Software Development

Part 4: Collections

Koen Pelsmaekers

Unit Informatie (GT 03.14.05)

email: koen.pelsmaekers@kuleuven.be

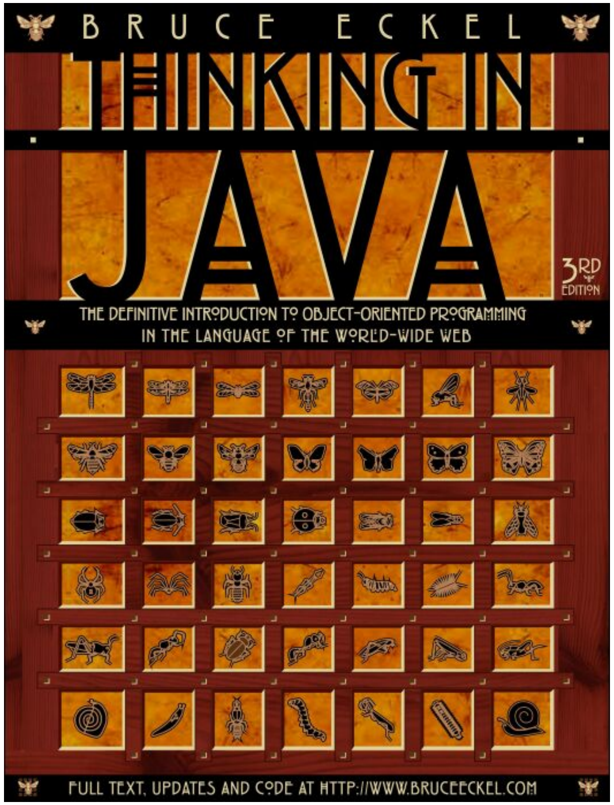
Good design principle:

Program to an interface

- Decouple declaration from implementation
  - "What" vs. "How", "specification" vs. "implementation"
- Information hiding or Encapsulation
  - Do not expose the internals of your implementation
- Defer choice of actual class

- Criteria for designing a good interface (details see later)
  - **Cohesion**: implements a single abstraction
  - **Completeness**: provides all operations necessary
  - **Convenience**: makes common tasks simple
  - **Clarity**: do not confuse your programmers
  - **Consistency:** keep the level of abstraction

Thinking in Java, Bruce Eckel



Source for some of the following slides:
"Thinking in Java, Bruce Eckel"
Version 3: available for free in pdf or html format

Java Collections Framework (from javadoc)

A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details

{=programming towards an interface}

Advantages

- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by requiring you to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by not requiring you to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms with which to manipulate them.

Only 25 classes and interfaces: "Our main design goal was to produce an API that was reasonably small, both in size, and (more importantly) in '**conceptual weight**.'"

Collections

- data structures
- interfaces
- implementations (general-purpose/specialized)
- algorithms

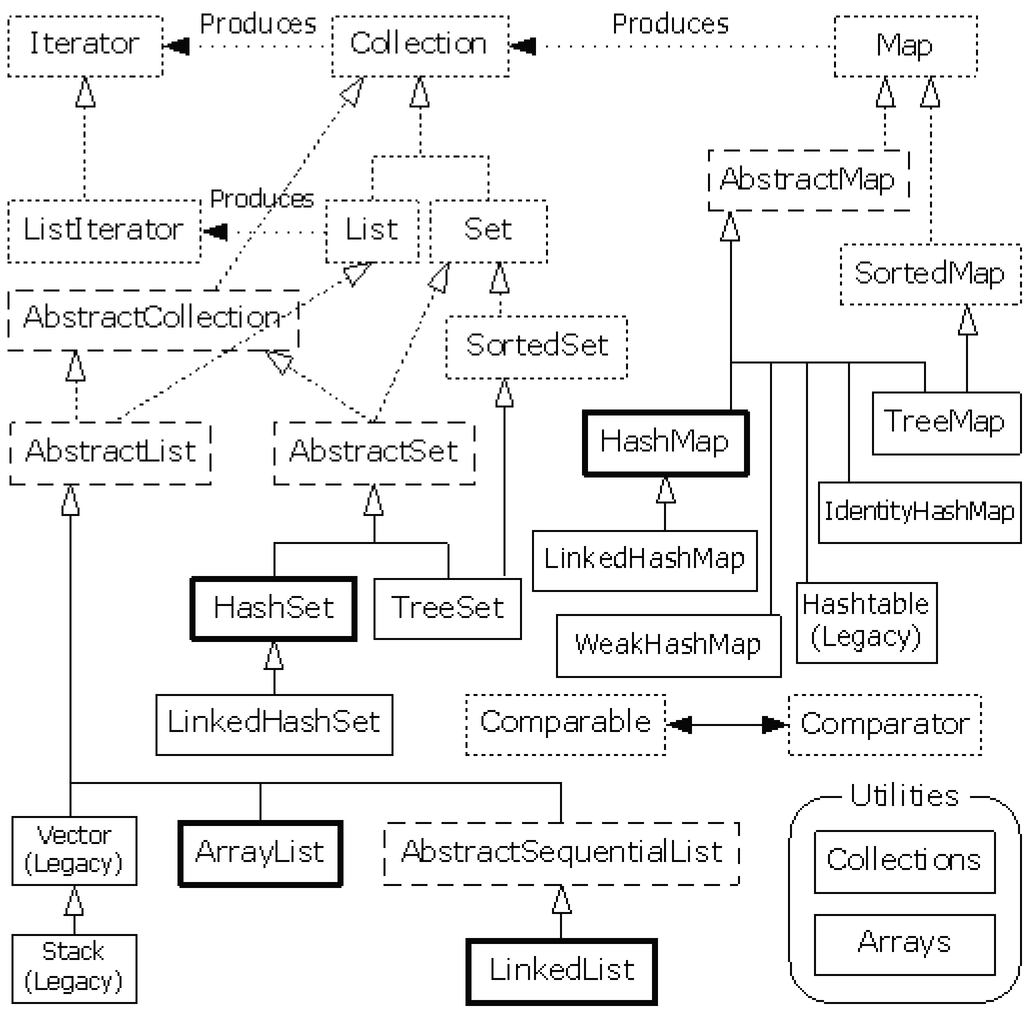
… (see “The Java Tutorial”)

