Collections

Using a library class

Suppose you want a compact array of bits, e.g. for a prime number program

You type `java array bits` into Google and find the `BitSet` class, then type `java 8 bitset`

You find that the full name of the class is `java.util.BitSet`

The full name is for world-uniqueness, and it means you have to use one of:

```
import java.util.BitSet;
import java.util.*;        all classes from java.util
```

Going to the source

Suppose the documentation for a class like `BitSet` doesn't tell you enough?

You can type `java source BitSet` into Google and read the source on a site like `docjar.com` or `grepcode.com`, but maybe not the most up-to-date version

The sources are also in a file `src.zip` which comes with your installation of the Java compiler, type

```
unzip -f src.zip java/util/BitSet.java
```

Packages

A *package* like `java.util` is the same thing as a folder or directory,

except that it uses `.` for platform independence instead of `/` or `\` or `:`,

and it may be packed into a zip or jar file

A pattern like `java.util.*` does not include sub-packages like `java.util.concurrent.*`

The package `java.lang` doesn't need to be imported

Clashes

```
import java.util.*;
import java.awt.*;
```

Both packages have a `List` class - if you don't need either, all is well

If you just want `java.util.List`, you need:

```
import java.util.List;                      choose this one
import java.util.*;
import java.awt.*;
```

If you also wanted to use `java.awt.List` as well, you would have to use the long name everywhere

import = abbreviation

An `import` doesn't mean what you think

It doesn't fetch and make classes available, all the standard java library classes are already available

It allows a class name like `BitSet` to be used without its prefix, i.e. you don't have to write `java.util.BitSet` everywhere

Classpaths

Classes are fetched and made available using classpaths

The standard libraries are always on the classpath

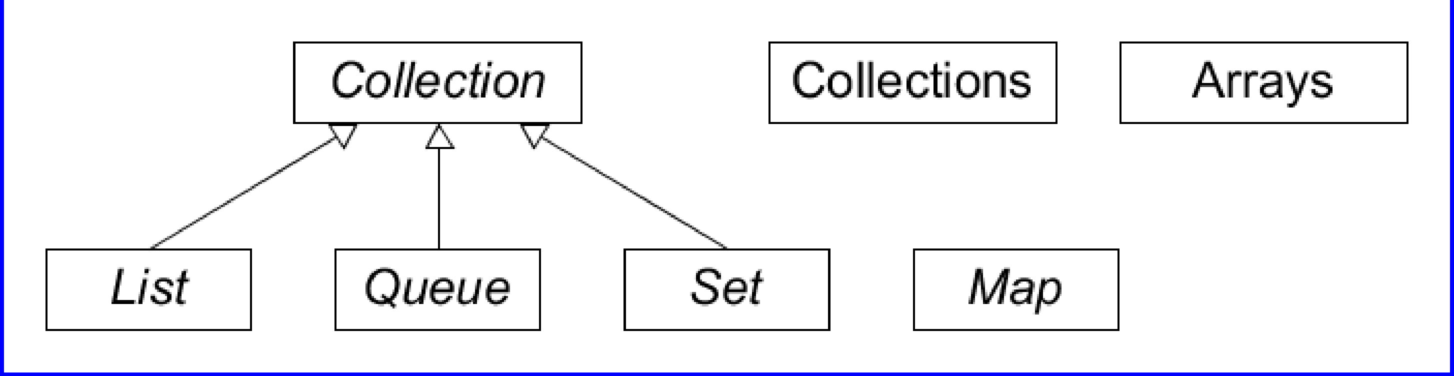
You can use an environment variable `CLASSPATH`

Or you can use the `-classpath` or `-cp` option

You need to include `.` in the classpath

Collections

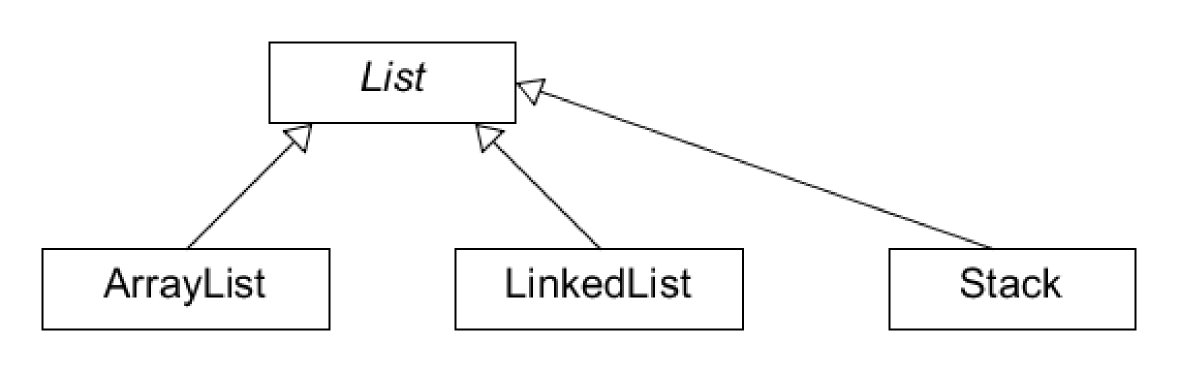
Java has classes which store lists, queues and sets which are together called `Collections`, plus utility classes such as `Collections` and some related `Map` classes:



