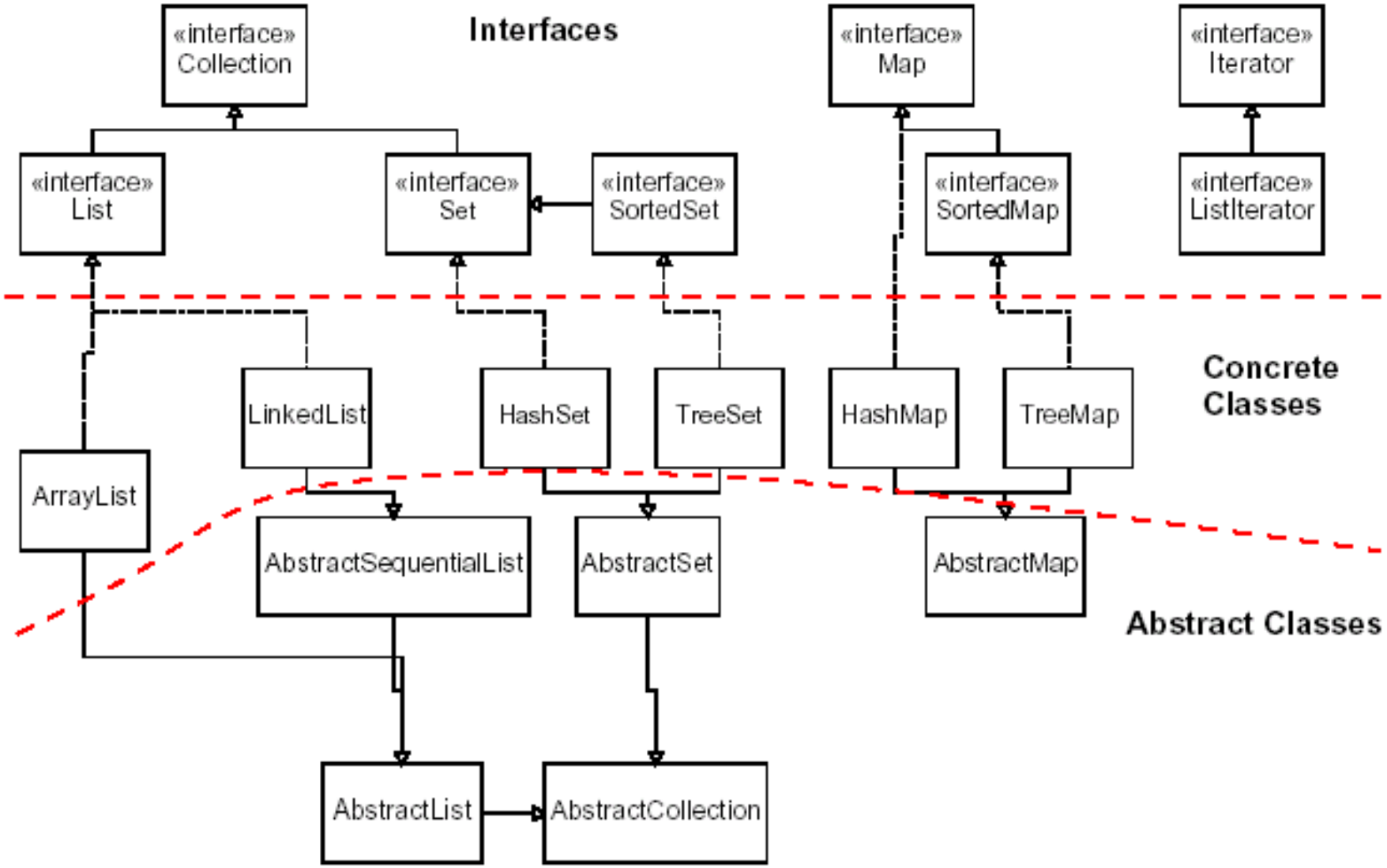
BUILDING JAVA PROGRAMS CHAPTER 11

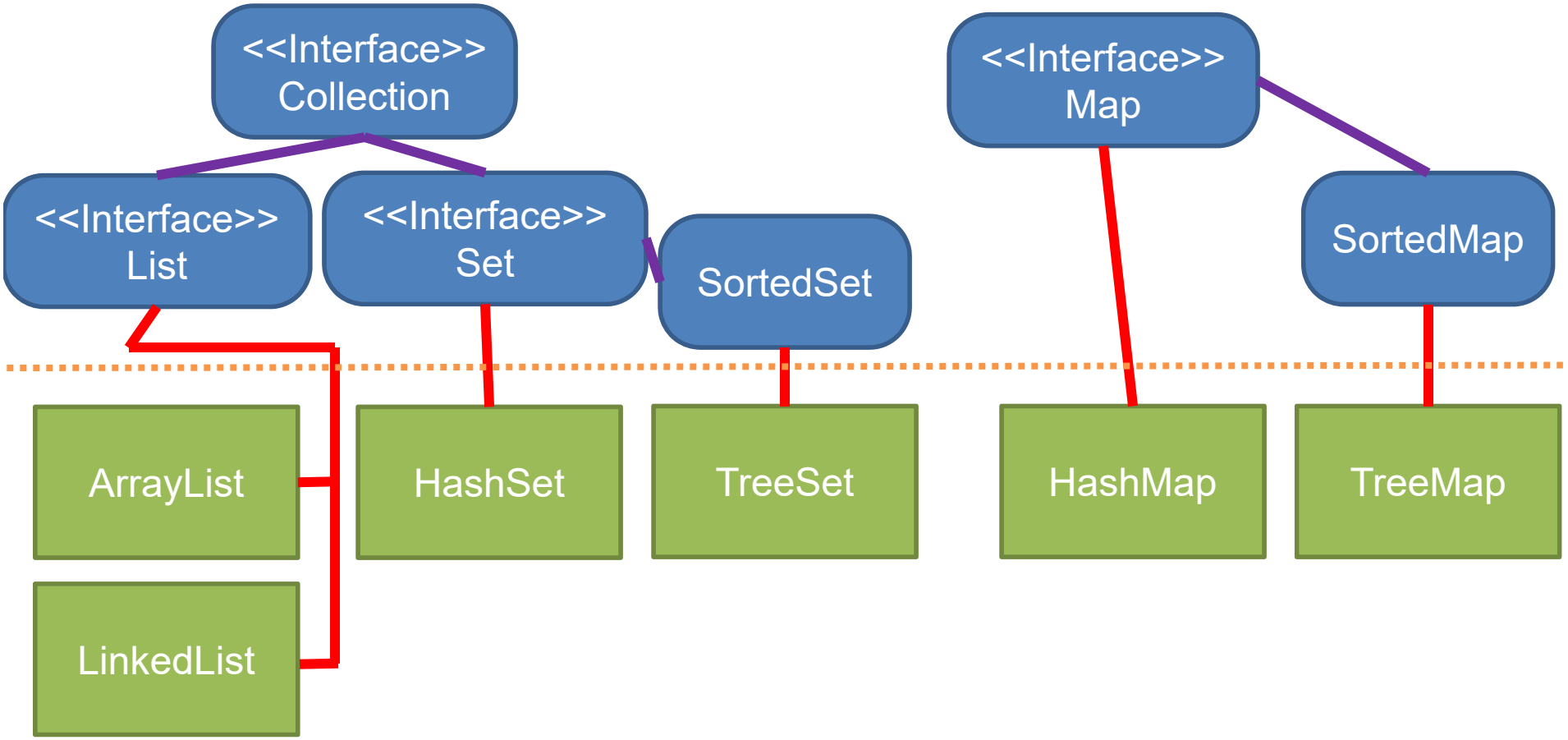
Java Collections Framework

Copyright (c) Pearson 2013.
All rights reserved.

Java collections framework



Java Collection Framework



Collections

- **collection**: an object that stores data; a.k.a. "data structure"
  - the objects stored are called **elements**
  - some collections maintain an ordering; some allow duplicates
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*