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

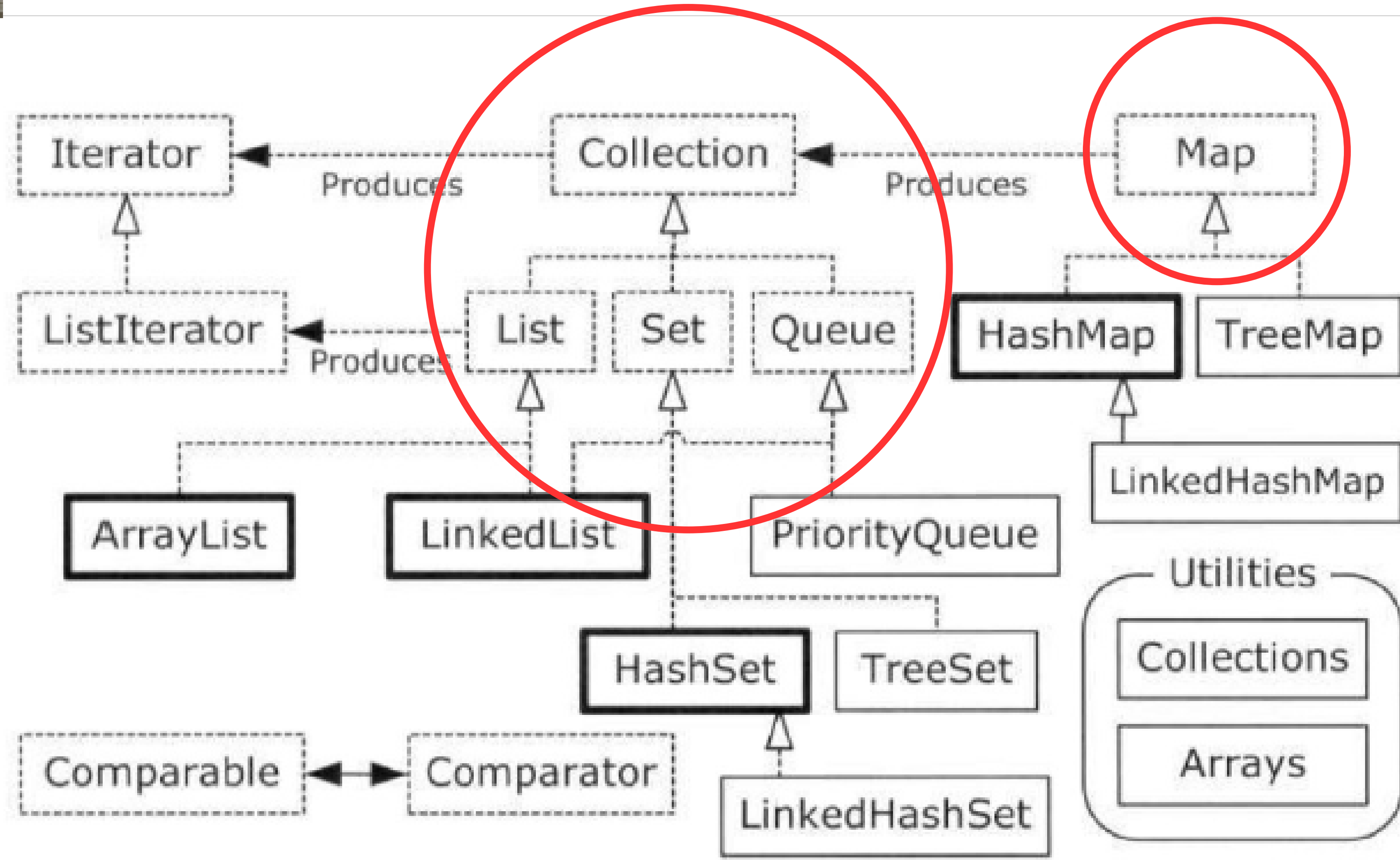
```
Module: java.base – Package: java.util.*
```

Java Collections Framework



Source: Thinking in Java
No standard UML!

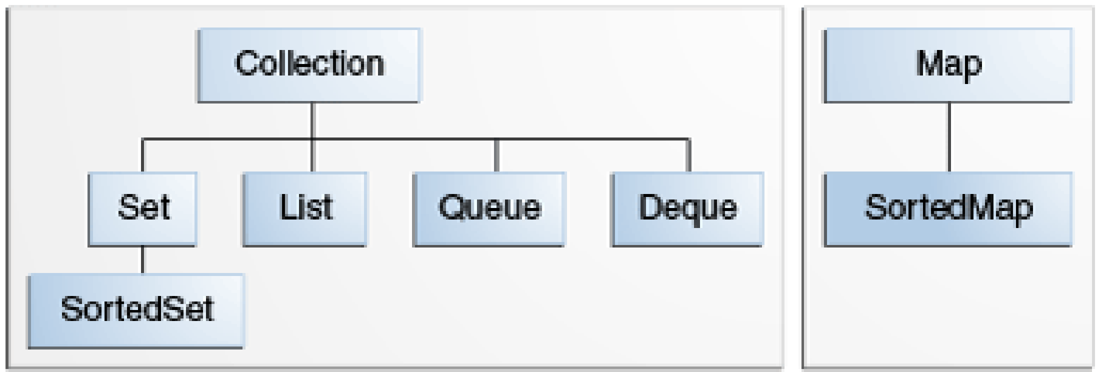
Java Collections Framework



Simple Container Taxonomy

Source: Thinking in Java
No standard UML!

Java Collections Framework



- <u>Interfaces</u>, implementations & algorithms
- No fixed size
- Objects --> <generics>

No standard UML!

