Effective Java Programming

algorithms

Structure

- algorithms
    - selection of algorithm
    - comparison of algorithms
    - elegance of solution
    - consideration of the problem domain
    - more than simple algorithms

Selection of algorithm

- may be the fastest algorithm for one type of data, and slow for another
- need to consider how the algorithm will be used
- choose the solution that is most likely the best
- complexity of algorithms
    - **memory complexity** - the number and size of the data structures used in the algorithm
    - **time complexity** - the relationship between the number of elementary operations performed during the run of the algorithm, and the size of the input data (given as a function of the size of the data)

Selection of algorithm

- time complexity
  - two algorithms of the same execution time $F_1(N)$ and $F_2(N)$ have order of complexity if:

    $\lim\limits_{N \to \infty} \frac{F1(N)}{F2(N)} = C$, where $0 < C < \infty$

  - If C = 0, algorithm with execution time F1(N) has a lower complexity order (better complexity)
  - If $C = \infty$, algorithm with execution time F1(N) has higher complexity order (worse complexity)

Selection of algorithm

- time complexity
  - allows comparison of algorithm speed
  - shows the trends in execution time with increasing problem
  - complexity algorithm defines the notation O (Landau)
  - O(F(N)) means that the algorithm has
    the complexity of the same order as the algorithm with run-time F(N)
  - examples:

    $O(1)$, $O(\log(N))$, $O(N)$, $O(N\log(N))$, $O(N^2)$, $O(N^k)$, $O(2^N)$, $O(N!)$

Example – sum of integers

```
class SimpleSum {
  public long sum(int start, int stop) {
    long sum = 0;
    for (int i = start; i <= stop; i++) {
      sum += i;
    }
    return sum;
  }
}
```

- number of calculations depends on amount of numbers
- linear complexity O(N)

Example – sum of integers

```
class SmartSum {
  public long sum(int start, int stop) {
    long big = stop * (stop + 1) / 2;
    start--; // result considers start
    long small = start * (start + 1) / 2;
    return big - small;
  }
}
```

- number of calculations is constant, independent of amount of numbers
- constant complexity O(1)

Comparision of algorithms

- complexity is an estimate
- estimates are not reality, but are good indicators
- we can have algorithms of similar comlexity
- having interesting alternatives you should compare them
  - easiest way – write a benchmark
  - best performed on:
    - different amount of data
    - many times

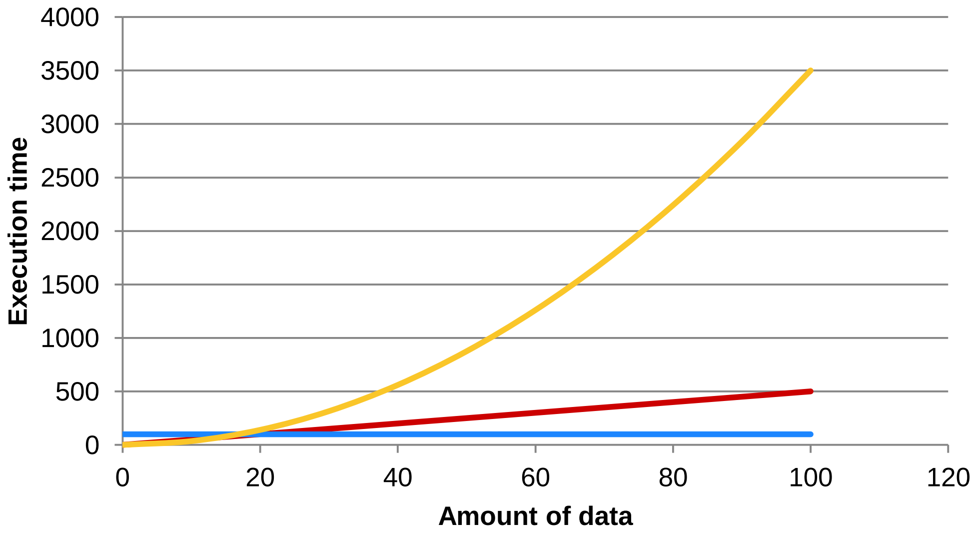
Comparison – simplified example

```
SimpleSum sims = new SimpleSum();
long before = System.currentTimeMilis();
for (int i = 0; i < 1000 * 1000; i++)
  sims.sum(10, 10000);
long after = System.currentTimeMilis();
System.out.println(„SimpleSum: ” + (after - before) + „ ms.”);
// about 26000ms

SmartSum sms = new SmartSum();
before = System.currentTimeMilis();
for (int i = 0; i < 1000 * 1000; i++)
  sms.sum(10, 10000);
after = System.currentTimeMilis();
System.out.println(„SmartSum: ” + (after - before) + „ ms.”);
// about 15ms
```

