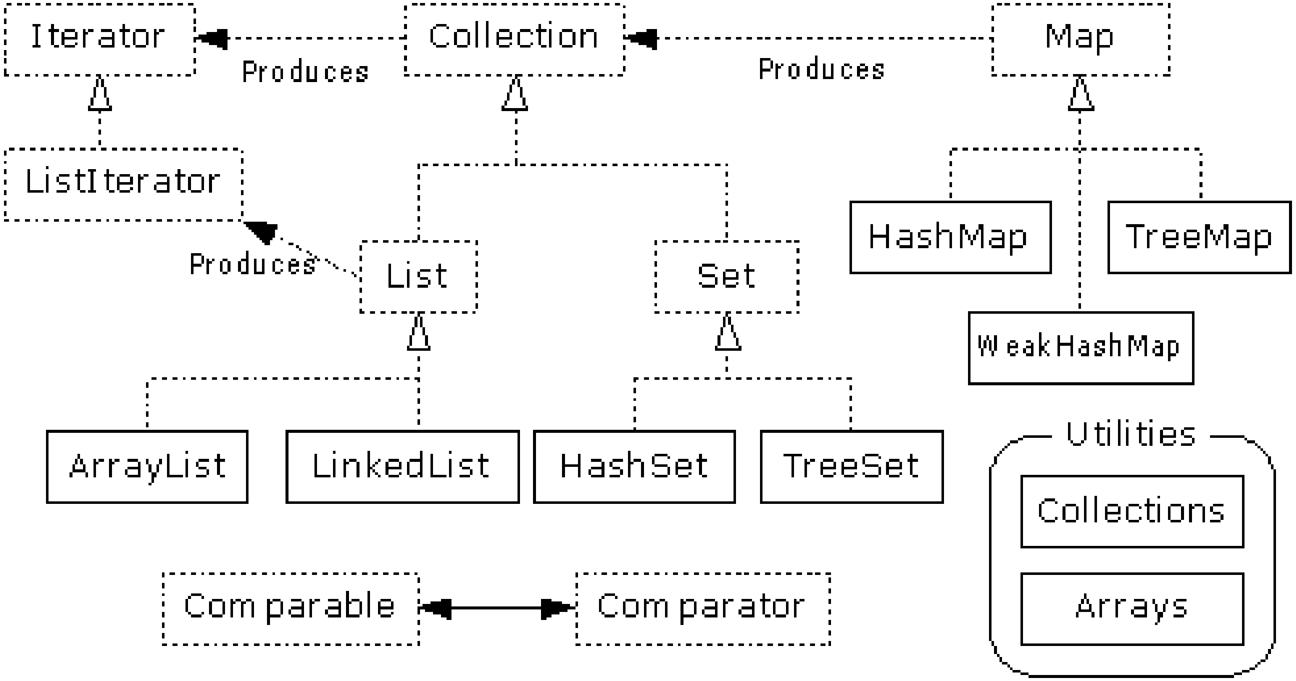
Java Collections

Object Oriented Concept

Collections Framework Diagram



- •Interfaces, Implementations, and Algorithms
- •From Thinking in Java, page 462