Collection interface (= "bag")

From the API:

*The root interface in the collection hierarchy. A collection represents a group of objects, known as its **elements**. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where **maximum generality** is desired.*

Set – SortedSet interface

From the API:

Set
*A collection that contains **no duplicate elements**. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.*

SortedSet
A Set that further provides a total ordering on its elements. The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of SortedMap.)

Some Collection/Set methods

`add(o)`	Add a new element
`size()`	Number of elements
`contains(o)`	Membership checking
`clear()`	Remove all elements
`remove(o)`	Remove the element o
`isEmpty()`	Whether it is empty
`iterator()`	Return an iterator

(only the most important methods are shown)

Using Set (since Java 5, Java 7)

```
Set<Person> personSet = new HashSet<>();
// ...
Person person = new Person();
personSet.add(person); // insert an element
// ...
int n = personSet.size(); // get size
// ...
if (personSet.contains(person)) {...}
  // check membership

// iterate through the set
Iterator<Person> iter = personSet.iterator();
while (iter.hasNext()) {
   Person person = iter.next();
   // no downcast [(Person) iter.next()] needed!
   // ...
}
```

General purpose Set implementations

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Details: see “Map” implementations

List implementations

List interface

From the API:

*An **ordered collection** (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer **index** (position in the list), and search for elements in the list.*

*Unlike sets, lists typically allow **duplicate elements**. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.*

Some List methods

`add(i,o)`	Insert o at position i (int i, Object o)
`add(o)`	Append o to end
`get(i)`	Return the i-th element
`remove(i)`	Remove the i-th element
`remove(o)`	Remove the element o
`set(i,o)`	Replace the i-th element with o

(only the most important methods are shown)

General purpose List implementations

General-purpose Implementations

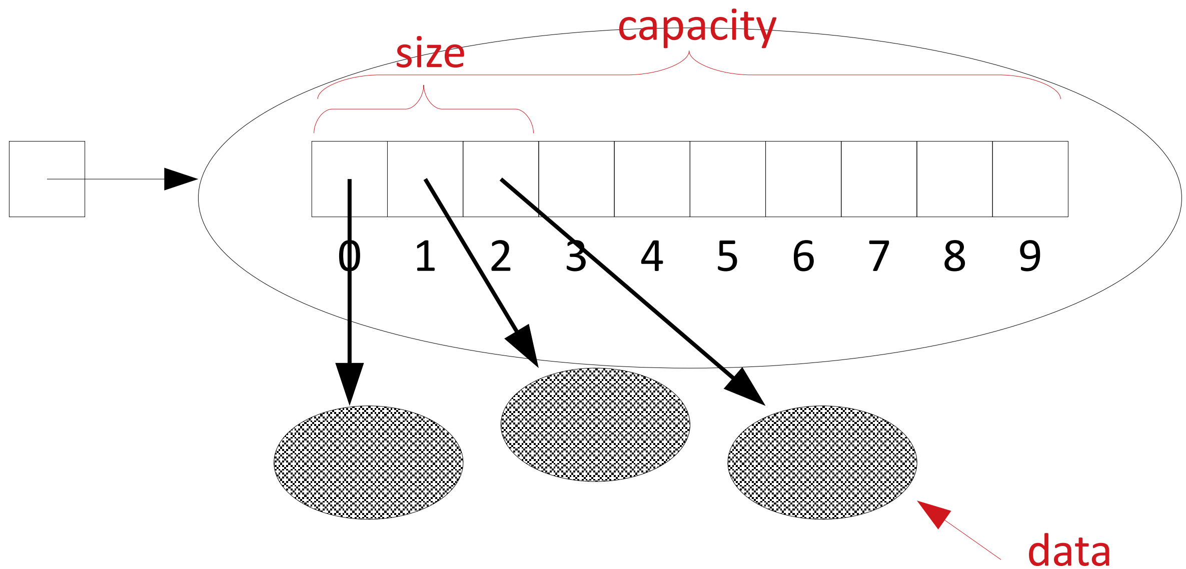
Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

List examples: see OOPDB-course (1st semester)

ArrayList vs. LinkedList

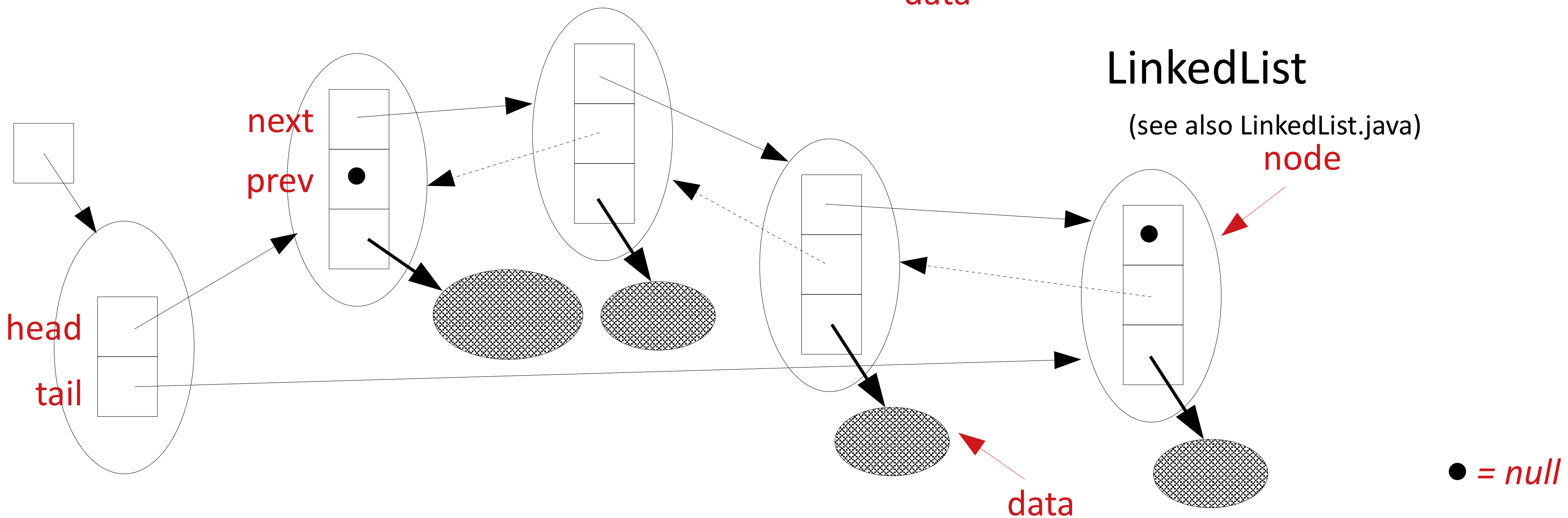
ArrayList

(see also ArrayList.java)

