COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

5. Java Collections framework

Aim

Introduce the Java Collections framework (JCF)

Objectives

…

5. Java Collections framework

Outline

- …

Readings

Core Java, vol. 2, chapter 2

Sun Java Tutorial: Java Collections Framework

`http://download.oracle.com/javase/tutorial/collections/index.html`

`http://download.oracle.com/javase/8/docs/technotes/guides/collections/index.html`

Introduction

- A *collection* is an object that groups multiple elements into a single unit.
- Java Collections Framework (JCF) provides a unified framework for data structures and algorithms that separates *implementation* from *interface*.
- Several possible data structures, depending on:
  - Is the collection fixed size or dynamic?
  - Is it an ordered sequence or an unordered set?
  - Do you need to be able to insert and delete items in the collection at arbitrary places, or only at the end?
  - Does the class have to provide a way to easily search through a collection (which may contain millions of items)?
  - Do you need random access to elements, or is sequential access adequate?

- The JCF depends on the idea of abstract classes and interfaces.

ArrayList

- It is not possible to dynamically change the size of an Array in Java.

```
String[] booksInLibrary = String[100]; // <—- what about book #101?
```

- The ArrayList class allows us to
  - add an object `a` using `add(a)`
  - retrieve an object at index i using `get(i)`
  - insert an object `a` at index i using `insert(i,a)`, in this case the indices of the other elements are advanced by 1
  - remove element i using `remove(i)`, in which case the elements with indices > i+1 are decremented, and the size of the ArrayList decreases by 1

- We can use an ArrayList to store objects of any type

ArrayList

Up until Java 1.5 it was possible to use inheritance and the `Object` class to store different types in an ArrayList:

```
List words = new ArrayList();    // ArrayList could store any type
words.add("Java");          // here we add a String
String s = (String)words.get(0); // but we need to cast
Person p = new Person();     // create a Person object
words.add(p);          // add this object to our ArrayList
```

Why is this a bad idea?

Seems the argument need to be cast everytime

ArrayList

Java 1.5 onwards solves this problem using type parameters, so our example becomes:

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

words.add("Java");      // add a String as before
String s = words.get(0);  // no need to cast
```

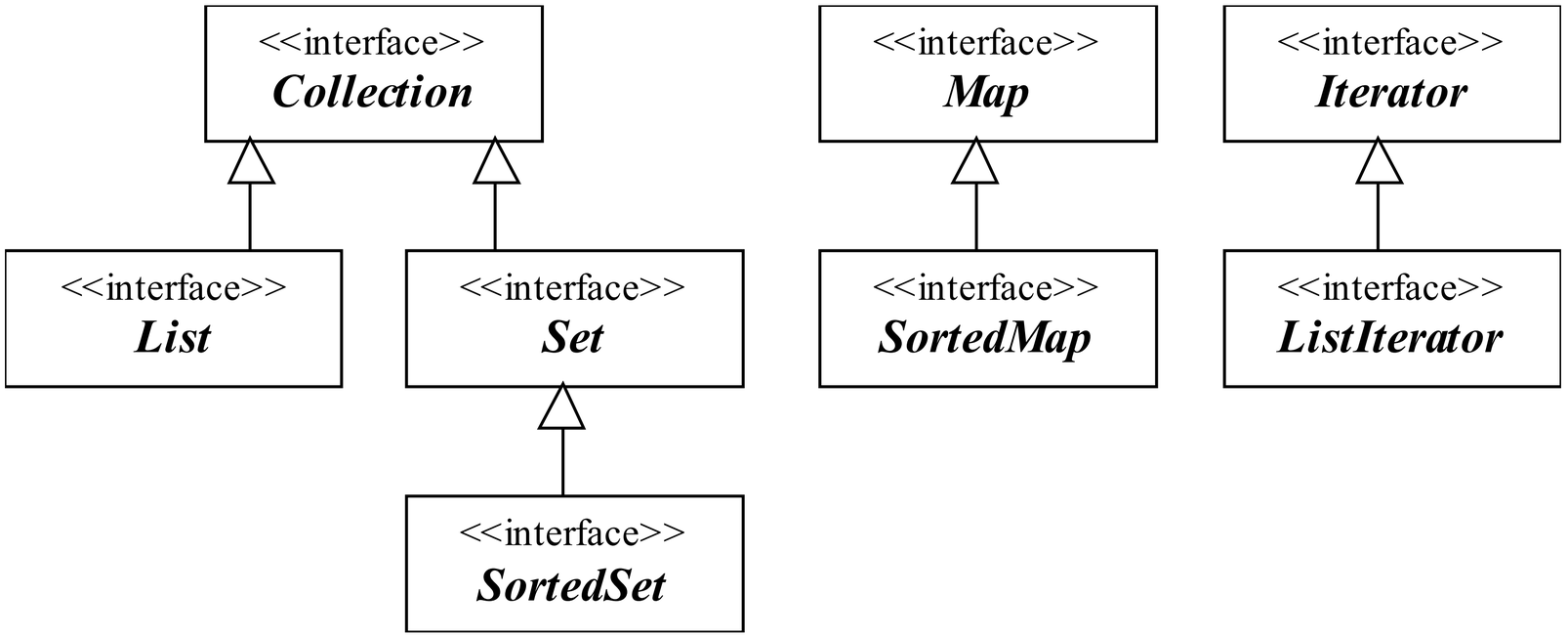
No need for a cast to retrieve data, since only Strings can be stored

The compiler can check that incorrect types are not added

Type parameters make programs **easier to read** and **safer**.

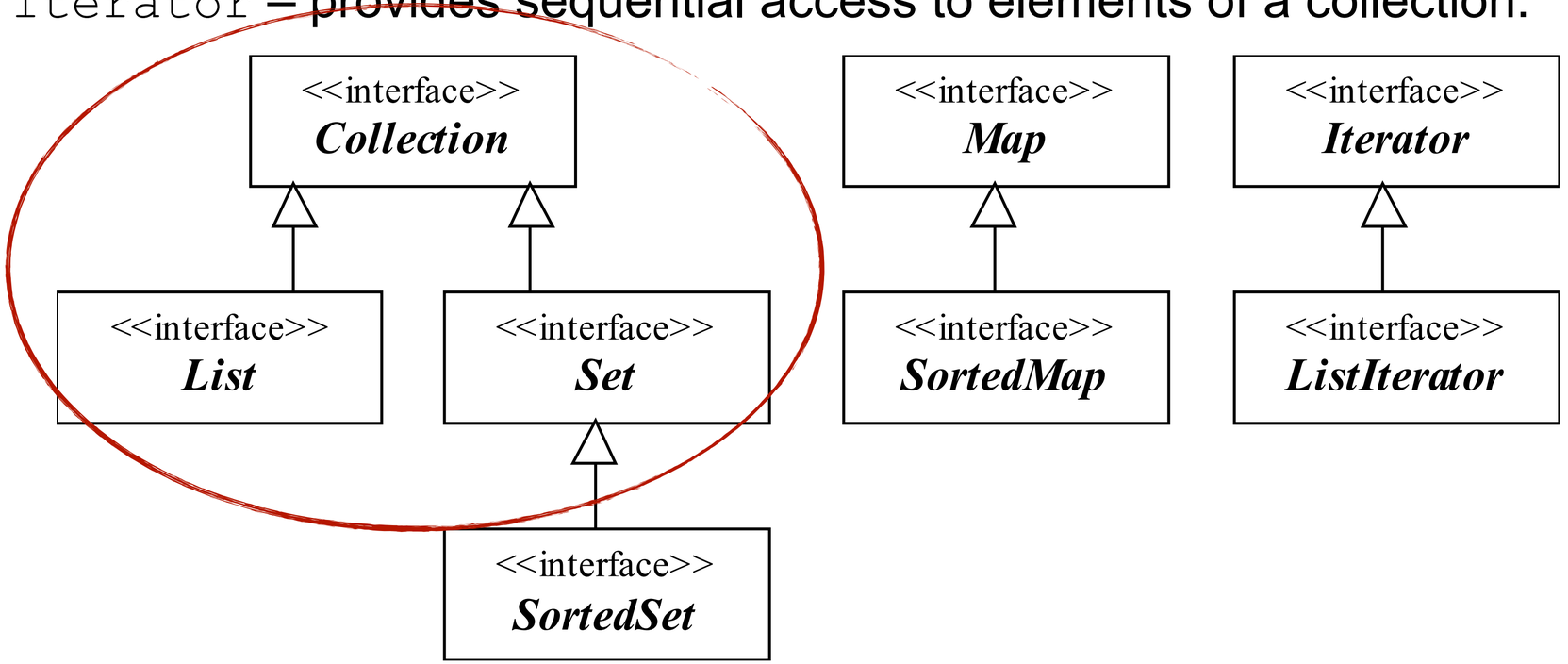
Interfaces in the JCF

- `Collection` – a collection of elements, most general;
- `List` – sorted, can contain duplicate elements;
- `Set` – unsorted, cannot contain duplicate elements;
- `SortedSet` – set that can be iterated through in sorted order;
- `Map` – mapping between key/value pairs, duplicate keys not allowed;
- `Iterator` – provides sequential access to elements of a collection.

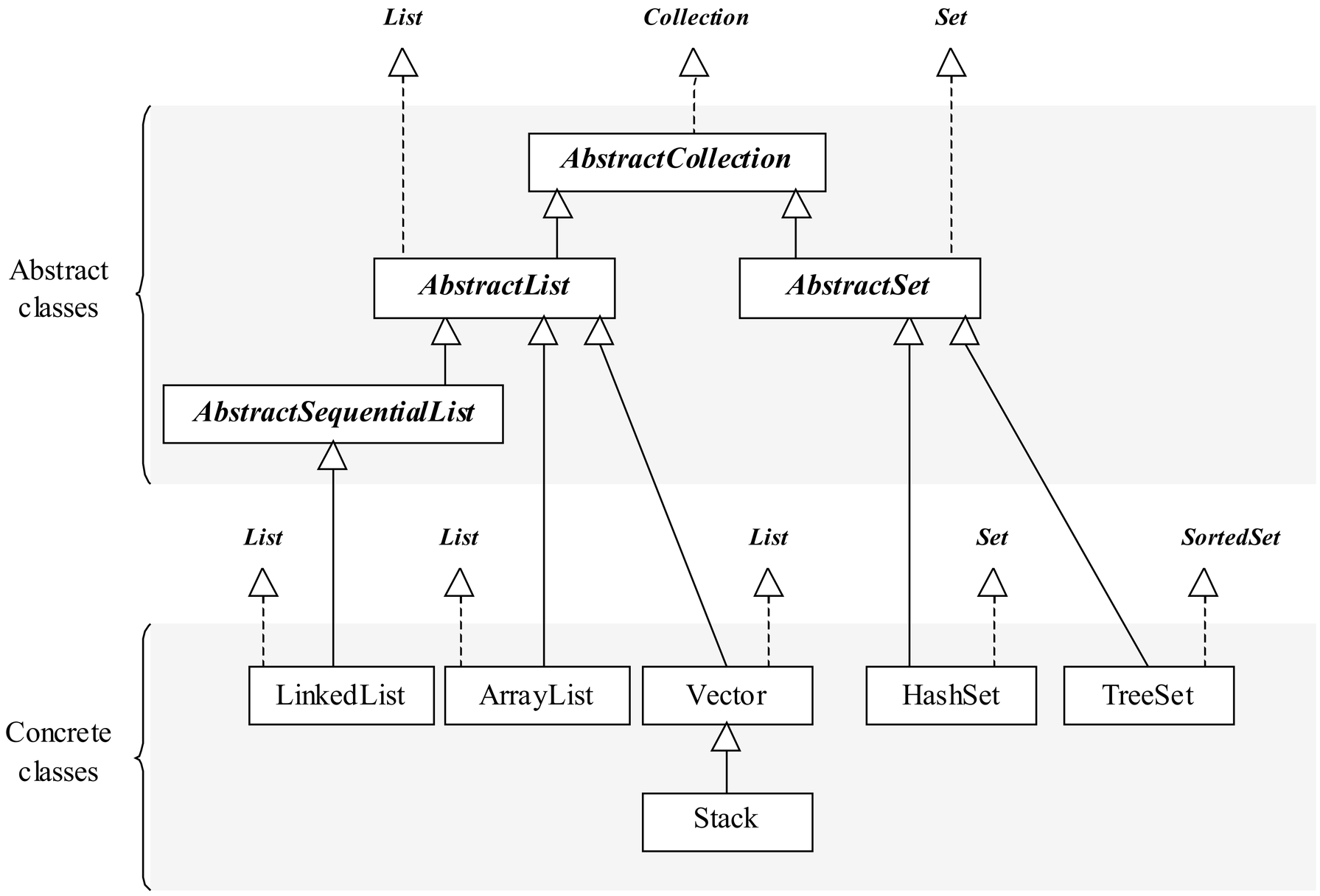


Interfaces in the JCF