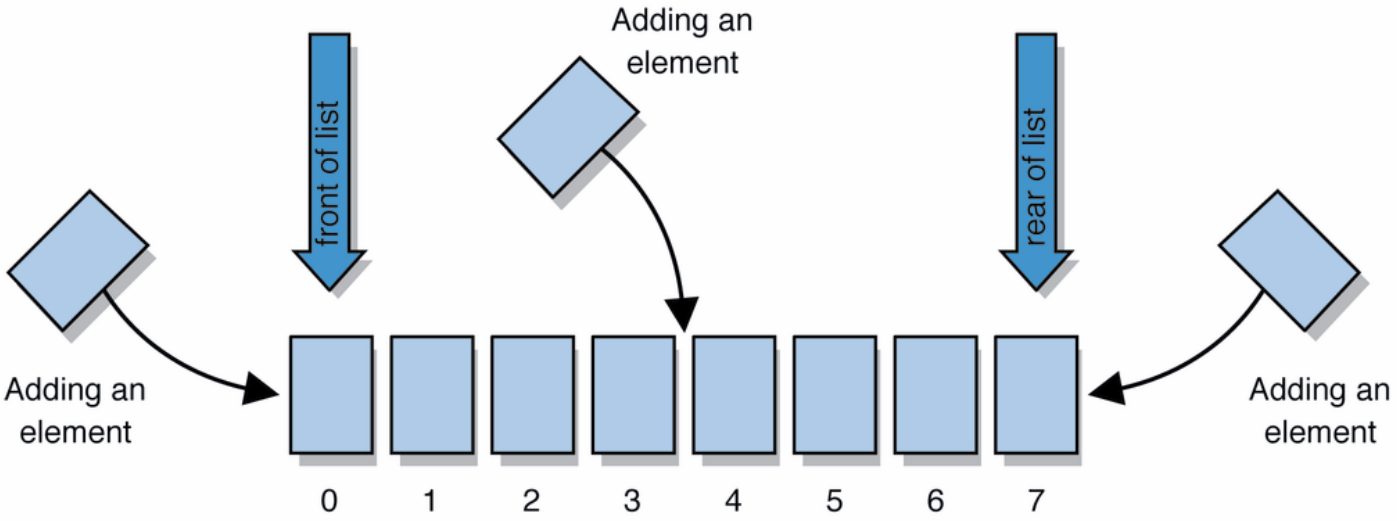
  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`

  - all collections are in the `java.util` package

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

Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **`ArrayList`** object



Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

```
[]
```

- You can add items to the list.
  - The default behavior is to add to the end of the list.

```
[hello, ABC, goodbye, okay]
```

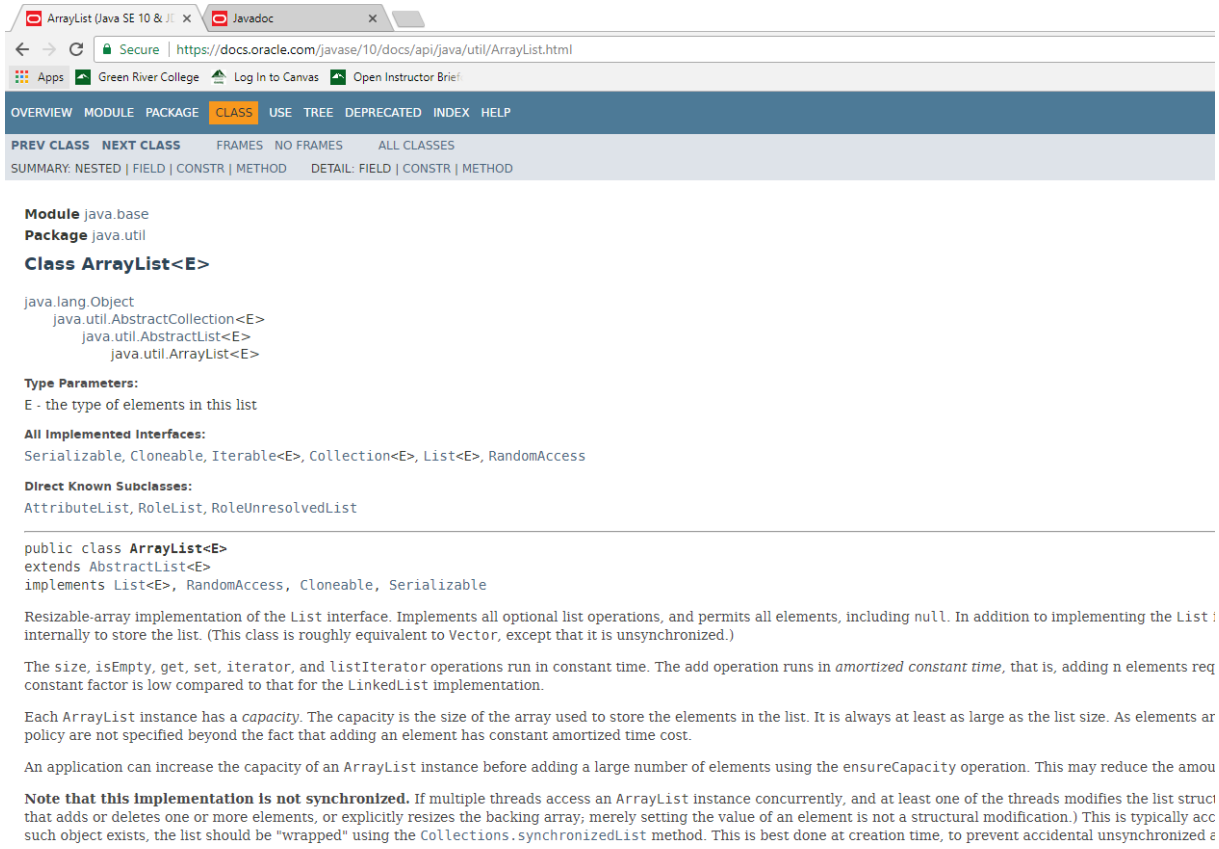
- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

