JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. **Collections**
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. Functional Programming

Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

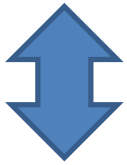
Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

Java Collection Framework

We group instances of some class into a Collection
to get a convenient way to handle them during iteration.

Collections



Data Structures

Collections util methods

Algorithms

JDK	Features
1.0	Vector, HashTable
1.2	Collection Framework
5	Generics
6	Deque, NavigableSet, NavigableMap
8	Functional programming
9	Static methods for collection creating
21	Sequenced Collections

Java Collection framework. Interfaces 1/3

- A collection, as its name implied, is simply an object that holds a collection (or a group, a container) of objects.
- Each item in a collection is called an element.
- A framework, by definition, is a set of interfaces that force you to adopt some design practices. A well-designed framework can improve your productivity and provide ease of maintenance.
- It also contains a set of classes that implements such interfaces
- It also contains classes with methods utilizing common algorithms (sort, search etc.)

Java Collection Framework Interfaces 2/3

There are four main interfaces in the Java Collections Framework:

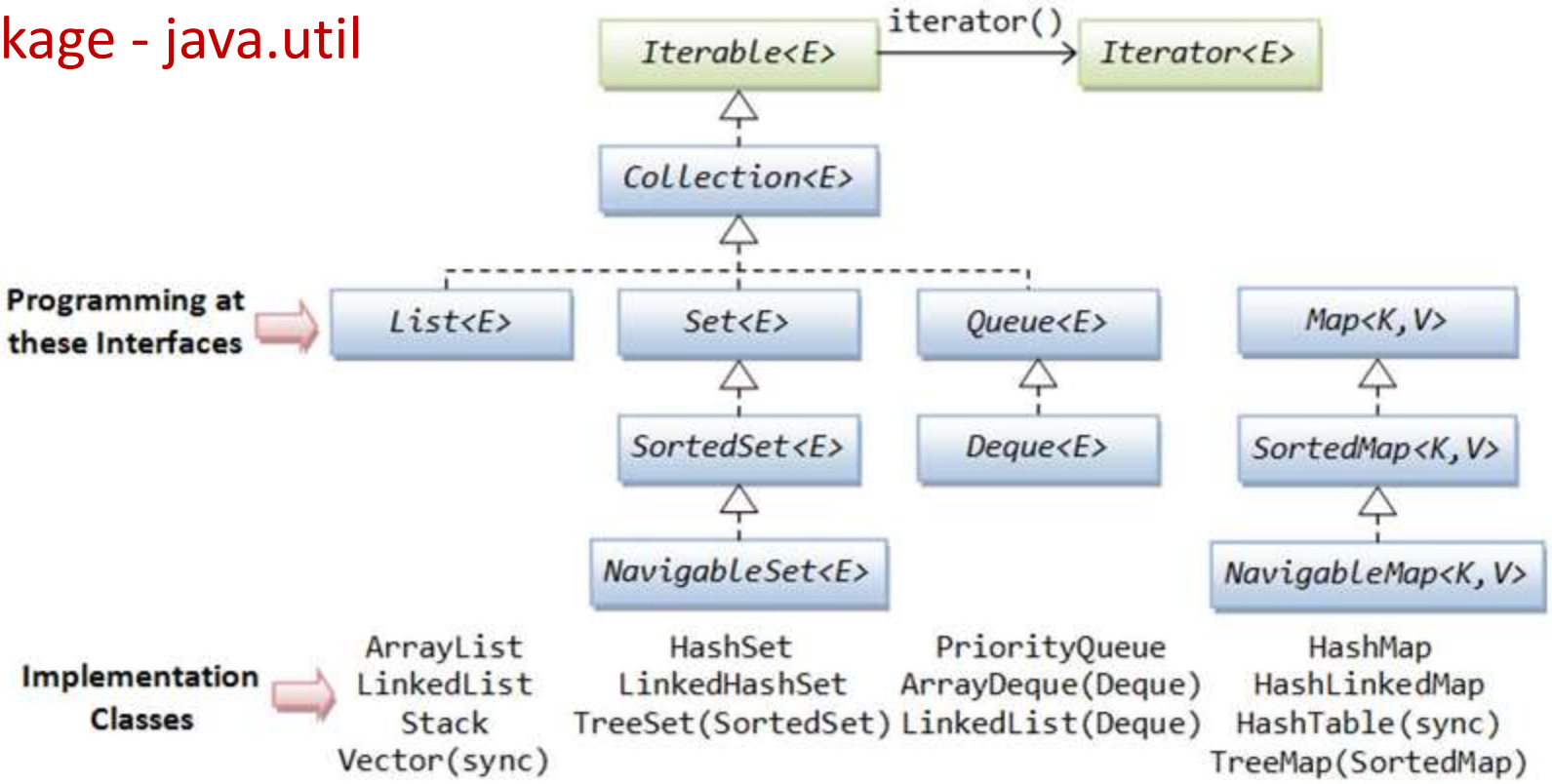
- **List:** A list is an ordered collection of elements that allows duplicate entries. Elements in a list can be accessed by an int index.
- **Set:** A set is a collection that does not allow duplicate entries.
- **Queue:** A queue is a collection that orders its elements in a specific order for processing. A Deque is a subinterface of Queue that allows access at both ends.
- **Map:** A map is a collection that maps keys to values, with no duplicate keys allowed. The elements in a map are key/value pairs.

since Java 1.2

Java Collection framework. Interfaces 3/3

- Collections

package - java.util



Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

The Collection interface 1/5

- The **Collection** interface is used to pass around collections of objects where maximum generality is desired
- No DIRECT implementations
- If method changes collection, it return true

The Collection interface 1/5

- The **Collection** interface is used to pass around collections of objects where maximum generality is desired

Collection
isEmpty() : boolean
parallelStream() : Stream<E>
removeIf(Predicate<? super E> filter) : boolean
spliterator() : Spliterator<E>
stream() : Stream<E>

See CollectionBasicOpsRunner

See CollectionBulkOpsRunner

See CollectionArrayOpsRunner

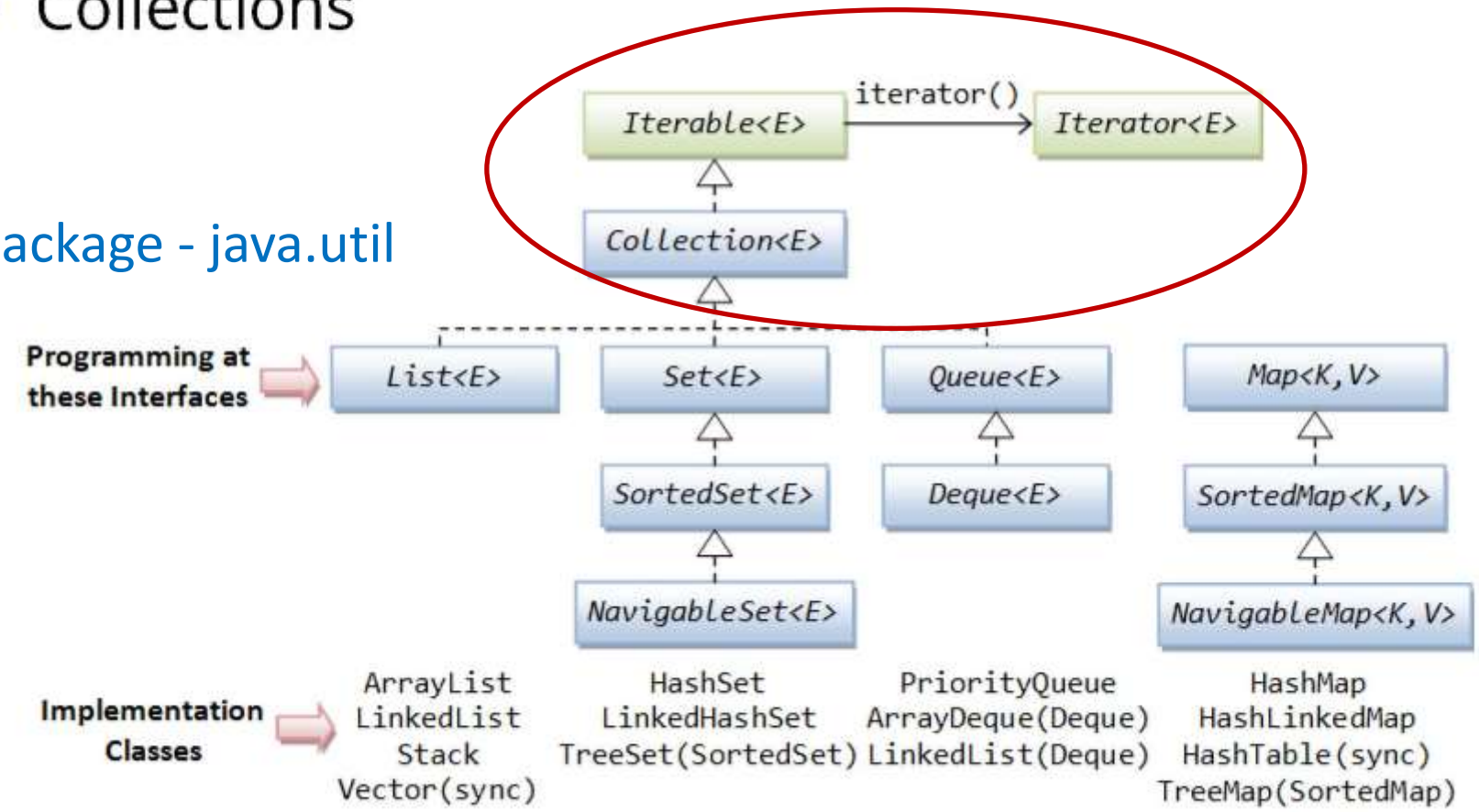
Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

Java Collection framework. Interfaces 2/3

- Collections

package - java.util



public interface java.lang.Iterable<T>

public interface Collection<E> extends Iterable<E> { .. }

Modifier and Type	Method	Description
default void	forEach(Consumer <? super T> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
Iterator<T>	iterator()	Returns an iterator over elements of type T.
default Spliterator<T>	spliterator()	Creates a Spliterator over the elements described by this Iterable.

Represents a data structure that can be iterated over.

Since Java 1.5

public interface java.util.Iterator<E>

Modifier and Type	Method	Description
default void	forEachRemaining(Consumer<? super E> action)	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
boolean	hasNext()	Returns true if the iteration has more elements.
E	next()	Returns the next element in the iteration.
default void	remove()	Removes from the underlying collection the last element returned by this iterator (optional operation).

Allows to retrieve or remove elements from a collection during the iteration.

Since Java 1.2

The Iterators

- By using of collection's iterator You can access each element in the collection.
- To re-traverse collection You must to obtain iterator again.
- You can not remove elements while collection traverse without iterator.
- You must use iterator's next() method before remove() method while collection traverse.

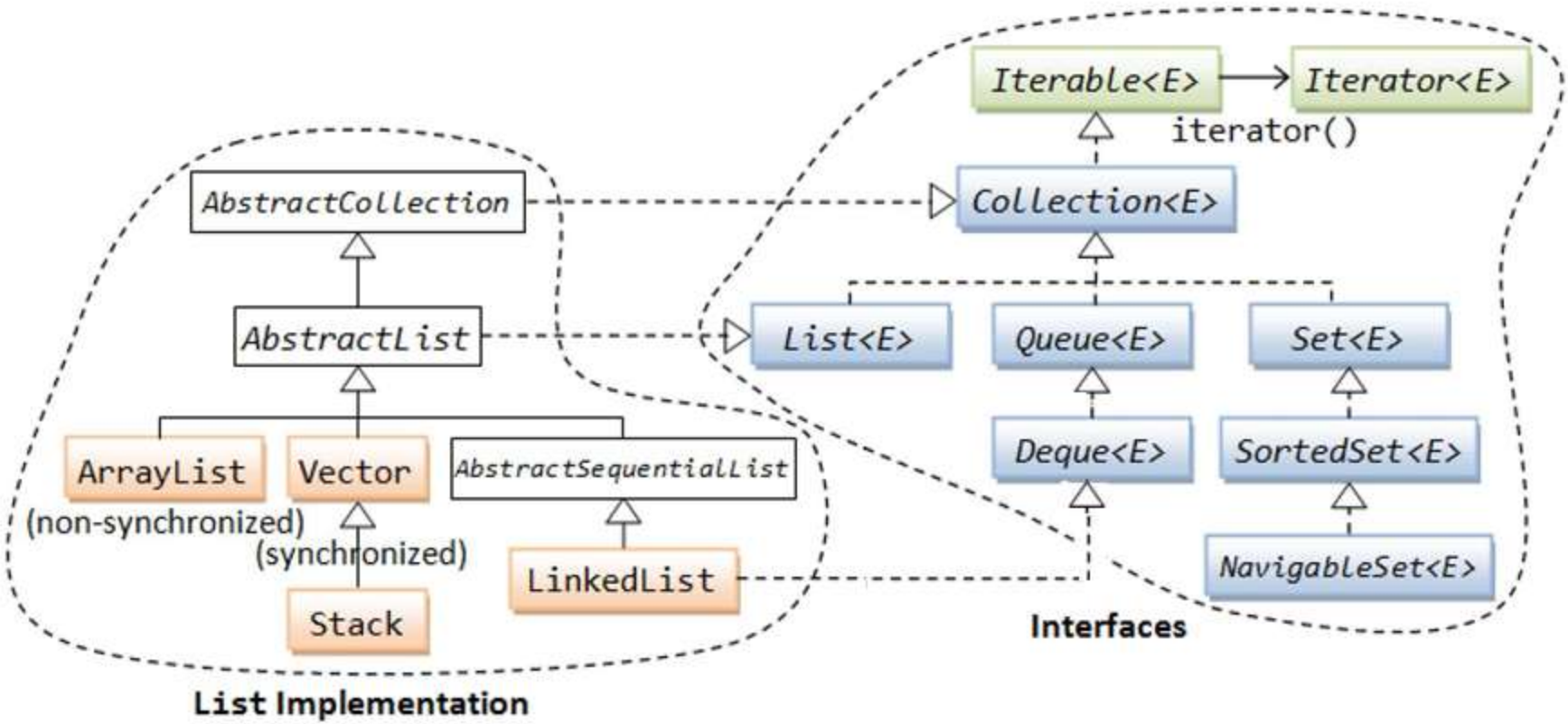
See IteratorRunner

Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

The List interface is a collection of ordered elements (for example, by the order of adding to the collection), which are accessed by index (positions in the collection).