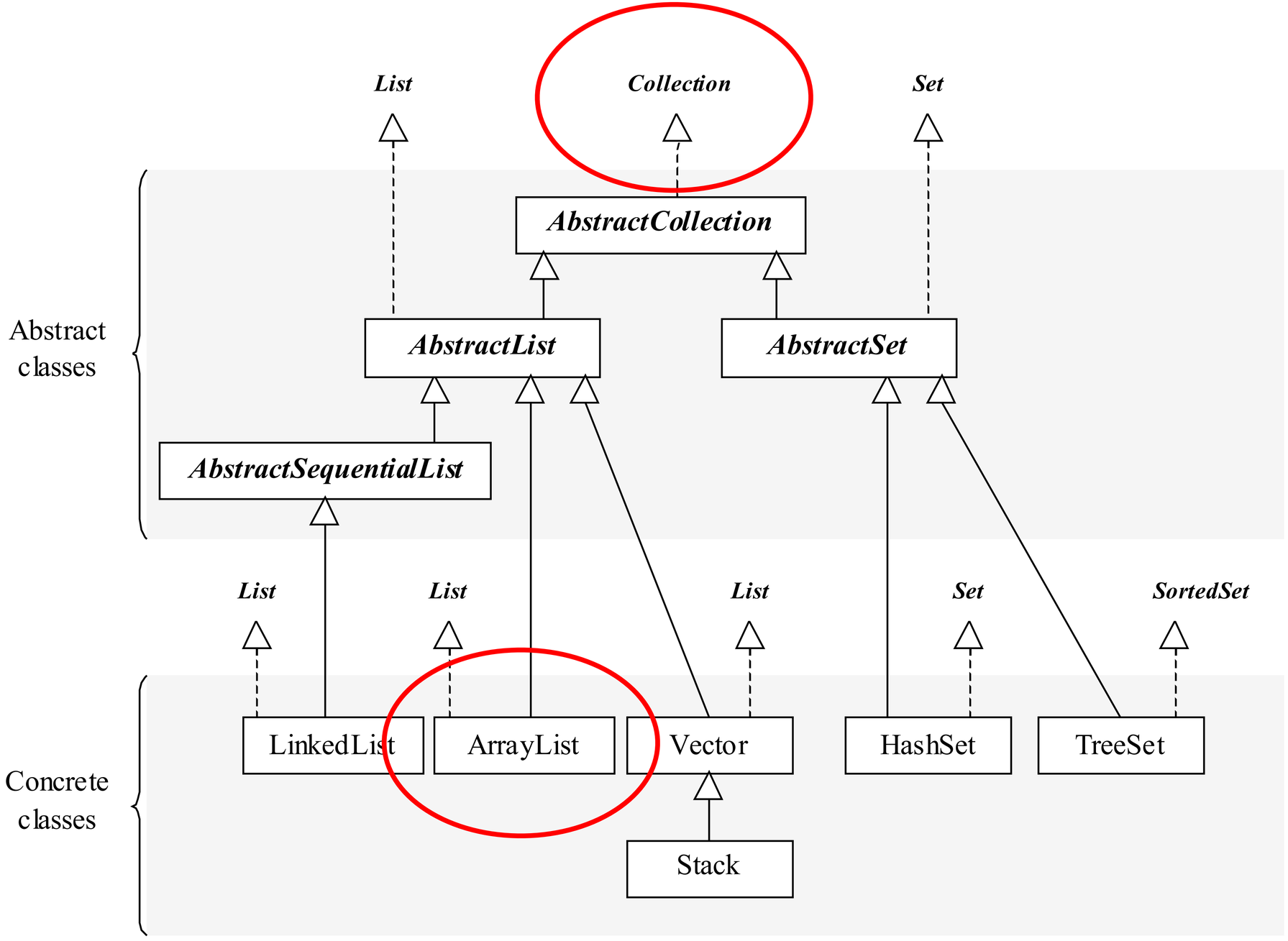
- `Collection` – a collection of elements, most general;
- `List` – sorted, can contain duplicate elements;
- `Set` – unsorted, cannot contain duplicate elements;
- `SortedSet` – set that can be iterated through in sorted order;
- `Map` – mapping between key/value pairs, duplicate keys not allowed;
- `Iterator` – provides sequential access to elements of a collection.



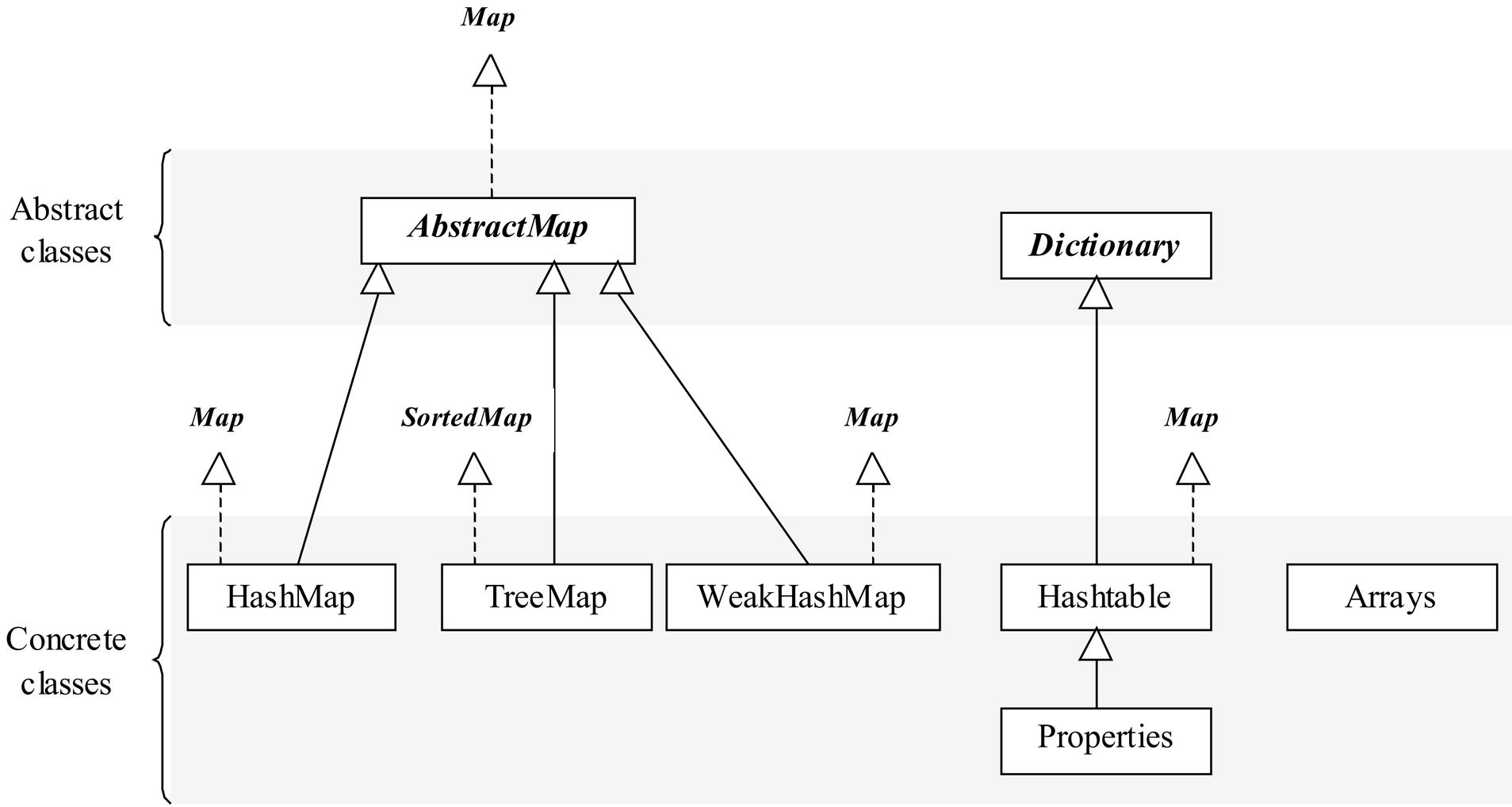
Abstract and concrete classes in the JCF



Abstract and concrete classes in the JCF



Abstract and concrete classes in the JCF



The `Collection` interface

- `Collection` has two fundamental methods:
  - `boolean add(Object obj)` – adds an object to the collection. Returns true if the collection was changed.
  - `Iterator iterator()` – returns an object that implements the `Iterator` interface. 迭代器
- Example – an ArrayList of Strings