The italics indicate that all except `Collections` and `Arrays` are interfaces

Lists

The main list classes are `ArrayList`, `LinkedList`, plus `Stack` which maybe shouldn't be there

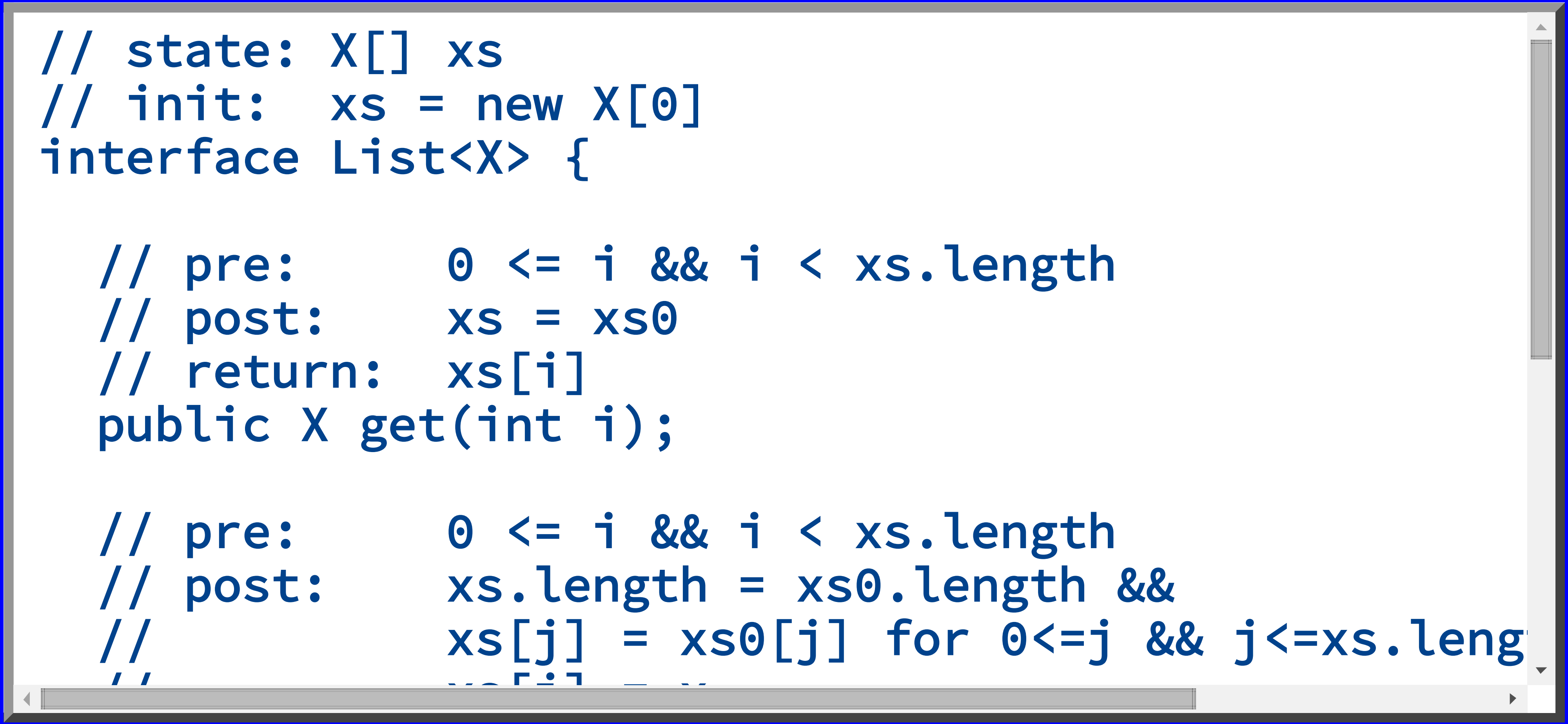


When you create a list, you have to choose between the `ArrayList` and `LinkedList` implementations

The List ADT

Java's *basic* notion of list is something like this

Using Java's arrays as a specification for Java's lists:



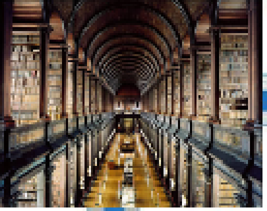
```
// state: X[] xs
// init:  xs = new X[0]
interface List<X> {

  // pre:     0 <= i && i < xs.length
  // post:    xs = xs0
  // return:  xs[i]
  public X get(int i);

  // pre:     0 <= i && i < xs.length
  // post:    xs.length = xs0.length &&
  //          xs[j] = xs0[j] for 0<=j && j<=xs.leng
  //          xs[i] = x
```

