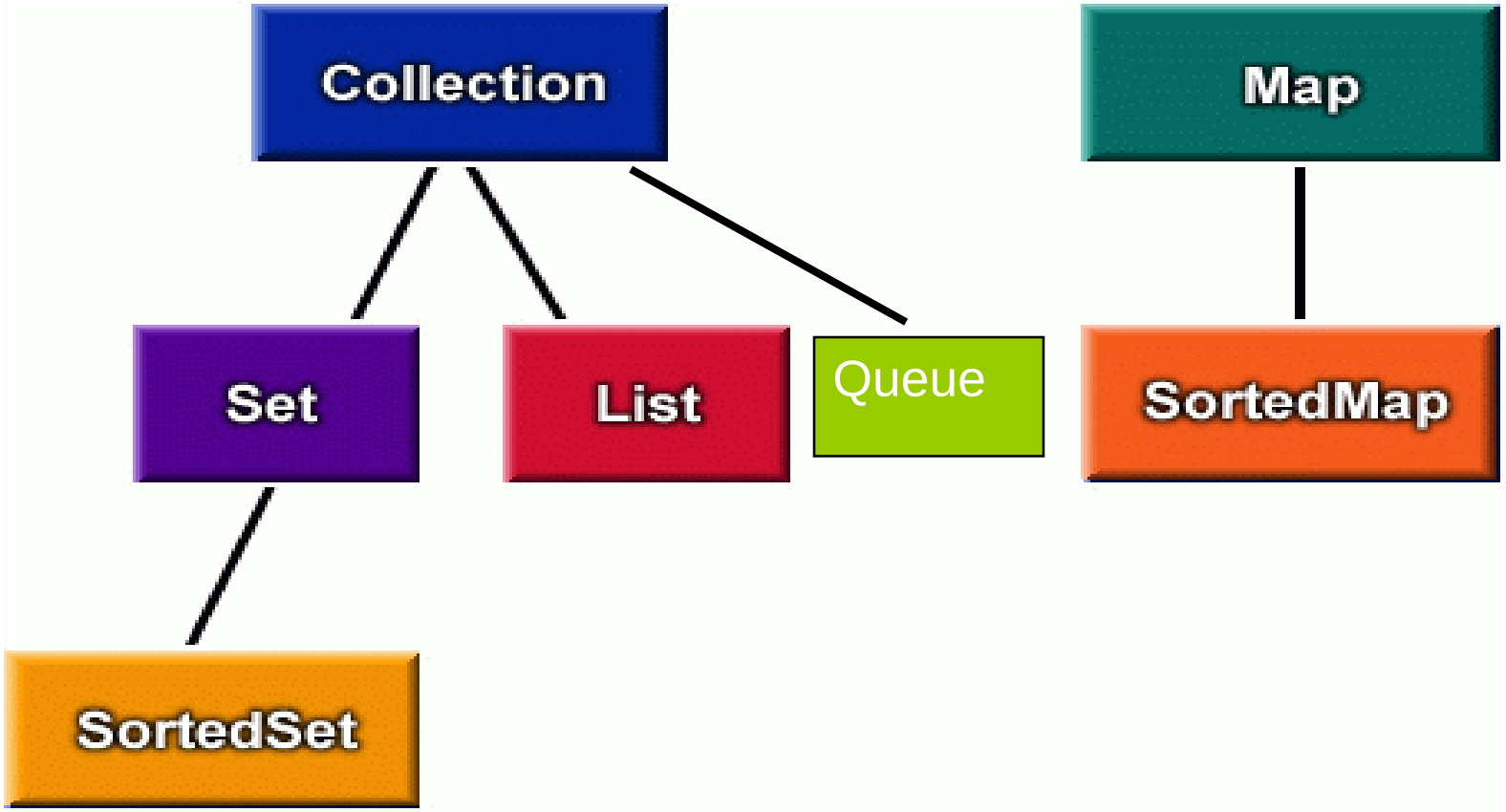
Java Collection Framework

Collection Framework

- A *collection framework* is a unified architecture for representing and manipulating collections. It has:
  - **Interfaces:** abstract data types representing collections
  - **Implementations:** concrete implementations of the collection interfaces
  - **Algorithms:** methods that perform useful computations, such as searching and sorting
    - These algorithms are said to be *polymorphic*: the same method can be used on different implementations