Comparison – take multiple series

- algorithms may behave differently with small and big amount of data
- they can be better for smaller data portions
- choice is not obvious
- see diagram
  - algorithm X – O(1)
  - algorithm Y – O(N)
  - algorithm Z – O(N^2)



Elegance of the solution

- elegant code is like poetry
  - hard to define
  - minimally complex
  - suited for the problem
  - result – almost always fast
- on the contrary – *brute force*
  - one after another (solution with loop)
  - works only because hardware is fast
  - bottlenecks
    - micro-optimizations won’t work
    - change the algorithm

Taking the problem domain into account

- having selected the algorithm it can be accelerated
- include the problem domain
    - is the algorithm a solution for it?
    - what the algorithm shouldn’t do is equally important
    - simplify, so it won’t be bothered by anything but the problem
- example
    - Bresenham’s algorithm vs. point by point (latter 25% better)
- specific solutions are faster then generic

More than simple algorithms

- algorithms are not only solutions
- are embedded in your code
- can give poor results when interacting with OS
- real life example:
  - JComboBox was initially inefficient
  - adding new elements had an order $O(N^2)$
  - interactions with other classes where to blame
  - after identifying the problem, order change to $O(N)$

Conclusions

- how the comlexity of algorithms is defined?
- what besides complexity is also important?