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

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

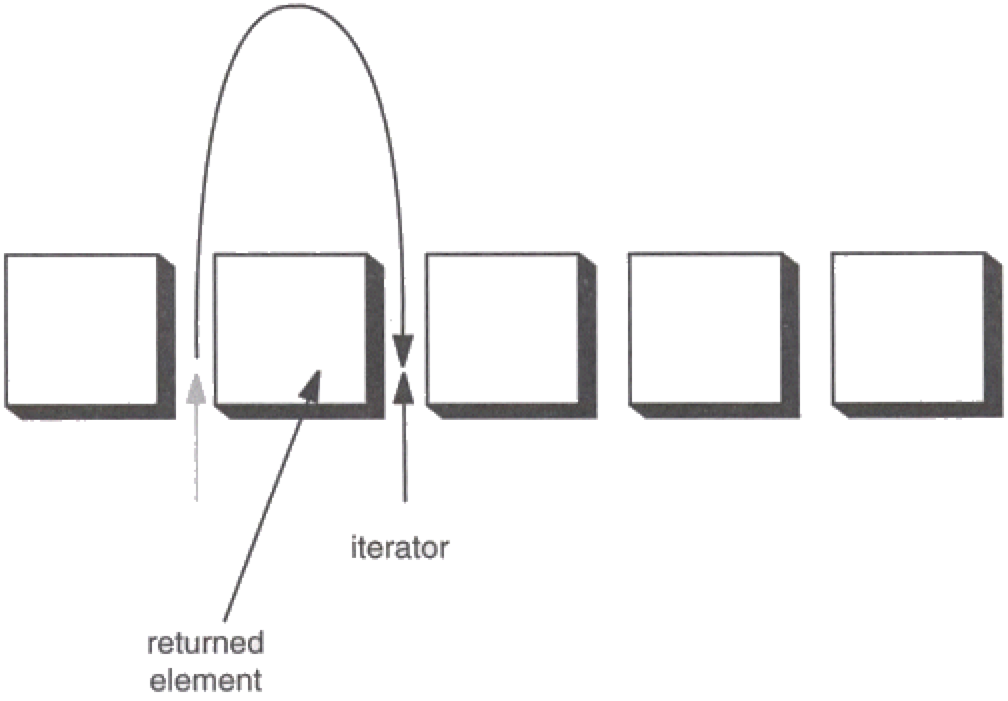
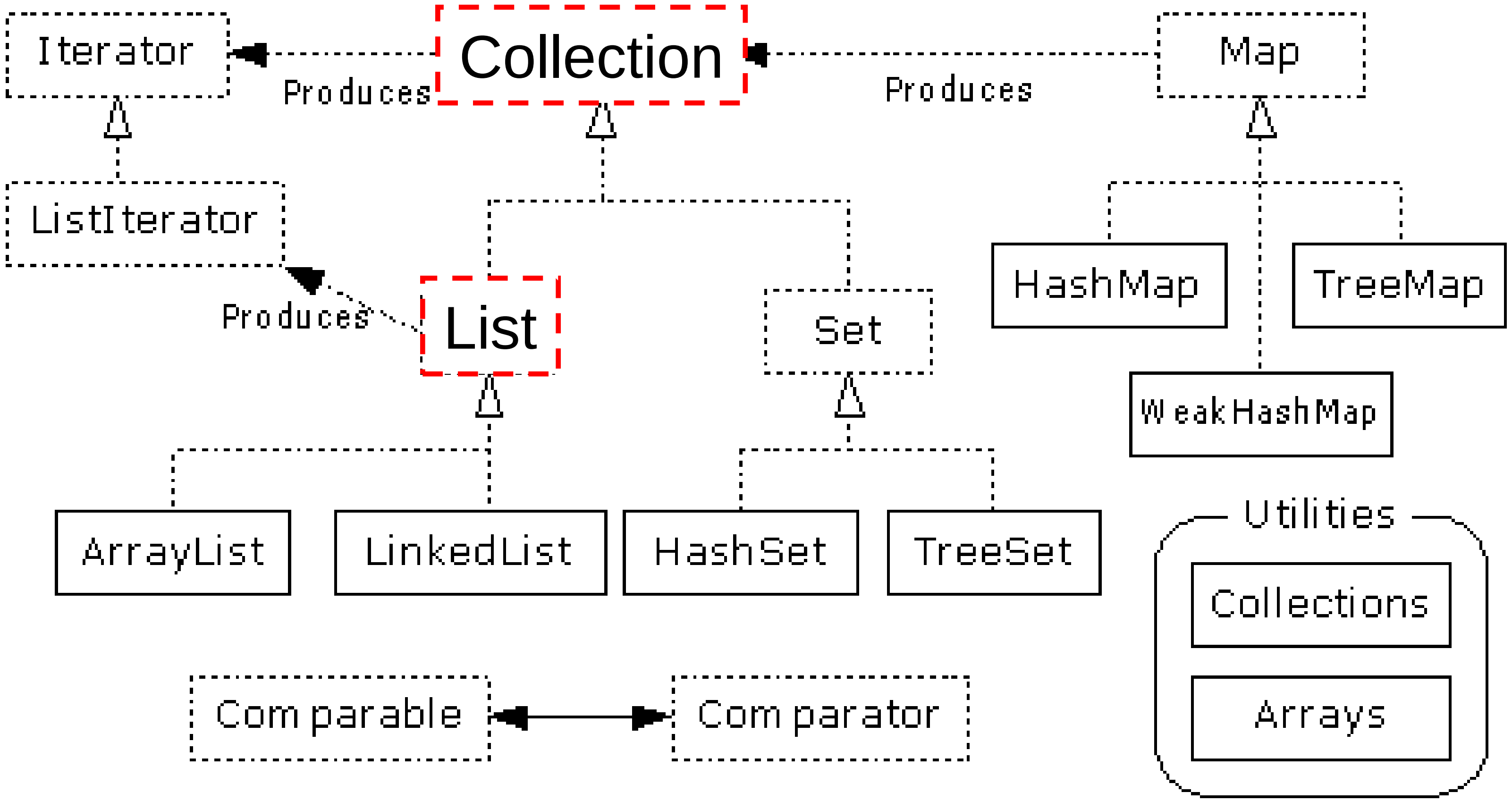