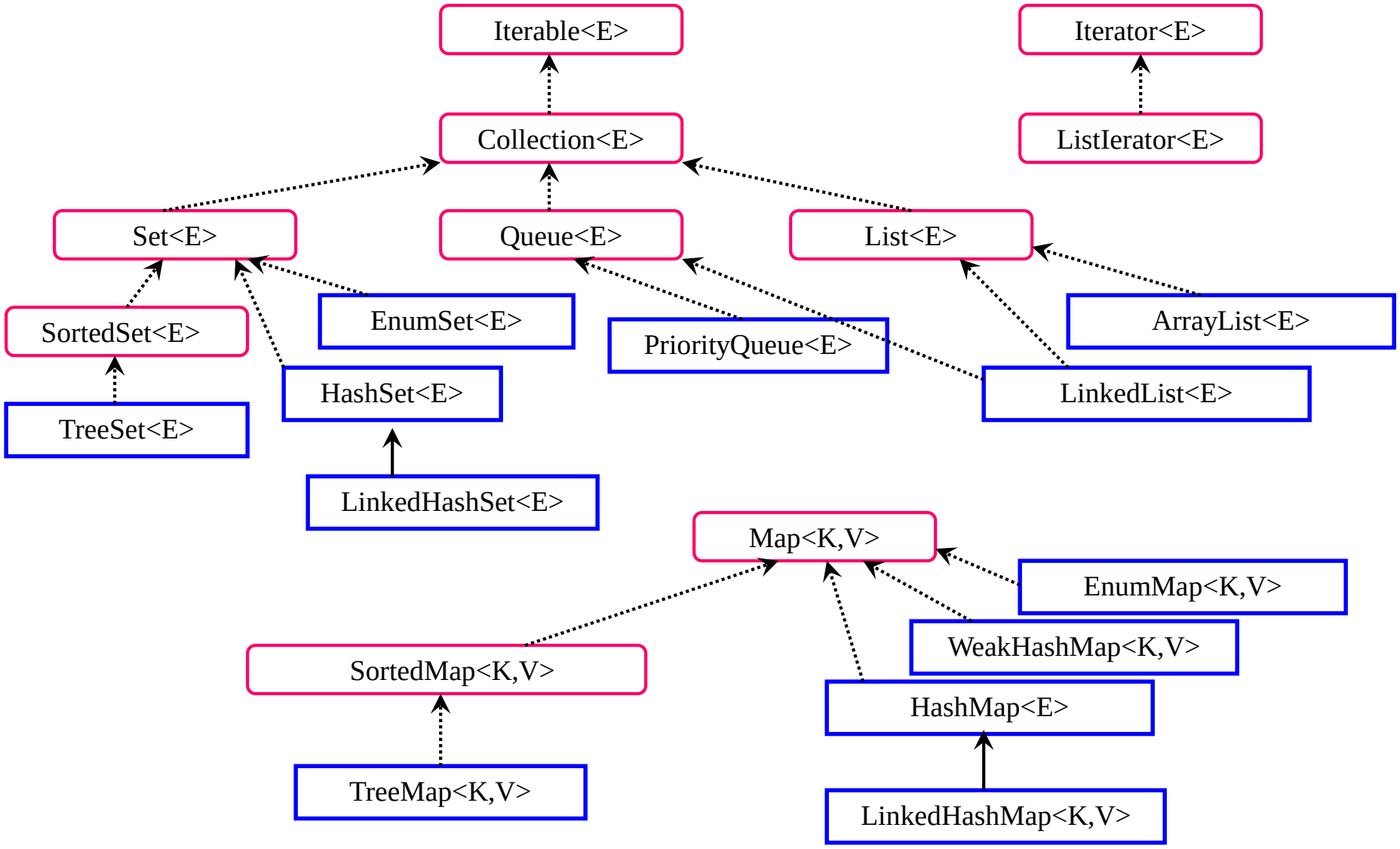
Collection interfaces



Collection Interface continued

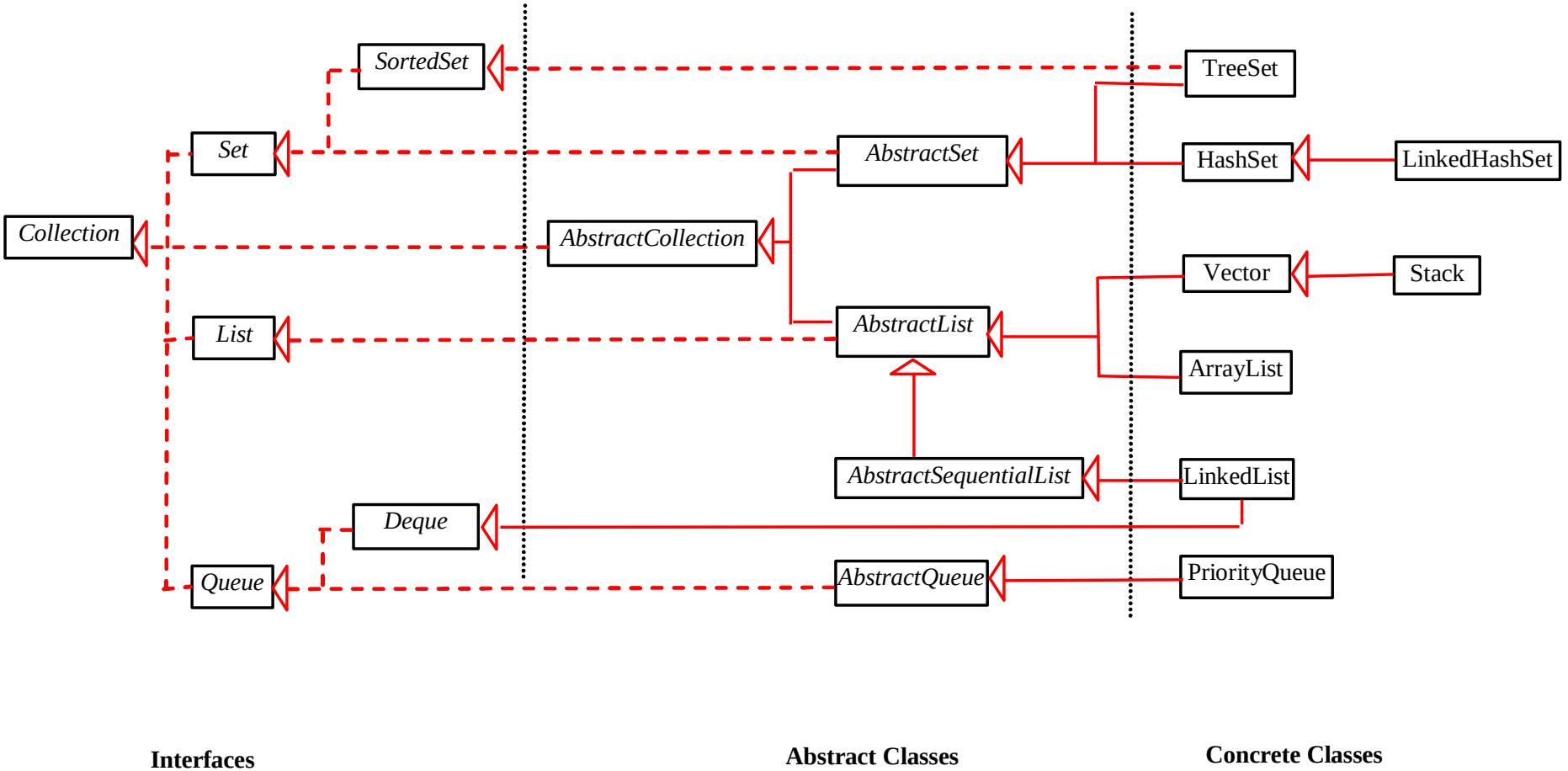
- **Set →**
  - The familiar set abstraction.
  - No duplicates; May or may not be ordered.
- **List →**
  - Ordered collection, also known as a sequence.
  - Duplicates permitted; Allows positional access
- **Map →**
  - A mapping from keys to values.
  - Each key can map to at most one value (function).
  - The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.
- **Queue →**
  - Ordered collection. FIFO (First In First Out)