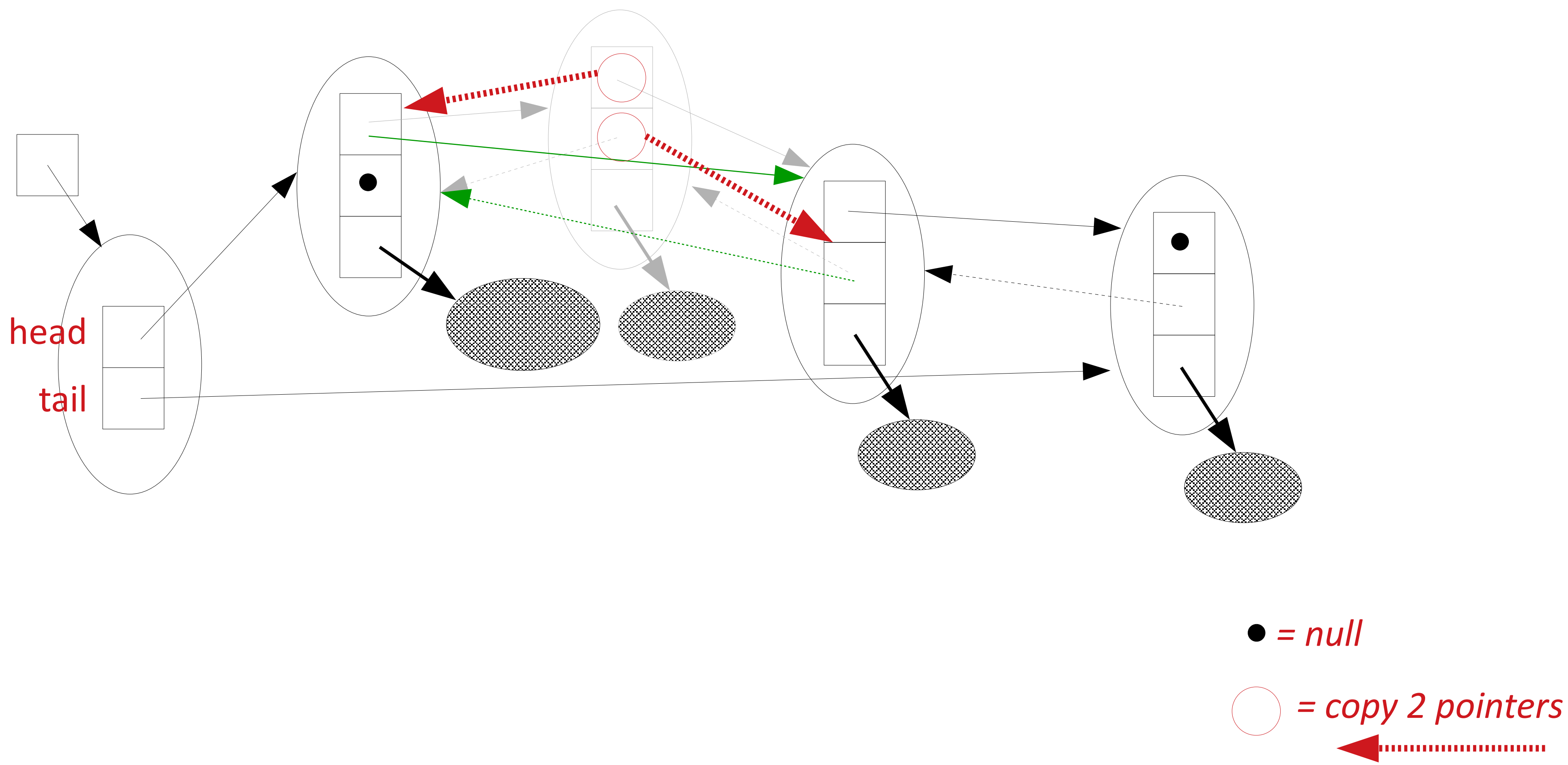


LinkedList

(see also LinkedList.java)



LinkedList: remove an element



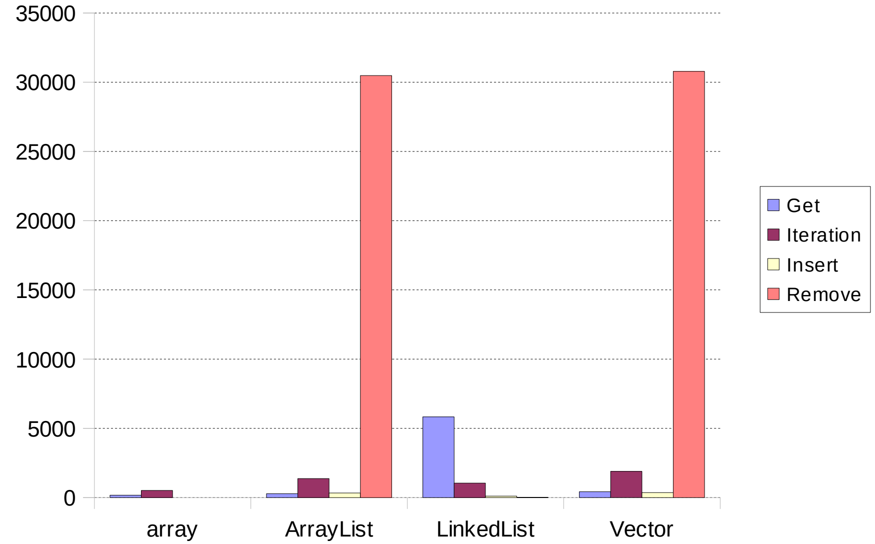
ArrayList vs. LinkedList

- ArrayList
  - + Fast get operation
  - + Fast add/remove at the back
  - - Slow insert/remove from front
  - - Contiguous memory (re)allocation needed
- LinkedList
  - - slow get operation
  - + fast add/remove operation at the back/front
  - + no memory reallocation needed

	get	*add*	*contains*	*next*	*remove(O)*	*Iterator.remove*
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