Figure 2–3: Advancing an iterator

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()`

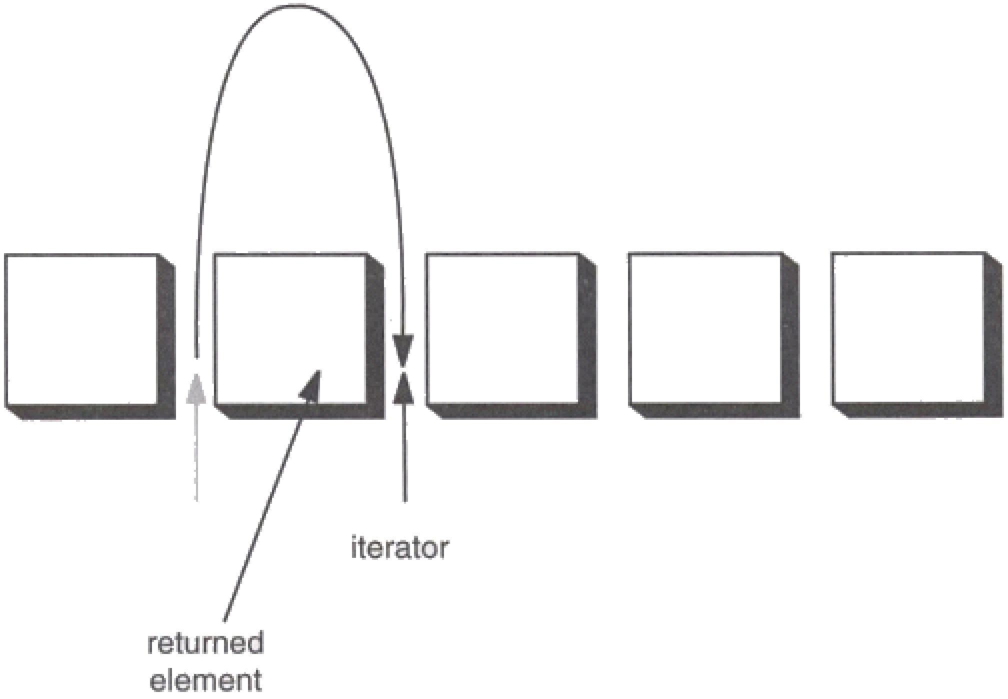
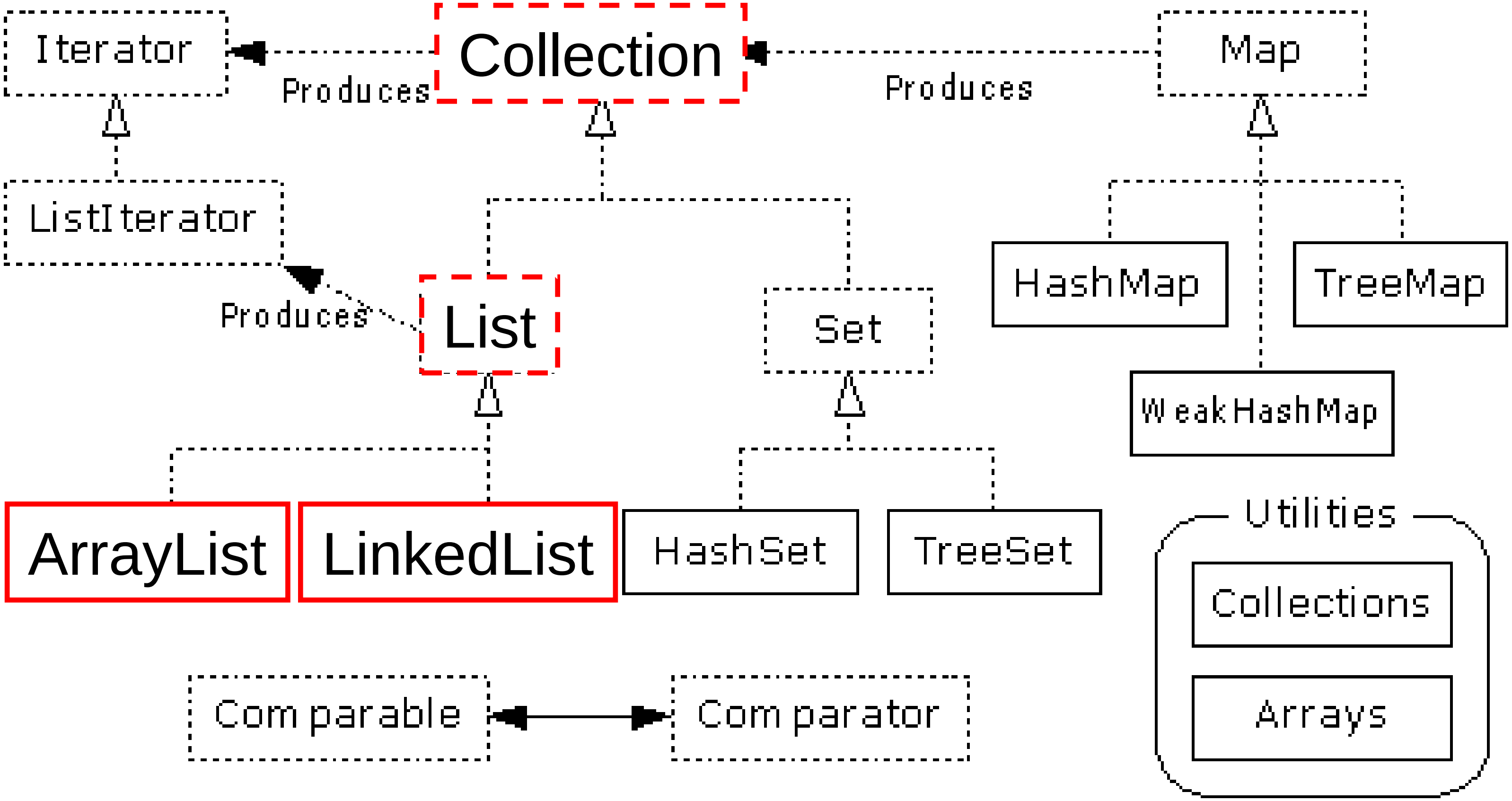