```
import java.util.*;
public class CollectionTest1 {
  public static void main(String[] args) {

    Collection<String> authors = new ArrayList<String>();
    authors.add("George Orwell");
    authors.add("JRR Tolkien");
    authors.add("Charles Dickens");
    authors.add("AA Milne");
    System.out.println("Before: "+authors);
    …
```

We could have declared authors to be `ArrayList`, but declaring it as the interface type allows us to change the list type easily at a later date

Type parameters identify the ArrayList as containing Strings

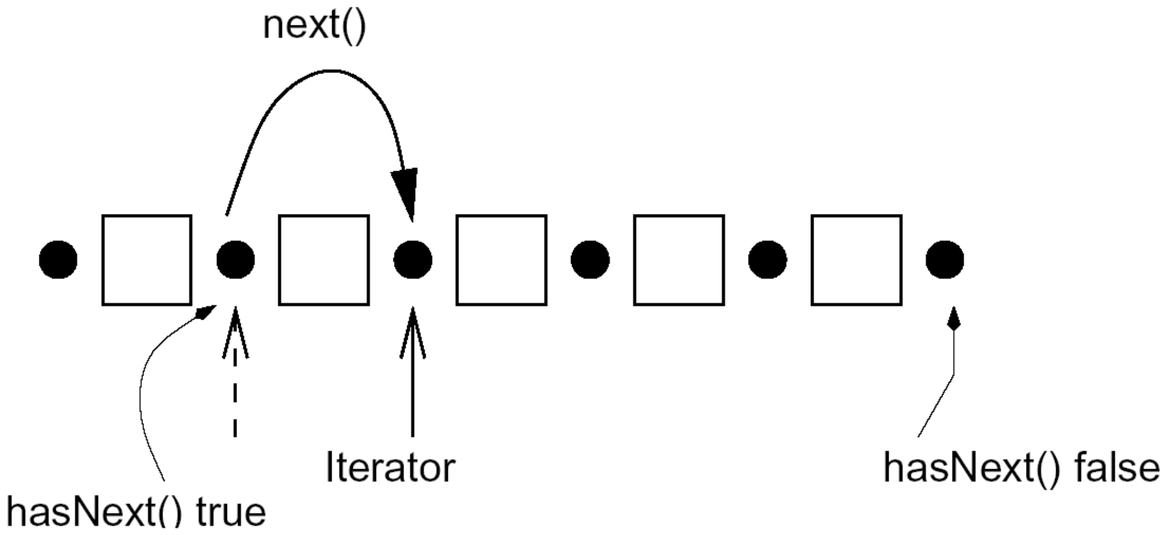
```
Before: [George Orwell, JRR Tolkien, Charles Dickens, AA Milne]
```

The `Iterator` interface

- Iterator has three principal methods:
  - `Object next()` – moves the iterator forward by one position and returns a reference to the element that was jumped over
  - `boolean hasNext()` – returns true if there are still elements to visit.
  - `void remove()` – removes the element that was returned by the last call to `next()`. `remove()` can only be called if the preceding call to the iterator was `next()`.

Best to think of an iterator as being between elements.

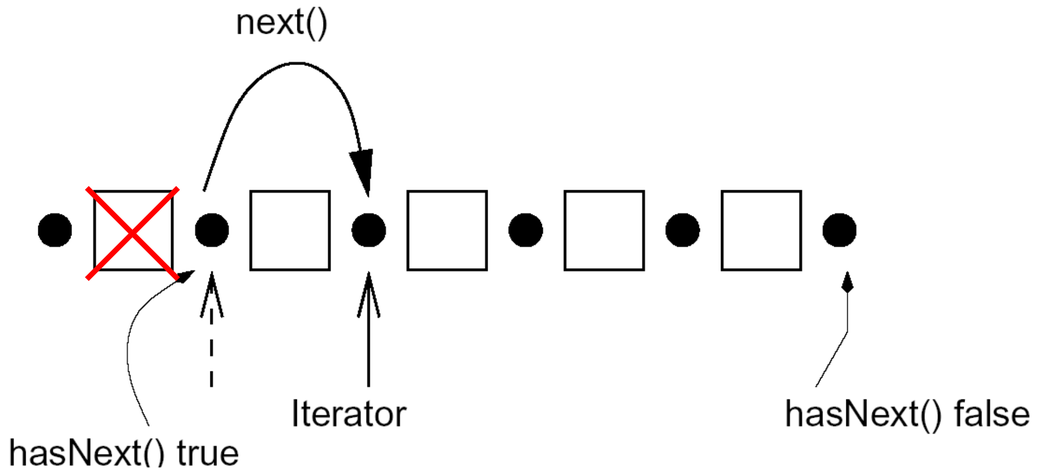
Note that add inserts a new element before the current iterator position.



The Collection interface

```
System.out.println("Before: "+authors);
Iterator aIter = authors.iterator();

while(aIter.hasNext()) {
      String author = (String) aIter.next();
      System.out.println(author);
      if (author.equals("Charles Dickens"
         aIter.remove();
   }

  System.out.println("After: "+authors);
}
}
```

author is the element that has just been jumped over




```
Before: [George Orwell, JRR Tolkien, Charles Dickens, AA Milne]
George Orwell
JRR Tolkien
Charles Dickens
AA Milne
After: [George Orwell, JRR Tolkien, AA Milne]
```

Iterator type

```
  System.out.println("Before: "+authors);
  Iterator aIter = authors.iterator();

  while(aIter.hasNext()) {
        String author = (String) aIter.next();
        System.out.println(author);
        if (author.equals("Charles Dickens"))
           aIter.remove();
     }

    System.out.println("After: "+authors);
  }
}
```

Why do we need a cast here?

Iterator type

```
  System.out.println("Before: "+authors);
  Iterator<String> aIter = authors.iterator();
          Type parameter is added
  while(aIter.hasNext()) {
        String author = aIter.next();
        System.out.println(author);
        if (author.equals("Charles Dickens"))
           aIter.remove();
     }