Type Trees for Collections

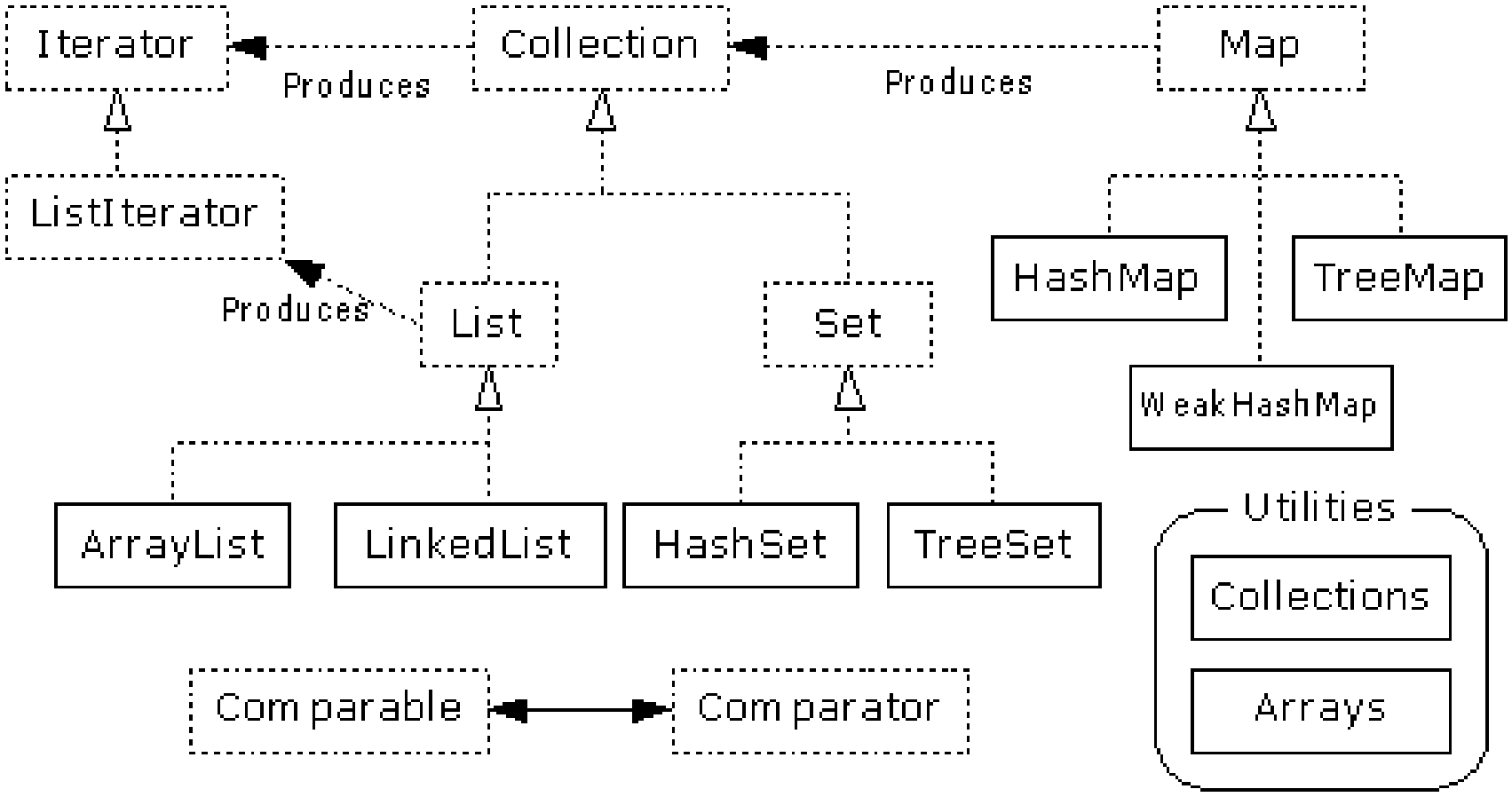


Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.



Collections Framework Diagram



Collection Interface

- Defines fundamental methods
  - **int size();**
  - **boolean isEmpty();**
  - **boolean contains(Object element);**
  - **boolean add(Object element); // Optional**
  - **boolean remove(Object element); // Optional**
  - **Iterator iterator();**
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

Interface `Collection`

- `add(o)` Add a new element
- `addAll(c)` Add a collection
- `clear()` Remove all elements
- `contains(o)` Membership checking.
- `containsAll(c)` Inclusion checking
- `isEmpty()` Whether it is empty
- `iterator()` Return an iterator
- `remove(o)` Remove an element
- `removeAll(c)` Remove a collection
- `retainAll(c)` Keep the elements
- `size()` The number of elements

Iterator Interface

- Defines three fundamental methods
  - **Object next()**
  - **boolean hasNext()**
  - **void remove()**
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() “reads” an element from the collection
  - Then you can use it or remove it

Iterator Position

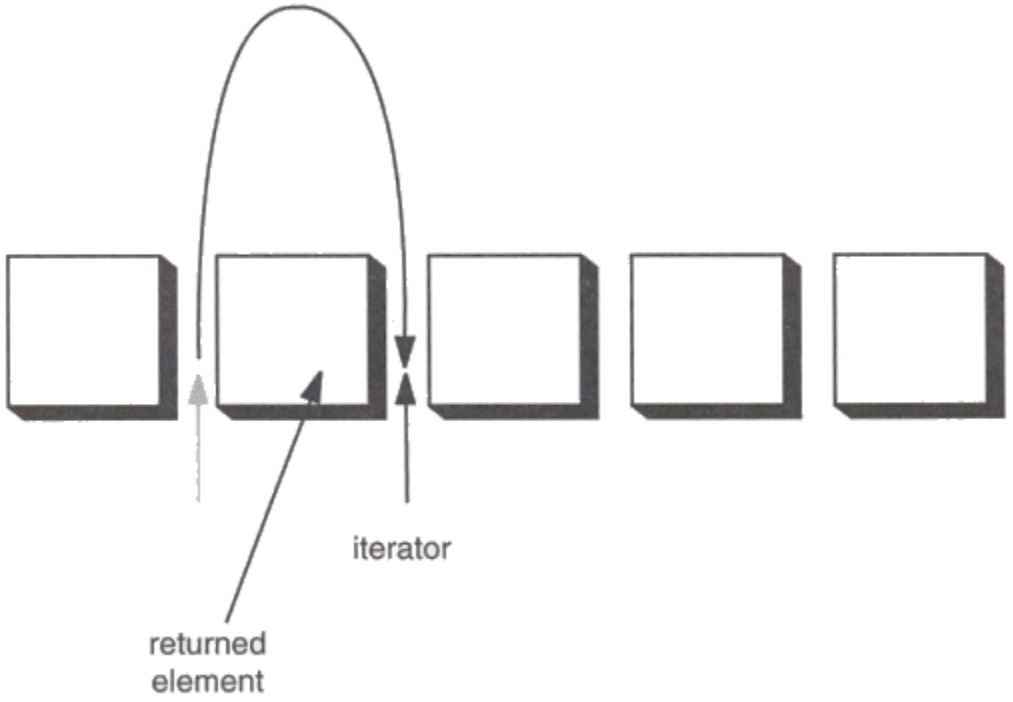
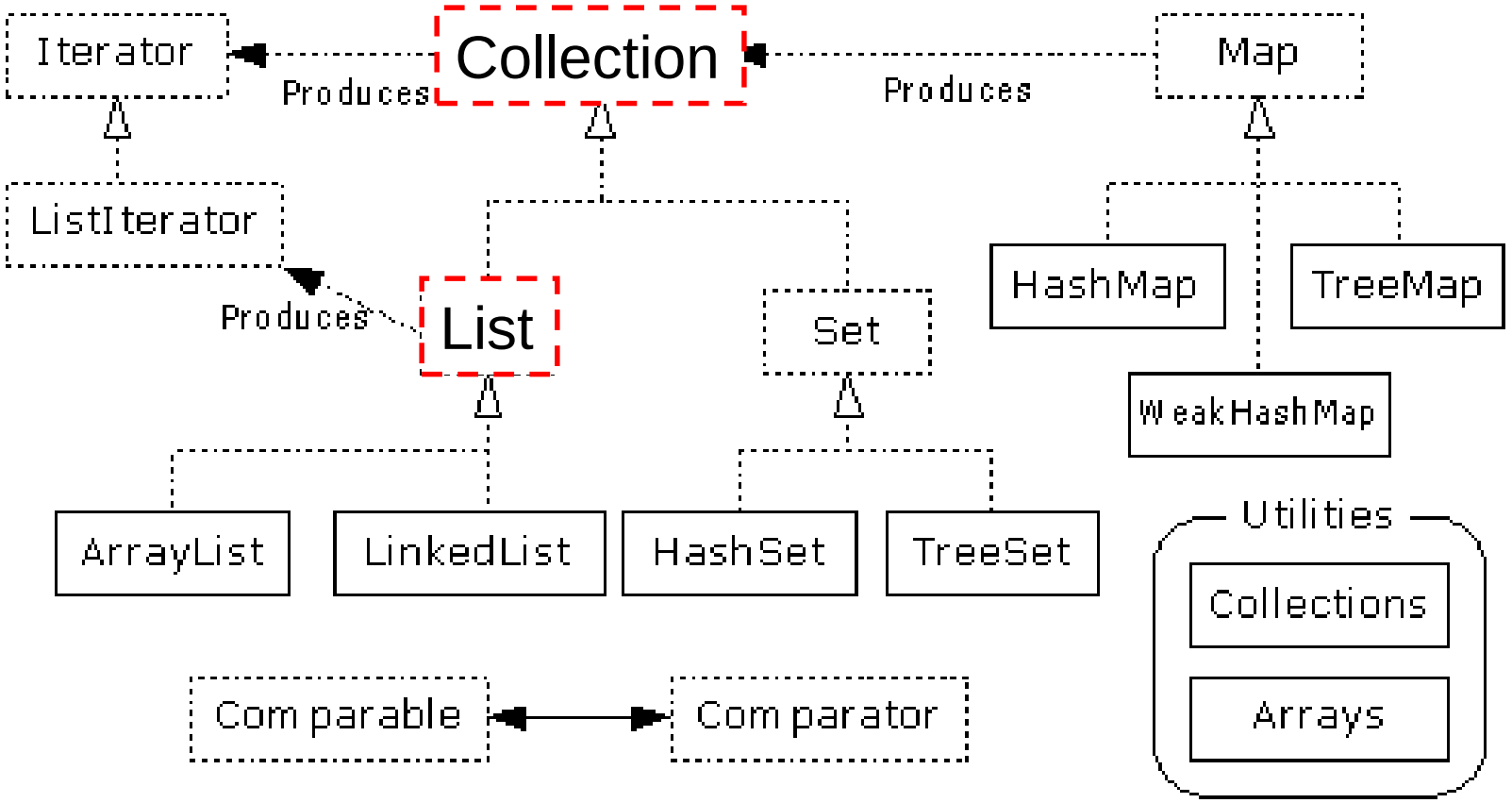