Figure 2–3: Advancing an iterator

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

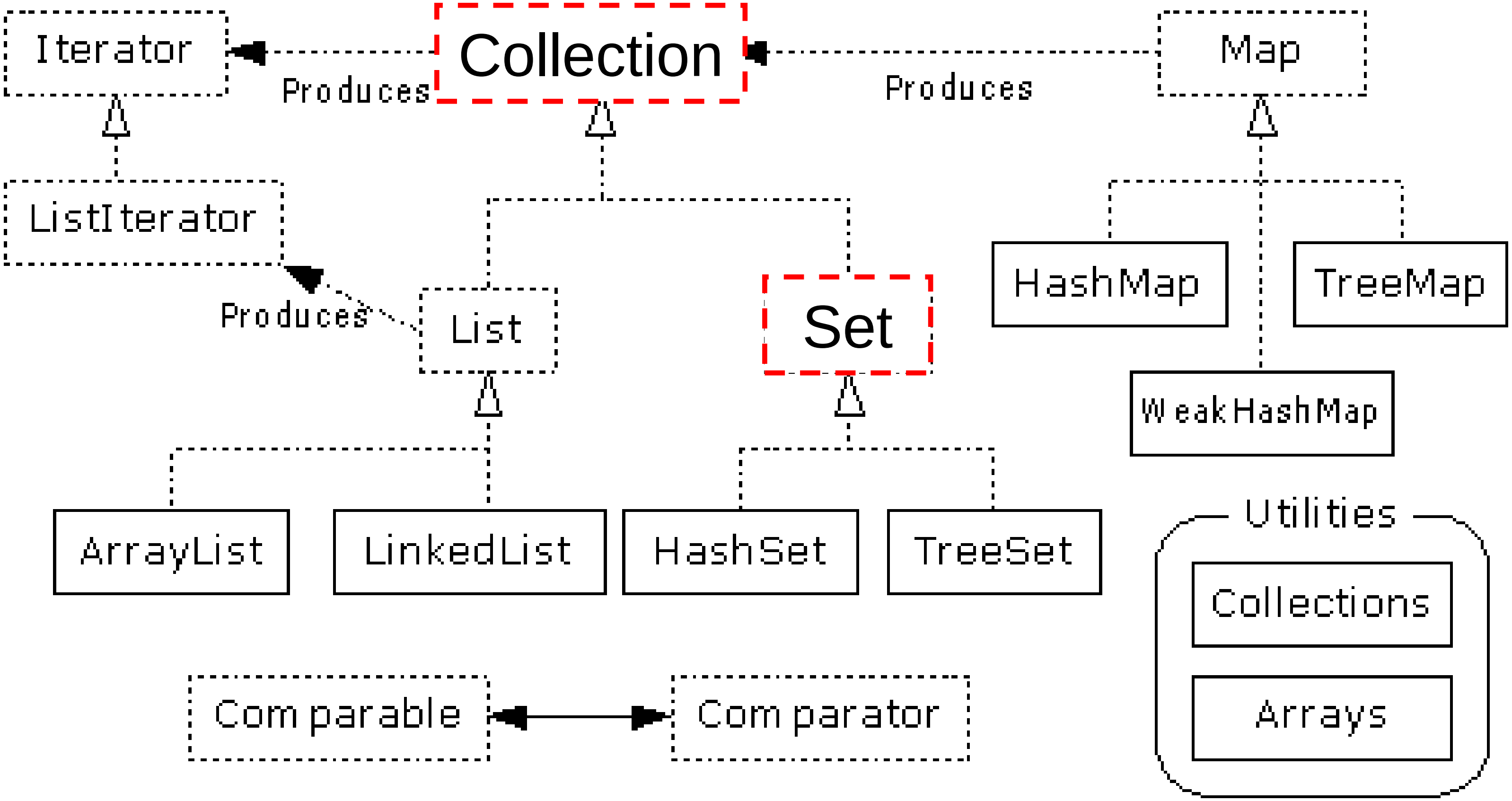
- Constant time positional access (it’s an array)
- One tuning parameter, the initial capacity

```
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)`

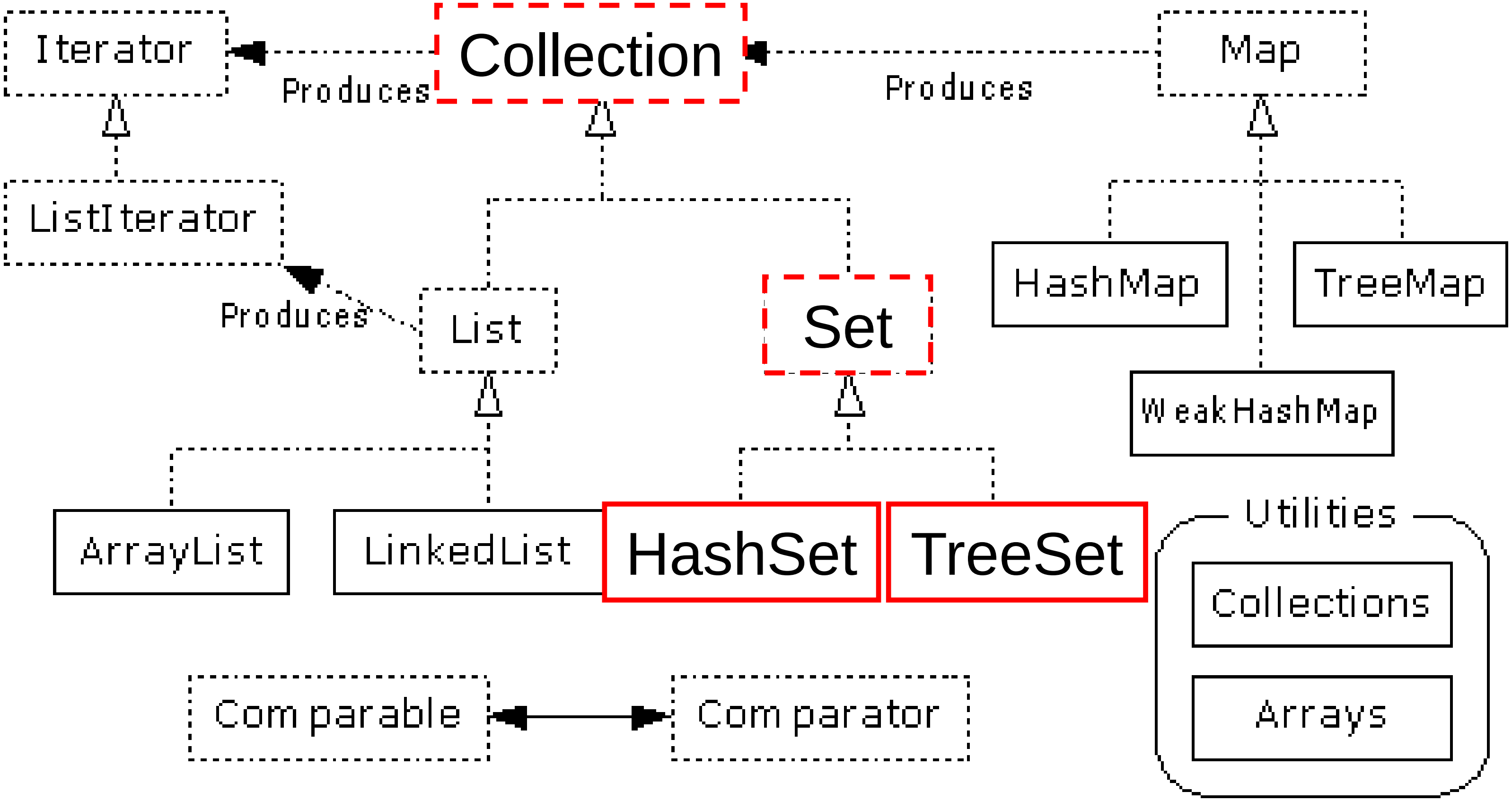
Set Interface Context



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



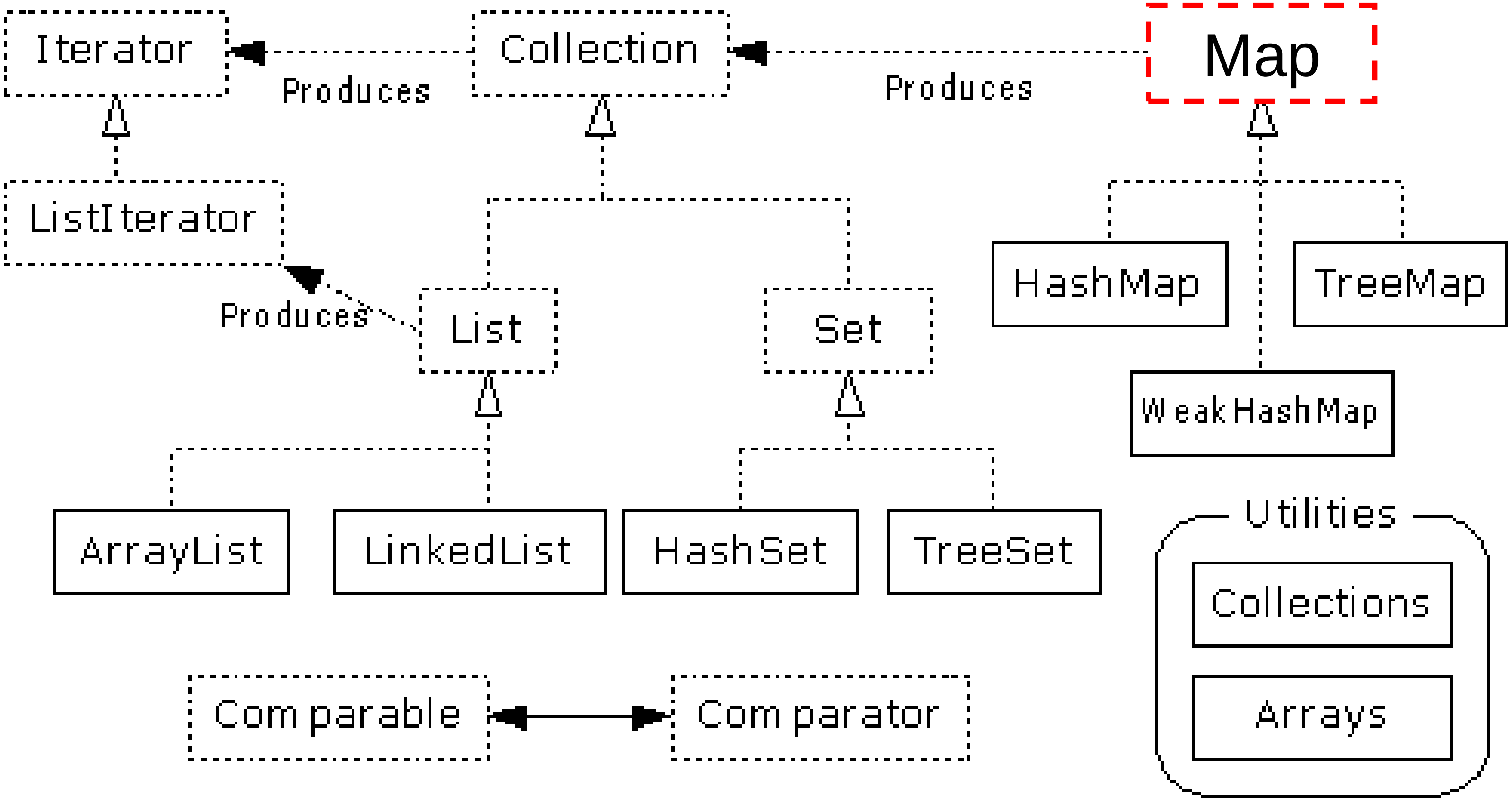
HashSet

- 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