    System.out.println("After: "+authors);
  }
}
```

Methods in the `Collection` interface

A class implementing `Collection` must supply the following methods:

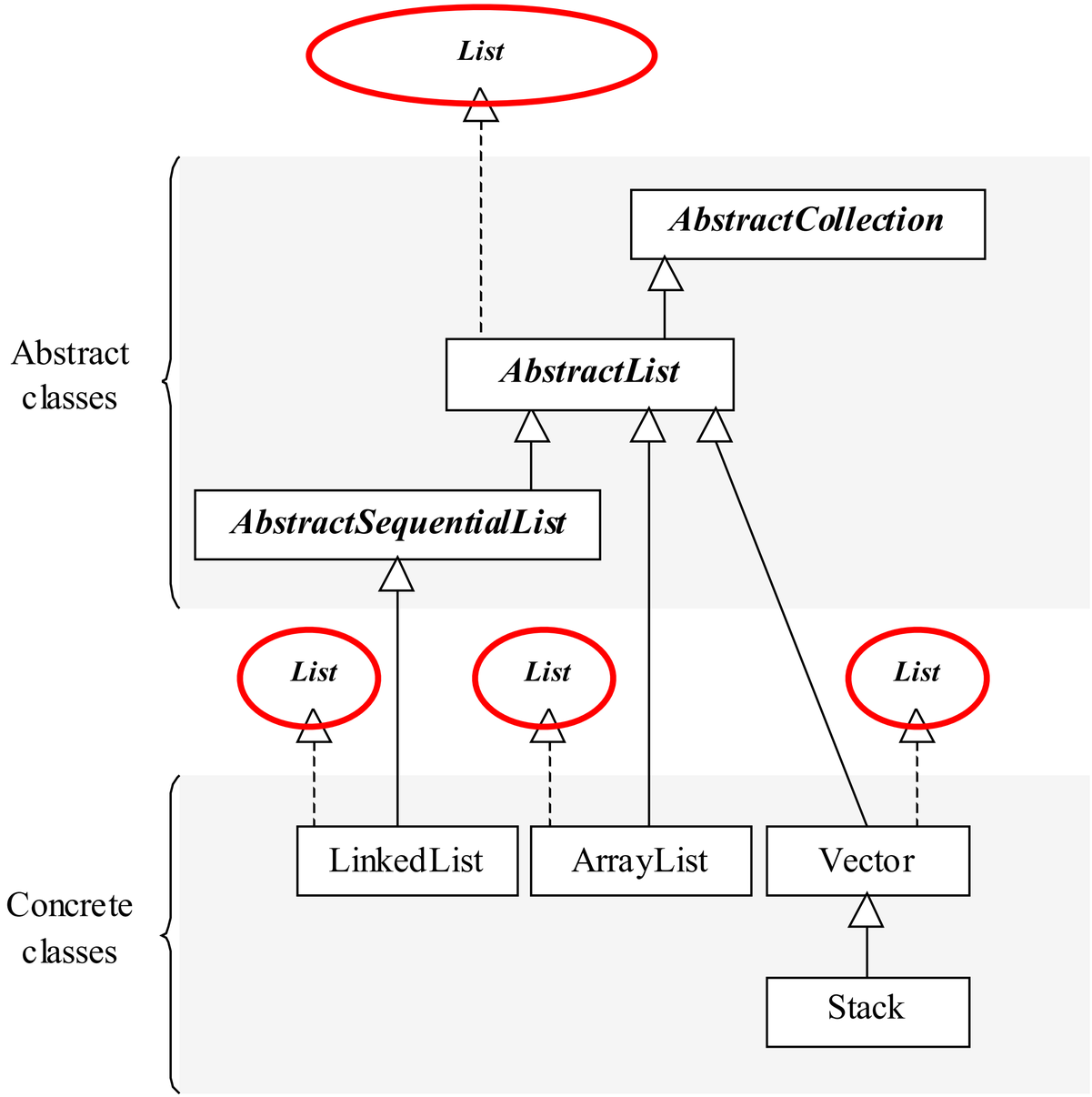
```
int size();
boolean isEmpty();
boolean contains(Object obj);
boolean containsAll(Collection c);
boolean equals(Object obj);
boolean add(Object obj);
boolean addAll(Collection c);
boolean remove(Object obj);
boolean removeAll(Collection c);
void clear();
boolean retainAll(Collection c);
Iterator iterator();
Object[] toArray();
```

For example, ArrayList implements Collection and includes all of these methods http://download.oracle.com/javase/8/docs/technotes/guides/collections/index.html

See also http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

`List` Interface, `ArrayList` and `LinkedList`

Sequences of things may be modelled using collections that implement the `List` interface.



`ArrayList` and `LinkedList`

- `ArrayList` (or `Vector`) appropriate when:
  - Items need to be inserted or deleted at the end of the sequence;
  - Accessing elements by index (e.g. `vec.get(5)`).

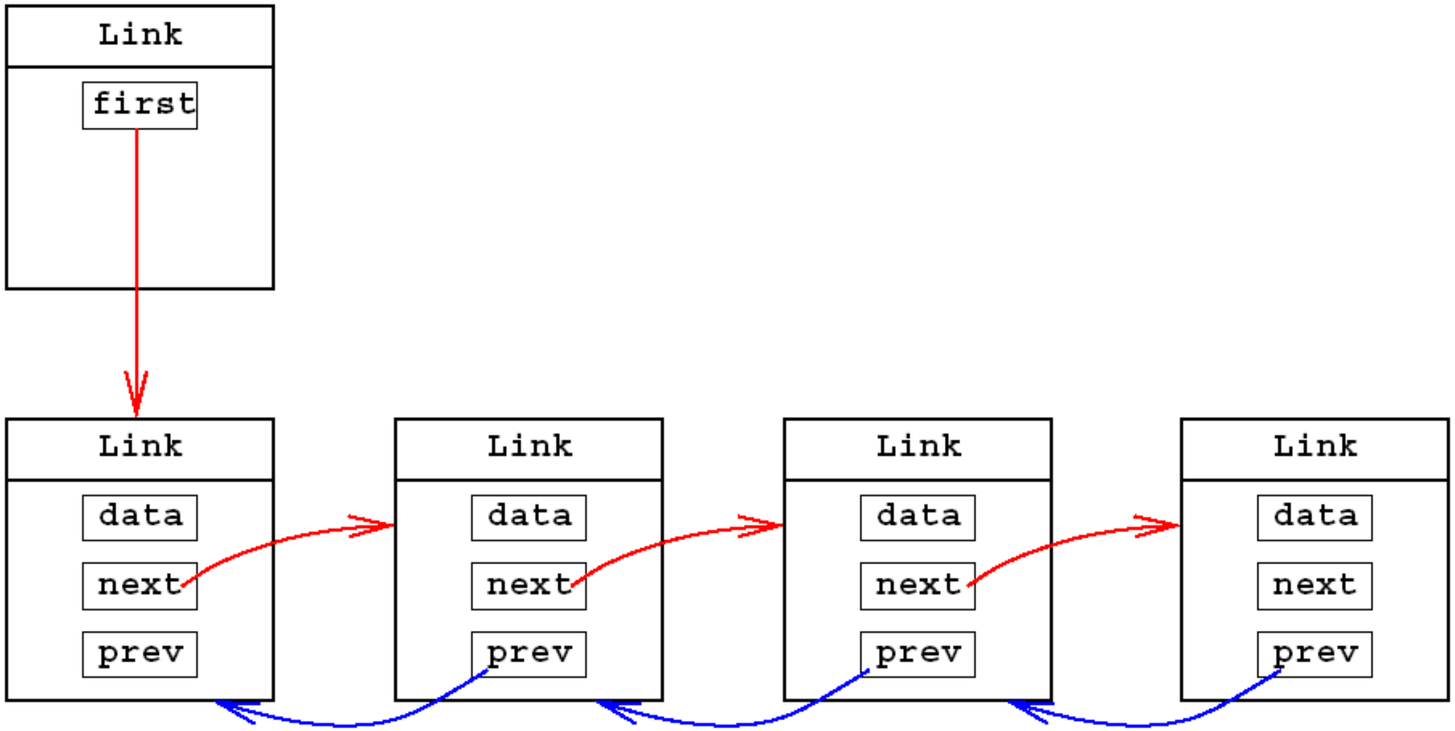
- `LinkedList` appropriate when:
  - Items in the middle of the list need to be inserted or deleted.

- But can use either in place of the other:

```
Collection authors = new ArrayList();
Collection authors = new LinkedList();
```

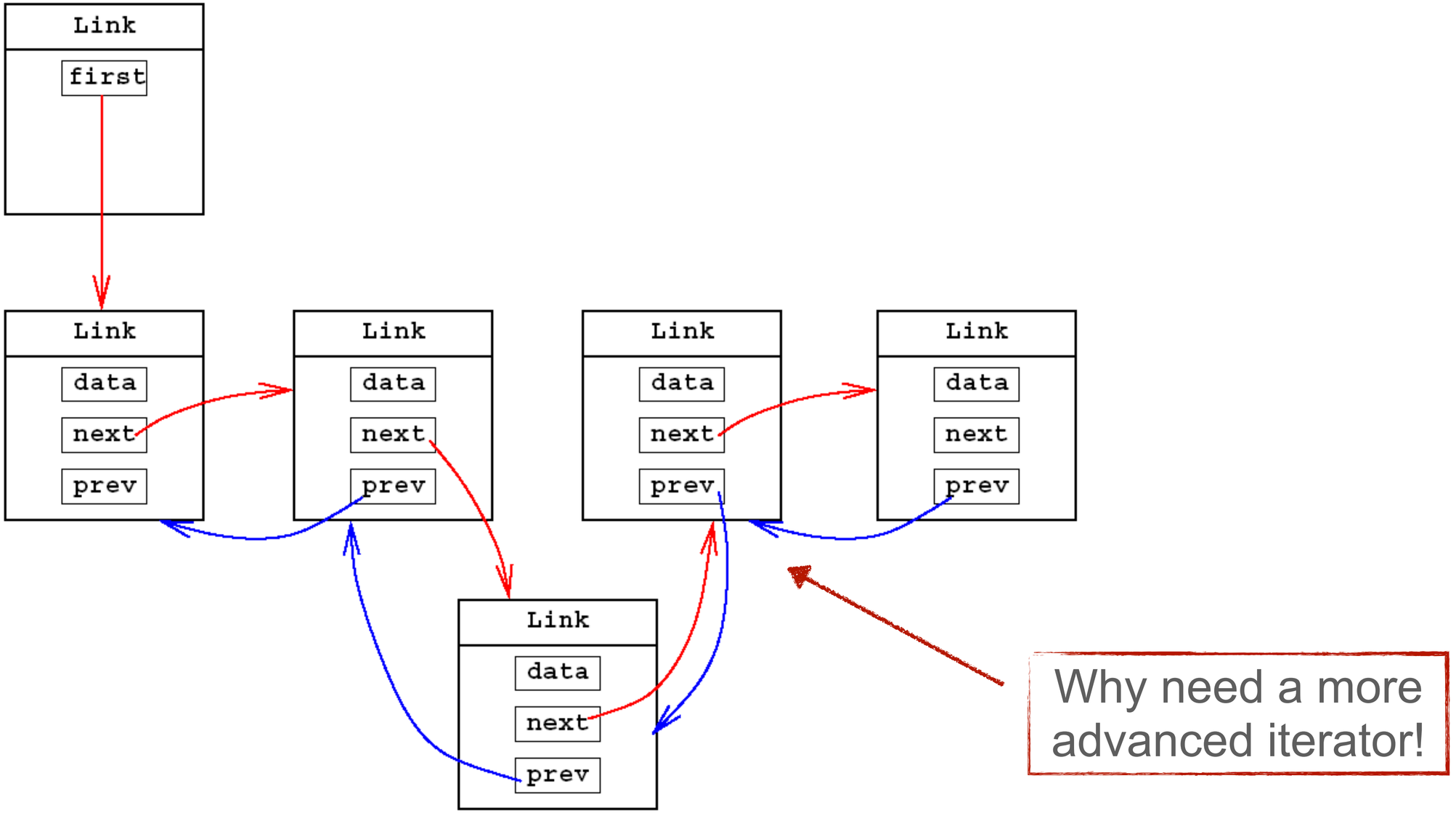
LinkedList

- A `LinkedList` is implemented using a doubly linked list



LinkedList

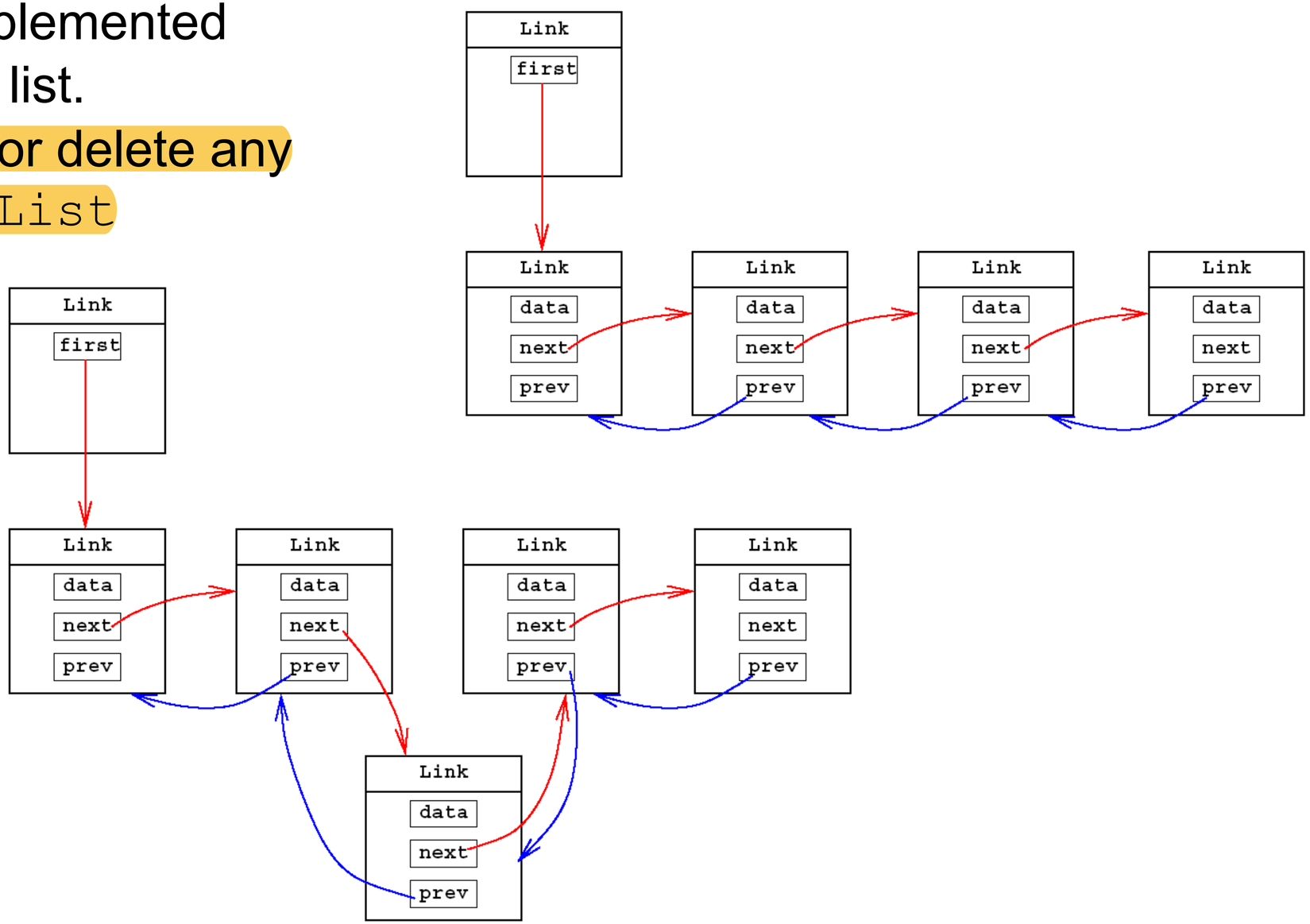
- It is efficient to insert or delete any element in a `LinkedList`



LinkedList

- A `LinkedList` is implemented using a doubly linked list.
- It is efficient to insert or delete any element in a `LinkedList`

add or delete: only need to change the link pointer/reference (next/ prev)



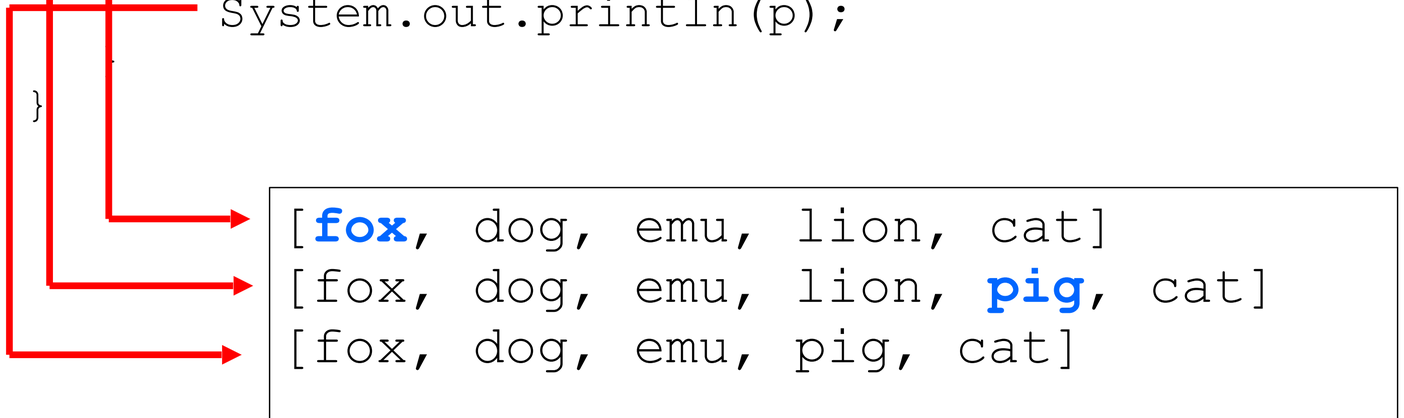
ListIterator

- In classes implementing the `List` interface, a `ListIterator` is provided.
