

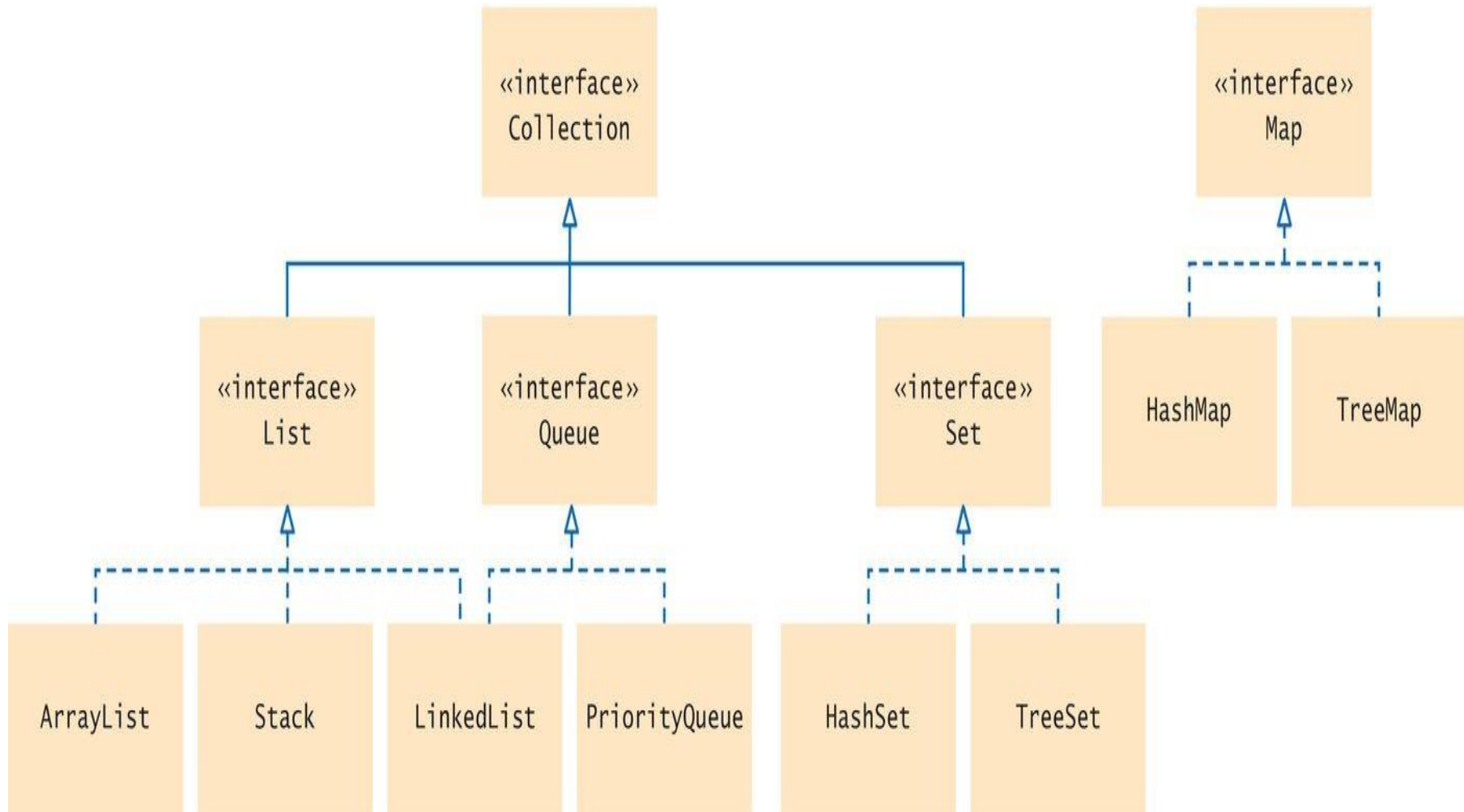


# The Java Collections Framework

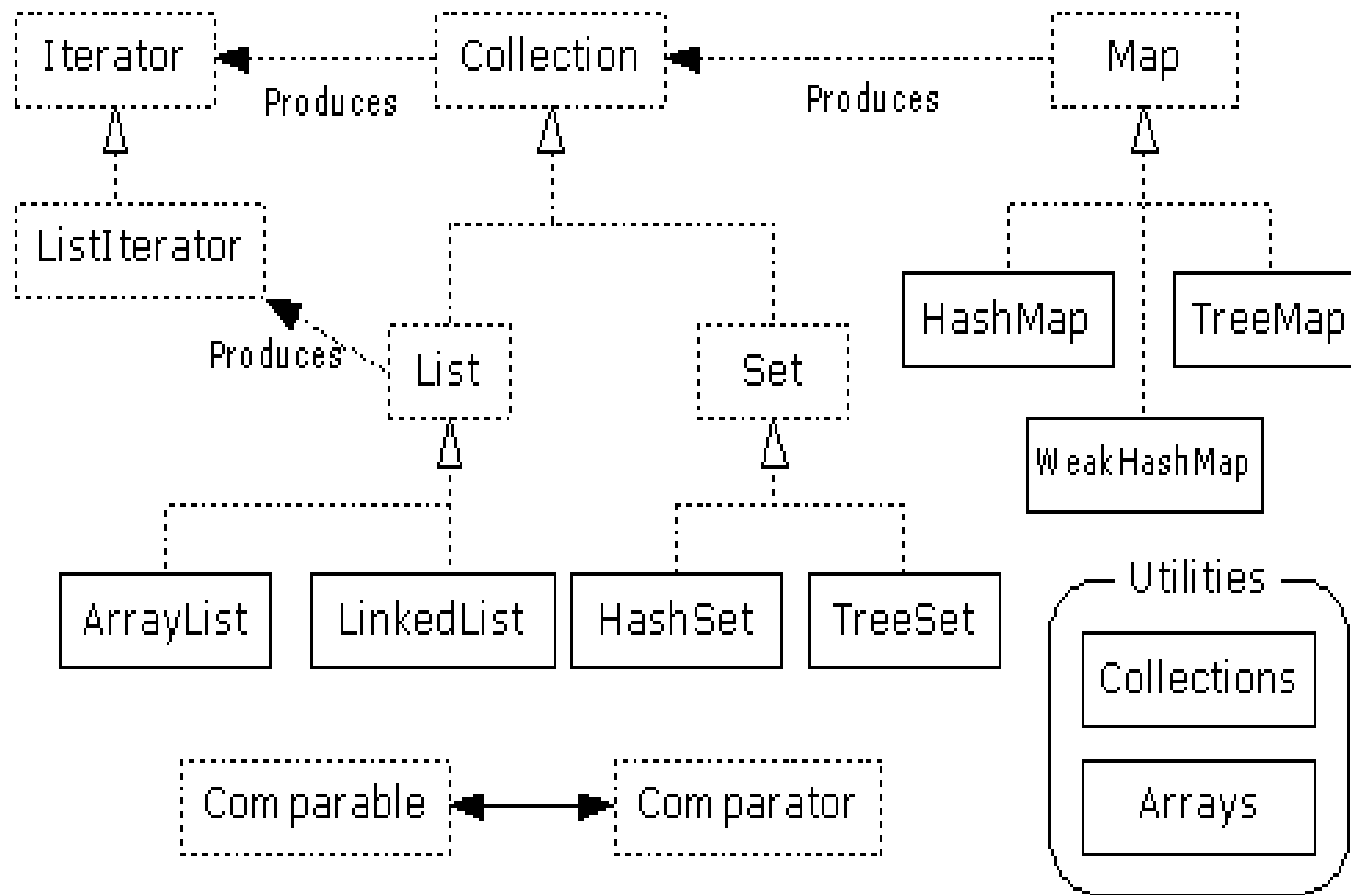
---



# An Overview of the Collections Framework

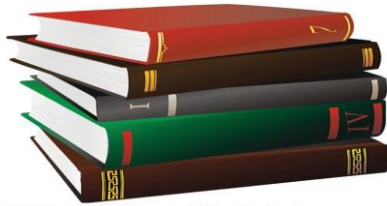


# Collection Interface



# Stack overview

- Remembers the order of elements
- But you can only add and remove at the top



© Vladimir Trenin/iStockphoto.



- Removes items in the opposite order than they were added Last-in, first-out or LIFO order
- Add and remove methods are called push and pop.



# Stack overview

- Stack class provides push, pop and peek methods.