Java Collection framework. Interfaces 3/3



List implements a mathematical abstraction of the list, can store the same elements at different indices.

The List interface 1/12

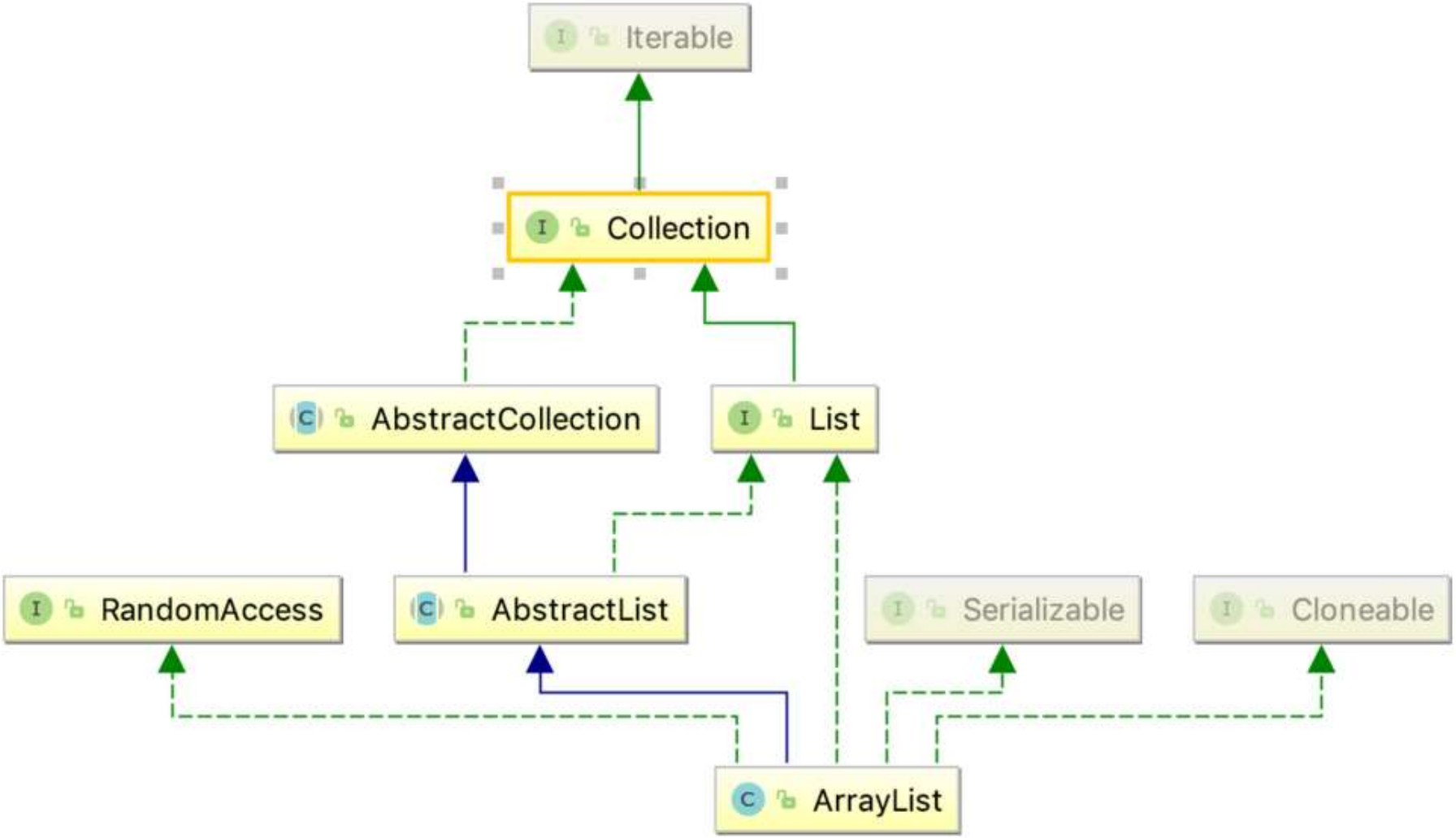
- The List Interface

+add(element : Object) : boolean
+add(index : int, element : Object) : void ✓
+addAll(collection : Collection) : boolean
+addAll(index : int, collection : Collection) : ✓
boolean
+clear() : void
+contains (element : Object) : boolean
+containsAll(collection : Collection) : boolean
+equals(object : Object) : boolean
+get(index : int) : Object ✓
+hashCode() : int
+indexOf(element : Object) : int ✓
+iterator() : Iterator

✓ - added in List interface

+lastIndexOf(element : Object) : int ✓
+listIterator() : ListIterator ✓
+listIterator(startIndex : Int) : ListIterator ✓
+remove(element : Object) : boolean
+remove(index : int) : Object ✓
+removeAll(collection : Collection) :
boolean
+retainAll(collection : Collection) :
boolean
+set(index : int, element : Object) : ✓
Object
+size() : int
+subList(fromIndex : int, toIndex : int) : ✓
List
+toArray() : Object[]
+toArray(array : Object[]) : Object[]

List implementation as ArrayList




```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Creating a List

Method	Description	Can add elements?	Can replace elements?	Can delete elements?
new ArrayList()	Returns **mutable** list	Yes	Yes	Yes
Arrays.asList(varargs)	Returns **fixed size** list backed by an array	No	Yes	No
List.of(varargs)	Returns **immutable** list	No	No	No
List.copyOf(collection)	Returns **immutable**	No	No	No

We can convert fixed-sized or immutable list through the list wrapping by List implementation's constructor

See ListRunner

public interface ListIterator<E> extends Iterator<E>

Modifier and Type	Method	Description
boolean	hasNext()	Returns true if the iteration has more elements in forward direction
E	next()	Returns the next element in the list and advances the cursor position
boolean	hasPrevious()	Returns true if the iteration has more elements in reverse direction
E	previous()	Returns the previous element in the list and move backwards the cursor position
int	nextIndex()	Returns the index of the element that would be returned by a subsequent call to next().
int	previousIndex()	Returns the index of the element that would be returned by a subsequent call to previous()

public interface ListIterator<E> extends Iterator<E>

Modifier and Type	Method	Description
void	remove()	Removes from the list the last element that was returned by next or previous (optional operation)
void	set(E e)	Replaces the last element returned by next or previous with the specified element (optional operation)
void	add(E e)	Inserts the specified element into the list (optional operation). The element is inserted immediately before the element that would be returned by next(), if any, and after the element that would be returned by previous(), if any. (If the list contains no elements, the new element becomes the sole element on the list)

See ListIteratorRunner

The ArrayList inner work

```
public class ArrayList<E> extends AbstractList<E>
      implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
...
private static final int DEFAULT_CAPACITY = 10;
...
/**
  * The array buffer into which the elements of the ArrayList are stored.
*/
transient Object[] elementData;
/**
 * The size of the ArrayList (the number of elements it contains).
*/
private int size;
```

See ArrayListInnerWork

The List interface 4/12

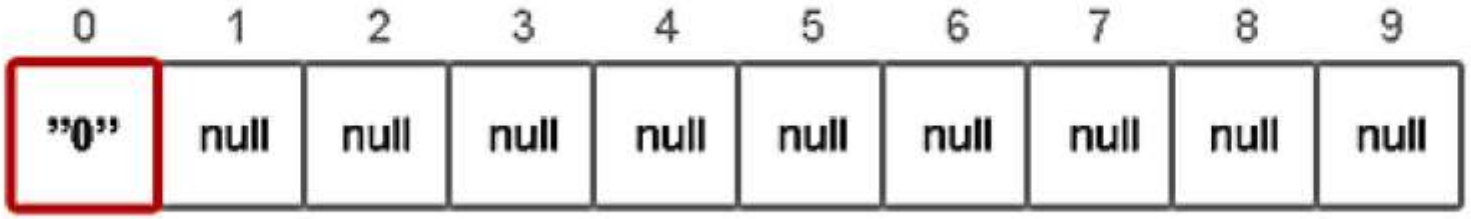
- ArrayList

1. List<String> myList = **new** ArrayList<>();

elementData = {}, size=0, capacity=0

2. myList.add("**0**");

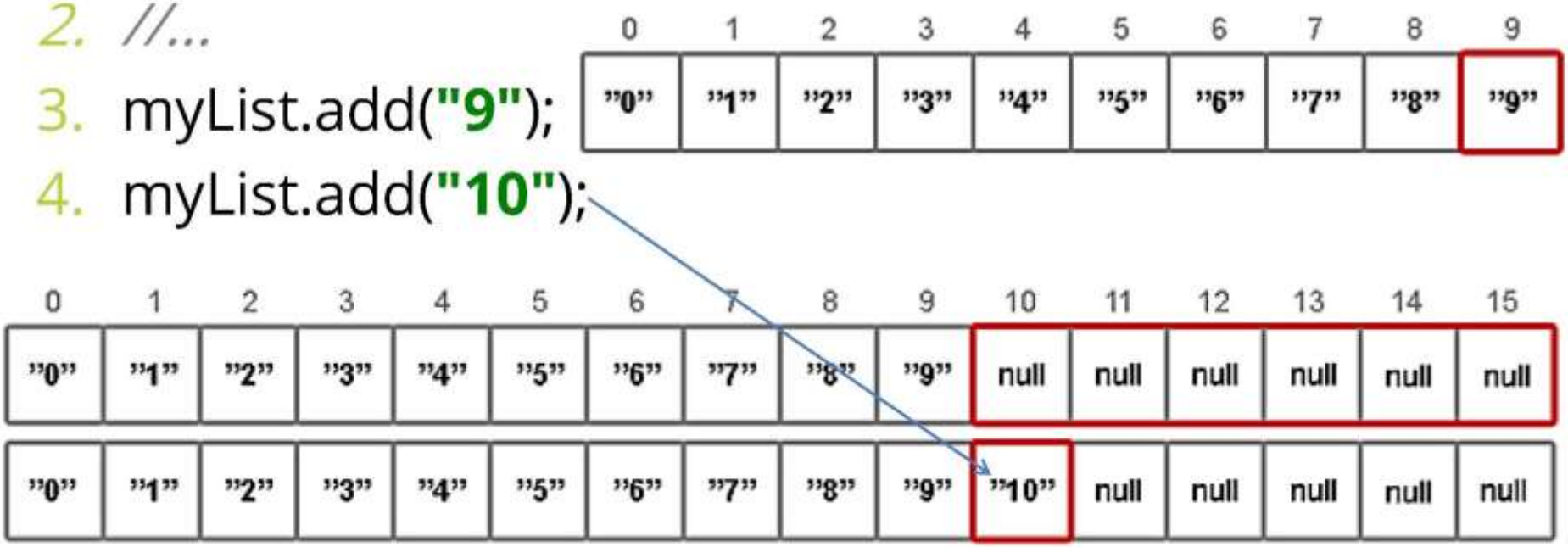
size=1, capacity=10



The List interface 5/12

- ArrayList

1. myList.add("1");
2. //...
3. myList.add("9");
4. myList.add("10");



newCapacity = oldCapacity + (oldCapacity >> 1)

15, 22, 33, ...

The List Interface methods implementation

```
public boolean add(E e) {
    modCount++;
    add(e, elementData, size);
    return true;
}
private void add(E e, Object[] elementData, int s) {
    if (s == elementData.length)
        elementData = grow();
        elementData[s] = e;
        size = s + 1;
}
private Object[] grow() {
    return grow(size + 1);
}
```

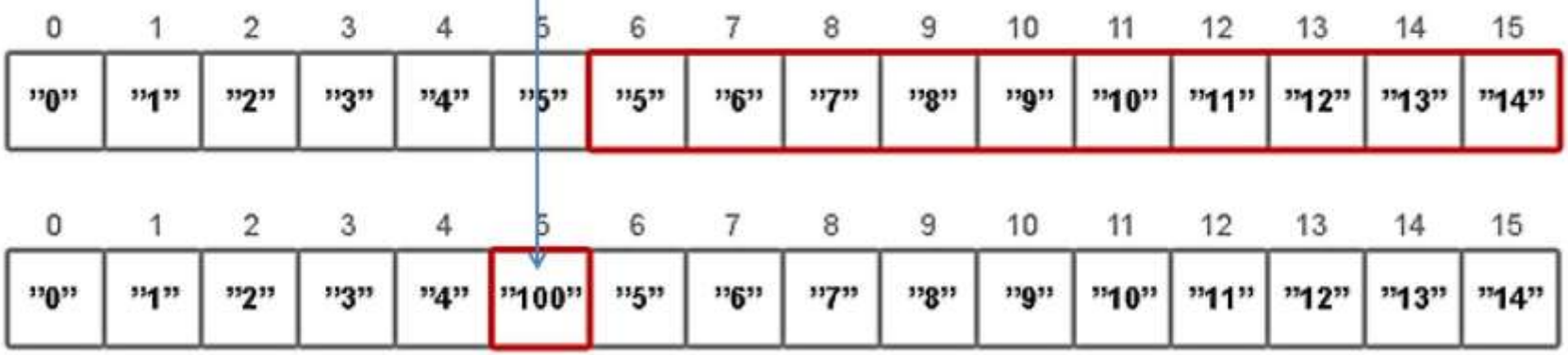
The List Interface methods implementation