- ListIterator extends `Iterator` by providing an add method and by allowing bidirectional motion
- Calling `remove()` after `previous()` removes the element last passed over.
- LinkedList and ArrayList implement List, which has a listIterator() method.

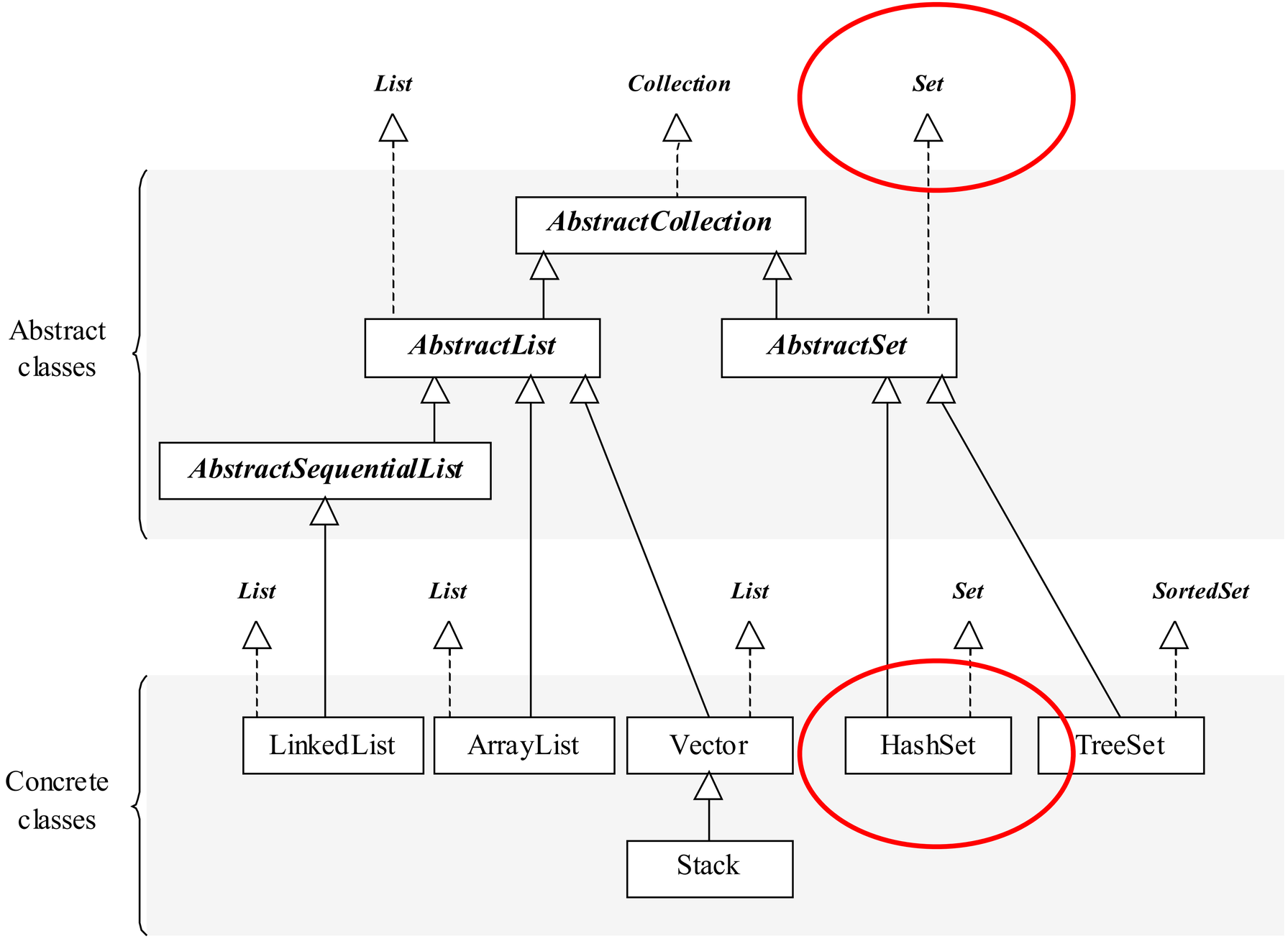
```
interface ListIterator extends Iterator {
  void add(Object);       //  Inserts obj into the list before the
                          //  current iterator position
  Object previous();      //  Support reverse iteration
  boolean hasPrevious(); //  Support reverse iteration
  ...
}
```

ListIterator