**Table 7** Working with Stacks

<code>Stack&lt;Integer&gt; s = new Stack&lt;&gt;();</code>	Constructs an empty stack.
<code>s.push(1);</code> <code>s.push(2);</code> <code>s.push(3);</code>	Adds to the top of the stack; s is now [1, 2, 3]. (Following the toString method of the Stack class, we show the top of the stack at the end.)
<code>int top = s.pop();</code>	Removes the top of the stack; top is set to 3 and s is now [1, 2].
<code>head = s.peek();</code>	Gets the top of the stack without removing it; head is set to 2.



# Stack overview

---

```
1⊕ import java.util.Scanner;
2
3
4
5 public class StackDemo {
6⊖     public static void main(String arg[])
7     {
8         Stack s = new Stack ();
9         s.push("A");
10        s.push("B");
11        s.push("C");
12        String word = (String) s.peek();    // Prints C
13        System.out.println(word);
14        while (s.size() > 0)
15        {
16            System.out.print(s.pop() + " "); // Prints C B A
17        }
18    }
19 }
20
```



# Calculator.java

3

```
4      /**
5      This calculator uses the reverse Polish notation.
6      */
7      public class Calculator
8      {
9      public static void main(String[] args)
10     {
11         StaScanner in = new Scanner(System.in);
12         Stack<Integer> results = new Stack<>();
13         System.out.println("Enter one number or operator per line, Q to quit.
14     ");
15         boolean done = false;
16         while (!done)
17         {
18             String input = in.nextLine();
19
20             // If the command is an operator, pop the arguments and push the result
21
22             if (input.equals("+"))
23             {
24                 results.push(results.pop() + results.pop());
25             }
26             else if (input.equals("-"))
27             {
28                 Integer arg2 = results.pop();
29                 results.push(results.pop() - arg2);
30             }
31             else if (input.equals("*") || input.equals("x"))
32             {
33                 results.push(results.pop() * results.pop());
34             }
35             else if (input.equals("/"))
36             {
37                 Integer arg2 = results.pop();
```

# Queue Overview

- Add items to one end (the tail) and remove them from the other end (the head)
- A queue of people



Photodisc/Punchstock.

- A priority queue
  - an unordered collection
  - has an efficient operation for removing the element with the highest priority





# Queue Overview

---

- The Queue interface in the standard Java library has:
  - an add method to add an element to the tail of the queue,
  - a remove method to remove the head of the queue, and
  - a peek method to get the head element of the queue without removing it.
- The LinkedList class implements the Queue interface. When you need a queue, initialize a Queue variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) {
    System.out.print(q.remove() + " "); // Prints A B C
}
```



# Queue Overview

**Table 8** Working with Queues

<code>Queue&lt;Integer&gt; q = new LinkedList&lt;&gt;();</code>	The <code>LinkedList</code> class implements the <code>Queue</code> interface.
<code>q.add(1);</code> <code>q.add(2);</code> <code>q.add(3);</code>	Adds to the tail of the queue; <code>q</code> is now <code>[1, 2, 3]</code> .
<code>int head = q.remove();</code>	Removes the head of the queue; <code>head</code> is set to 1 and <code>q</code> is <code>[2, 3]</code> .
<code>head = q.peek();</code>	Gets the head of the queue without removing it; <code>head</code> is set to 2.

# Priority Queue Overview

- A priority queue collects elements, each of which has a priority.
- Example: a collection of work requests, some of which may be more urgent than others.
- Does not maintain a first-in, first-out discipline.
- Elements are retrieved according to their priority.
- Priority 1 denotes the most urgent priority.
- Each removal extracts the minimum element.
- When you retrieve an item from a priority queue, you always get the most urgent one.



© paul kline/iStockphoto.



# Priority Queue Overview