The List methods

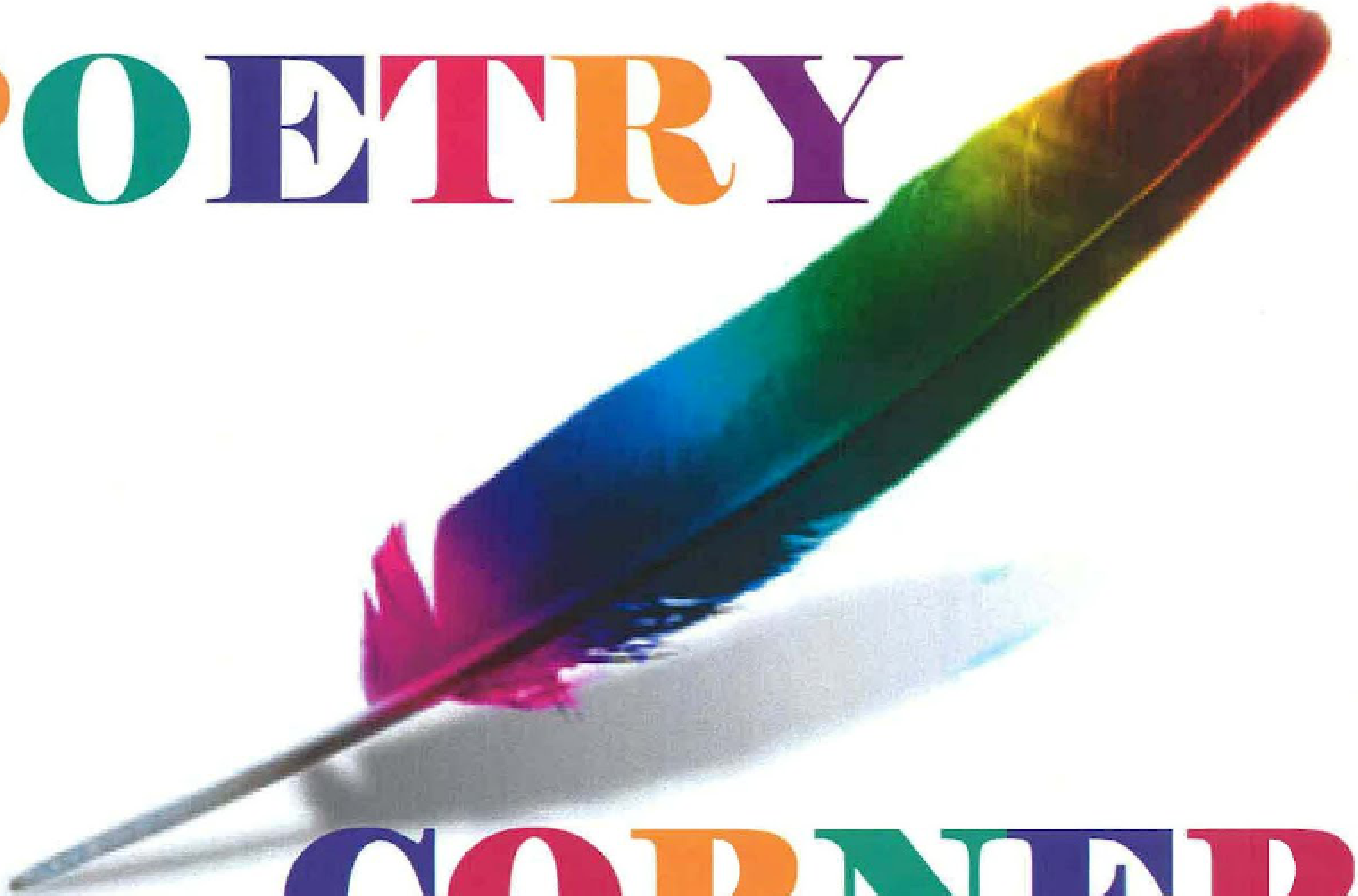
```
add(e) add(i,e) addAll(c) addAll(i,e)
clear() contains(o) containsAll(c)
equals(o) get(i) hashCode() indexOf(o)
isEmpty() iterator() lastIndexOf(o)
listIterator(i) remove(i) remove(o)
removeAll(c) retainAll(c) set(i,e)
size() subList(i,j) toArray() toArray(a)
```