Some statistics (Thinking in Java): List (+ demo)

Type	Get	Iteration	Insert	Remove
array	172	516	na	na
ArrayList	281	1375	328	30484
LinkedList	5828	1047	109	16
Vector	422	1890	360	30781



As expected, arrays are faster than any container for random-access lookups and iteration. You can see that random accesses (get( )) are cheap for **ArrayLists** and expensive for **LinkedLists**. (Oddly, iteration is faster for a **LinkedList** than an **ArrayList**, which is a bit counterintuitive.) On the other hand, insertions and removals from the middle of a list are dramatically cheaper for a **LinkedList** than for an **ArrayList**—especially removals. **Vector** is generally not as fast as **ArrayList**, and it should be avoided; it's only in the library for legacy code support (the only reason it works in this program is because it was adapted to be a List in Java 2). The best approach is probably to choose an **ArrayList** as your default and to change to a **LinkedList** if you discover performance problems due to many insertions and removals from the middle of the list. And of course, if you are working with a fixed-sized group of elements, use an array.

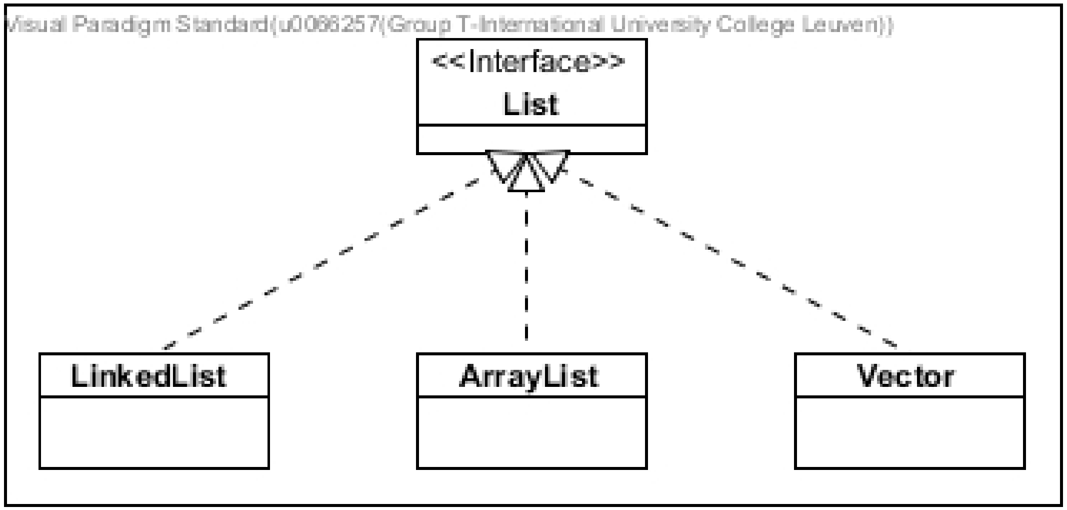
Other List or Collection methods

- Sort all elements
  - sort(Comparator or null[*]) (since Java 8, see "interface")
- Replace all elements
  - replaceAll(UnaryOperator) (since Java 8, lambda expr.)
- Set theory operations (Collection interface)
  - Intersection (A ∩ B)
    - retainAll()
  - Union (A U B)
    - addAll()
  - Complement (A \ B)
    - removeAll()

[*] Comparable interface is used: “internal ordering”

Static type vs. Dynamic type

Program towards the interface



List +

addFirst()
addLast()
getFirst()
getLast()

List

List (legacy)

```
List<Person> myList1 = new ArrayList<>();
List<Person> myList2 = new LinkedList<>();
myList1.addFirst(p); // valid?
myList2.getLast(p); // valid?
```

Compile-time type vs. run-time type

```
List<Person> myList1 = new ArrayList<>();
List<Person> myList2 = new ArrayList<>();
myList1.addFirst(); // valid?
// compiler error: addFirst() is no List method
myList2.getLast(); // valid?
// compiler error: getLast() is no List method

myList1 = new LinkedList<>(); // valid?
myList1.addFirst(); // valid?

((LinkedList)myList1).addFirst(); // valid?

((LinkedList)myList2).getLast(); // valid?
```

Try it out!

Queue interface

Queue interface

From the API:

*A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide **additional insertion, extraction, and inspection operations**. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.*

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.

Some queue methods

`add(o)`	Insert o (throws exception)
`offer(o)`	Insert o (returns special value: boolean)
`remove()`	Remove and return the head of the queue (throws exception)
`poll()`	Remove and return the head of the queue (returns special value: null)
`element()`	Return the head of the queue (throws exception)
`peek()`	Return the head of the queue (returns special value: null)

in case of empty queue or queue with no capacity

(only the most important methods are shown)

Using Queue (since Java 6)

```java
Queue<Integer> queue = new ArrayDeque<>(); // FIFO

queue.add(10);
queue.offer(20);
System.out.println(queue.peek()); //10

Integer io = queue.remove();
System.out.println(queue.element()); //20

io = queue.remove();
io = queue.poll(); // returns null
io = queue.remove(); // NoSuchElementException
```

Map implementations

Map interface

From the API:

*An object that maps **keys to values**. A map cannot contain duplicate keys; each key can map to at most one value.*

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

*The Map interface provides **three collection views**, which allow a map's contents to be viewed as a <u>set of keys</u>, <u>collection of values</u>, or <u>set of key-value mappings</u>. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.*