```
3 |
4 public class PriorityQ {
5     public static void main(String arg[]){
6         PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
7         pq.add(10);
8         pq.add(7);
9         pq.add(3);
10        pq.add(6);
11        pq.add(2);
12        System.out.println(pq.peek());
13        System.out.println(pq);
14        pq.poll();
15        System.out.println(pq);
16        PriorityQueue<String> spq = new PriorityQueue<String>();
17        spq.add("Zoo");
18        spq.add("Book");
19        spq.add("Apple");
20        spq.add("Yello");
21        System.out.println(spq.peek());
22        System.out.println(spq);
23        spq.poll();
24        System.out.println(spq);
25    }
26
27 }
```

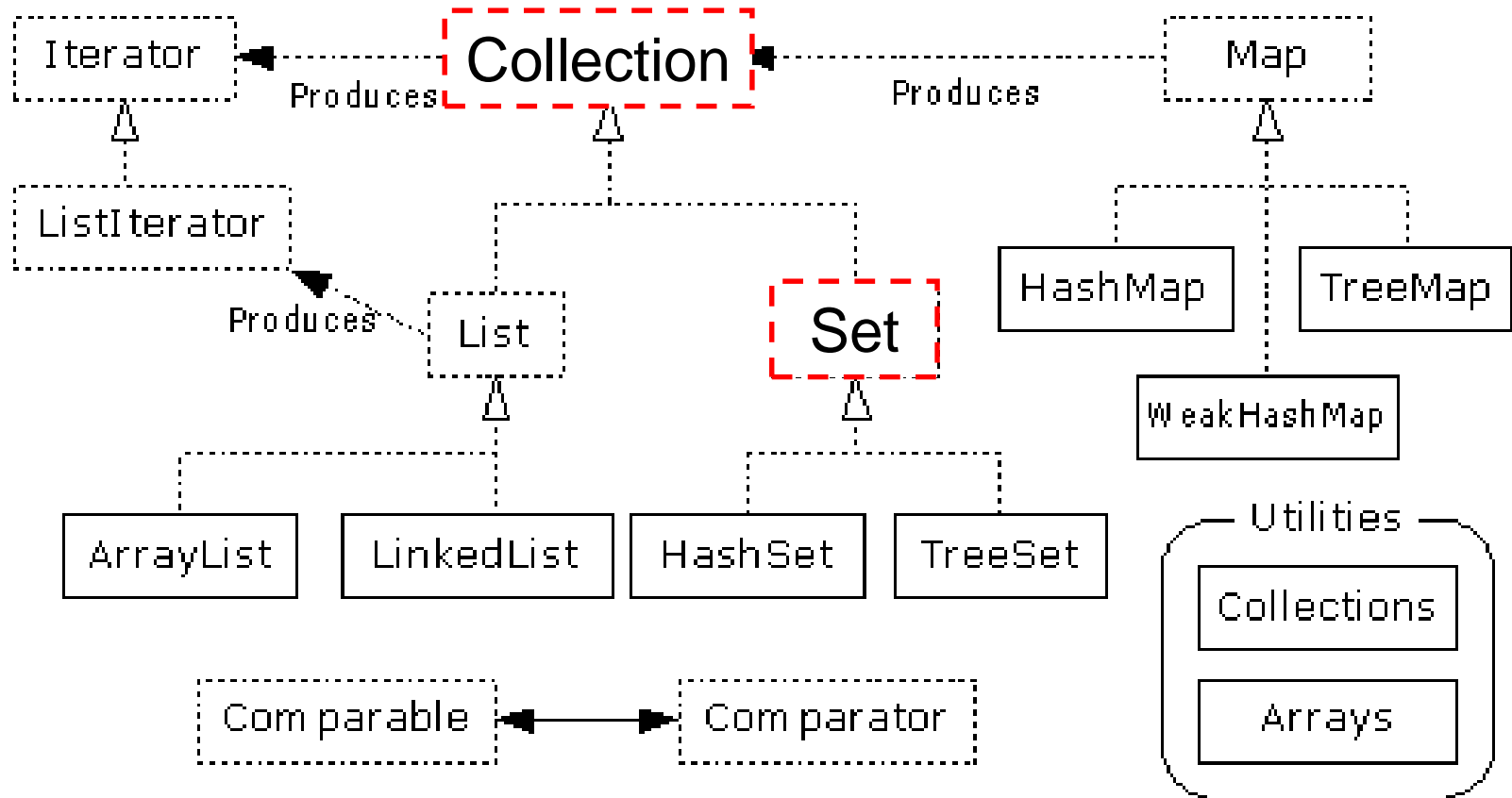


# Priority Queue Overview

---

```
PriorityQueue<WorkOrder> q = new PriorityQueue<>();  
q.add(new WorkOrder(3, "Shampoo carpets"));  
q.add(new WorkOrder(1, "Fix broken sink"));  
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

# Set Interface Context





# Set Interface

---

- Set is a child interface of Collection.
- The Set interface present group of individual objects as a single entity where duplicate are not allowed and insertion order is not preserved.
- Inserting and removing elements is more efficient with a set than with a list.