```
private Object[] grow(int minCapacity) {
   return elementData = Arrays.copyOf(elementData,
                                      newCapacity(minCapacity));
}
private int newCapacity(int minCapacity) {
   int oldCapacity = elementData.length;
   int newCapacity = oldCapacity + (oldCapacity >> 1);
      if (newCapacity - minCapacity <= 0) {
         if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
            return Math.max(DEFAULT_CAPACITY, minCapacity);
         if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
         return minCapacity;
      }
      return (newCapacity - MAX_ARRAY_SIZE <= 0) ? newCapacity
               : hugeCapacity(minCapacity);          }
```

The List interface 6/12

- myList.add("14");

- myList.add(5, "100");

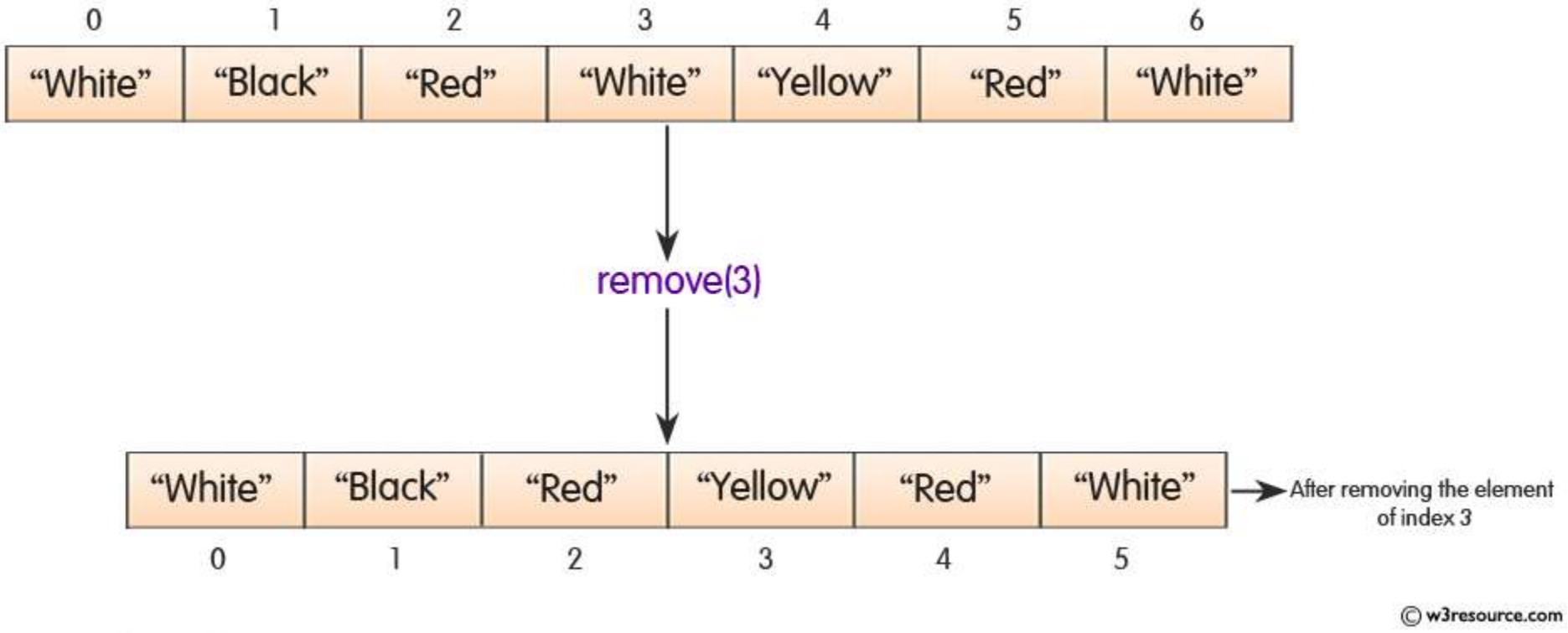


The List Interface methods implementation

```
public void add(int index, E element) {
    rangeCheckForAdd(index);
    modCount++;
    final int s;
    Object[] elementData;
    if ((s = size) == (elementData = this.elementData).length)
        elementData = grow();
        System.arraycopy(elementData, index,
                         elementData, index + 1,
                         s - index);
    elementData[index] = element;
    size = s + 1;
}
```

See ArrayListInnerWork

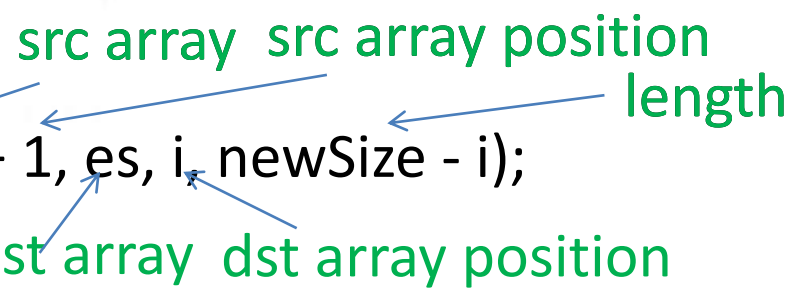
The List Interface



The List Interface methods implementation

```
public E remove(int index) {
        Objects.checkIndex(index, size);
        final Object[] es = elementData;
        @SuppressWarnings("unchecked")
        E oldValue = (E) es[index];
        fastRemove(es, index);
        return oldValue;
}

private void fastRemove(Object[] es, int i) {
        modCount++;
        final int newSize;
        if ((newSize = size - 1) > i)
            System.arraycopy(es, i + 1, es, i, newSize - i);
        es[size = newSize] = null;
}
```



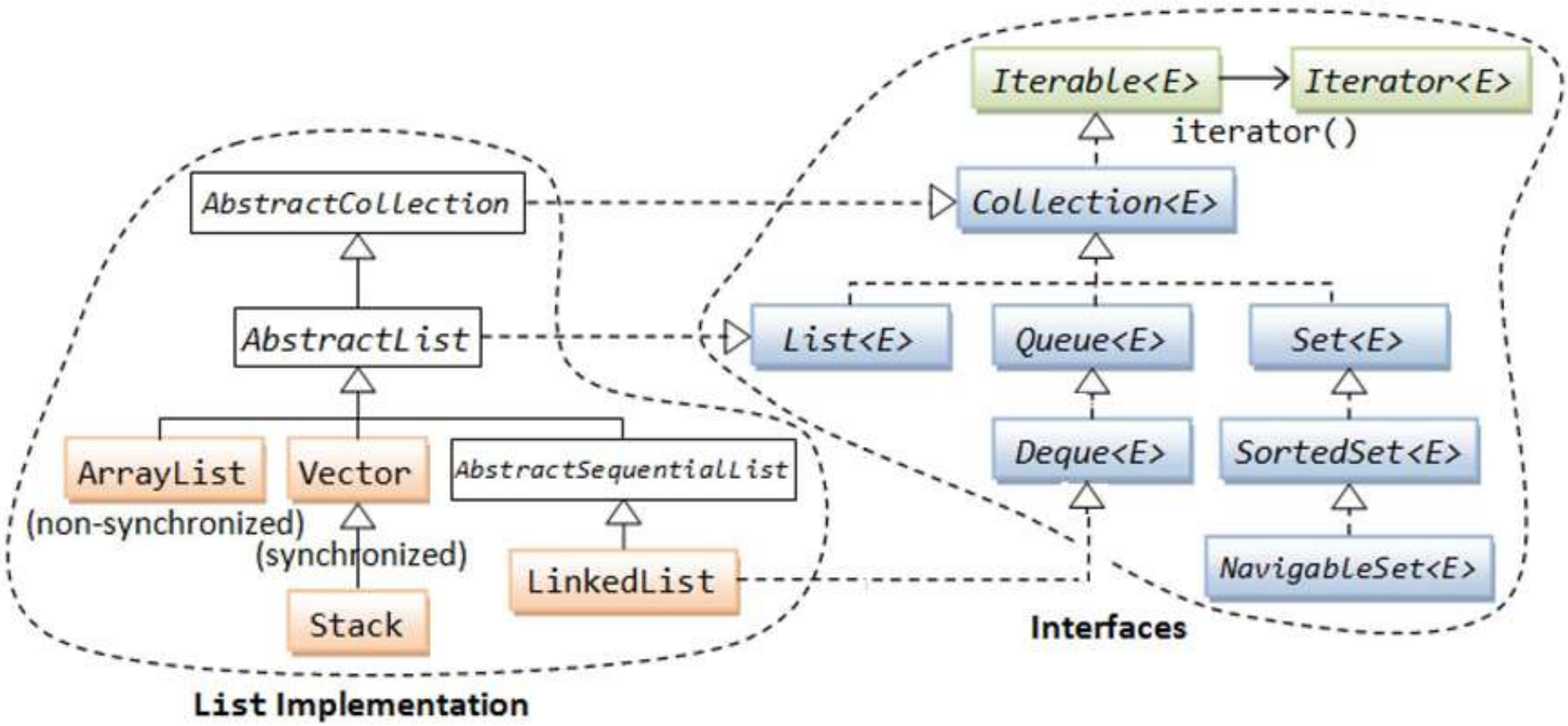
See ArrayListInnerWork

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable

Modifier and Type	Method	Description
boolean	ensureCapacity(int minCapacity)	Increases the capacity of this ArrayList instance
void	trimToSize()()	Trims the capacity of this ArrayList instance to be the list's current size
protected void	removeRange(int fromIndex, int toIndex)	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index)

See ArrayListRunner

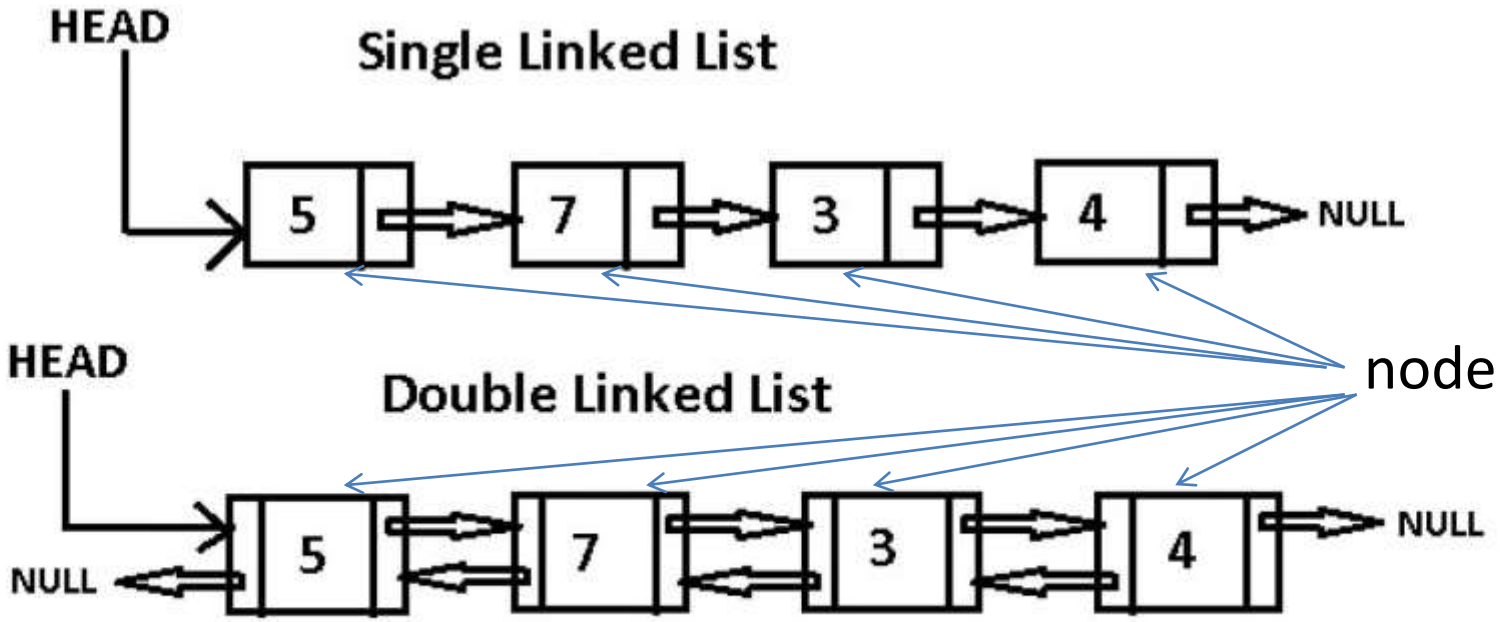
List implementation as LinkedList




```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

The List interface 7/12

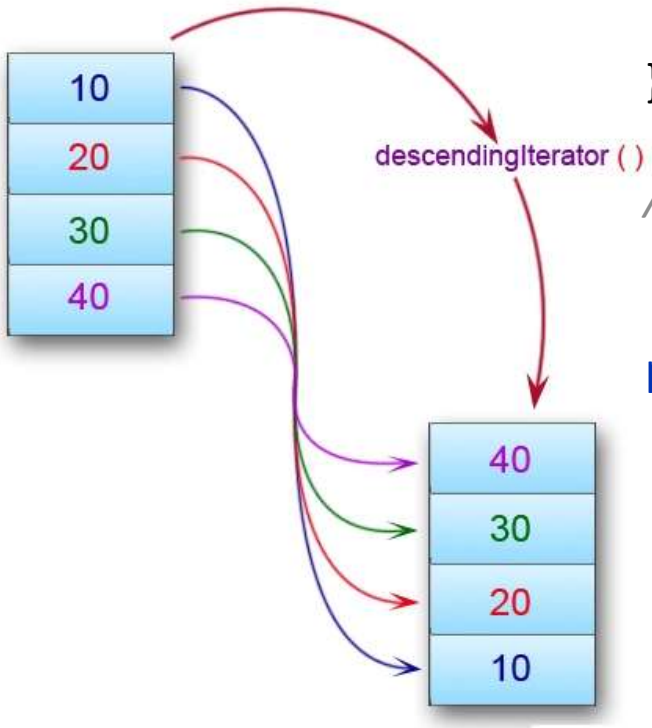
- A **linked list** is a data structure in which data is stored in structures called nodes and every node has a pointer to the next node in the list



The LinkedList specific methods

	Add element to start of list	Add element to end of list	Get element from start of list	Get element from end of list
Element does not remove from the list			throws NoSuchElementException for empty list	
	void addFirst(E e) (Deque)	void addLast(E e) (Deque)	E getFirst() (Deque) E element() (Queue)	E getLast() (Deque)
			returns null for empty list	
	boolean offerFirst(E e) (Deque) void push(E e) (Deque)	boolean offer(E e) (Queue) boolean offerLast(E e) (Deque)	E peek() (Queue) E peekFirst() (Deque)	E peekLast() (Deque)
Element removes from the list			throws NoSuchElementException for empty list	
			E remove() (Queue) E removeFirst() (Deque) E pop() (Deque)	E removeLast() (Deque)
			returns null for empty list	
			E poll() (Queue) E pollFirst() (Deque)	E pollLast() (Deque)

The LinkedList specific methods




```
public Iterator<E> descendingIterator() {
    return new DescendingIterator();
}

/**
 * Adapter to provide descending iterators via ListItr.previous
 */
private class DescendingIterator implements Iterator<E> {
    private final ListItr itr = new ListItr(size());
    public boolean hasNext() {
        return itr.hasPrevious();
    }
    public E next() {
        return itr.previous();
    }
    public void remove() {
        itr.remove();
    }
}
```

public boolean removeFirstOccurrence(Object o) and
public boolean removeLastOccurrence(Object o) were added too.