Figure 2–3: Advancing an iterator

Example - SimpleCollection

```
public class SimpleCollection  {
    public static void main(String[] args) {
        Collection c;
        c = new ArrayList();
        System.out.println(c.getClass().getName());
        for (int i=1; i <= 10; i++) {
                c.add(i + " * " + i + " = "+i*i);
        }
        Iterator iter = c.iterator();
        while (iter.hasNext())
                System.out.println(iter.next());
}}
```

List Interface Context



List Interface

- The List interface adds the notion of *order* to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow *duplicate* elements
- Provides a ListIterator to step through the elements in the list.

ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
  - **void add(Object o) -** before current position
  - **boolean hasPrevious()**
  - **Object previous()**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

Iterator Position - `next()`, `previous()`

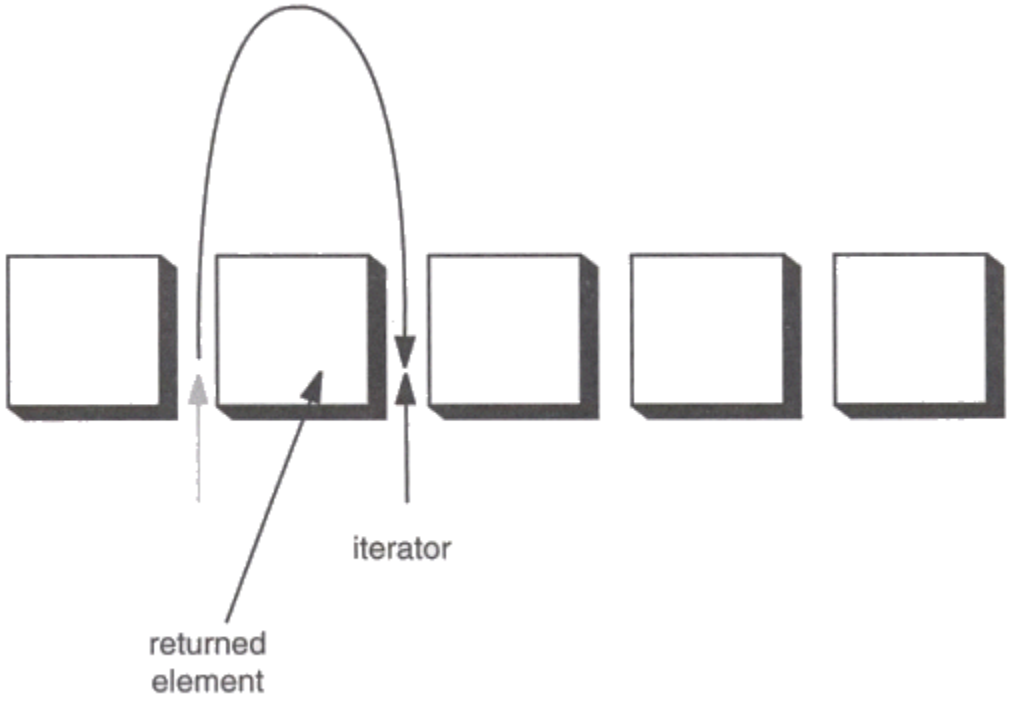
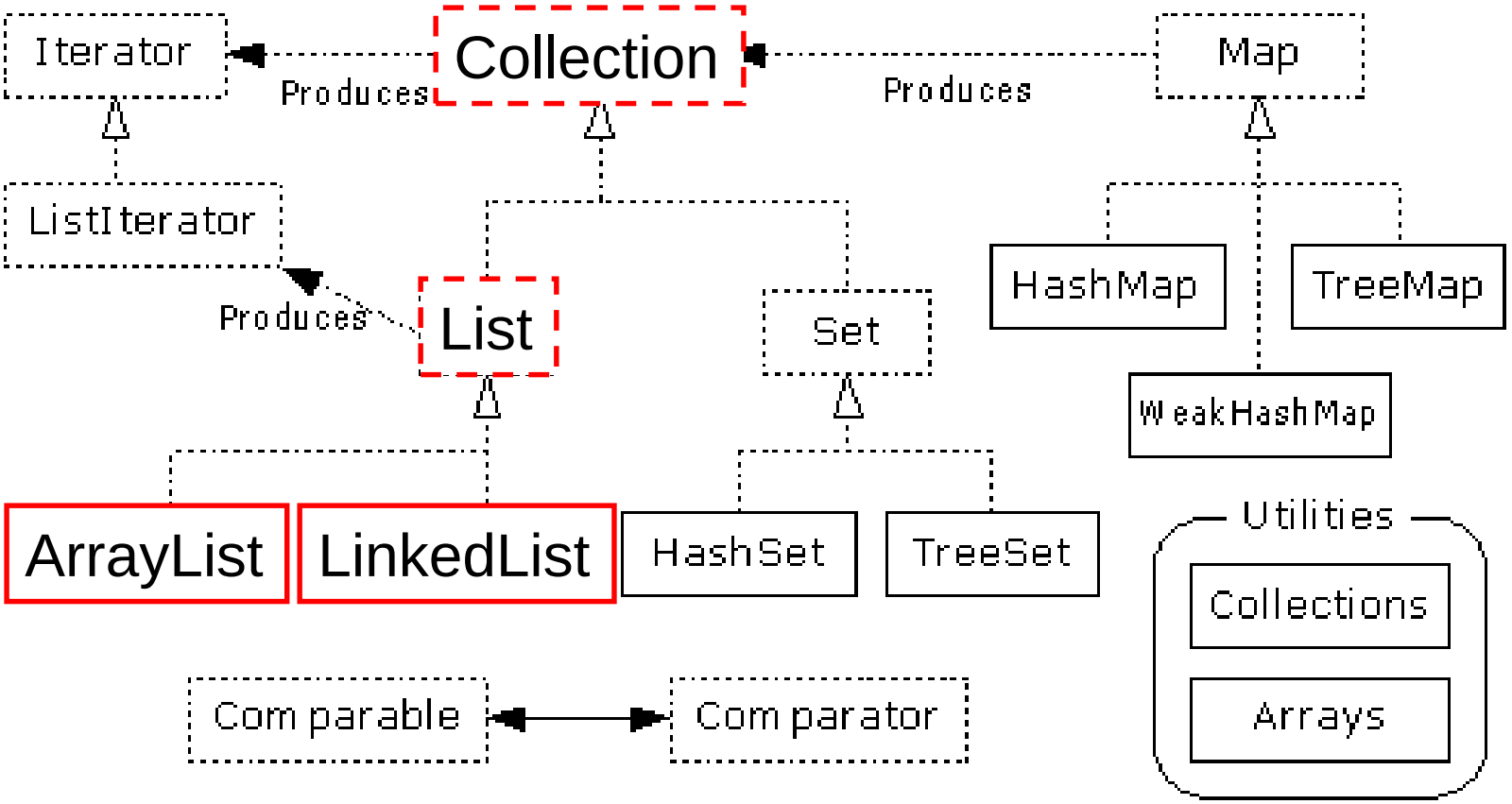