# 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





# Set Interface

---

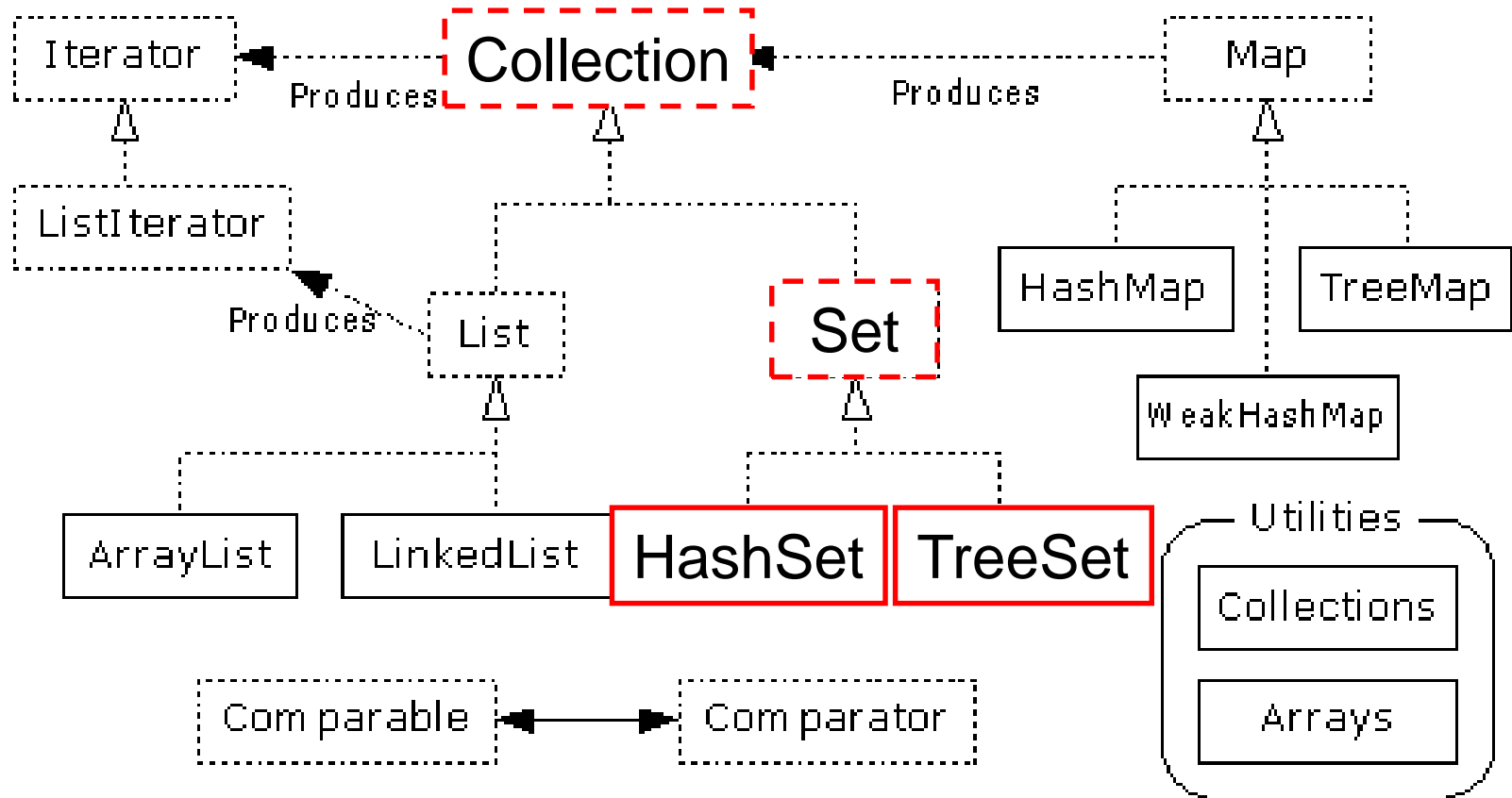
- Testing if `s2` is a *subset* of `s1`  
`s1.containsAll(s2)`
- Setting `s1` to the *union* of `s1` and `s2`  
`s1.addAll(s2)`
- Setting `s1` to the *intersection* of `s1` and `s2`  
`s1.retainAll(s2)`
- Setting `s1` to the *set difference* of `s1` and `s2`  
`s1.removeAll(s2)`