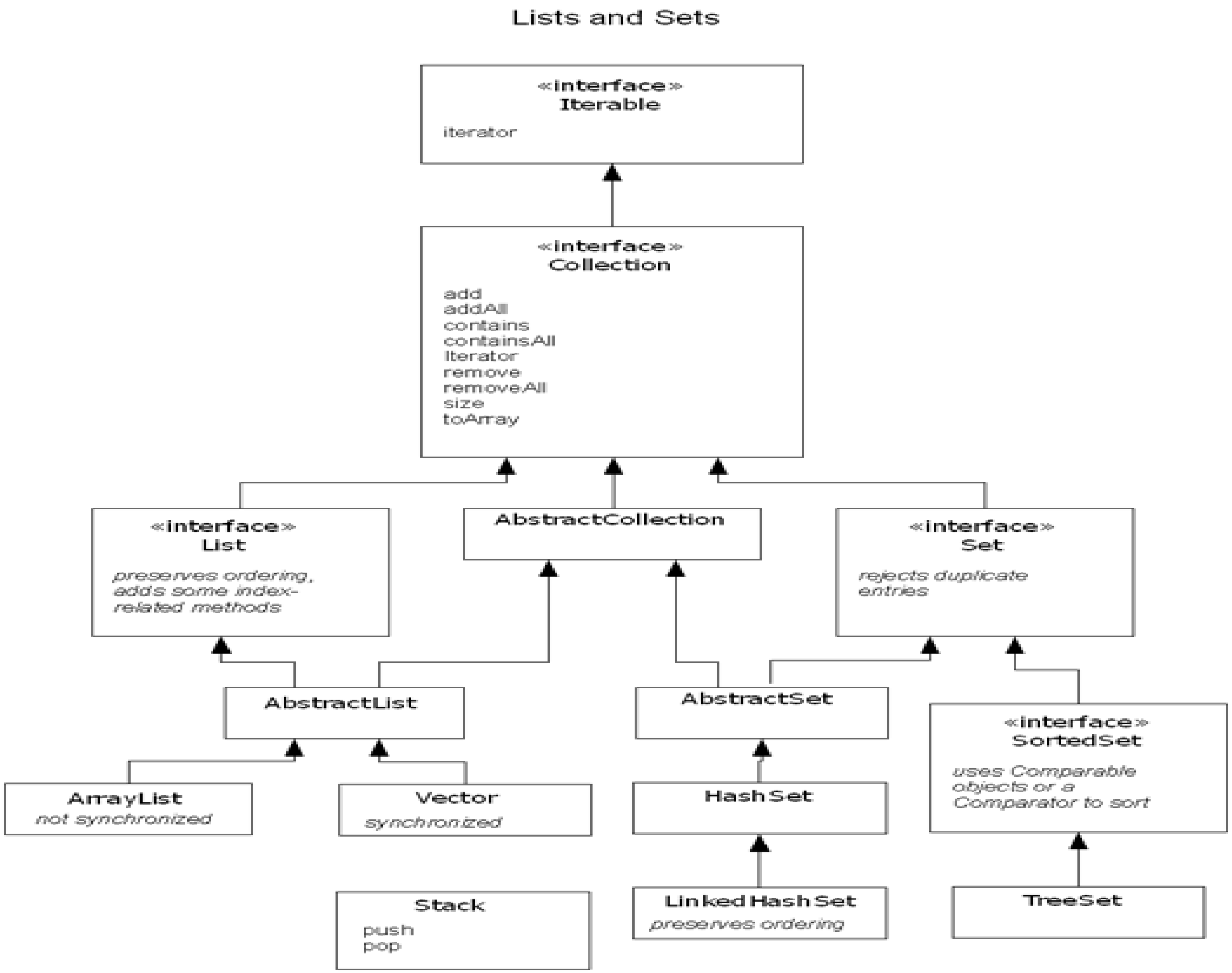
Effective Java Programming

data structures (collections and arrays)

Structure

- data structures (collections and arrays)
    - collections API
    - sets
    - lists
    - maps
    - arrays

Collections API – basic interfaces



Collections API – basic interfaces

- Iterator<E> - one way iterator
- Iterable<E> - get the iterator
- Collection<E>
  - Set<E> - no order, no duplicates
  - List<E> - order through indexes, duplicates
  - Queue<E> - duplicates, order managed by queue (FIFO, LIFO, natural, priorities, …)
  - Deque<E> - bidirectional
- Map<K, V> - key – value relations
  - keys cannot be duplicated

API – important functions

- Iterator<E>
  - hasNext() – true if next element exists
  - next() – gets next element and moves cursor
  - remove() – removes last element
- Iterable<E>
  - iterator() – get Iterator<E>
- Collection<E>
  - add(E)
  - remove(Object)
  - contains(Object)
  - size() – number of elements

API – important functions

- Set<E>
  - only inherited from Collection<E>
- List<E>
  - add(E) – adds to the end
  - add(int, E) – adds at position, rest mover right
  - set(int, E) – set at position
  - get(int)
  - remove(int)
  - indexOf(Object)
  - lastIndexOf(Object)

API – important functions

- Queue<E>

add to queue	*offer(E)* false if impossible	*add(E)* exception if impossible
get and remove from beginning	*poll()* null if empty	*remove()* exception if empty
get without removing	*peek()* null if empty	*element()* exception if empty

API – important functions

- Map<K, V>
  - put(K, V) – set value for given key
  - get(Object) – get value for key
  - remove(Object) – remove value for key
  - containsKey(Object)
  - containsValue(Object)
  - keySet() – set of keys
  - values() – collection of values
  - entrySet() – set of key-value pairs

Collections API – further interfaces

- ListIterator<E> - bidirectional iterator
- Comparable<T>
  - compare this object with another
  - used for natural order
  - class can only have one natural order
- Comparator<T>
  - compare two objects
  - need for new comparator instance
  - non-natural order
  - a class can have many comparators
- SortedSet<E> - set of ordered elements *
- SortedMap<E> - map with ordered keys *

API – important functions

- ListIterator<E>
  - hasPrevious(), previous()
- Comparable<T>
  - compareTo(T) – compares *this* with given object
    - < 0 – *this* is smaller, 0 – equal, > 0 – *this* is bigger
- Comparator<T>
  - compare(T, T) – compare two objects
- SortedSet<E>
  - first(), last()
- SortedMap<K, V>
  - firstKey(), lastKey()