See LinkedListRunner

Since JDK 6

LinkedList inner work

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable {

    transient int size = 0;
    transient Node<E> first;
    transient Node<E> last;
...

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
        Node(Node<E> prev, E element,
                Node<E> next) {...}
...}
}
```

first node

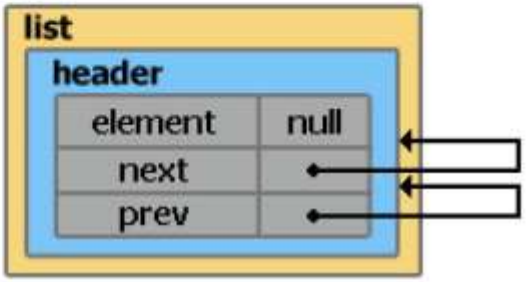
last node

next node

previous node

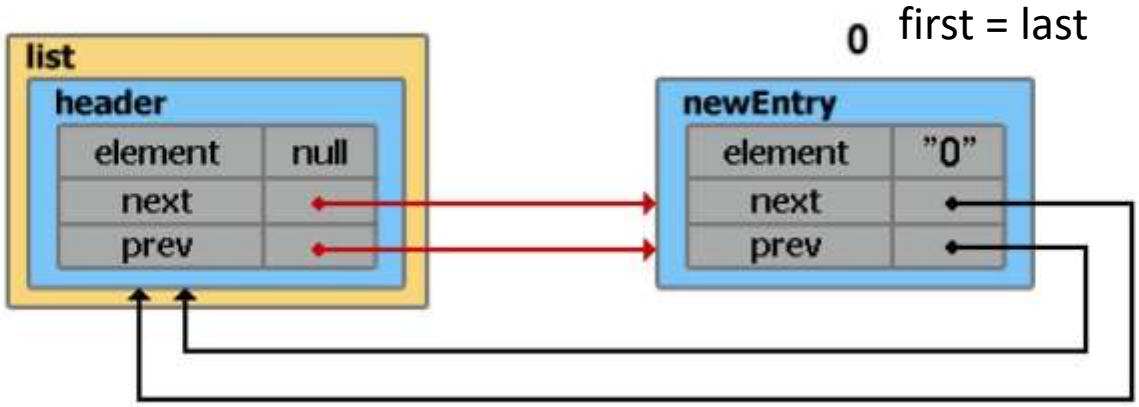
The List interface 9/12

- List<String> list = **new** LinkedList<>();



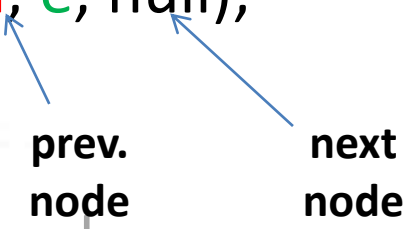

```
assert (size == 0)
    ? (first == null && last == null)
    : (first.prev == null && last.next == null);
```

- list.add("0");



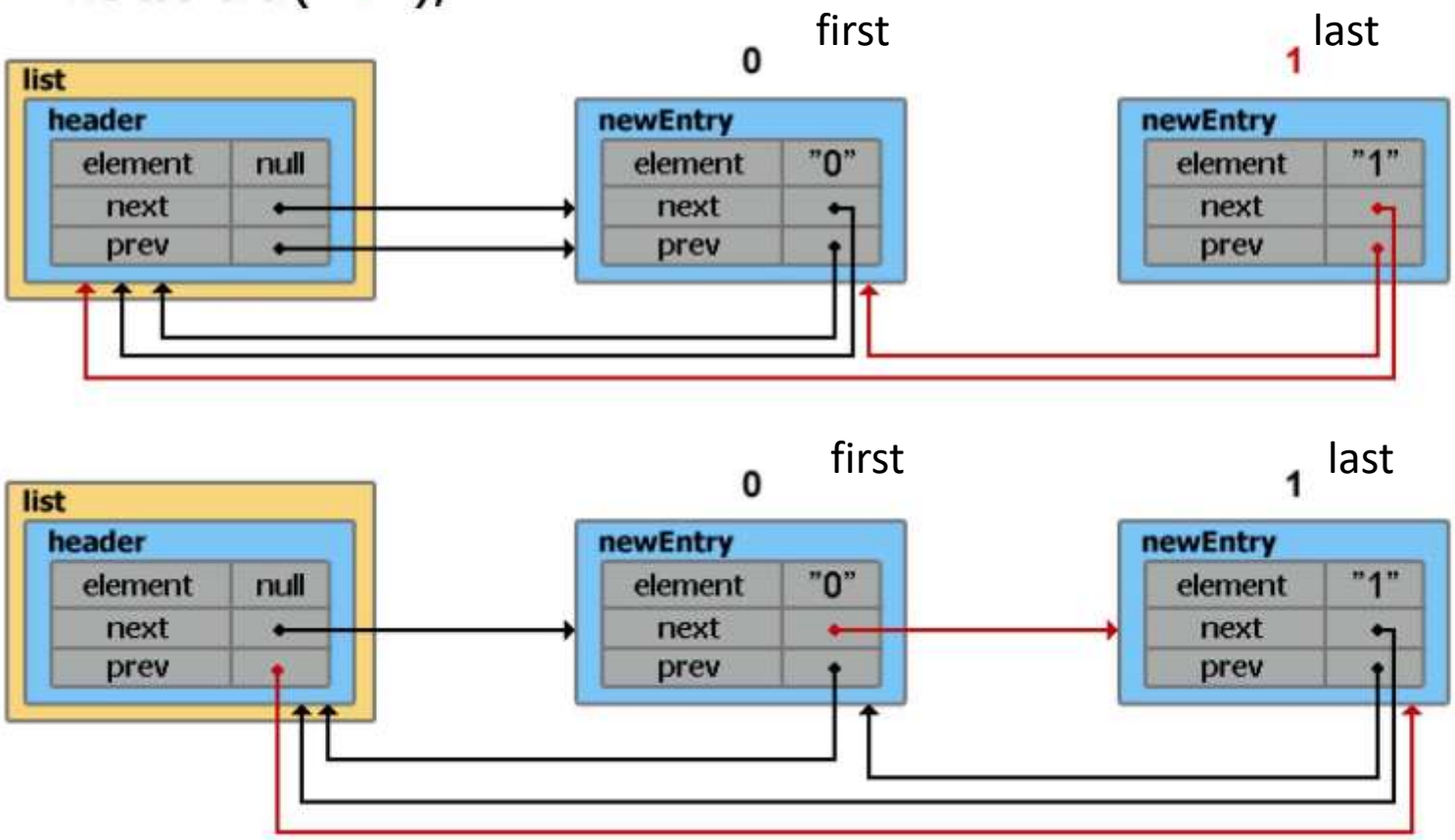
LinkedList inner work

```
public boolean add(E e) {
        linkLast(e);
        return true;
}
void linkLast(E e) {
        final Node<E> l = last; //take header.last value
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode; // set last to last node
        if (l == null) //if list empty
            first = newNode;       //set first to first node
        else
            l.next = newNode; //set next pointer of last node
        size++;
        modCount++;
}
```



The List interface 10/12

- list.add("1");



LinkedList inner work

```
public void add(int index, E element) {
      … if (index == size)
            linkLast(element);
      else
            linkBefore(element, node(index));
   }
void linkBefore(E e, Node<E> succ) {  // assert succ != null
      final Node<E> pred = succ.prev; //ук-ль на пред.
      final Node<E> newNode = new Node<>(pred, e, succ);
      succ.prev = newNode;
      if (pred == null)
         first = newNode;
      else
         pred.next = newNode;
      size++;        modCount++;     }
```

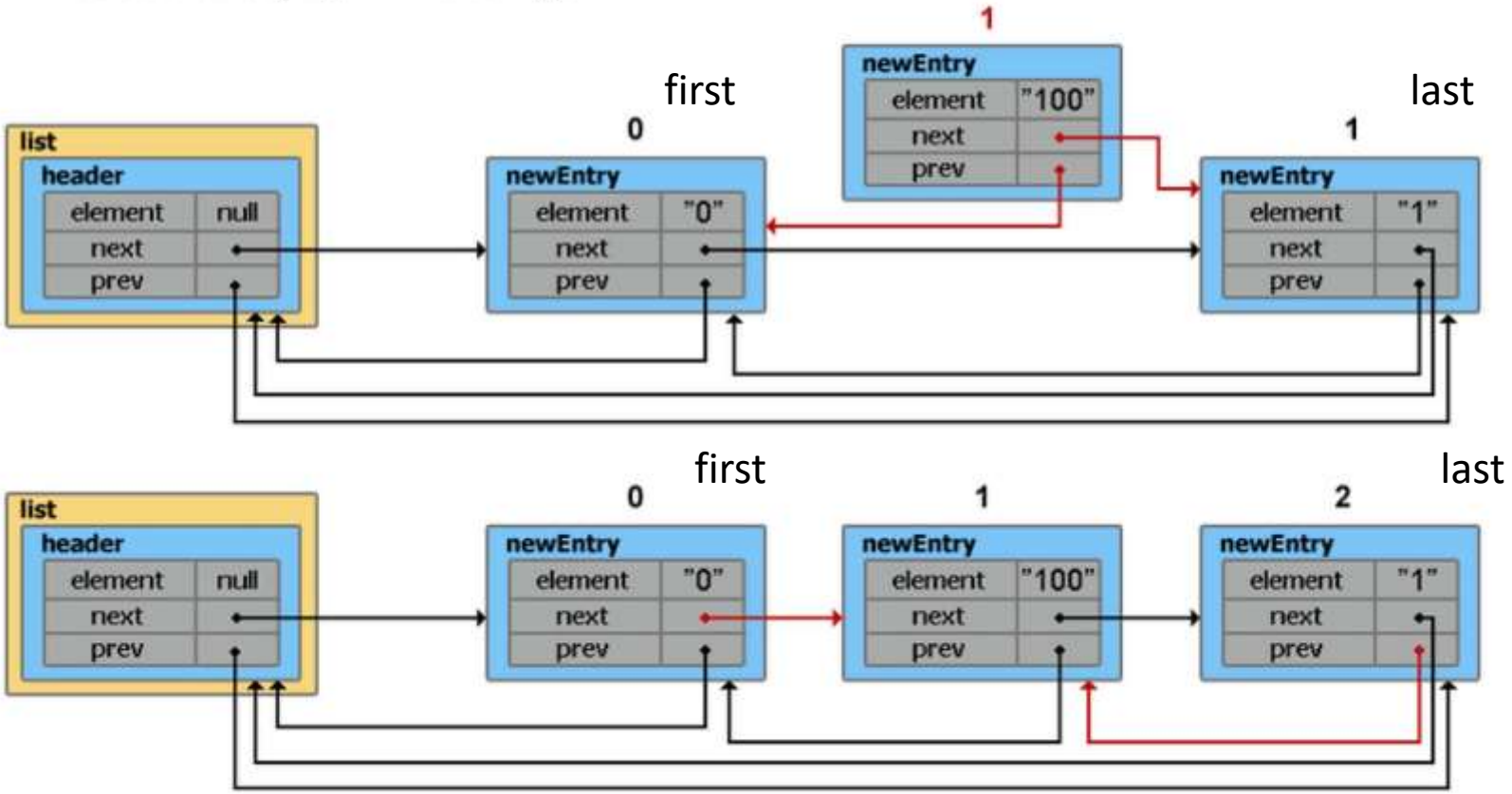
узел, после которого вставка

пред. узел

след. узел

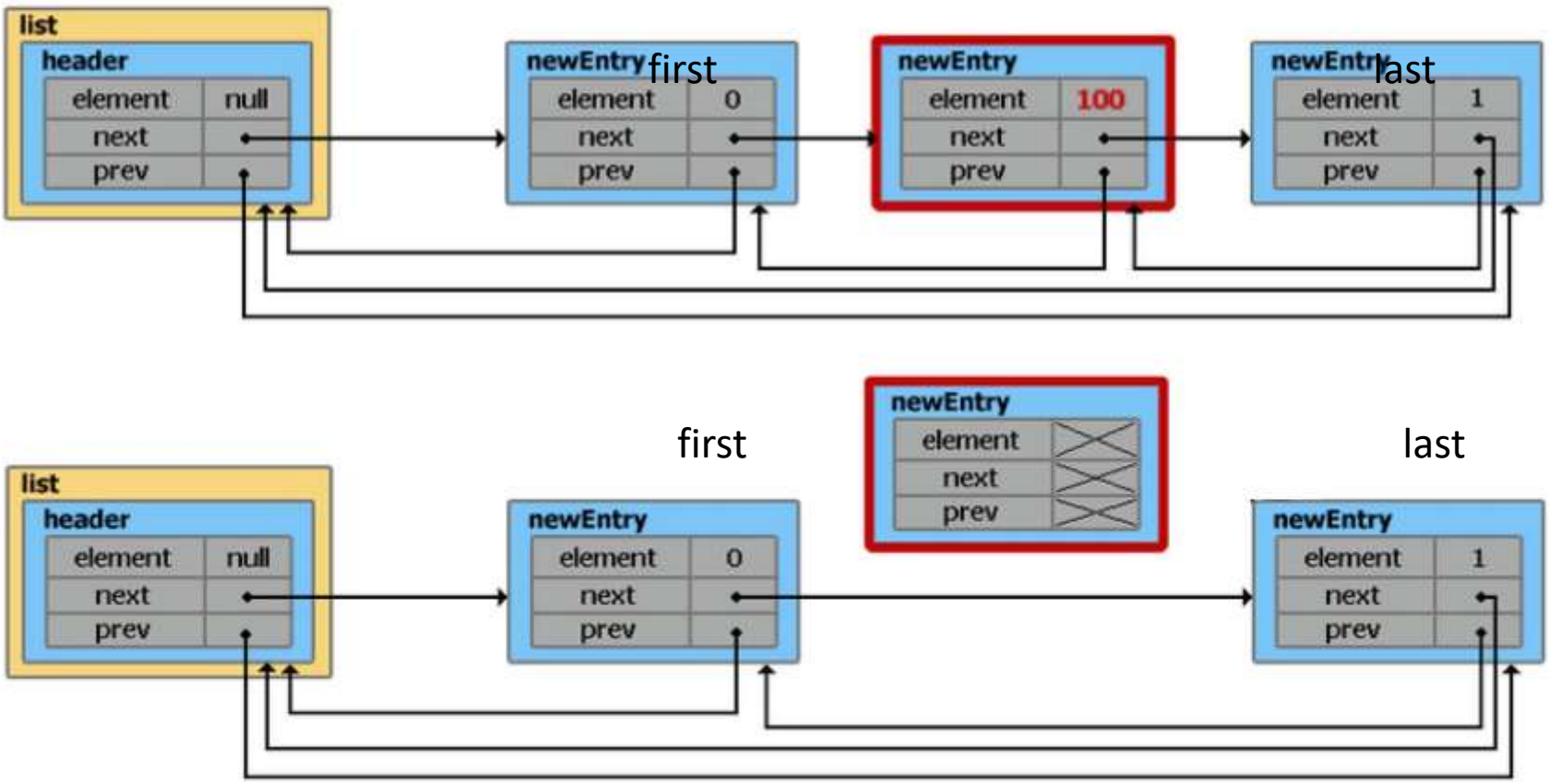
The List interface 11/12

- list.add(1, **"100"**);



The List interface 12/12

- list.remove("100");



LinkedList inner work