This illustrates how libraries accumulate stuff

Most of the non-index methods are from `Collection`

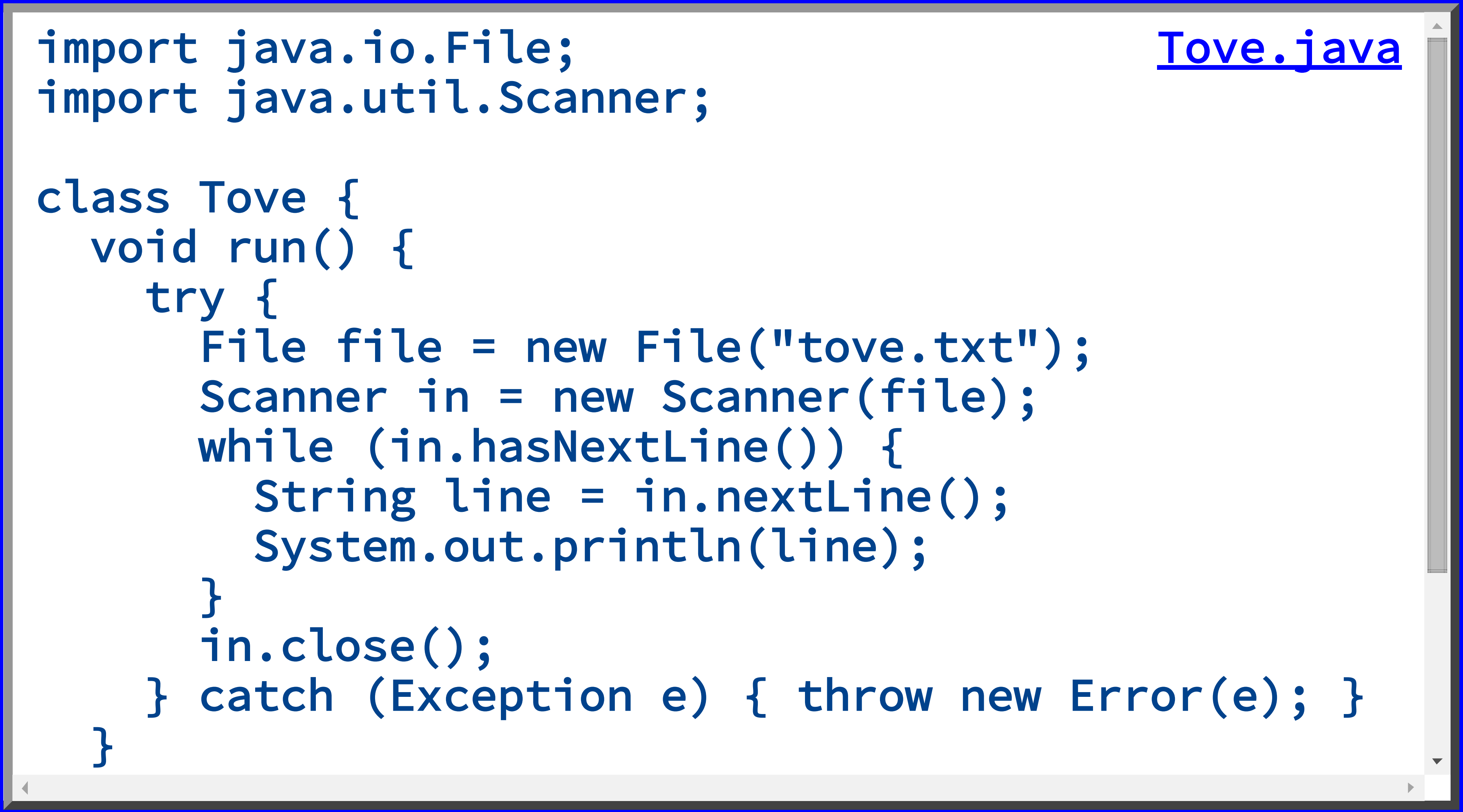


POETRY



CORNER

Poetry Reading



Tove.java

```
import java.io.File;
import java.util.Scanner;

class Tove {
  void run() {
    try {
      File file = new File("tove.txt");
      Scanner in = new Scanner(file);
      while (in.hasNextLine()) {
        String line = in.nextLine();
        System.out.println(line);
      }
      in.close();
    } catch (Exception e) { throw new Error(e); }
  }
```