```
import java.util.*;
public class EditLinkedList {
   public static void main(String[] args) {
        String[] animals = {"dog","emu","lion","cat"};

        List<String> p = new LinkedList<String>(Arrays.asList(animals));
        ListIterator<String> i = p.listIterator();

        i.add("fox");
        System.out.println(p);
        // calling next() without hasNext() not recommended!!!
        i.next(); i.next(); i.next();
        i.add("pig");
        System.out.println(p);
        // calling previous() without hasPrevious() not recommended!!!
        i.previous(); i.previous();
        i.remove();
        System.out.println(p);

}
```



Abstract and concrete classes in the JCF



Sets: Collections accessed by content

- These data structures are optimised for access by content, e.g. quickly finding if a particular word is a member of the set or not.
- Hash tables are key-value pairs of information
- The hash code is a unique integer representing each object, which is the index where the object is stored.
  - e.g for a small telephone directory

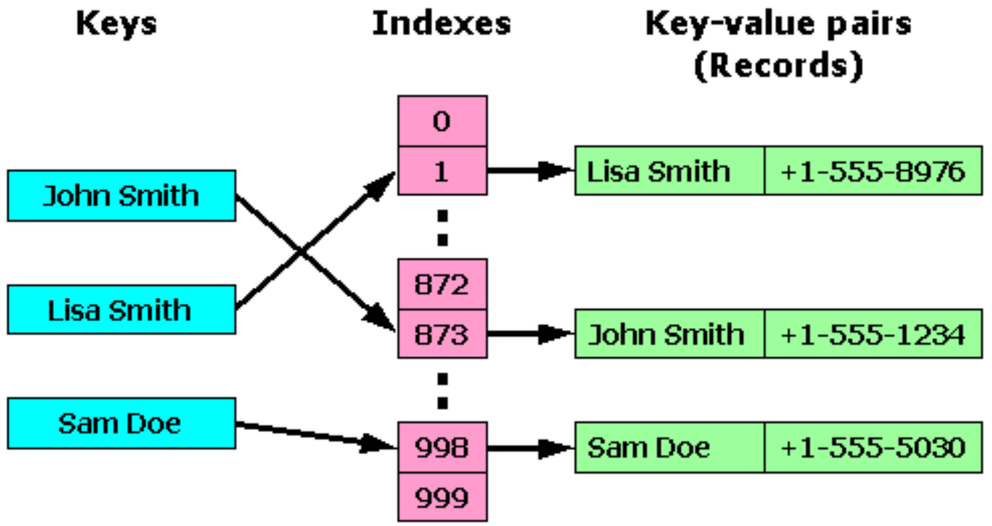


image from http://en.wikipedia.org/wiki/Hash_table

Sets: Collections accessed by content

- `HashSet` (based on a hash table) implements `Set`, and `TreeSet` implements `SortedSet`
- These data structures are optimised for access by content, e.g. quickly finding if a particular word is a member of the set or not.
- *Example*: create a set of all words used in a text.

```
Collection inputText;
...
Set wordSet = new HashSet();
String wd;
for(Iterator i = inputText.iterator(); i.hasNext();) {
  wd = (String) i.next();
  wordSet.add(wd);    // only adds unique words
}
```

- To use `HashSet`, your class must supply `hashCode` and `equals` methods

HashSet

`hashCode` works out where to insert an Object into the `HashSet`.

Default `hashCode` method (inherited from `Object`) returns a hash code directly related to the memory address of the object. Must override.

`hashCode` is usually a function of the object's contents, e.g. for the `Person` class:

```
public int hashCode() {
  return 13*surname.hashCode() +
         17*firstName.hashCode() +
         11*age;
}
```

HashSet

Default `equals` method (inherited from Object) returns true only if the reference of two objects is the same. Must override.

An equal object is not null, is of the same class and has the same attribute values, e.g. for the Person class:

```
public boolean equals(Object obj) {
   if (obj == null || getClass() != obj.getClass())
     return false;
   Person p = (Person) obj;
   return surname.equals(p.surname) && firstName.equals(p.firstName)&& age == p.age;
}
```

TreeSet

`TreeSet` implements `SortedSet`, a subinterface of `Set`.

A `TreeSet` is a sorted collection:

Elements may be inserted in any order;

Iterating through the collection returns them in sorted order.

Objects stored in a `TreeSet` must implement *Comparator* interface, so must provide a *compare* method.

```
TreeSet wds = new TreeSet();
wds.add("Java");
wds.add("C");
wds.add("C++");
wds.add("Scheme");
wds.add("lisp");
for (Iterator iter = wds.iterator(); iter.hasNext();)
  System.out.println(iter.next());
```

```
C
C++
Java
Scheme
lisp
```

for loops and collections

- Using a while loop to iterate a list

```
Collection<String> c = …
Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
  String element = iter.next();
  // do something with element
}
```

- Shortcut for this (Java 1.5 onwards):

```
Collection<String> c = …
for (String element : c) {
  // do something with element
}
```

Summary

A collection is an object that groups multiple elements into a single unit.

The Java Collections Framework is a generic framework in which interface is separated from implementation.

The `Collection` interface specifies the operations that should be applicable to any collection.

The `Iterator` interface is a generic way to step through a collection.

ListIterator

Collection has two subinterfaces: List and Set. A List is ordered and can contain duplicate elements, whereas a Set cannot contain duplicates.