  - » Red-Black Trees out of Cormen-Leiserson-Rivest
- 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
- Can use a different Comparator
  - » provide Comparator to the TreeSet constructor

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

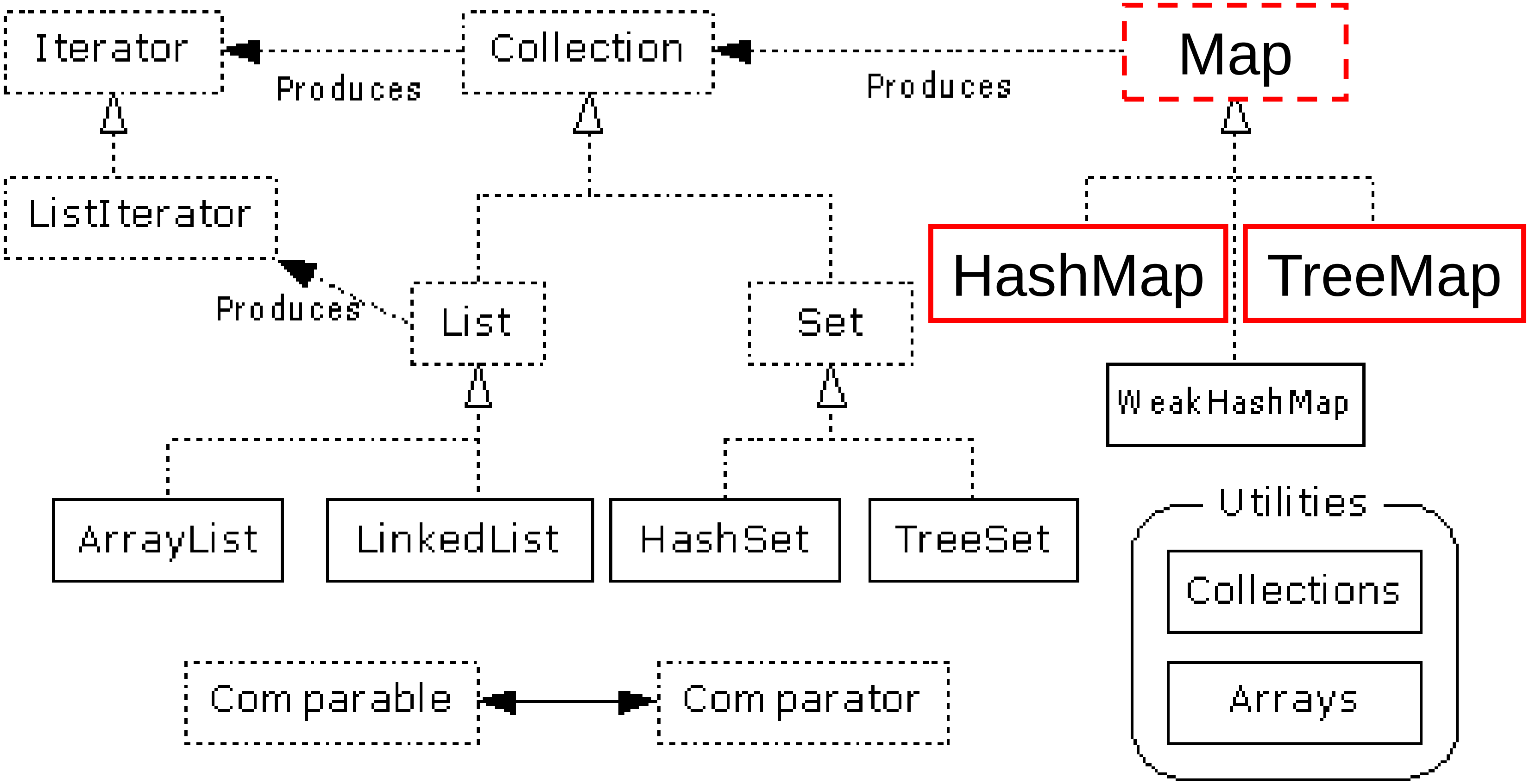
Map methods

```
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))*

HashMap and TreeMap

HashMap

HashMap can contain one null key.

HashMap maintains no order.

TreeMap

TreeMap can not contain any null key.

TreeMap maintains ascending order.

HashMap

```
{
      HashMap<Integer,String> hm=new HashMap<Integer,String>();

        hm.put(100,“Tahil");
        hm.put(102,“Rifad");
       hm.put(101,“Jubayer");

        for(Map.Entry m:hm.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }
      If ( hm.containsKey (102))
           System.out.println(hm.get(102));

      hm.remove(102);
}
```

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[]);
```

HashMap and Hashtable

- HashMap is **non synchronized**. It is not-thread safe
- HashMap **allows one null key and multiple null values**.
- HashMap is **fast**.
- HashMap is **traversed by Iterator**.
- HashMap inherits **AbstractMap** class.

- Hashtable is **synchronized**. It is thread-safe
- Hashtable **doesn't allow any null key or value**.
- Hashtable is **sl**
- Hashtable is **traversed by Enumerator and Iterator**.**ow**.
- Hashtable inherits **Dictionary** class.