequency order.
042:     TreeSet<WordWithFrequency> wordWithFrequencyValues
043:       = new TreeSet<WordWithFrequency>(wordMappedToWordWithFrequency.values());
044:
045:     String result = "";
046:     for (WordWithFrequency wordWithFrequency : wordWithFrequencyValues)
047:       result += String.format("%s%n", wordWithFrequency);
048:
049:     return result;
050:   } // toString
051:
052: } // class WordFrequencyMap
```

The `WordFrequency` class

- Same as previous version!



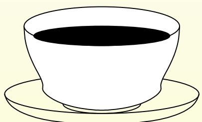
Coffee time: What would happen if `String` did not **override** `hashCode()` – would our program here work? What would it do instead?

Trying it

Console Input / Output

```
$ java WordFrequency input.txt
 (Output shown using multiple columns to save space.)
name 6        any 2         'tis 1        but 1      foot 1      no 1            so 1         title 1
nor 5         he 2          all 1         by 1       for 1       o 1             some 1       to 1
that 5        montague 2    and 1         call 1     hand 1      of 1            sweet 1      we 1
a 4           not 2         arm 1         call'd 1   in 1        owes 1          take 1       were 1
is 3          part 2        art 1         dear 1     it 1        perfection 1    thee 1       without 1
other 3       thy 2         as 1          doff 1     man 1       retain 1        thou 1
romeo 3       what's 2      be 1          enemy 1    my 1        rose 1          though 1
which 3       would 2       belonging 1   face 1     myself 1    smell 1         thyself 1
$ _
```

Run



Coffee time: Now that you know about **maps**, are you tempted to re-implement some of the program for translating documents, perhaps in particular the way that `Dictionary` works, in Section **??** on page **??**?

Coursework: Finding duplicate voters, using a `HashMap`

(Summary only)

Write a program to detect people voting more than once in voting records, using a `HashMap`.

Section 9

Collections of collections

THE UNIVERSITY of MANCHESTER

Aim

AIM: To explore the idea that the elements of a **collection** can themselves be collections, and so quite complex **data structure**s can be built.

Collections of collections

- No example here
  - just idea
  - and coursework.
- Idea might be obvious
  - collections can contain any kinds of **object**
    - including **collection**s.
- E.g. `ArrayList` of `ArrayList`s
  - **collections framework**'s answer to **two-dimensional array**s
- E.g. `TreeMap` of `LinkedList`s, if say
  - making index of all occurrences of identifiers in directory of Java **source code file**s.
- Etc..

Coursework: Finding duplicate voters, using a `HashMap` of `LinkedList`s

(Summary only)

Write a program to detect people voting more than once in voting records, using a `HashMap` of **objects** containing a `LinkedList`.

THE UNIVERSITY of MANCHESTER

Concepts covered in this chapter

- Each book chapter ends with a list of concepts covered in it.
- Each concept has with it
  - a self-test question,
  - and a page reference to where it was covered.
- Please use these to check your understanding before we start the next chapter.