Using lists

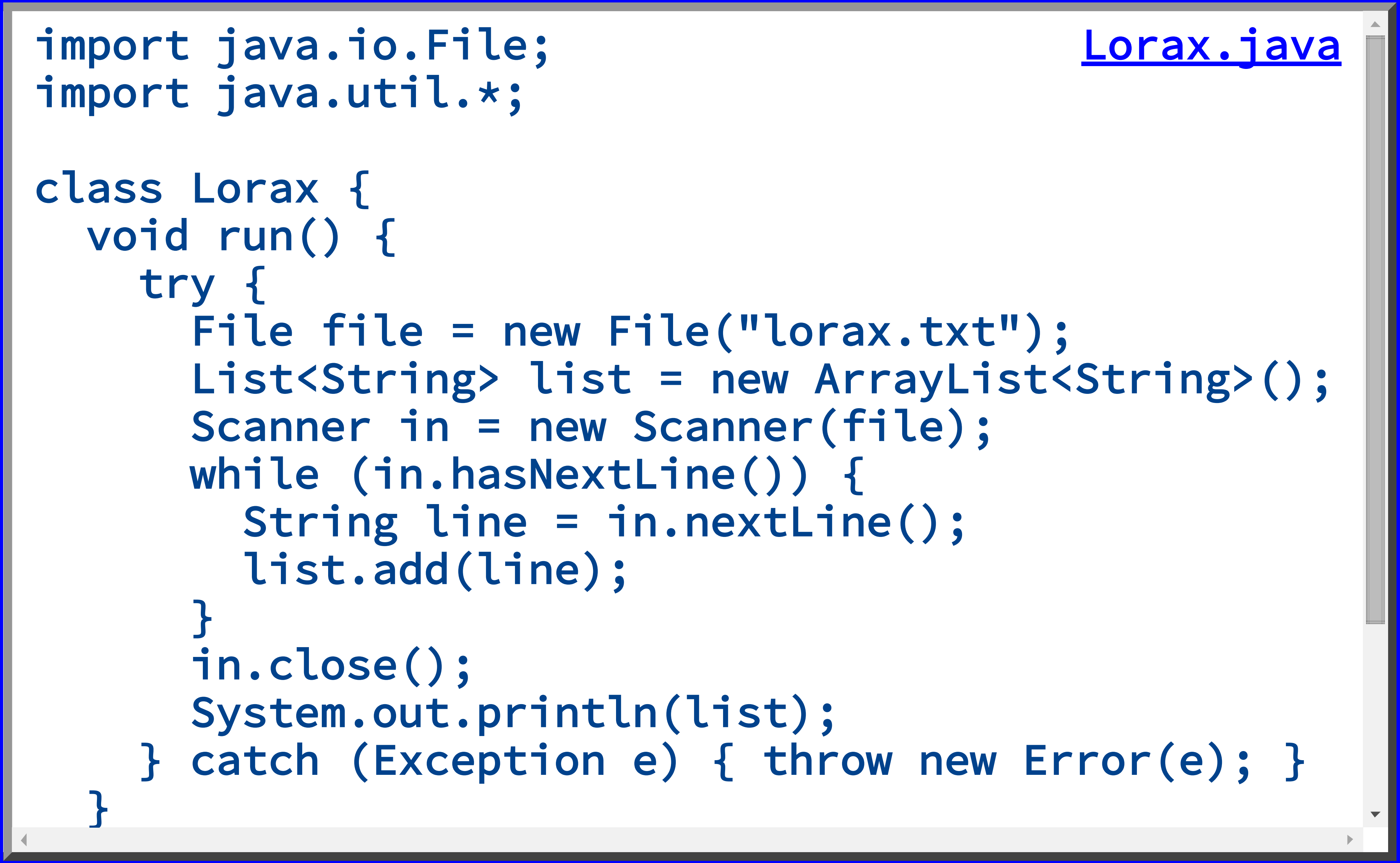
Suppose you want to collect the lines as you read them

You have no (easy) way of knowing how many lines there will be

You could use an array and extend it

But there is no good excuse here for avoiding the libraries, so let's use a list

Gathering lines



Lorax.java

```
import java.io.File;
import java.util.*;

class Lorax {
  void run() {
    try {
      File file = new File("lorax.txt");
      List<String> list = new ArrayList<String>();
      Scanner in = new Scanner(file);
      while (in.hasNextLine()) {
        String line = in.nextLine();
        list.add(line);
      }
      in.close();
      System.out.println(list);
    } catch (Exception e) { throw new Error(e); }
  }
```

Alternatives

There are always lots of ways of doing things

Is there a better way of reading in the lines of a file?

There is a new package `java.nio` with a whole lot of new complicated and incompatible classes

But if you wade through the mess, you can find some gems, e.g. international characters in UTF-8 are easier to handle

Using nio

Hibou.java

```
import java.nio.file.*;
import java.util.*;

class Hibou {
  void run() {
    try {
      Path path = Paths.get("hibou.txt");
      List<String> list = Files.readAllLines(path);
      System.out.println(list);
    } catch (Exception e) { throw new Error(e); }
  }
  public static void main(String[] args) {
    Hibou program = new Hibou();
    program.run();
  }
}
```

List to array

Once we have gathered a list of lines, *if* we don't need to change it again from now on, *and if* we want to do a lot of indexing, we can convert it to an array:

Array.java

```
...
  String[] lines = new String[list.size()];
  list.toArray(lines);
  System.out.println(Arrays.toString(lines));
...
```

Iteration

Suppose we have a list, and decide to keep it instead of converting it into an array

What interesting things can we do with it?

We can use "foreach" loops:

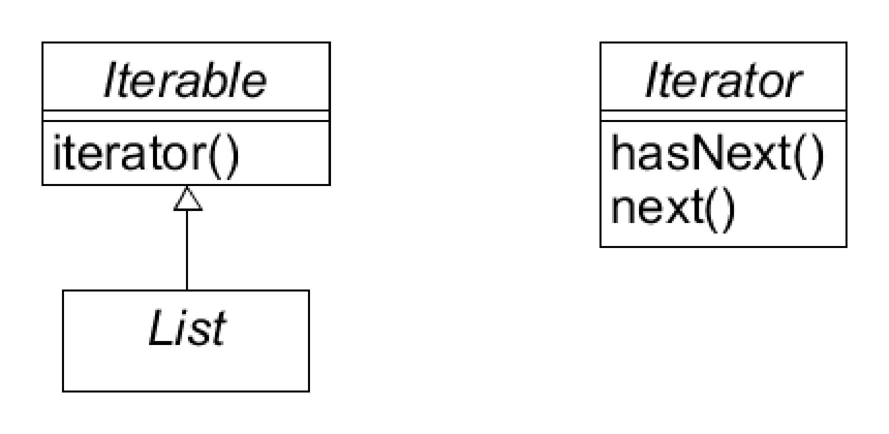
Loop.java

```
...
  for (String line : list) {
    System.out.println(line);
  }
...
```

How does this work?