```
public boolean remove(Object o) {
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
    return false;     }
```

LinkedList inner work

```
E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;
    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }
    if (next == null) {
        last = prev;
    } else {
        next.prev = prev;
        x.next = null;
    } ...
```

```
...
        x.item = null;
        size--;
        modCount++;
        return element;
    }
```

See LinkedListInnerWork

ArrayList and LinkedList Performance Comparison

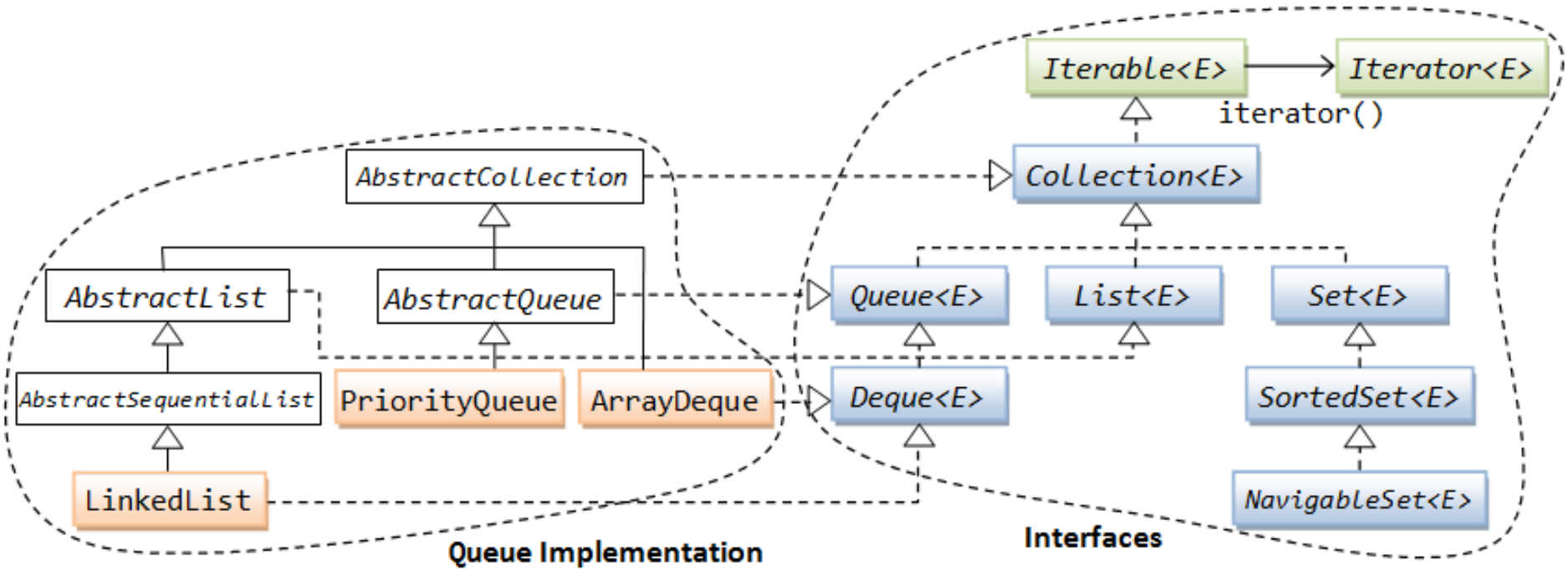
	Start of the list	The list middle	End of the list	Homogeneous in the list
Insert	LinkedList	ArrayList	≈	ArrayList
Remove	LinkedList	ArrayList	ArrayList	ArrayList
Update	ArrayList	ArrayList	ArrayList	ArrayList
Get	ArrayList	ArrayList	ArrayList	ArrayList

See ArrayListAndLinkedListComparision

Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

The Queue Interface 1/6



The Queue interface 2/6

- The Queue
- The first element that is inserted is the first to be removed
- Often referred to as First-in First-out (FIFO) collection

- The Deque is a linear collection that supports element insertion and removal at both ends

Queue vs Deque

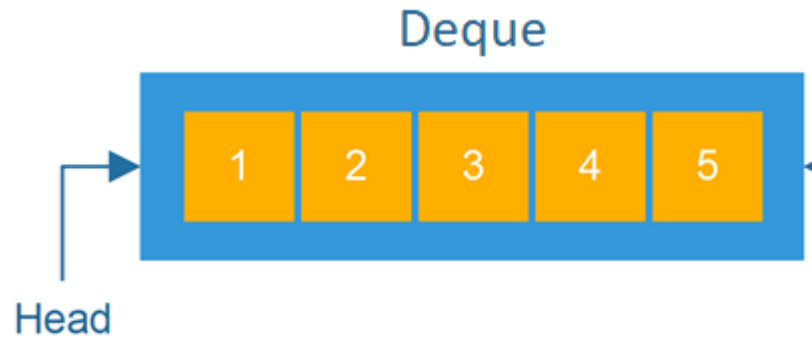
E peek()
E element()
gets, doesn't remove

boolean offer(E e)

E poll()
E remove()
gets, remove

E peekFirst()
E getFirst()
gets, doesn't remove

E peekLast()
E getLast()
gets, doesn't remove



void addFirst(E e)
boolean offerFirst(E e)
void push(E e)

void addLast(E e)
boolean offerLast(E e)

E pollFirst()
E removeFirst()
E pop()
gets, remove

See QueueRunner
See DequeRunner

E pollLast()
E removeFLast()
gets, remove

The Queue interface methods

LinkedList implementation

```
boolean add(E e)        -> linkLast(e);
                           return true;
boolean offer(E e);     -> return add(e);

E element()             -> return getFirst();
E peek();               -> final Node<E> f = first;
                           return (f == null) ? null : f.item;

E poll();               -> final Node<E> f = first;
                           return (f == null) ? null : unlinkFirst(f);
E remove();             ->return removeFirst();
```

remain in the queue

removed from the queue

```
public E getFirst() {
   final Node<E> f = first;
   if (f == null)
      throw
      new NoSuchElementException();
   return f.item; }
```

```
public E removeFirst() {
      final Node<E> f = first;
      if (f == null)
         throw
new NoSuchElementException();
      return unlinkFirst(f);    }
```

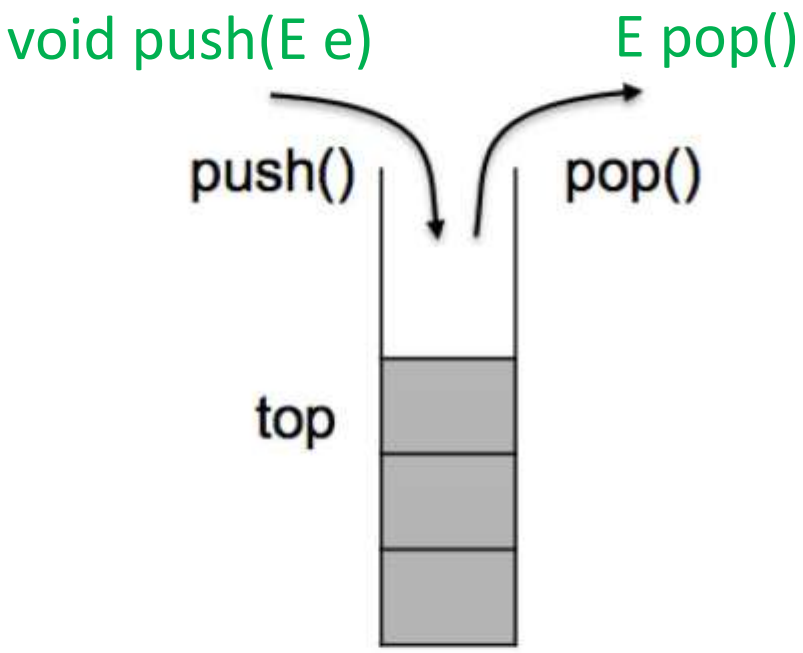
The Queue interface methods

LinkedList implementation

```
private E unlinkFirst(Node<E> f) {
    // assert f == first && f != null;
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;
    f.next = null; // help GC
    first = next;
    if (next == null)
        last = null;
    else
        next.prev = null;
    size--;
    modCount++;
    return element;
}
```

The Deque interface 5/6

- Stack
- Elements are stored in order of insertion and the elements are removed in reverse order
- Often referred to as a Last-in First-out (LIFO) collection



See StackRunner

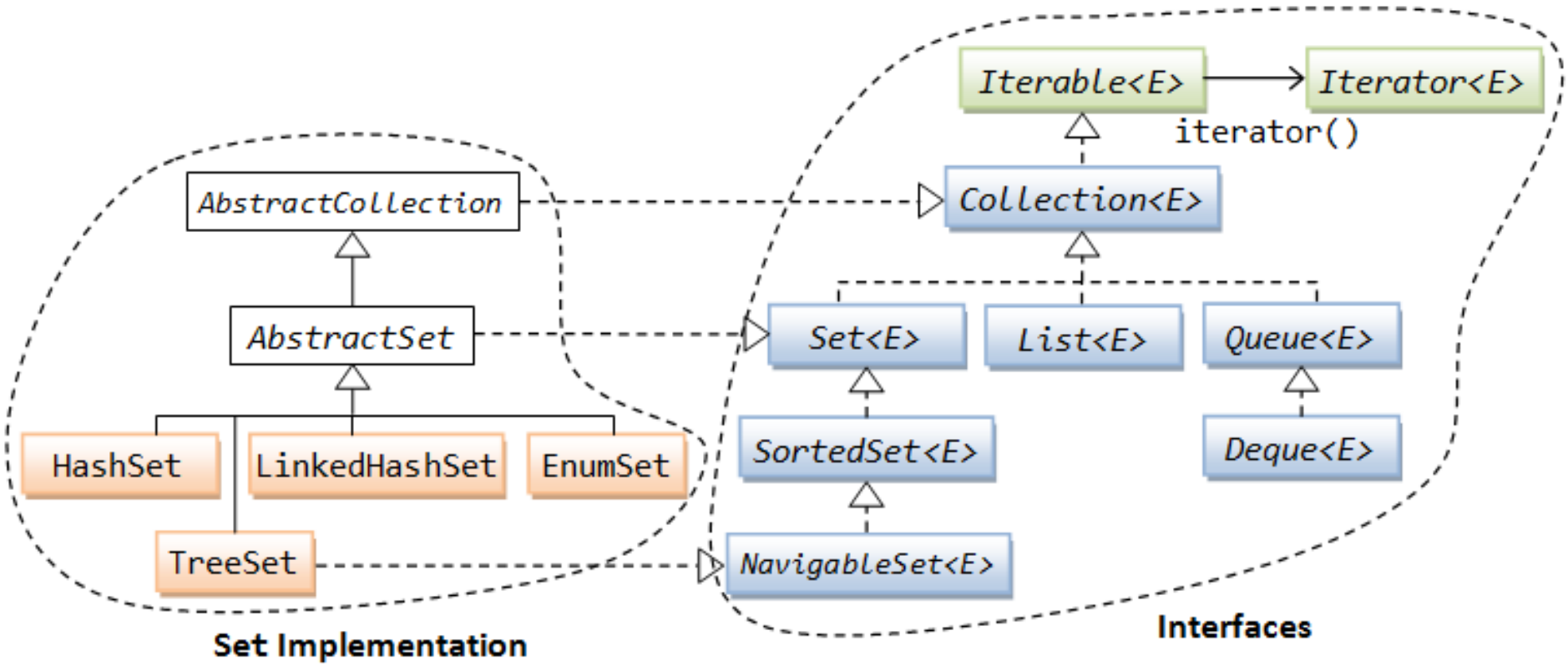
Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

The Set interface

- A Set is a Collection that cannot contain duplicate elements
- Set interface models the mathematical set abstraction.
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.
- Two Set instances are equal if they contain the same elements.

The Set interface



The HashSet implementation

- Hashset no guarantees concerning the order of elements
- The class of instances added to HashSet has to override equals() and hashcode() methods

See HashSetRunner

See Student & StudentMain

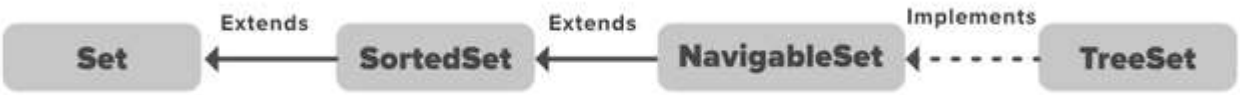
The HashSet inner work