Lists - Interface

- **List** methods
  - add
  - addAll
  - clear
  - contains
  - containsAll
  - get
  - indexOf
  - isEmpty
  - remove
  - removeAll
  - set
  - size
  - Sort
  - toArray
  - subList

Learning about classes

- The Java API Specification is a huge web page containing documentation about every Java class and its methods.



Exercise solution (Better)

```
ArrayList<String> allWords = new ArrayList<String>();
Scanner input = new Scanner(new File("words.txt"));
while (input.hasNext()) {
    String word = input.next();
    allWords.add(word);
}
System.out.println(allWords);

ArrayList<String> temp = new ArrayList<String>();
// remove all “E” words
for (int i = 0; i < allWords.size(); i++) {
    String word = allWords.get(i);
    if (word.toUpperCase().contains("E"))
      {temp.add(word);}
}

allWords.removeAll(temp);
System.out.println(allWords);
```

Collections Tips

- What is wrong with the code below?

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

- It isn't the List vs ArrayList…
- It's the lack of <> on the ArrayList
- The above code will compile, and "may" possibly still work.
- BUT it isn't good code. The right side is using generic objects, so you are losing the good code.

Collections Tips

- What is wrong with the code below?

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

- NOTHING! ☺

- The above code is actually allowed and good.
- As of version 7 Java added the ability for an operator to “look” ahead and see what kind of class is necessary.
- So in this case Java sees that <> should be <String> and puts it in.
- This is called the “diamond” operator.

Collections Tips

- This makes something like this:

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

- A lot easier to type

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

Collections Tips

- Note that the code below, is not considered good code.

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

- Although it is similar to the code on the previous slides, the fact that it is now on two different lines means that this now becomes harder to track.
- In this case you really “should” fill in the diamond yourself.

Reading Moby Dick

- In the book Moby Dick,
  - How many UNIQUE words are there.
  - Not words in total, but different words?
  - How would you do this, using ArrayList
  - How long would it take?
  - Is there a better way to do it.

Pseudo Code

- Open the file
- Start the timer

- Read one word
- Check to see if it is in the list
  - If it is not in the list, add it to the list.
- End the timer

- Find the list of the list

Empirical analysis

Running a program and measuring its performance

`System.currentTimeMillis()`

- Returns an integer representing the number of milliseconds that have passed since 12:00am, January 1, 1970.
  - The result is returned as a value of type `long`, which is like `int` but with a larger numeric range (64 bits vs. 32).
- Can be called twice to see how many milliseconds have elapsed between two points in a program.