Iterable

A list is *iterable*, i.e. it has a method which returns an `Iterator` which allows an ordered traversal

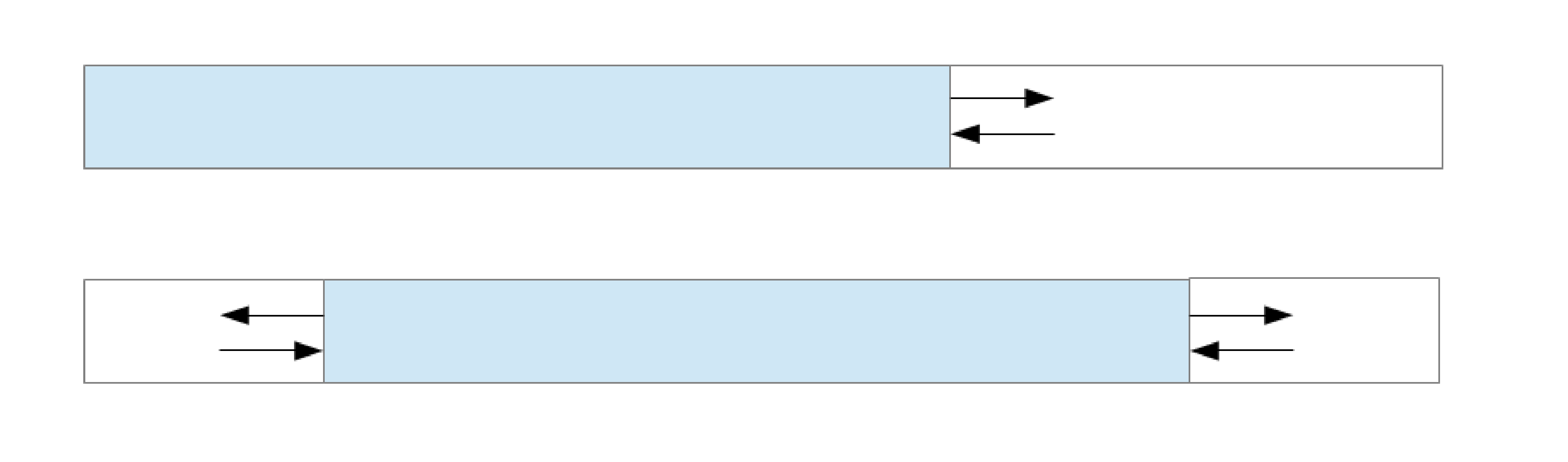


Java's "foreach" loop syntax is compiled into iterator calls

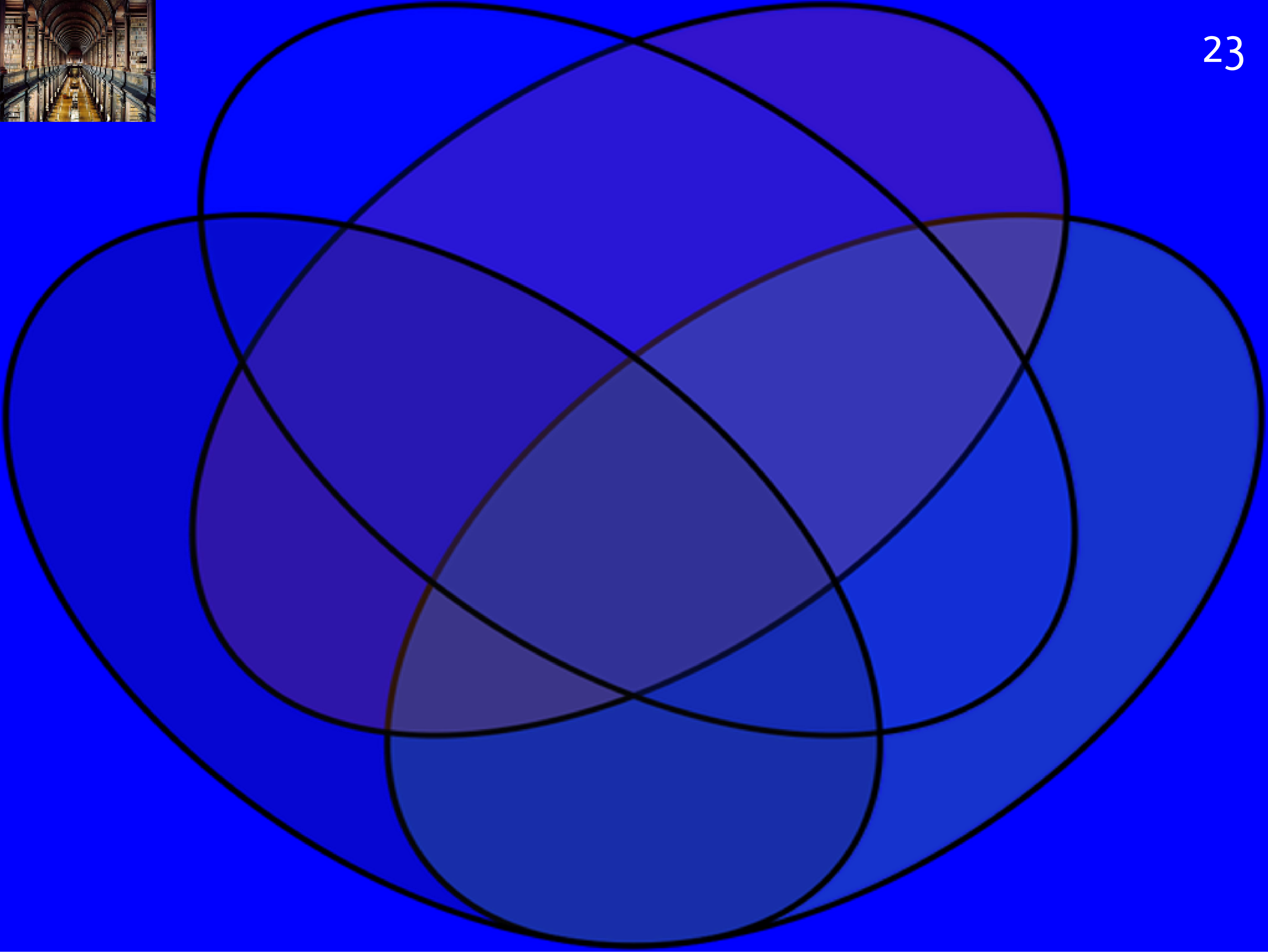
Any suitable class, e.g. your own, can be made iterable