```
public class HashSet<E> extends AbstractSet<E>
                        implements Set<E>, Cloneable, java.io.Serializable {
    ...
    private transient HashMap<E,Object> map;
    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();
...
    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
...
public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }
```

Inner work will be explored for Map<K,V>

The TreeSet implementation

- *TreeSet* is one of the most important implementations of the *SortedSet* interface that uses a binary search tree for storage.
- The ordering of the elements is maintained by a set using their natural ordering (*element must be comparable*) whether or not an *explicit comparator* is provided.
- *TreeSet* is basically an implementation of a *self-balancing binary search tree* like a *Red-Black Tree*.



See TreeSetRunner

The TreeSet inner work

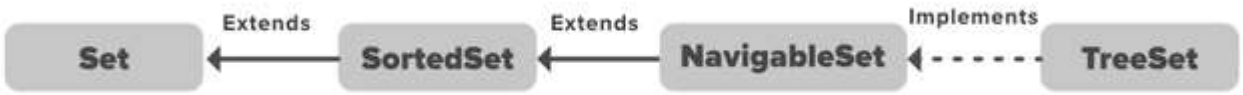
```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable {
     ...
    public TreeSet() {
        this(new TreeMap<>());
    }

    public TreeSet(Comparator<? super E> comparator) {
        this(new TreeMap<>(comparator));
    }
...
}
```

Inner work will be explored for TreeMap<K,V>

The SortedSet interface

- The *SortedSet* is a Set that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a Comparator provided at SortedSet creation time.
- In addition to the normal Set operations, the SortedSet interface provides operations for the following:
  - allows arbitrary range operations on the sorted set
  - returns the first or last element in the sorted set
  - returns the Comparator, if any, used to sort the set



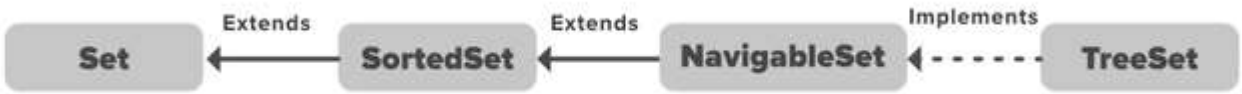
The SortedSet interface

```
public interface SortedSet<E> extends Set<E> {
   Comparator<? super E> comparator();
   /*Returns a range of this set whose elements:
     are strictly less than toElement.*/
   SortedSet<E> headSet(E toElement);
   /*are strictly greater than or equal fromElement.*/
   SortedSet<E> tailSet(E fromElement);
   /*range from fromElement, inclusive, to toElement, exclusive*/
   SortedSet<E> subSet(E fromElement, E toElement);
   E first();
   E last();
   default Spliterator<E> spliterator() {...}
```

See SortedSetRunner

The NavigableSet interface

- The *NavigableSet* is a SortedSet extended with navigation methods reporting closest matches for given search targets and include/exclude bound(s).
- Methods *lower()*, *floor()*, *ceiling()*, and *higher()* return elements respectively *less than*, *less than or equal*, *greater than or equal*, and *greater* than a given element, returning null if there is no such element.
- A NavigableSet may be accessed and traversed in either *ascending* or *descending* order.
- The *descendingSet* method returns a view of the set with the senses of all relational and directional methods inverted.



The NavigableSet interface

```
public interface NavigableSet<E> extends SortedSet<E> {
   Iterator<E> descendingIterator();
   NavigableSet<E> descendingSet();
   NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
                          E toElement,   boolean toInclusive);
   NavigableSet<E> headSet(E toElement, boolean inclusive);
   NavigableSet<E> tailSet(E fromElement, boolean inclusive);
   E floor(E e);
   E ceiling(E e);
   E lower(E e);
   E higher(E e);
   E pollFirst();
   E pollLast();
}
```

allows to retrieve elements based on their values

See NavigableSetRunner

since JDK 6

The LinkedHashSet interface

- The *LinkedHashSet* is an ordered version of HashSet that maintains a doubly-linked List across all elements.
- When the iteration order is needed to be maintained this class is used. *The iteration order is defined by the insertion order of the elements into the set*.
- When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.
- LinkedHashSet does not add extra methods to HashSet methods.

See LinkedHashSetRunner

since JDK 1.4

LinkedHashSet inner work

```
public class LinkedHashSet<E> extends HashSet<E>
   implements Set<E>, Cloneable, java.io.Serializable {
...
   public LinkedHashSet(int initialCapacity) {
    super(initialCapacity, .75f, true);
}
...
}
public class HashSet<E> extends AbstractSet<E>
   implements Set<E>, Cloneable, java.io.Serializable {
...
HashSet(int initialCapacity, float loadFactor, boolean dummy) {
   map = new LinkedHashMap<>(initialCapacity, loadFactor);
}
...
}
```

Inner work will be explored for LinkedHashMap<K,V>

See LinkedHashSetRunner

The Set interface 6/6

Constructs a new set containing the elements in the specified collection

```
1. List<String> myList = new ArrayList<>();
2. myList.add("aaa");
3. myList.add("bbbb");
4. myList.add("bbbb");
5. myList.add("aaa");
6. myList.add("ccccc");
7. Set<String> mySet = new HashSet<>(myList);
8. System.out.println(myList);
9. System.out.println(mySet);
```

We can change collection type by use as constructor argument

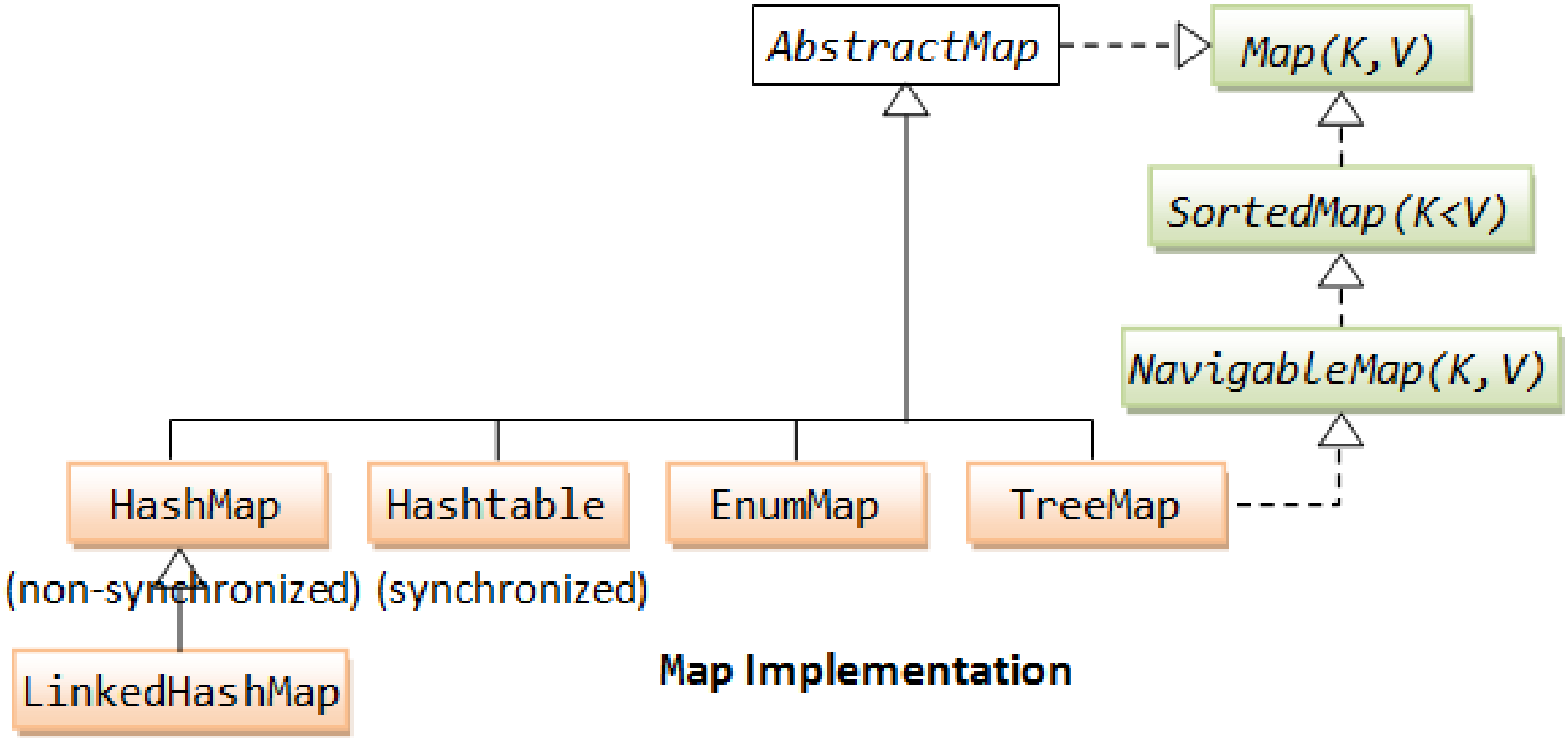
Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

The Map interface

- A Map contains *values on the basis of key*, i.e. *key* and *value pair*.
- Each key and value pair is known as an *entry*.
- A Map contains *unique keys*.
- A Map is useful if you have to search, update or delete elements on the basis of a key.

The Map interface 1/13



Map Implementation

The Map interface methods

Method	Description
V **put**(K key, V value)	Inserts the key-argument and associated with it value argument as an entry to the map (optional operation). Returns the previous value associated with key, or null if there was no mapping for key.
void **putAll**(Map<? extends K,? extends V> m)	Copies all of the mappings from the map argument to this map (optional operation).
V **putIfAbsent**(K key, V value)	If the key argument is not already associated with a value argument (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V **get**(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V **getOrDefault**(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

The Map interface methods

Method	Description
Set>K> **keySet**()	Returns the Set view containing all the keys.
Collection<V> **values**()	Returns a Collection view of the values contained in this map.
Set<Map.Entry<K,V>> **entrySet**()	Returns the Set view containing all the key-value pairs.
V **replace**(K key, V value)	Replaces the entry for the specified key only if it is currently mapped to some value.
boolean **replace**(K key, V oldValue, V newValue)	Replaces the entry for the specified key only if currently mapped to the specified value.
void **replaceAll**(BiFunction<? super K,? super V,? extends V> function)	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
V **merge**(K key, V value, BiFunction<? super V,? super V,? extends V> remapFunc)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

The Map interface methods

Method	Description
V **compute**(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	Attempts to update a value for the key argument with remapping function argument (if remapping function is null removes the entry).
V **computeIfPresent**(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	Updates a value for the key argument with remapping function argument only if the map contains the entry of key argument.
V **computeIfAbsent**(K key, Function<? super K,? extends V> mappingFunction)	Updates a value for the key argument with remapping function argument only if the map does not contain the entry of key argument.
V **remove**(Object key)	Removes an entry for key-argument if it is present (optional operation).
boolean **remove**(Object key, Object value)	Removes the entry for the specified key only if it is currently mapped to the specified value.

The Map interface methods

Method	Description
void **clear**()	Removes all entries from the map.
void **forEach**(BiConsumer<? super K,? super V> action)	Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
boolean **containsKey**(Object key)	Returns true if this map contains a mapping for the specified key.
boolean **containsValue**(Object value)	Returns true if this map maps one or more keys to the specified value.
boolean **isEmpty**()	Returns true if this map contains no key-value mappings.
boolean **equals**(Object o)	Compares the specified object with this map for equality.
int **hashCode**()	Returns the hash code value for this map.
int **size**()	Returns the number of key-value mappings in this map.

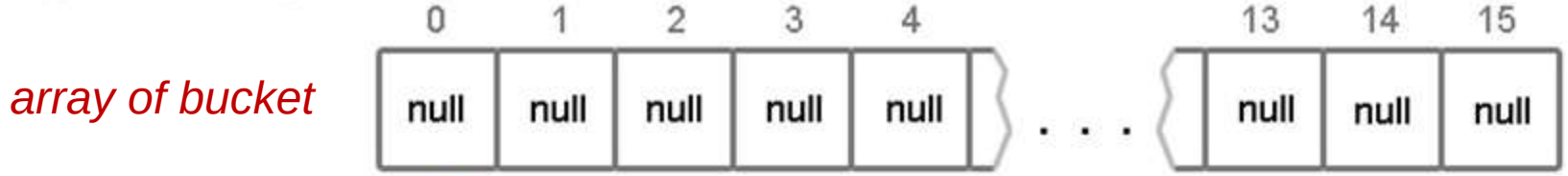
Map creation

- JDK 9 adds standard for all collections static fabric overloaded static <K, V> Map<K, V> of(K k1, V v1) methods for immutable map creation.
- JDK 9 adds
  static <K, V> Entry<K, V> entry(K k, V v)
  and
  static <K, V> Map<K, V> ofEntries(Entry<? extends K, ? extends V>... entries)
  methods for immutable map creation in convenient way.
- JDK 10 adds
  static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> map) method for map copy.

See HashMapRunner

The HashMap inner work

- Map<String, String> hashmap = **new** HashMap<>();