"Maps" are everywhere

- Examples
  - Dictionary
    - word -> explanation
  - Contact list on mobile device
    - name -> extra information about a contact
  - HTTP-headers/Request parameters
    - Host -> iiw.kuleuven.be
    - Accept -> text/html
    - Accept-language -> en-US
    - ...
    - <url>?source=xy&q=Berlin&output=json...

key

value

Some Map methods

`put(k,v)`	Put mapping for k to v
`get(k)`	Get the value associated with k
`clear()`	Remove all mappings
`size()`	The number of pairs
`keySet()`	The set of keys
`values()`	The collection of values
`entrySet()`	The set of key-value pairs
`containsKey(k)`	Whether contains a mapping for k
`containsValue(v)`	Whether contains a mapping to v
`isEmpty()`	Whether it is empty

(only the most important methods are shown; k=key; v=value)

Using Map (since Java 5, Java 7)

```
Map<String,Person> map = new HashMap<>();
// ...
map.put("Jef", new Person()); // insert a key-value pair
// ...
// get the value associated with key "Jef"
Person val = map.get("Jef");
map.remove("Jef"); // remove a key-value pair
// ...
if (map.containsValue(val)) { ... }
if (map.containsKey("Jef")) { ... }
Set<String> keys = map.keySet(); // get the set of keys
// iterate through the set of keys
Iterator<String> iter = keys.iterator();
while (iter.hasNext()) {
  String key = iter.next();
  // ...
}
```

General purpose Map implementations

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

It is a challenge to find a performant Map implementation:

- Lineair list (no implementation in Java Collections API)
- Hashtable (HashMap in Java)
- Balanced Binary Tree (TreeMap in Java)

Hashtable implementation

- Table with limited number of buckets (rows)
- **put**
  - hashcode() % tablelength => index in table
  - no collision: put key-value pair into bucket
  - collision: follow an algorithm to find an empty bucket
    - lineair search
    - more sophisticated search
    - Java: array of linked lists or TreeMap (since Java 8)
- **get**
  - calculate index
  - check for the key value
    - keys equal => element found
    - keys !equal => search algorithm (cf. collision)
  - goal: maximize **hit ratio**

https://www.ee.ryerson.ca/~courses/coe428/structures/hash.html

hashCode() and equals()

- If two objects are equal, they MUST have matching hashcodes.
- If two objects are equal, calling equals() on either object MUST return true. In other words, if (a.equals(b)) then (b.equals(a)).
- If two objects have the same hashcode value, they are NOT required to be equal. But if they're equal, they MUST have the same hashcode value.
- So, if you override equals(), you MUST override hashCode().

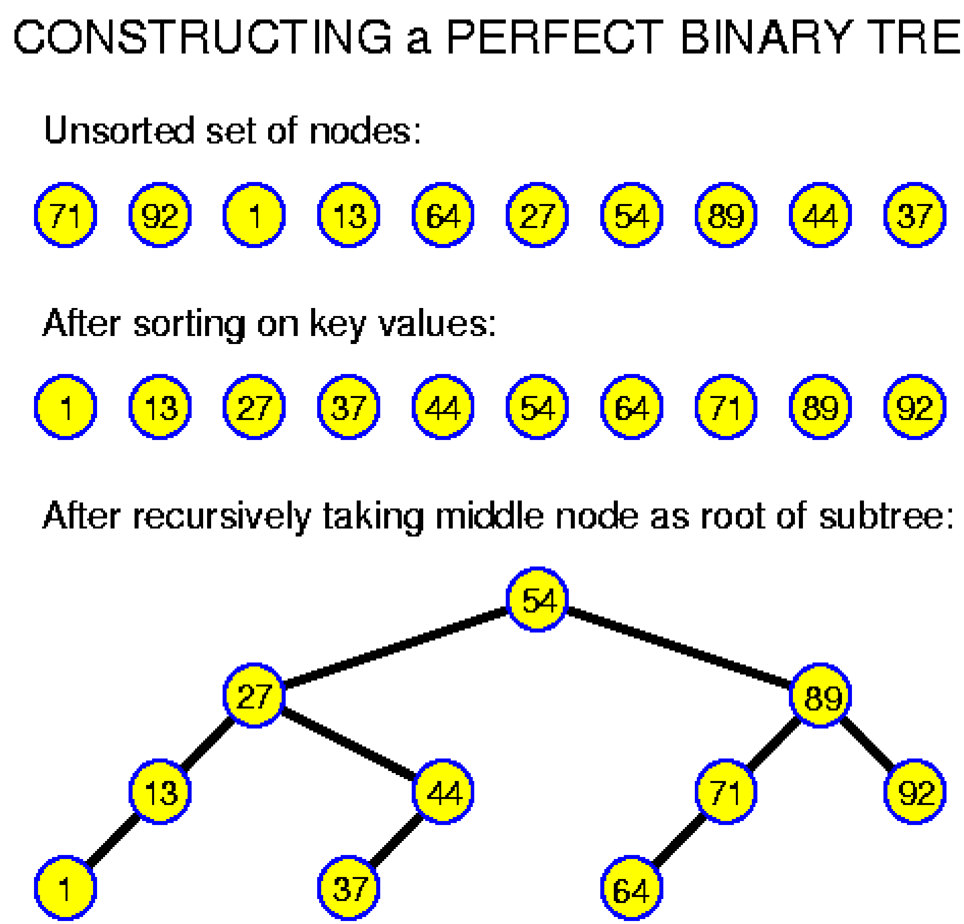
Hashtable: conclusion

- Array based
- Maximize hit ratio
  - Avoid key collisions
  - `hashCode()`: as unique as possible!
  - `equals()` should "follow" hashcode
- Size – Capacity – Load factor
- Special case: LinkedHashMap