Figure 2–3: Advancing an iterator

ArrayList and LinkedList Context



List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access

ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException(
                "Illegal Capacity: "+initialCapacity);
    this.elementData = new Object[initialCapacity];
}
```

ArrayList methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - **Object get(int index)**
  - **Object set(int index, Object element)**
- Indexed add and remove are provided, but can be costly if used frequently
  - **void add(int index, Object element)**
  - **Object remove(int index)**
- May want to resize in one shot if adding many elements
  - **void ensureCapacity(int minCapacity)**

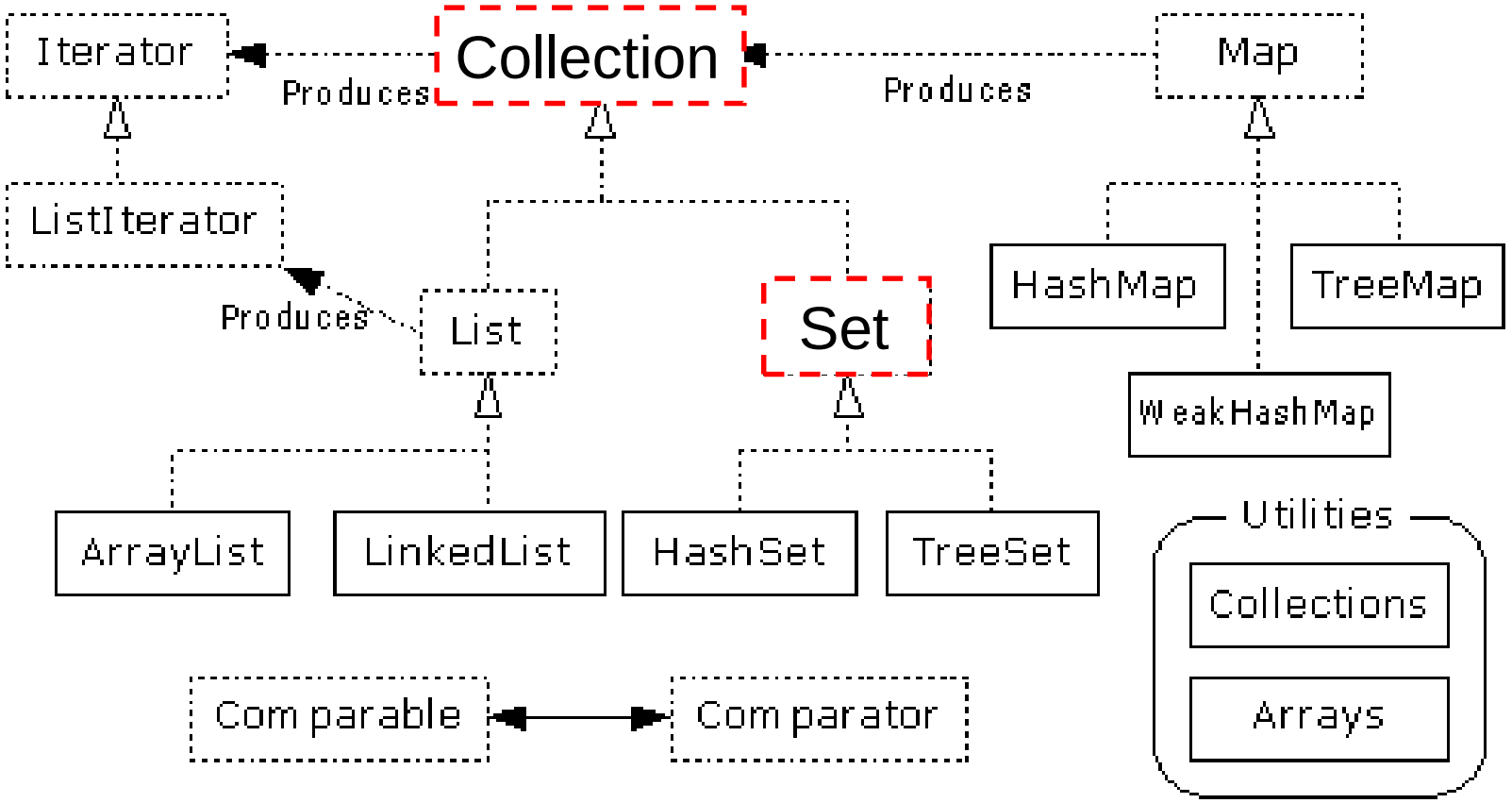
LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
  - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
  - Start from beginning or end and traverse each node while counting

LinkedList methods

- The list is sequential, so access it that way
  - **ListIterator listIterator()**
- ListIterator knows about position
  - use **add()** from ListIterator to add at a position
  - use **remove()** from ListIterator to remove at a position
- LinkedList knows a few things too
  - **void addFirst(Object o), void addLast(Object o)**
  - **Object getFirst(), Object getLast()**
  - **Object removeFirst(), Object removeLast()**

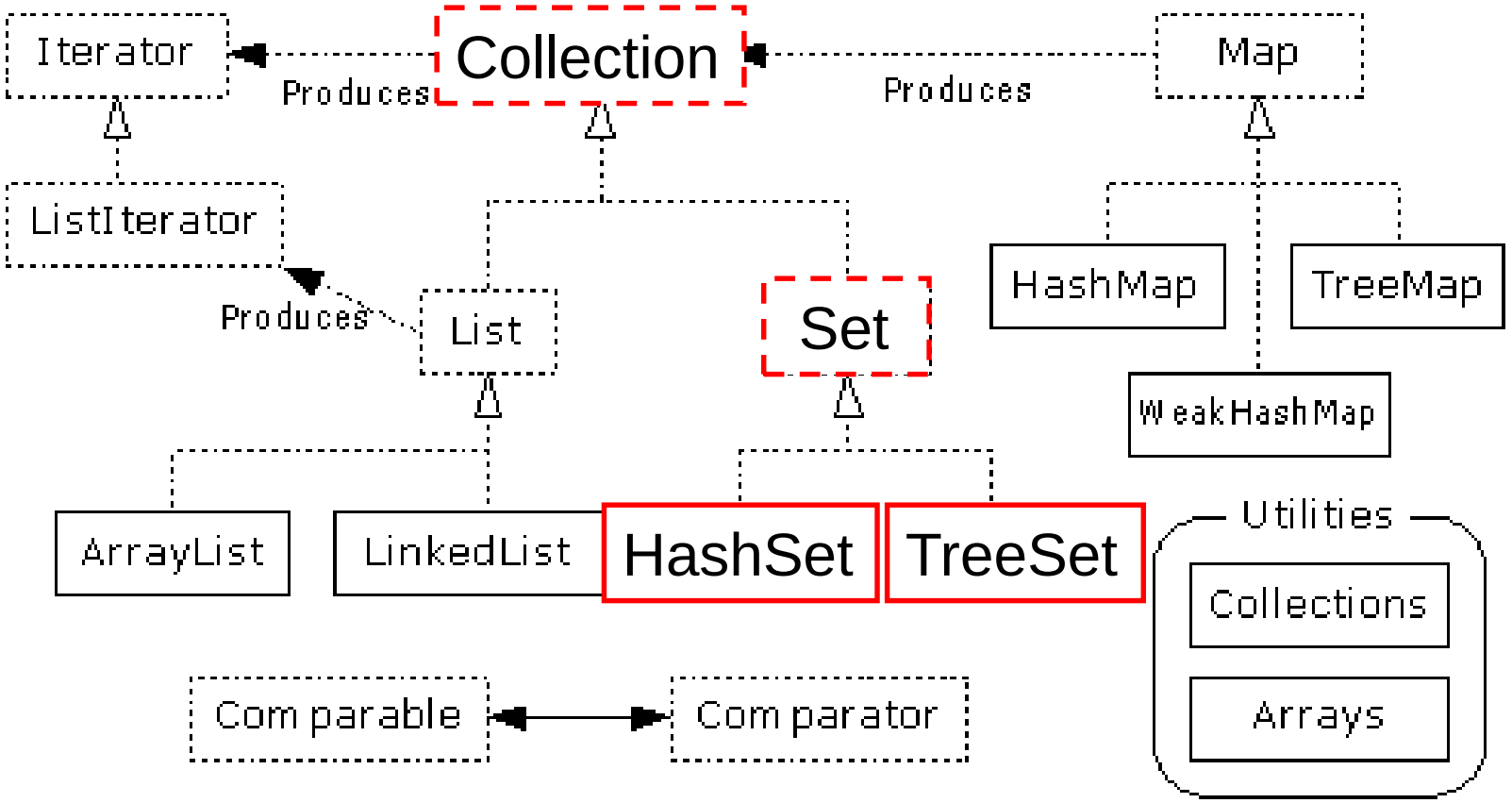
Set Interface Context



Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Defines two fundamental methods
  - **boolean add(Object o) -** reject duplicates
  - **Iterator iterator()**
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface
  - There is a SortedSet interface that extends Set

HashSet and TreeSet Context



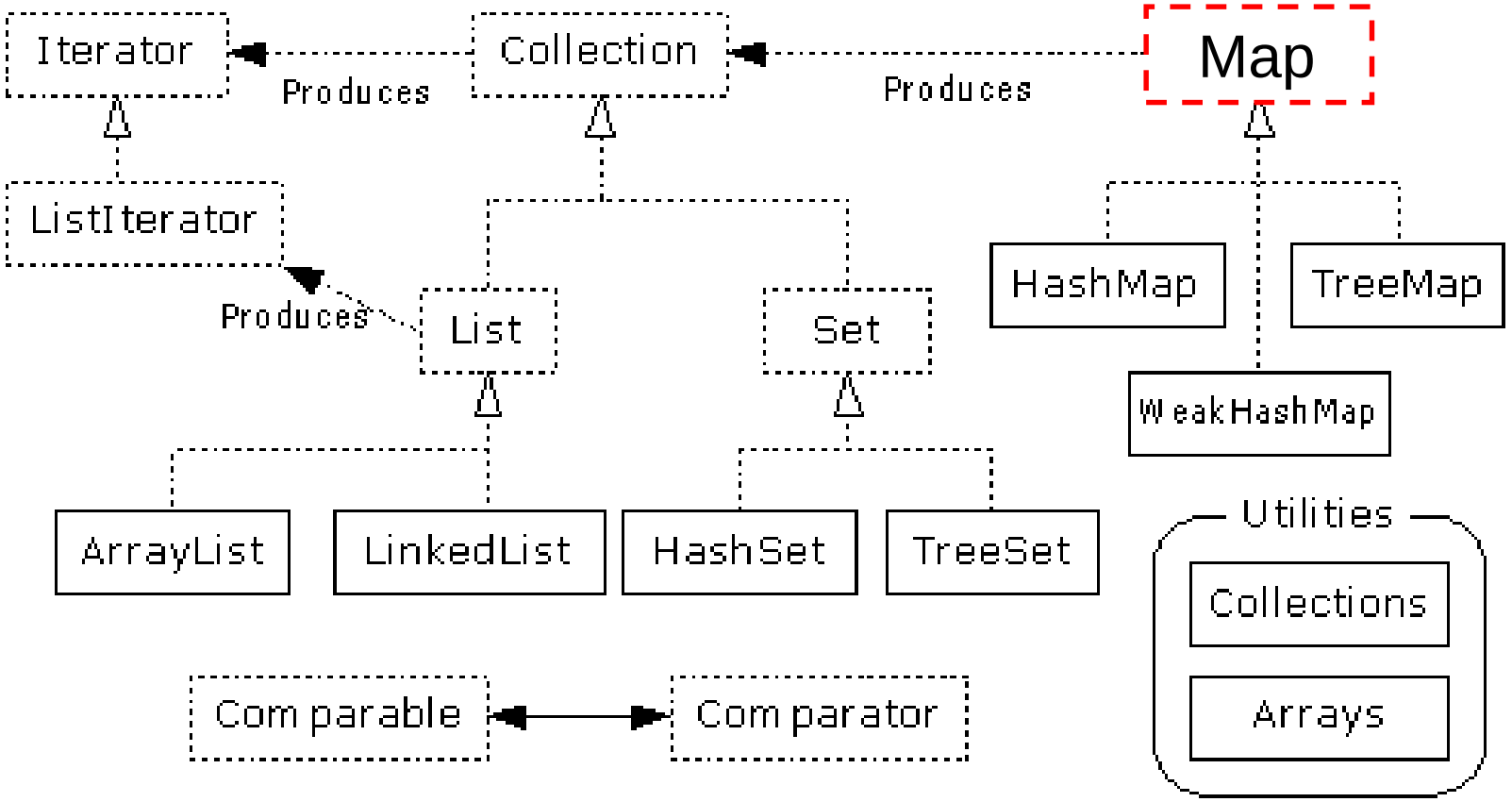
HashSet