# The difference between List and Set?

List	Set
Duplicate is allowed	Duplicate is not allowed
Insertion order is preserved	Insertion order is not preserved

# HashSet and TreeSet Context





# Hashset overview

---

- The underlying structure is Hashtable.
- Duplicates are not allowed. What happened if you insert duplicate?
- Insertion order is not preserved and all the objects will be inserted based on hash-code object.
- Set elements are grouped into smaller collections of elements that share the same characteristic.
  - Grouped by an integer hash code that is computed from the element.
- Heterogeneous objects are allowed (int, string, etc).
- Null insertion is allowed.
- Hashset is the best option for frequent search operations.
- Demo

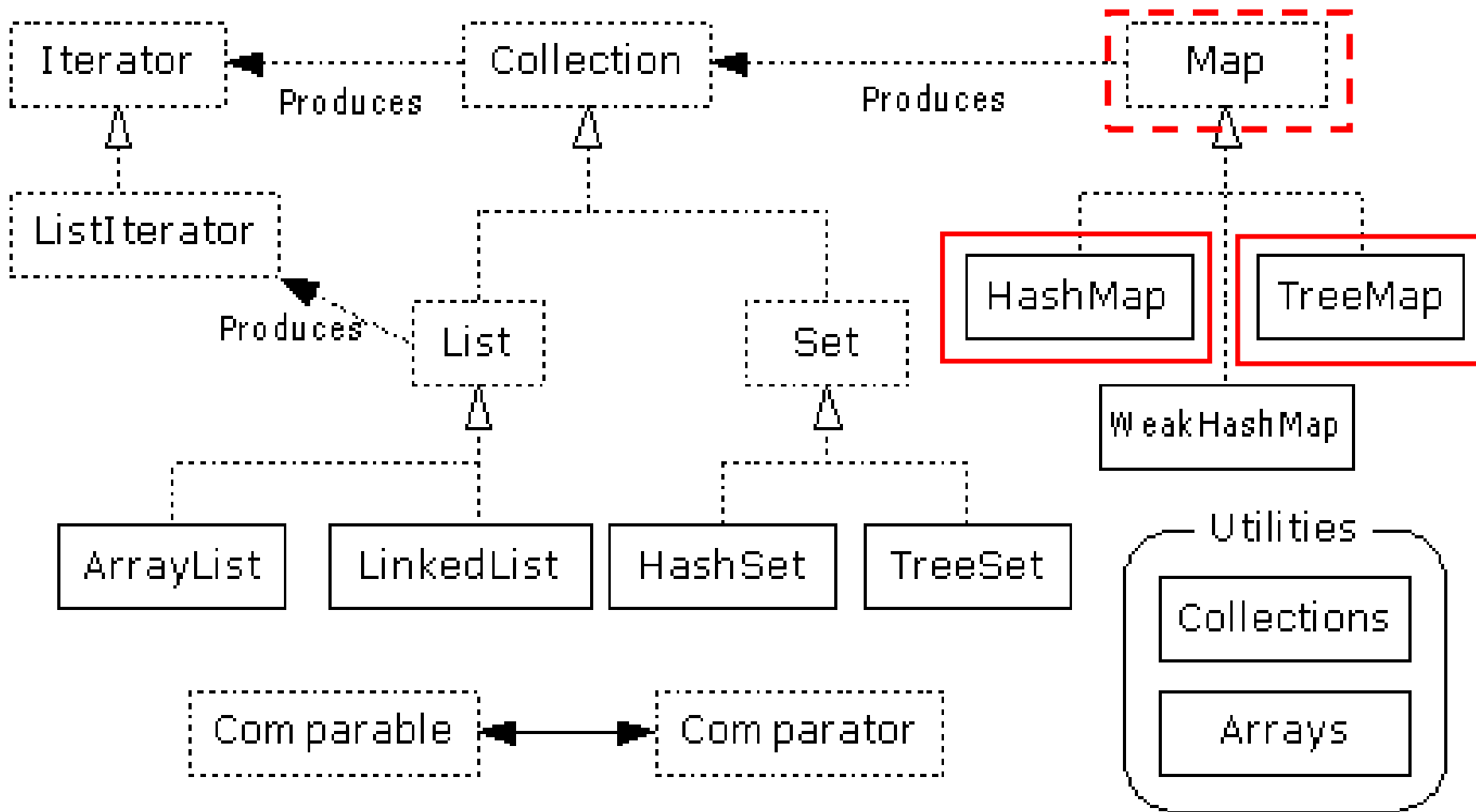


# TreeSet Overview

---

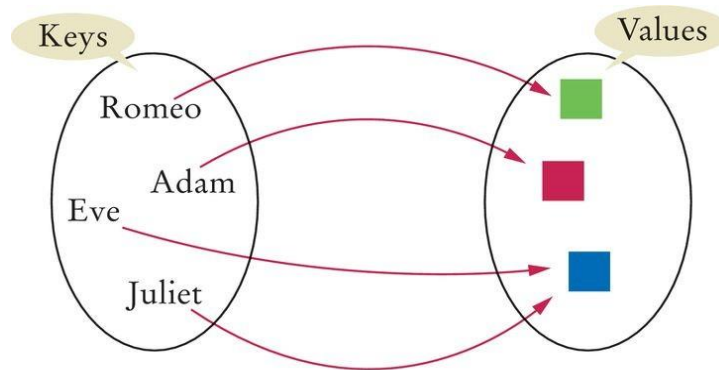
- The underlying structure for TreeSet is `BalancedTree`
- Duplicates are not allowed.
- Insertion order is not preserved, however, elements can be inserted according to some sorting order.
- Heterogeneous objects are not allowed. If you try to insert heterogeneous objects, you will get a runtime exception.
- Null insertion is not allowed.

# Collection Interface



# Maps

- A **Map** is an object that maps keys to values
- A map cannot contain duplicate keys
- Each key can map to at most one value
- Examples: dictionary, phone book, etc.





# Hash Map

---

- `Map` is an interface; you can't say `new Map( )`
- Here are two implementations:
  - `HashMap` is the faster
  - `TreeMap` guarantees the order of iteration
- 
- Good: `Map map = new HashMap( );`  
Fair: `HashMap map = new HashMap( );`





# HashMap – Basic operations

---

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( );



# More about put

---

- If the map already contains a given key, `put(key, value)` replaces the value associated with that key