HashMap: Java implementation

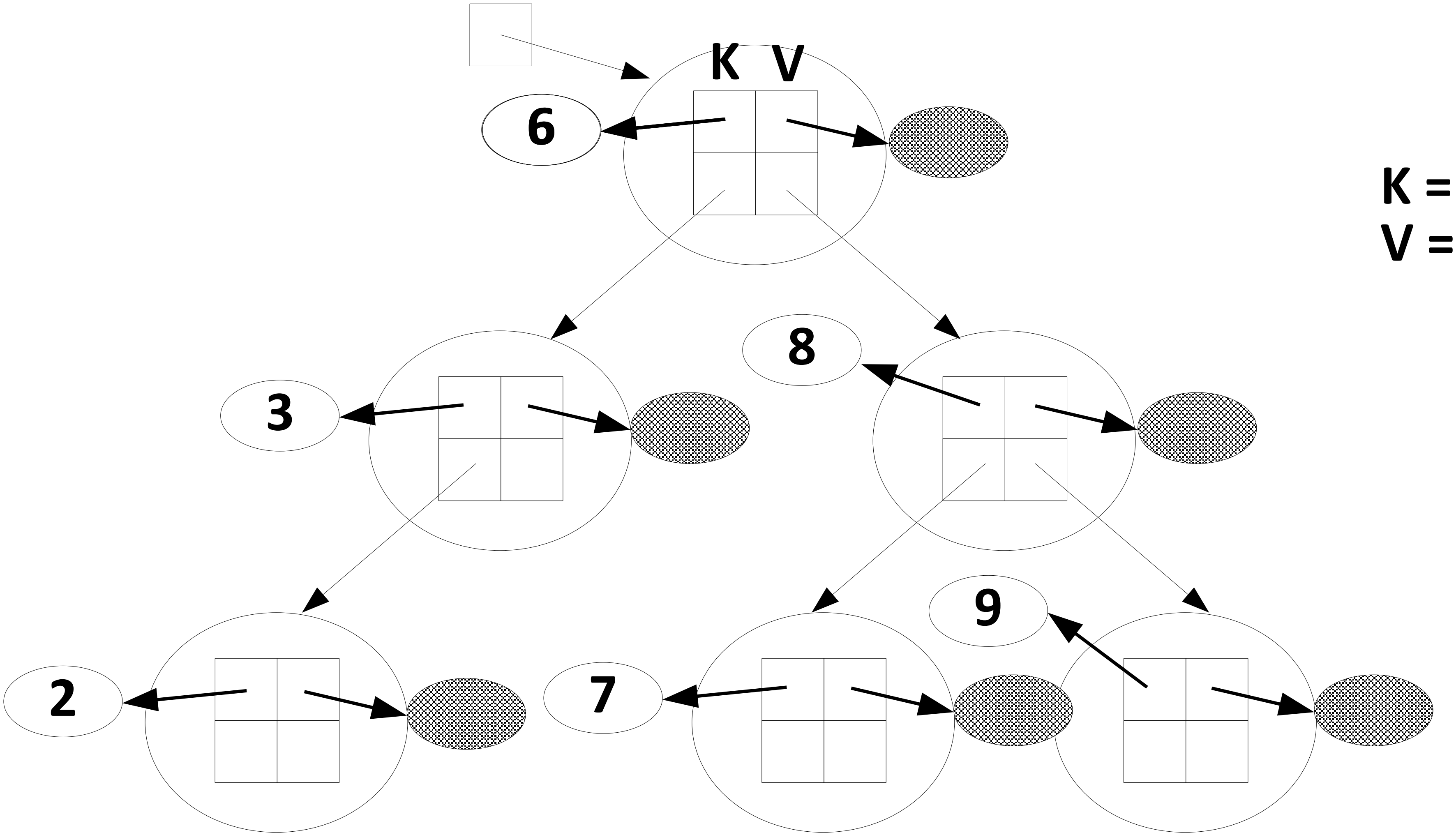
- constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets
- iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings)
- #entries in the hash table > load factor * current capacity: the hash table is rehashed (internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets
- collisions: multiple entries per bucket, which must be searched sequentially (linked list); since Java 8: TreeMap

Balanced binary tree implementation



https://visualgo.net/en

TreeMap<Integer, String>



K = Key
V = Value

TreeMap: conclusion

- Keys have to implement Comparable and/or Comparator interface
  - Keys are sorted
- Balanced binary tree
  - Internal ordering (Comparable)
  - External ordering (Comparator)
- Re-balance if necessary
- Binary search

Set implementations

- Similar to Map implementations
  - keys only (unique)
  - no values
- Most common concrete implementations
  - HashSet
  - TreeSet
  - LinkedHashSet

Some statistics (Thinking in Java): Set

Type	Test size	Add	Contains	Iteration
TreeSet	10	25	23,4	39,1
	100	17,2	27,5	45,9
	1000	26	30,2	9
HashSet	10	18,7	17,2	64,1
	100	17,2	19,1	65,2
	1000	8,8	16,6	12,8
LinkedHashSet	10	20,3	18,7	64,1
	100	18,6	19,5	49,2
	1000	10	16,3	10

The performance of **HashSet** is generally superior to **TreeSet** for all operations (but in particular for addition and lookup, the two most important operations). The only reason **TreeSet** exists is because it maintains its elements in sorted order, so you use it only when you need a sorted **Set**.

Note that **LinkedHashSet** is slightly more expensive for insertions than **HashSet**; this is because of the extra cost of maintaining the linked list along with the hashed container. However, traversal is cheaper with **LinkedHashSet** because of the linked list.