- Find and add elements very quickly
  - uses hashing implementation in HashMap
- Hashing uses an array of linked lists
  - The **hashCode()** is used to index into the array
  - Then **equals()** is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- The **hashCode()** method and the **equals()** method must be compatible
  - if two objects are equal, they must have the same **hashCode()** value

TreeSet

- Elements can be inserted in any order
- The TreeSet stores them in order
- An iterator always presents them in order
- Default order is defined by natural order
  - objects implement the Comparable interface
  - TreeSet uses **compareTo(Object o)** to sort

Map Interface Context



Map Interface

- Stores key/value pairs
- Maps from the key to the value
- Keys are unique
  - a single key only appears once in the Map
  - a key can map to only one value
- Values do not have to be unique

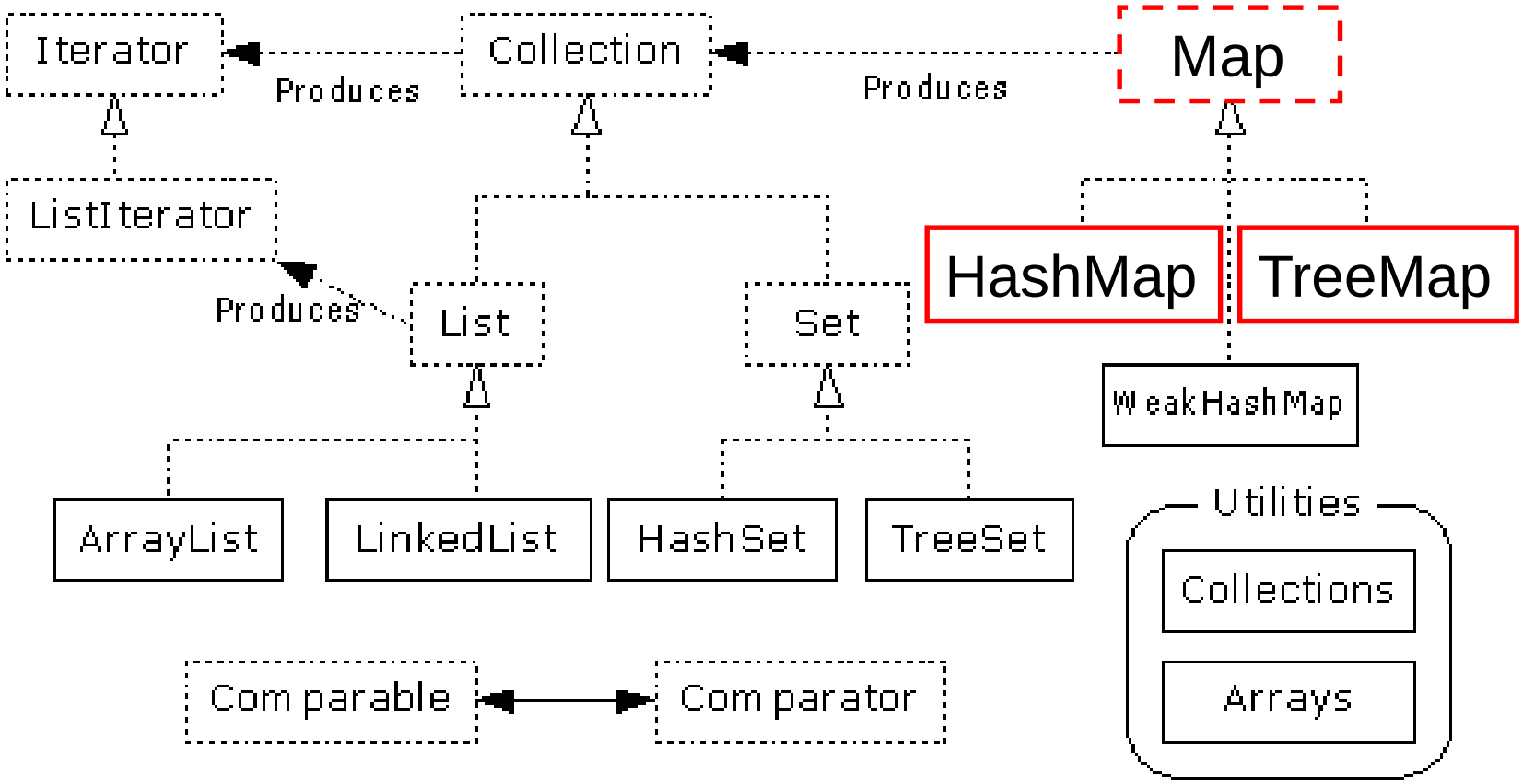
Map methods

Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()

Map views

- A means of iterating over the keys and values in a Map
- **Set keySet()**
  - returns the Set of keys contained in the Map
- **Collection values()**
  - returns the Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
- **Set entrySet()**
  - returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

HashMap and TreeMap Context



HashMap and TreeMap

- HashMap
  - The keys are a set - unique, unordered
  - Fast

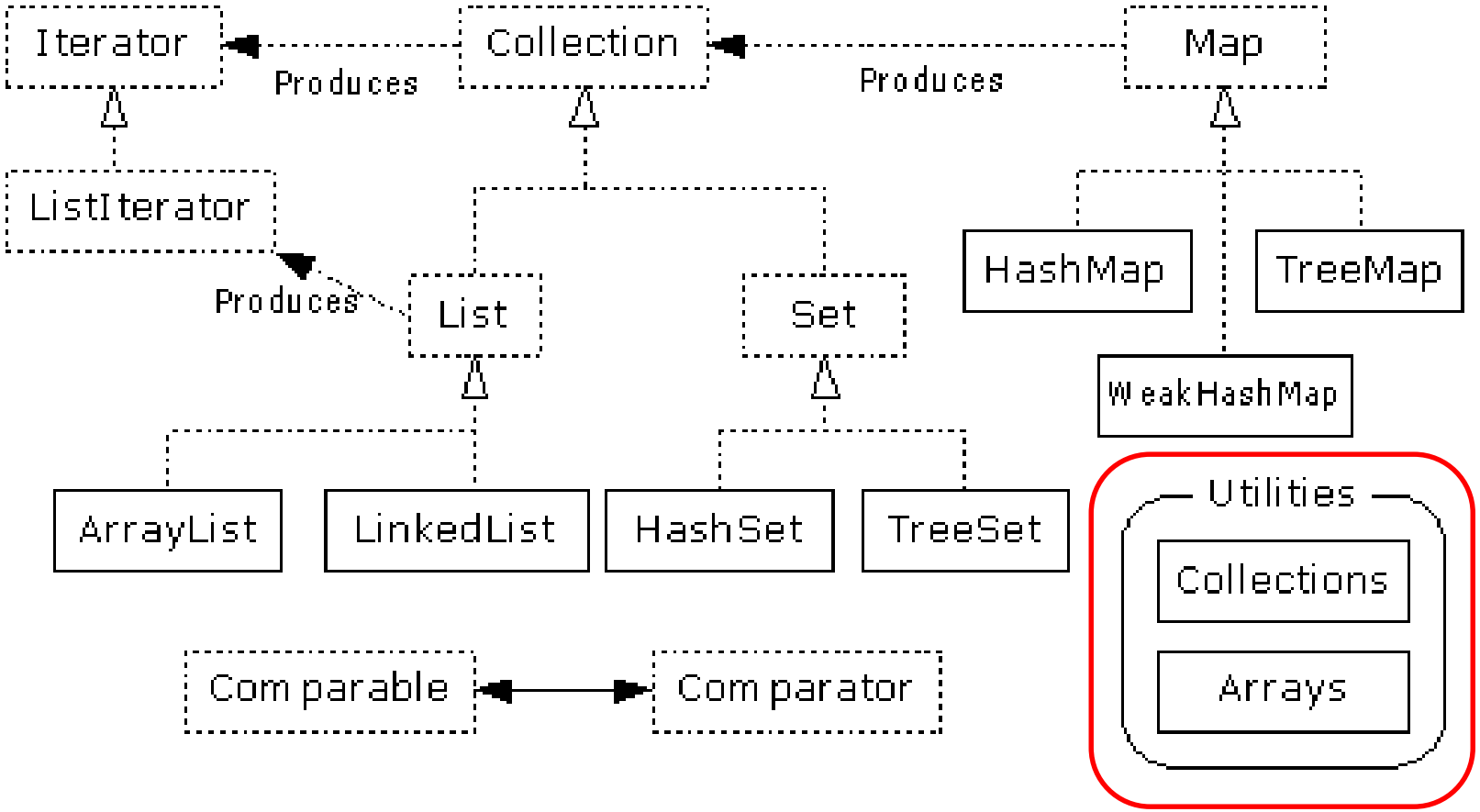
- TreeMap
  - The keys are a set - unique, ordered
  - Same options for ordering as a TreeSet
    - *Natural order (Comparable, compareTo(Object))*
    - *Special order (Comparator, compare(Object, Object))*

Bulk Operations

- In addition to the basic operations, a Collection may provide “bulk” operations