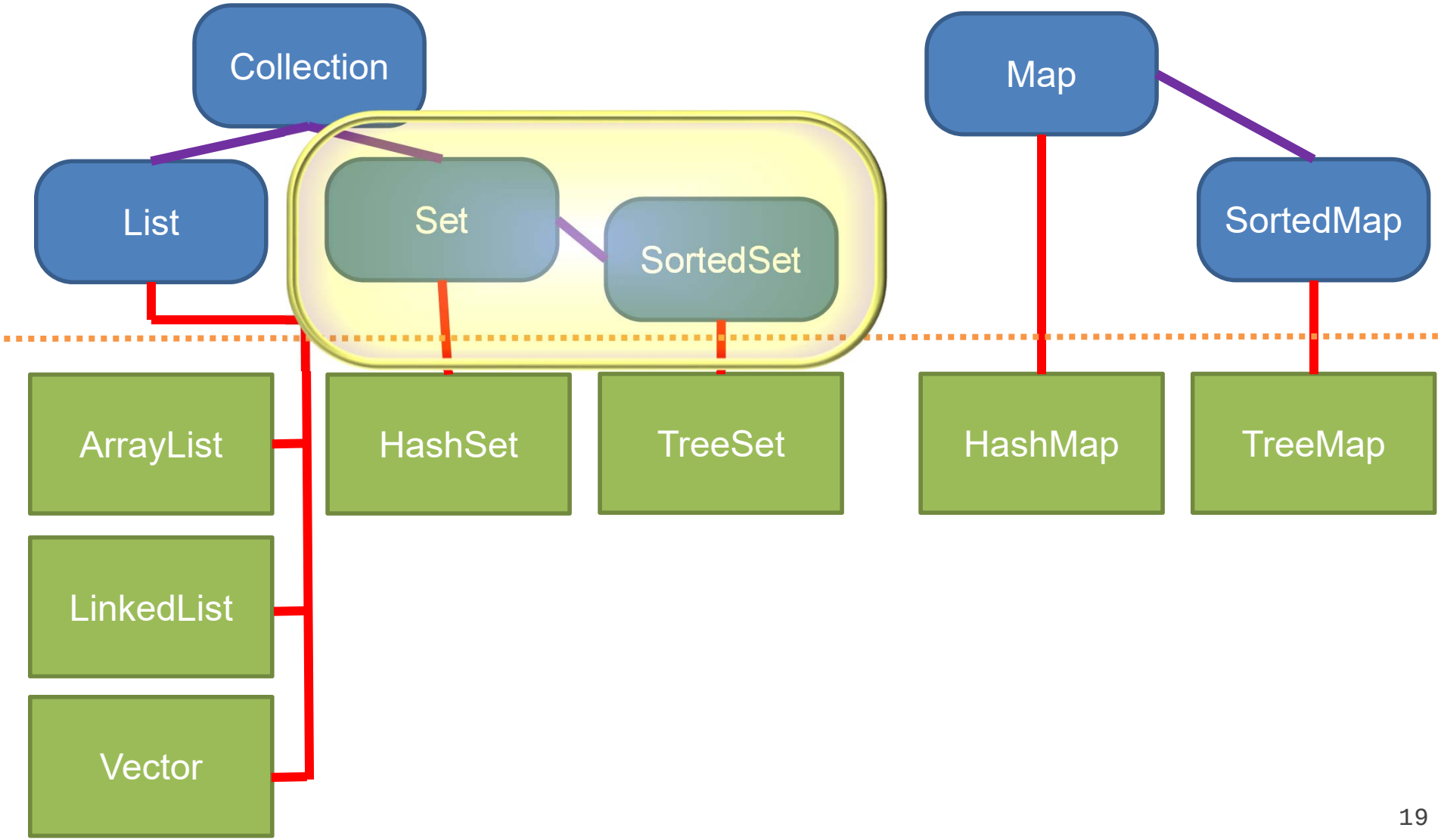
- How much time does it take to store *Moby Dick* into a `List`?