Lists and Queues

With a list, one "end" is flexible, but with a queue, both "ends" are flexible:

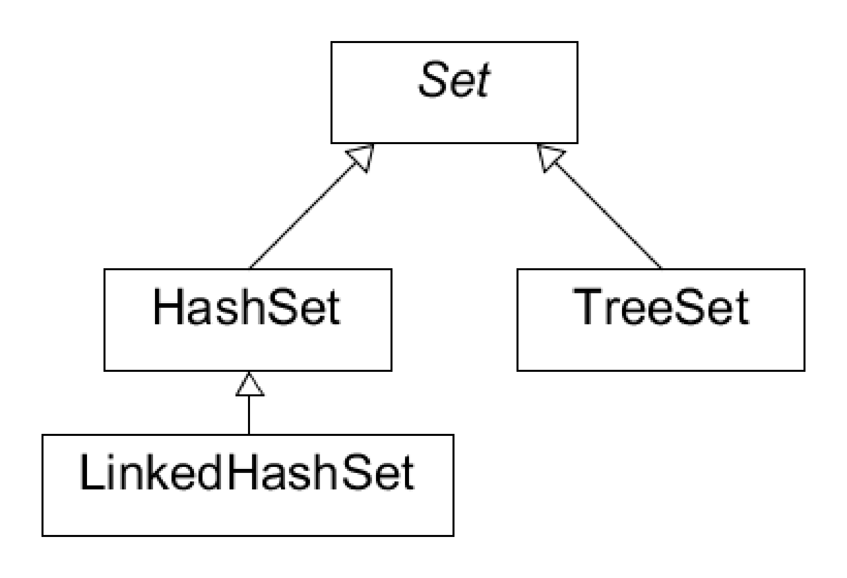


Java's queues are double ended (deques) where you can put things and take things off at either end