```
boolean containsAll(Collection c);
boolean addAll(Collection c);    // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear();                    // Optional
Object[] toArray();
Object[] toArray(Object a[]);
```

Utilities Context



Utilities

- The Collections class provides a number of static methods for fundamental algorithms
- Most operate on Lists, some on all Collections
  - Sort, Search, Shuffle
  - Reverse, fill, copy
  - Min, max
- Wrappers
  - synchronized Collections, Lists, Sets, etc
  - unmodifiable Collections, Lists, Sets, etc

Concrete Collections

concrete collection	implements	description
`HashSet`	`Set`	hash table
`TreeSet`	`SortedSet`	balanced binary tree
`ArrayList`	`List`	resizable-array
`LinkedList`	`List`	linked list
`Vector`	`List`	resizable-array
`HashMap`	`Map`	hash table
`TreeMap`	`SortedMap`	balanced binary tree

General Purpose Implementations

	Hash Table	Resizable array	balanced tree	linked list
Set	HashSet		TreeSet (sortedSet)	
List		ArrayList Vector		LinkedList
Map	HashMap Hashtable		TreeMap (sortedMap)	

The Arrays Class

The Arrays class contains various static methods

- sorting arrays
- searching arrays
- comparing arrays
- filling array elements
- convert array to list.

The Arrays Class UML Diagram

Arrays	
+asList(a: Object[]): List	Returns a list from an array of objects
Overloaded binarySearch method for byte, char, short, int, long, float, double, and Object. +binarySearch(a: xType[], key: xType): int	Overloaded binary search method to search a key in the array of byte, char, short, int, long, float, double, and Object
Overloaded equals method for boolean, byte, char, short, int, long, float, double, and Object. +equals(a: xType[], a2: xType[]): boolean	Overloaded equals method that returns true if a is equal to a2 for a and a2 of the boolean, byte, char, short, int, long, float, and Object type
Overloaded fill method for boolean char, byte, short, int, long, float, double, and Object. +fill(a: xType[], val: xType): void +fill(a: xType[], fromIndex: int, toIndex: xType, val: xType): void	Overloaded fill method to fill in the specified value into the array of the boolean, byte, char, short, int, long, float, and Object type
Overloaded sort method for char, byte, short, int, long, float, double, and Object. +sort(a: xType[]): void +sort(a: xType[], fromIndex: int, toIndex: int): void	Overloaded sort method to sort the specified array of the char, byte, short, int, long, float, double, and Object type

Convert to and from an Array

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

       List<String> sun = new ArrayList<String>();
      sun.add("Feel"); sun.add("the");
      sun.add("power");
      sun.add("of"); sun.add("the"); sun.add("Sun");

       String[] s1 = sun.toArray(new String[0]);
  //Collection to array
     for(int i = 0; i < s1.length; ++i){
     String contents = s1[i];
     System.out.print(contents); }

      System.out.println();
```