Some statistics (Thinking in Java): Map

Type	Test size	Put	Get
TreeMap	10	26,6	20,3
	100	34,1	27,2
	1000	27,8	29,3
HashMap	10	21,9	18,8
	100	21,9	18,6
	1000	11,5	18,8
LinkedHashMap	10	23,4	18,8
	100	24,2	19,5
	1000	12,3	19
IdentityHashMap	10	20,3	25
	100	19,7	25,9
	1000	13,1	24,3
WeakHashMap	10	26,6	18,8
	100	26,1	21,6
	1000	14,7	19,2
Hashtable	10	18,8	18,7
	100	19,4	20,9
	1000	13,1	19,9

Some statistics (Map): explanation

As you might expect, **Hashtable** performance is roughly equivalent to **HashMap**. (You can also see that **HashMap** is generally a bit faster; **HashMap** is intended to replace **Hashtable**.) The **TreeMap** is generally slower than the **HashMap**, so why would you use it? As a way to create an ordered list. The behavior of a tree is such that it's always in order and doesn't have to be specially sorted. Once you fill a **TreeMap**, you can call **keySet( )** to get a **Set** view of the keys, then **toArray( )** to produce an array of those keys. You can then use the static method **Arrays.binarySearch( )** (discussed later) to rapidly find objects in your sorted array. Of course, you would probably only do this if, for some reason, the behavior of a **HashMap** was unacceptable, since **HashMap** is designed to rapidly find things. Also, you can easily create a **HashMap** from a **TreeMap** with a single object creation. In the end, when you're using a **Map**, your first choice should be **HashMap**, and only if you need a constantly sorted **Map** will you need **TreeMap**.

LinkedHashMap is slightly slower than **HashMap** because it maintains the linked list in addition to the hashed data structure. **IdentityHashMap** has different performance because it uses == rather than equals( ) for comparisons.

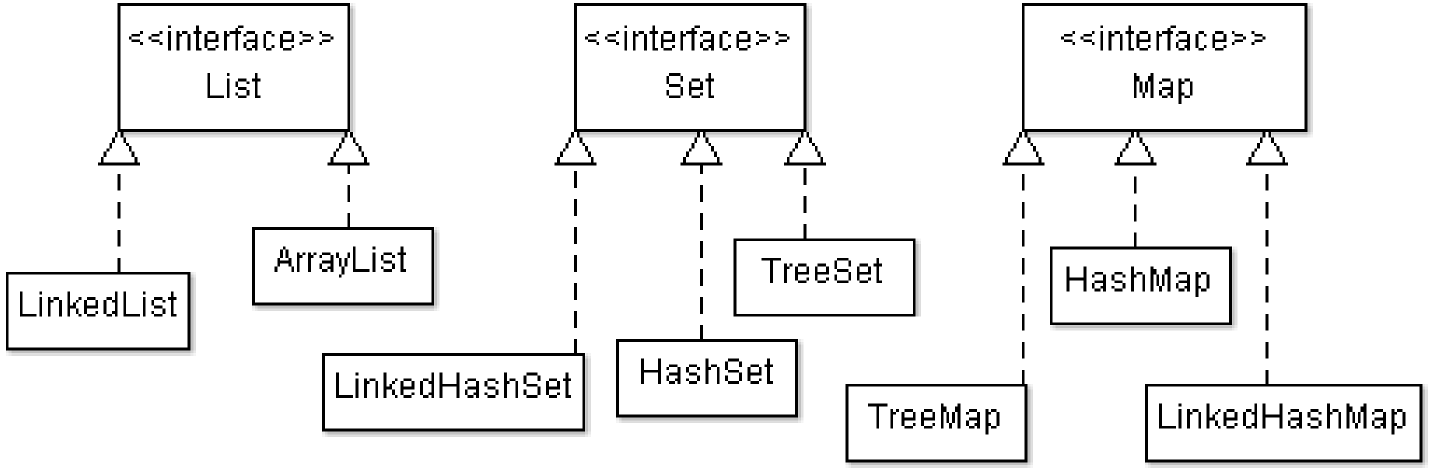
Collections: Conclusion and other collection characteristics

Collection implementations overview

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
Hashtable	Map	hash table

Collections: conclusion

- If possible: program towards the interface
- Defer the choice for a concrete collection implementation until you know about the actions to be performed on the collection
- Performance should be you last concern



Big "O" notation

- The Big O notation is used to indicate the time taken by algorithms to run:
  (N is the number of elements)
  - O(N):- The time taken is linearly dependent to the number of elements
  - O(log N):- The time taken is logarithmic to the number of elements
  - O(1):- The time taken is constant time, regardless of the number of elements

http://en.wikipedia.org/wiki/Big_O_notation

Big-O and Java Collections

	get	add	contains	next	remove(O)	Iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

	get	containsKey	next	Note
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

	add	contains	next	Note
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

	offer	peek	poll	size
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDequeue	O(1)	O(1)	O(1)	O(1)

Other collection-related Java 8 features

- Iterable interface
  - `forEach()`: (internal) iteration
- Collections interface
  - `removeIf()`
- List interface
  - `replaceAll()`
  - `sort()` (no need for `Collections.sort()` any more)!
- Map interface
  - `putIfAbsent()`
  - `getOrDefault()`

Iterate through Collections

- The Iterator interface
- Java 5: enhanced for-loop
- Java 8: forEach with a lambda expression

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

```
for ( Object o: collection) {
   ... o ...
}
```

```
collection.forEach(e -> e.doSomething())
```

Ordering and sorting

- Two ways to define order on objects
  - Natural order: implement `Comparable` interface
    - `int compareTo(Object o)`
  - Arbitrary order(s): use class(es) that implement the `Comparator` interface
    - `int compare(Object o1, Object o2)`

Natural order

- Implemented in the class definition
- Comparable interface
  - `int compareTo(o)`
    - < 0: `this` precedes o
    - ==0: equals
    - > 0: o precedes `this`
  - If `a.compareTo(b) == 0` then `a.equals(b)`
- Examples
  - `String`, `Integer`, `Long`, …

Arbitrary ordering

- Use a class that implements the Comparator interface
- Comparator interface
  - `int compare(o1,o2)`
    - < 0: o1 precedes o2
    - ==0: equals
    - > 0: o2 precedes o1
- If `compareTo(a,b) == 0` then `a.equals(b)`

Collections: some exercises

Collections: exercises

1) Count the number of different words in a text
   a) show the number
   b) show the words
   c) show the words in alphabetic order
   d) Show the words in reverse alphabetic order
2) Count the word frequency in a text and
   a) show the result
   b) show the result with the words in the order they occur in the text
   c) show the result with the words in an alphabetic order
   d) show the result with the words in a reverse alphabetic order
   e) show the result from highest to lowest frequency