Sets

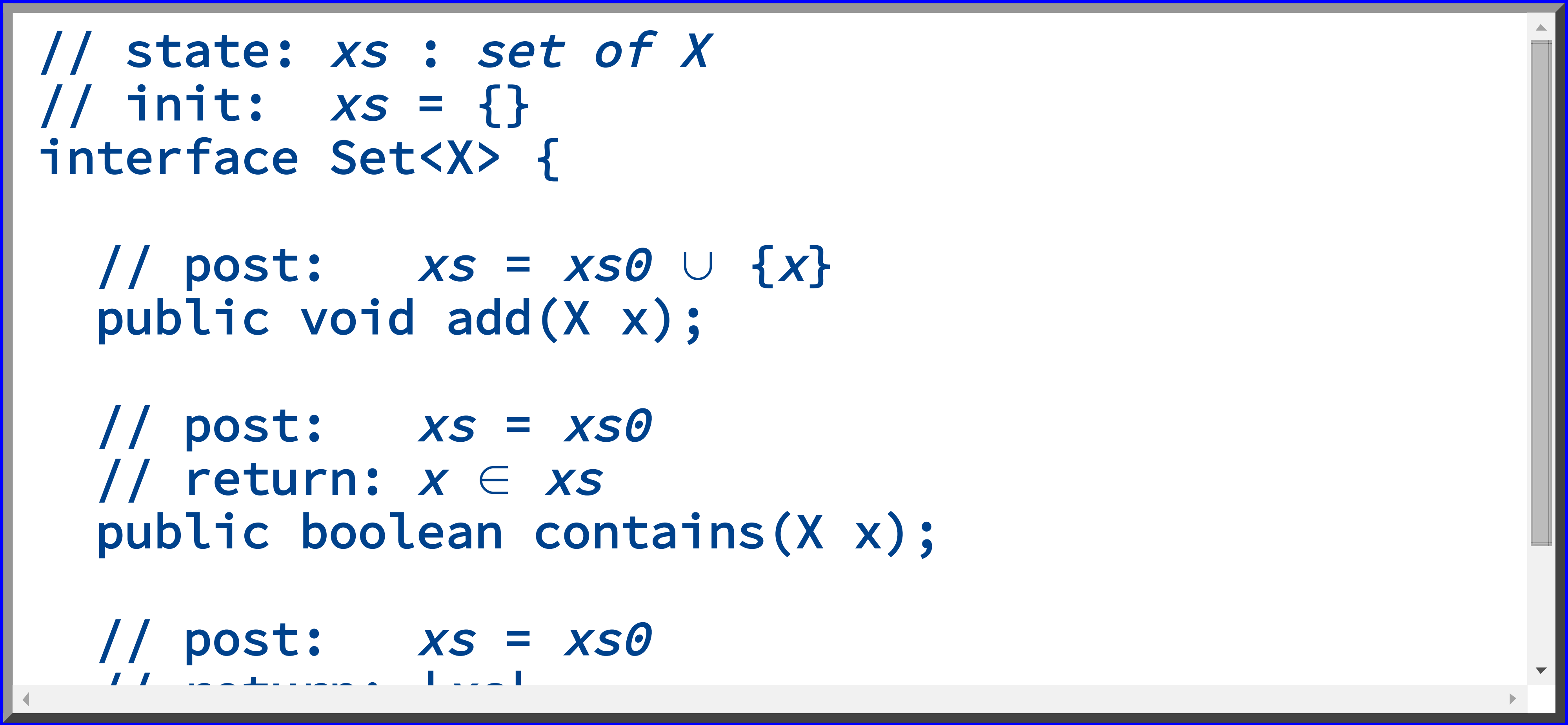
The main set classes are `HashSet`, `TreeSet`



These are the easy parts of the full diagram!

A set ADT

Mathematics has a lot to say about sets, so let's use maths notation for our ADT spec for ***finite*** sets:



```
// state: xs : set of X
// init:  xs = {}
interface Set<X> {

  // post:   xs = xs0 ∪ {x}
  public void add(X x);

  // post:   xs = xs0
  // return: x ∈ xs
  public boolean contains(X x);

  // post:   xs = xs0
  // return: |xs|
```

The Set methods

```
add(e) addAll(c) clear() contains(o)
containsAll(c) equals(o) hashCode()
isEmpty() iterator() remove(o)
removeAll(c) retainAll(c) size()
toArray() toArray(a)
```

Again, this is ADT methods plus utility methods, mostly from `Collection`

Words

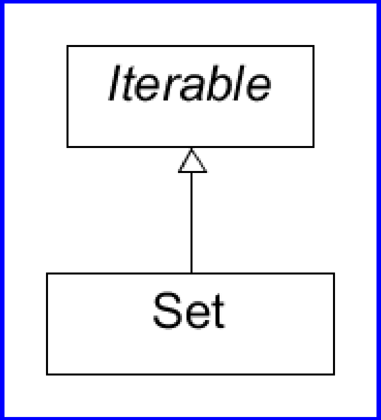
How many different words do poets use?

Vocab.java

```
import java.io.File;
import java.util.*;

class Vocab {
  void run() {
    try {
      File file = new File("lorax.txt");
      Set<String> words = new TreeSet<String>();
      Scanner in = new Scanner(file);
      while (in.hasNext()) {
        String word = in.next();
        words.add(word);
      }
```

Iterating through sets

Here's a difficult part of the library classes



Mathematics says: "sets are unordered"

Computer Science says: "we must be able to process sets element-by-element"

Set order

Computer Science says:

- "*we must be able to process sets element-by-element*"
- "*the processing should be independent of ordering*"
- "*with the possible exception of printing*"
- "*the independence is almost always the programmer's responsibility, because anything else would be too expensive*"

Actual set order

`TreeSet`: the elements are in *sorted* order (in a binary search tree, with red-black balancing)

`HashSet`: the elements are in an *unpredictable* ("random") and *unstable* order (in a flexible hash table)

`LinkedHashSet`: the elements are kept in *insertion* order (in a combined hash table and linked list)

Maps

A *map* is like a finite function held as data – you give it a key and it gives back a value

To count occurrences of words in poems:

Count.java

```
...
  Map<String,Integer> words;
  words = new HashMap<String,Integer>();
  ...
  if (! words.containsKey(word)) words.put(word, 1);
  else words.put(word, words.get(word) + 1);
  ...
  System.out.println(words);
...
```