- This means Java has to do equality testing on keys
- `void putAll(Map t);`
  - Copies one `Map` into another
  - Example: `newMap.putAll(oldMap);`
- `void clear();`
  - Example: `oldMap.clear();`



# Map: Collection views

---

- `public Set keySet( );` // duplicate is prohibited
- `public Collection values( );` // duplicate is allowed
- `public Set entrySet( );`
  - returns a set of `Map.Entry` (key-value) pairs
- You can create iterators for the key set, the value set, or the entry set (the set of entries, that is, key-value pairs)
- The above views provide the *only* way to iterate over a `Map`



# Map.Entry

---

- Entry Specific methods that can only apply on Entry object:
  - ```
public interface Entry { // Inner interface of Map
    Object getKey( );
    Object getValue( );
    Object setValue(Object value);
}
```
- This is a small interface for working with the Collection returned by `entrySet( )`
- Can get elements *only* from the `Iterator`, and they are only valid during the iteration



# Map example

---

- `import java.util.*;`

```
public class MapExample {
```

```
    public static void main(String[] args) {  
        Map<String, String> fruit = new HashMap<String, String>();  
        fruit.put("Apple", "red");  
        fruit.put("Pear", "yellow");  
        fruit.put("Plum", "purple");  
        fruit.put("Cherry", "red");  
        for (String key : fruit.keySet()) {  
            System.out.println(key + ": " + fruit.get(key));  
        }  
    }  
}
```

- Plum: purple  
Apple: red  
Pear: yellow  
Cherry: red



# HashMap

---

- The underlying structure is hash table.
- Insertion order is not preserved.
- Duplicate keys are not allowed however duplicate values are allowed.
- Heterogeneous keys and objects are allowed for keys and values.
- Null is allowed once for the key and multiple for the values.
- Best choice for frequent search operation.