Enhanced for loop

```
List<String> sun2 = Arrays.asList(s1);
//Array back to Collection
for(String s2: sun2)
    { String s3 = s2;
      System.out.print(s3);
    }
 }
}
```

A note on iterators

- An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator()` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Iterate Through Collections

- The `Iterator` interface:

```
interface Iterator {
  boolean hasNext();
  Object next();
  void remove();
}
```

- The `iterator()` method defined in the `Collection` interface:

```
Iterator iterator()
```

Iterators

- Iterators provide a general way to traverse all elements in a collection

```
ArrayList<String> list = new ArrayList<String>();
   list.add("1-FiRsT");
   list.add("2-SeCoND");
   list.add("3-ThIrD");
   Iterator<String> itr = list.iterator();
   while (itr.hasNext()) {
System.out.println(itr.next().toLowerCase());
   }
```

Output

```
1-first
2-second
3-third
```

Enhanced for loop

- If a class **extends Iterable<E>**
- (e.g. class **Set<E>** implements Iterable),
- Java's enhanced for loop of this general form

  for (*E refVar* **:** *collection<E>* **) {**
  refVar refers to each element in *collection<E>*
  }
  - example

```
ArrayList<String> list = new ArrayList<String>();
list.add("first");
for (String s : list)
      System.out.println(s.toLowerCase());
```

Map and SortedMap

- The Map interface defines methods
  - **`get`**, **`put`**, **`contains`**, **`keySet`**, **`values`**, **`entrySet`**
- **`TreeMap`** implements **`Map`**
  - put, get, remove: O(log n)
- **`HashMap`** implements **`Map`**
  - put, get, remove: O(1)

Set and SortedSet

- Some Map methods return **`Set`**
- The Set interface
  - **`add`**, **`addAll`**, **`remove`**, **`size`**, but no **`get`**!
- Some implementations
  - **`TreeSet`**: values stored in order, O(log n)
  - **`HashSet`**: values in a hash table, no order, O(1)

Choosing the datatype

- When you declare a Set, List or Map, you should use Set, List or Map interface as the datatype instead of the implementing class. That will allow you to change the implementation by changing a single line of code!

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
      Set<String> ss = new LinkedHashSet<String>();

        for (int i = 0; i < args.length; i++)
            ss.add(args[i]);

        Iterator i = ss.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

```
import java.util.*;

public class Test {

  public static void main(String[] args)
  {
     //map to hold student grades
     Map<String, Integer> theMap = new
  HashMap<String, Integer>();

     theMap.put("Korth, Evan", 100);
     theMap.put("Plant, Robert", 90);
     theMap.put("Coyne, Wayne", 92);
     theMap.put("Franti, Michael", 98);
     theMap.put("Lennon, John", 88);

     System.out.println(theMap);
     System.out.println("--------------------");
     System.out.println(theMap.get("Korth,
  Evan"));
     System.out.println(theMap.get("Franti,
  Michael"));  }}
```

Using Sets to find duplicate elements

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate
detected: " + a);

        System.out.println(s.size() + "
distinct words: " + s);
    }}
```

Which class should I use?

- The difference between the different classes is how the structure is implemented.
  - This generally has an impact on performance.
- Use Vector
  - Fast access to elements using index
  - Optimized for storage space
  - Not optimized for inserts and deletes
- Use ArrayList
  - Same as Vector except the methods are not synchronized. Better performance
- Use linked list
  - Fast inserts and deletes
  - Stacks and Queues (accessing elements near the beginning or end)
  - Not optimized for random access

Which class should I use?

- Use Sets
  - When you need a collection which does not allow duplicate entries
- Use Maps
  - Very Fast access to elements using keys
  - Fast addition and removal of elements
  - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

Data Structures	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known.	Slow search, slow deletion, and fixed size.
Ordered array	Quicker search than unsorted array	Slow insertion and deletion, fixed size
Stack	Last in first out	Slow access to other items
Queue	First in first out access.	Slow access to other items
Linked list	Quick insertion, quick deletion	Slow search
Array List	Random Access	Slow insertion, deletion