```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
  transient Node<K, V>[] table;
  //...
  {
    static class Node<K, V> implements Map.Entry<K, V> {
      final K key;
      V value;
      Node<K, V> next;
      final int hash;
      //...
```

The HashMap inner work

size - number of HashMap elements

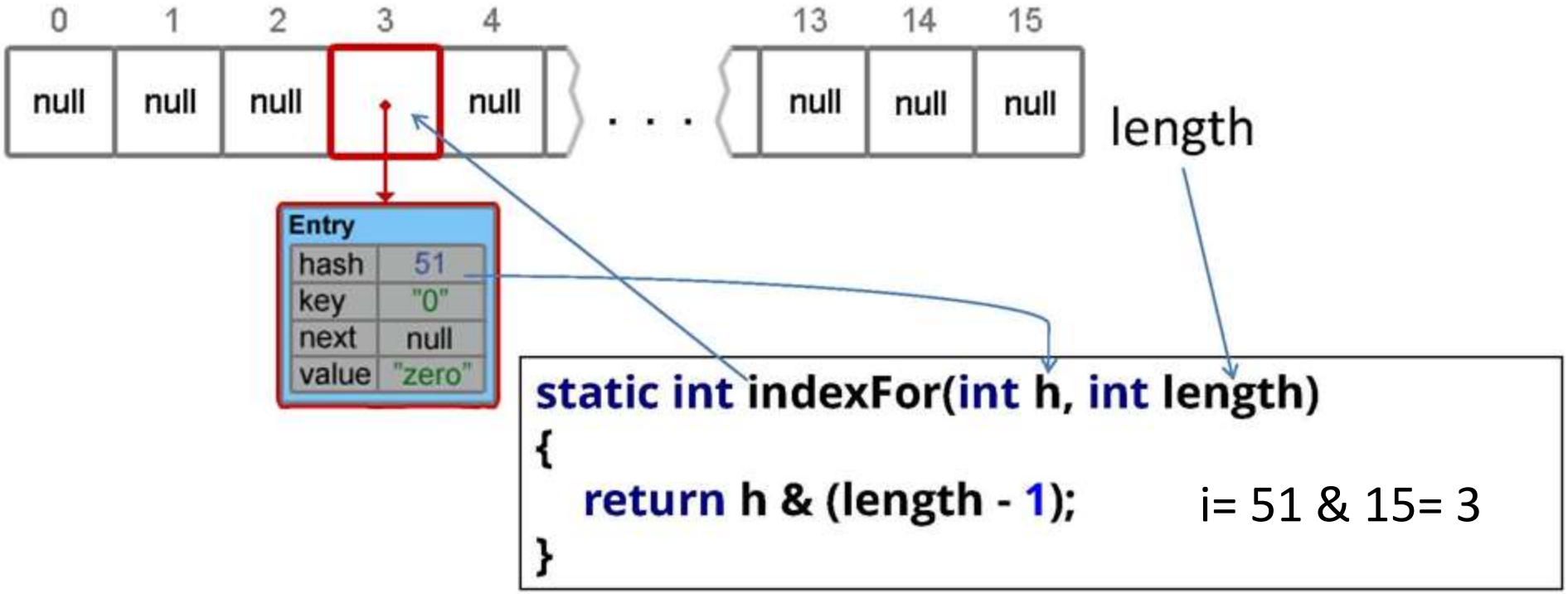
capacity - size of Node<K, V> array (initial = 16, max = 1 073 741 824);

loadFactor - the measure that decides when to increase the capacity of the Map, the default value is 0.75;

threshold - the maximum number of elements at which the Node<K, V> array size is doubled. It is calculated by the formula (threshold = capacity * loadFactor) - default is 16 * 0.75 = 12;

The HashMap inner work

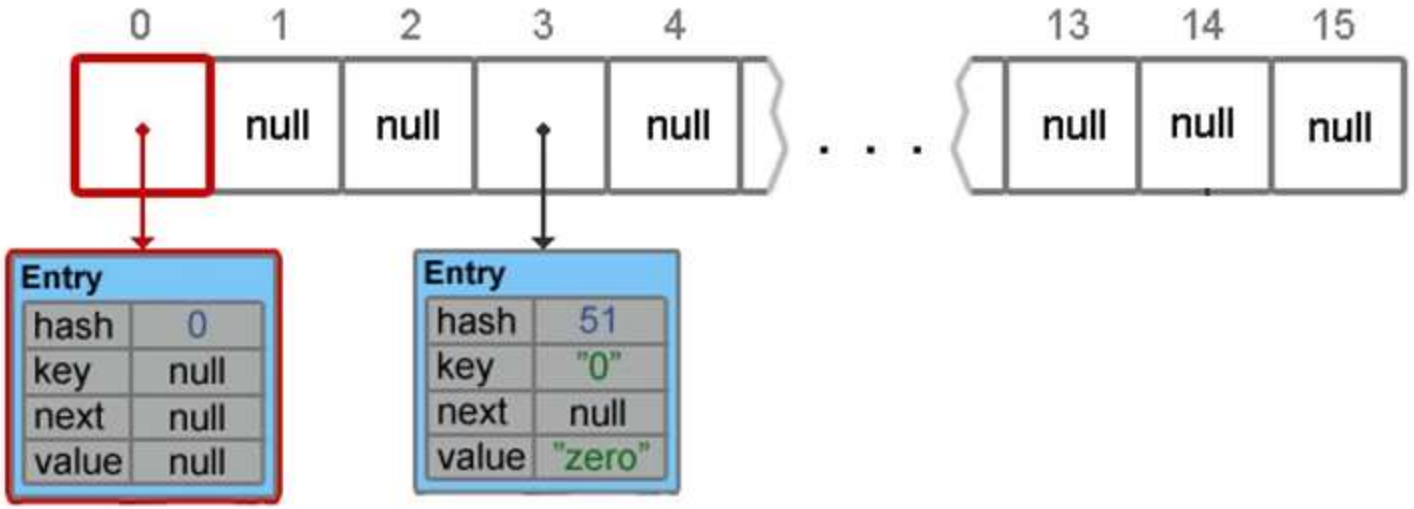
1. hashmap.put("0", "zero");



The HashMap inner work

value can be arbitrary

- hashmap.put(**null**, **null**);

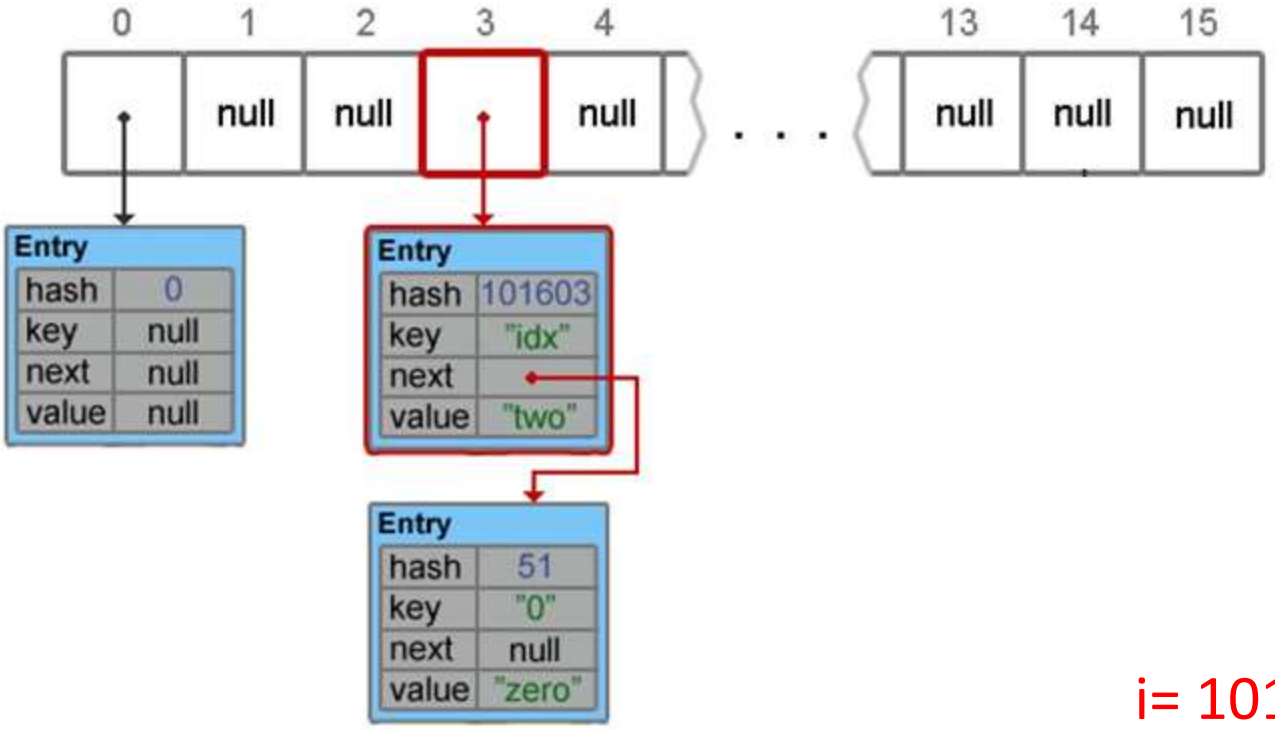



```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

i= 0 & 15= 0

The HashMap inner work

- hashmap.put("idx", "two");



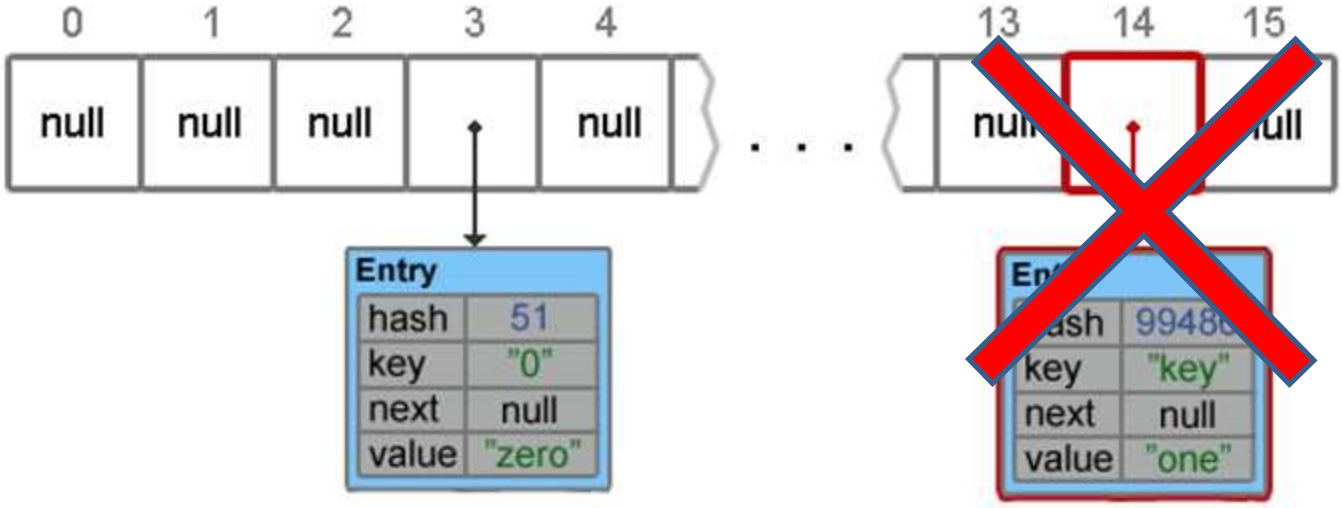
i= 101603 & 15= 3

- collision

The HashMap inner work

- hashmap.put("key", "one");

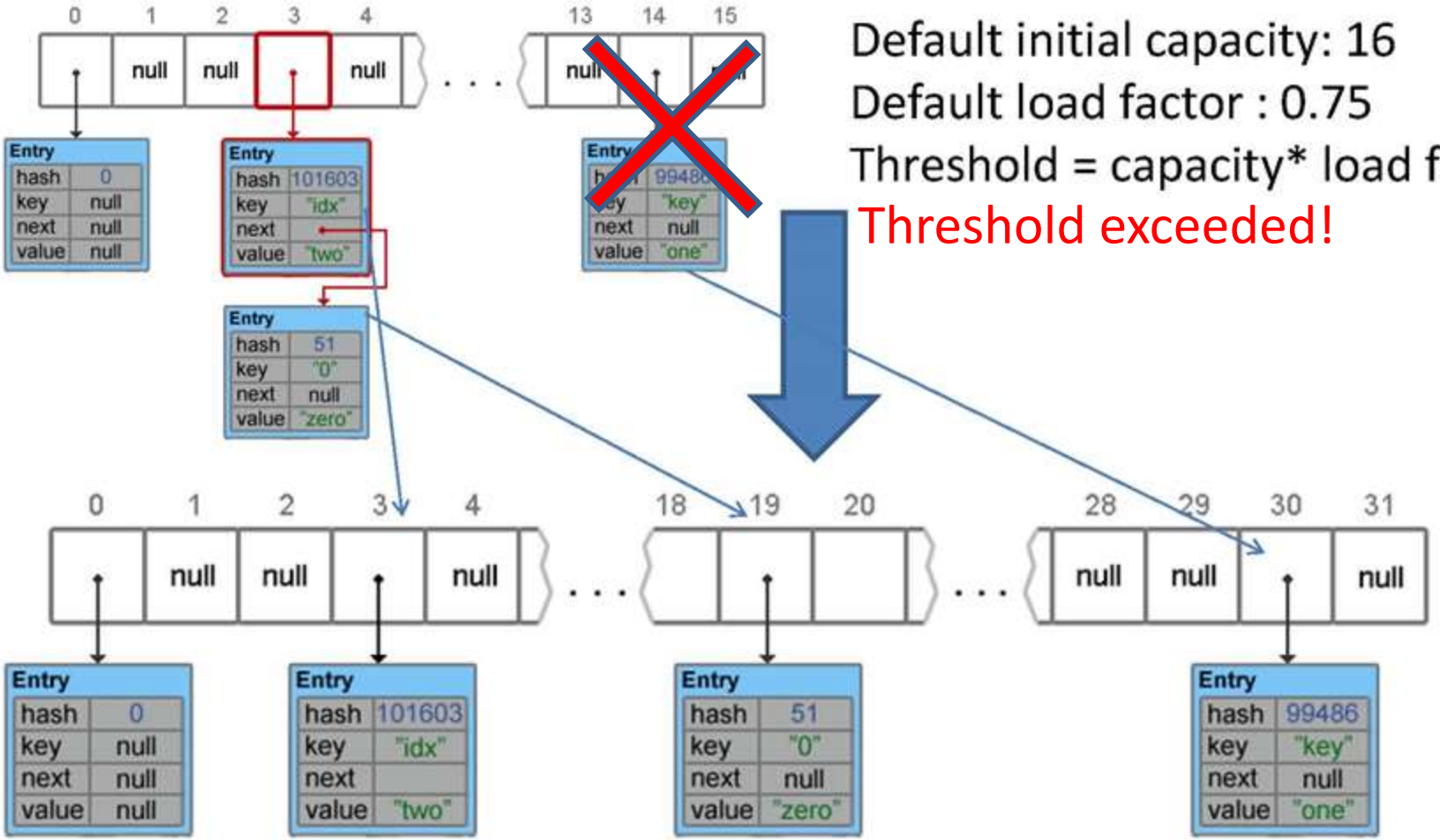
Default initial capacity: 16
Default load factor : 0.75
Threshold = capacity* load factor
: 12



i= 99486 & 15= 14

Threshold exceeded!