Demo the code

write sample code using ArrayList

SETS

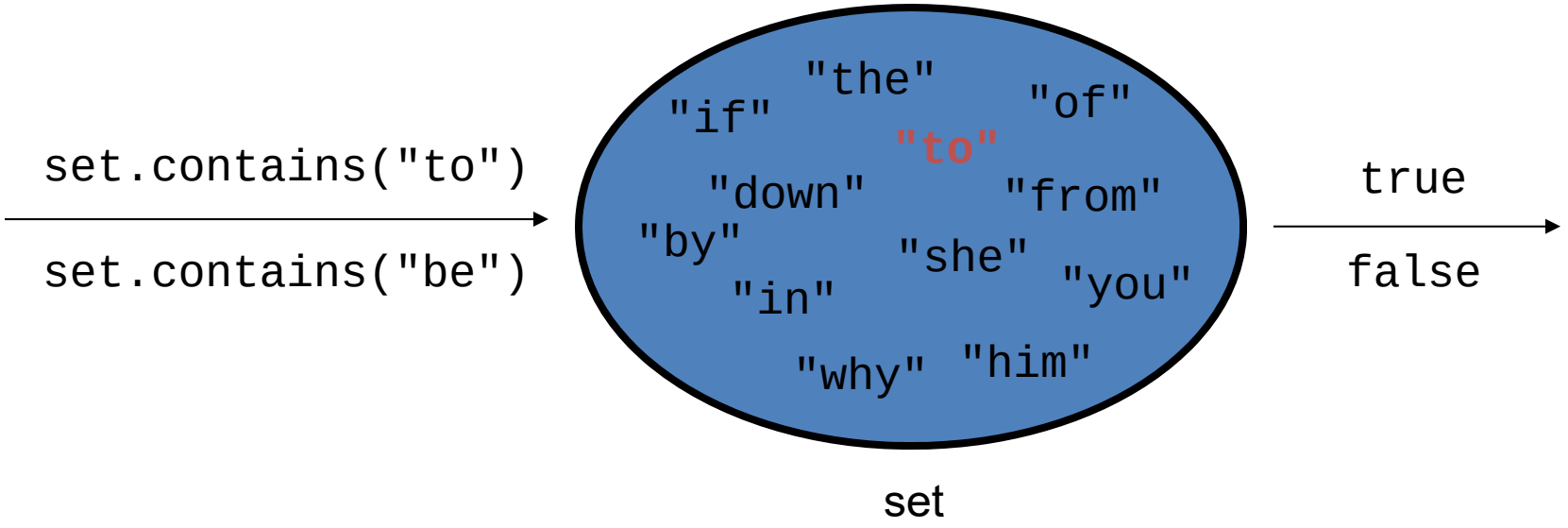
Java Collection Framework



Sets (11.2)

- **set**: A collection of **unique values** (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)

  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



`Set` implementation

- in Java, sets are represented by `Set` interface in `java.util`

- `Set` is implemented by `HashSet` and `TreeSet` classes

  - `HashSet`: implemented using a "hash table" array;
    very fast: **O(1)** for all operations
    elements are stored in unpredictable order

  - `TreeSet`: implemented using a "binary search tree";
    pretty fast: **O(log N)** for all operations
    elements are stored in sorted order

  - `LinkedHashSet`: **O(1)** but stores in order of insertion

Set methods

```
List<String> list = new ArrayList<String>();
…

Set<Integer> setA = new TreeSet<Integer>();
or
Set<Integer> setB = new HashSet<Integer>();


Set<String> setZ = new HashSet<String>(list);
```

- can construct an empty set, or one based on a given collection

`Set` methods

`add(`**value**`)`	adds the given value to the set
`contains(`**value**`)`	returns `true` if the given value is found in this set
`remove(`**value**`)`	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the set's size is 0
`toString()`	returns a string such as `"[3, 42, -7, 15]"`

Set operations

`addAll(`**collection**`)`	adds all elements from the given collection to this set
`containsAll(`**coll**`)`	returns `true` if this set contains every element from given set
`equals(`**set**`)`	returns `true` if given other set contains the same elements
`iterator()`	returns an object used to examine set's contents *(seen later)*
`removeAll(`**coll**`)`	removes all elements in the given collection from this set
`retainAll(`**coll**`)`	removes elements *not* found in given collection from this set
`toArray()`	returns an array of the elements in this set

Sets and ordering

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

- `LinkedHashSet` : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```

The "for each" loop (7.1)

```
for (type name : collection) {
    statements;
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, array, or other collection

```
Set<Double> gradeSet = new HashSet<Double>();
…

for (double oneGrade : gradeSet) {
    System.out.println("Student's grade: " + oneGrade);
}
```

  - needed because sets have no indexes; can't `get` element `i`

Examining sets and maps

- elements of Java `Sets` and `Maps` can't be accessed by index
  - must use a "for each" loop:

```
Set<Integer> scoreSet = new HashSet<Integer>();
for (int aScore : scoreSet) {
    System.out.println("The score is " + aScore);
}
```

  - Problem: for each is read-only; cannot modify set while looping

```
for (int aScore : scores) {
    if (score < 60) {
    // throws a ConcurrentModificationException
        scores.remove(aScore);
    }
}
```

Summary

- Sets
  - Are a collection
  - Are an interface
    - With two primary implementations
      - TreeSet
      - HashSet
  - Are unindexed
    - Insertion order doesn't matter
  - Are significantly faster than lists.
  - TreeSet has a side effect
  - Must use a "for each" loop.
  - Removal should not be done while iterating.