The HashMap inner work



See HashMapInnerWork

HashMap additional methods

- JDK 19 adds
  `static <K, V> HashMap<K, V> newHashMap(int numMappings)`
  method - creates a new, empty HashMap suitable for the expected number of mappings - the method argument. The returned map uses the default load factor of 0.75, and its initial capacity is generally large enough so that the expected number of mappings can be added without resizing the map.

See HashMapRunner

The TreeMap inner work

by keys comparing

Rules:

1.Each node is either red or black.

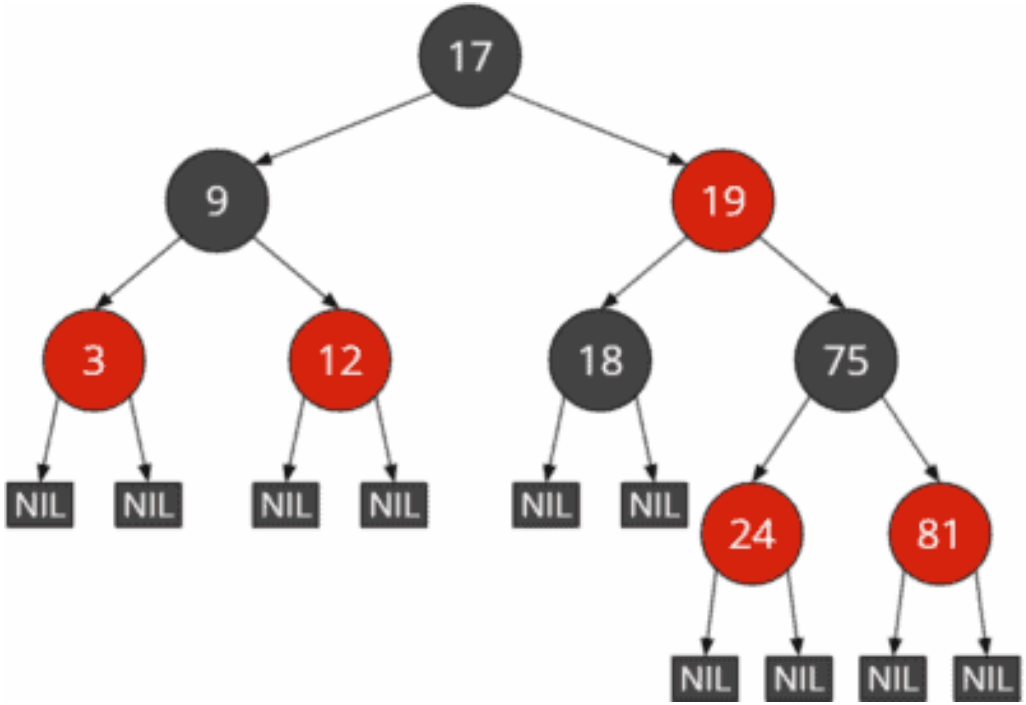
2.(The root is black.)

3.All NIL leaves are black.

4.A red node must not have red children.

5.All paths from a node to the leaves below contain the same number of black nodes.

6.Left/right auto-rotations supply rules above



Numbers value

See TreeMapRunner

The TreeMap inner work

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable {
...
    private static final boolean RED   = false;
    private static final boolean BLACK = true;

    static final class Entry<K,V> implements Map.Entry<K,V> {
        K key;
        V value;
        Entry<K,V> left;
        Entry<K,V> right;
        Entry<K,V> parent;
        boolean color = BLACK;
...
}
```

See TreeMapRunner

TreeMap additional methods

Method	Description
Comparator<? super K> **comparator**()	Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.
NavigableSet<K> **descendingKeySet**()	Returns a reverse order NavigableSet view of the keys contained in this map.
NavigableMap<K,V> **descendingMap**()	Returns a reverse order view of the mappings contained in this map.
NavigableSet<K> **navigableKeySet**()	Returns a NavigableSet view of the keys contained in this map.
SortedMap<K,V> **headMap**(K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
NavigableMap<K,V> **headMap**(K toKey, boolean inclusive)	Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.
SortedMap<K,V> **tailMap**(K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

TreeMap additional methods

Method	Description
NavigableMap<K,V> **tailMap**(K fromKey, boolean inclusive)	Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.
SortedMap<K,V> **subMap**(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
NavigableMap<K,V> **subMap**(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	Returns a view of the portion of this map whose keys range from fromKey to toKey.
K **higherKey**(K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
Map.Entry<K,V> **higherEntry**(K key)	Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.

TreeMap additional methods

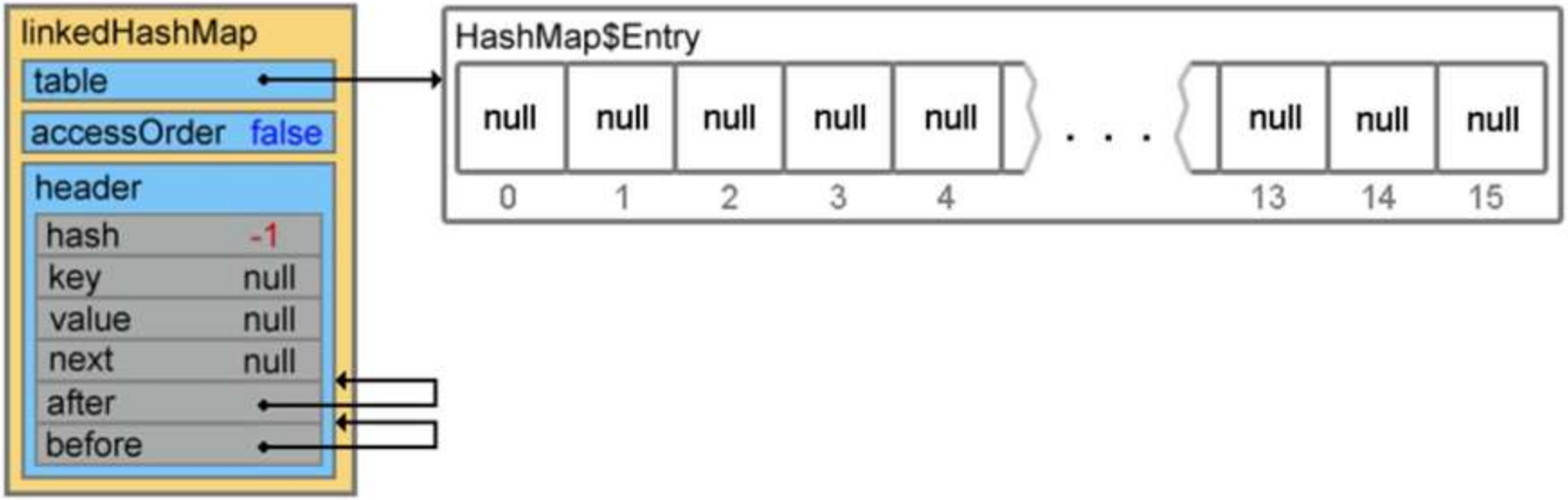
Method	Description
K **lowerKey**(K key)	Returns the greatest key strictly less than the given key, or null if there is no such key.
Map.Entry<K,V> **lowerEntry**(K key)	Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K **ceilingKey**(K key)	Returns the least key greater than or equal to the given key, or null if there is no such key.
Map.Entry<K,V> **ceilingEntry**(K key)	Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
K **floorKey**(K key)	Returns the greatest key less than or equal to the given key, or null if there is no such key.
Map.Entry<K,V> **floorEntry**(K key)	Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.

TreeMap additional methods

Method	Description
K **firstKey()**	Returns the first (lowest) key currently in this map.
Map.Entry<K,V> **firstEntry()**	Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
K **lastKey()**	Returns the last (highest) key currently in this map.
Map.Entry<K,V> **lastEntry()**	Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
Map.Entry<K,V> **pollFirstEntry**()	Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V> **pollLastEntry**()	Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.

The LinkedHashMap inner work

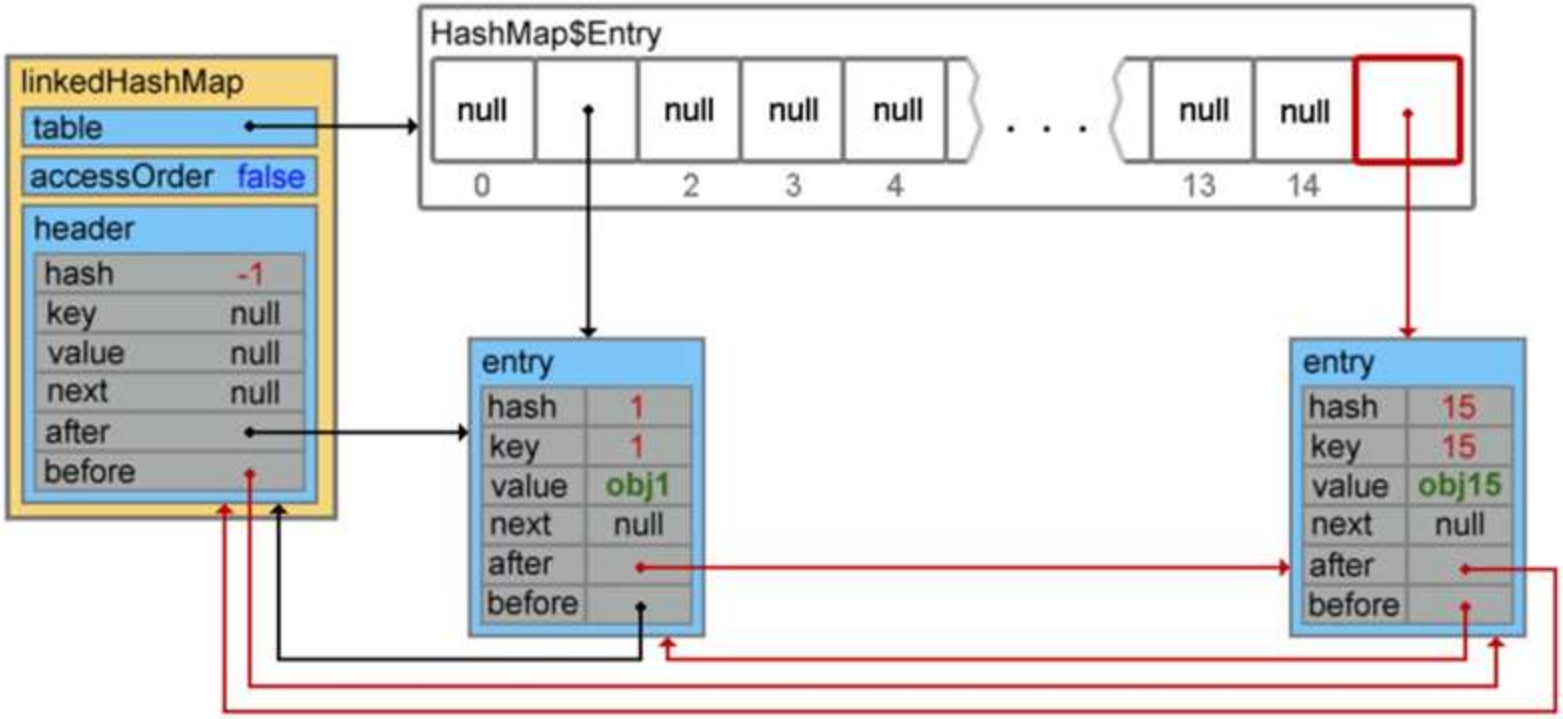
- LinkedHashMap - defines the iteration ordering, which is normally the order in which keys were inserted
- Map<Integer, String> hm = **new** LinkedHashMap<>();



See LinkedHashMapRunner

The LinkedHashMap implementation

1. linkedHashMap.put(1, **"obj1"**);
2. linkedHashMap.put(15, **"obj15"**);



LinkedHashMap additional methods

- JDK 19 adds:
- static <K, V> LinkedHashMap<K, V> newLinkedHashMap(
  int numMappings)
  - creates a new, empty HashMap suitable for the expected number of mappings - the method argument. The returned map uses the default load factor of 0.75, and its initial capacity is generally large enough so that the expected number of mappings can be added without resizing the map.
- protected boolean removeEldestEntry(
  Map.Entry<K,V> eldest)
  - returns true if this map should remove its eldest entry. this override will allow the map to grow up to defined number entries and then delete the eldest entry each time a new entry is added.

See LinkedHashMapRunner

Module contents

- Collections
  - Java Collection Framework, Interfaces
  - The Collection Interface
  - The Iterators
  - The List Interface
  - The Queue and Deque Interfaces
  - The Set Interface
  - The Map Interface
  - The Collection Class

The Collections class 1/6

- **The java.util.Collections** class consists exclusively of static methods that operate on or return collections
- It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection
- The methods of this class all throw a NullPointerException if the collections or class objects provided to them are null.

The Collections class 4/6

- Returns an unmodifiable view of the specified collection

- unmodifiableCollection(Collection<? extends T> c)
- unmodifiableList(List<? extends T> list)
- unmodifiableMap(Map<? extends K,? extends V> m)
- unmodifiableSet(Set<? extends T> s)
- unmodifiableSortedMap(SortedMap<K,? extends V> m)
- unmodifiableSortedSet(SortedSet<T> s)

The Collections class 5/6

```
1. List<String> myList = new ArrayList<String>();
2. myList.add("aaa");
3. myList.add("bbbb");
4. myList.add("ccccc");
5. List<String> readOnlyList =
6.         Collections.unmodifiableList(myList);
7. readOnlyList.add("fff");
```

```
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add
```

The Collections class 6/6

- Returns a synchronized (thread-safe) collection backed by the specified collection

- synchronizedCollection(Collection<T> c)
- synchronizedList(List<T> list)
- synchronizedMap(Map<K,V> m)
- synchronizedSet(Set<T> s)
- synchronizedSortedMap(SortedMap<K,V> m)
- synchronizedSortedSet(SortedSet<T> s)

See CollectionsRunner