Sets – standard implementations

- HashSet<E>
  - based on hashtable
  - elements must implement hashCode()
  - add(E), remove(Object), contains(Object) – O(1)
- TreeSet<E>
  - sorted (implements SortedSet<E>)
  - moves through elements in natural order or given by comparator
    - elements must implement Comparable<T> for natural order
  - additional functionality = time overhead

Sets – standard implementations

- LinkedhashSet<E>
  - similar to HashSet<E>
  - offers viewing in order of adding
  - faster browsing thanks to bidirectional list
  - add and remove are slower
  - search stays equally fast – hashtable
- EnumSet<E>
  - set only for one type of enumeration at once
  - extremely fast
  - based on vector of bits
    - every enumeration has known number of elements
    - element in set, bit set to 1, otherwise 0

Hashtable

- linear access to elements is too slow
- hashtable divides elements into sublists
- every sublist is in a seperate cell (bucket)
- element’s position is calculated from hashCode() and length of table
- for a perfect hashing function O(1)
  - every bucket has only one element
  - only possible if every possible value is known at beginning
- therefore the table contains lists
- access time depends on load factor
  - ratio of size of table to number of elements

Hashtable - efficiency

- efficency of hashtable depends on
  - capacity
  - load factor
  - hash function used on elements (hashCode)
- Java collections based on hashtables (HashSet, HashMap)
  - can set the initial capacity
  - can define maximum load factor
  - automatically reorganize when hitting boundary
    - allocate hashtable with doubled size
    - rehash and reorder elements in table

Hashtable - efficiency

- bigger initial capacity
  - greedy memory management
  - shorter access time
  - rare reorganization – time saving
  - longer element review
- smaller initial capacity – the opposite
- smaller load factor
  - shorter access time
  - often reorganization – time consuming, array gets bigger
  - longer element review (array grows)
- bigger load factor – the opposite
- both parameters should be set for expected amount of data

Hashtable - efficiency

- you can override hashCode()
  - should always return varied values
  - worst case – returns constant
    - hashtable becomes a list
    - hashCode and array become redundant
  - every wrapper class and String have optimized hash functions
  - contract with equals() has to be preserved
- ATTENTION
  - attributes used to calculate hashCode() cannot change after adding to structure
  - after change hashCode() will return new result and the object will be lost

Contract between *equals()* and *hashCode()*

- if called more then once on same object has to return same value
  - assuming data for equals stays the same
  - value can change between application runs
- if two objects are the same for equals, they must have same hashCode
  - does not work other way round!!!
  - if hashCode equal, objects are not necessarily equal
- if objects are not equal, hashCode does not have to be different. It is still better when it does
- CONCLUSION
  - when overriding hashCode override equals
  - calculate hashCode in a deterministic manner using ONLY data used in equals

equals contract

- Well written equals method must fulfill:
  - x.equals(x) == true
  - if x.equals(y) == true then y.equals(x) == true
  - if x.equals(y) == true and y.equals(z) == true then x.equals(z) == true
  - x.equals(y) returns the same as long all x and y are the same
  - x.equals(null) == false

Sets – NavigableSet (Java 6)

- extends SortedSet<E>
  - allows forward and backward navigation
    - descendigSet() returns reverse
    - iterating in reverse is slower
  - additionally
    - return subsets with indication whether top and bottom border range are included
    - nearest to target element
    - returning and deleting biggest and smallest element
- implemented in TreeSet<E>

Sets – efficiency comparison

- adding words from books to sets

source document	number of words	number of different words	HashSet [s]	TreeSet [s]
Alice in Wonderland	28 195	5 909	5	7
The Count of Monte Cristo	466 300	37 545	75	98

Taken from „Java 2. Advanced techniques", Horstmann, Cornell

Sets – efficiency comparison

- adding elements – add (time in ns)

Size	Implementation		
	TreeSet	HashSet	LinkedHashSet
10	746	308	350
100	**501**	**178**	**270**
1 000	714	216	303
10 000	1 975	711	1 615

The anomalies are caused by an error in the test environment – collections were cleared after every run, not recreated (some collections change their infrastructure)

Sets – efficiency comparison

- Checking elements – contains (time in ns)

Size	Implementation		
	TreeSet	HashSet	LinkedHashSet
10	**173**	91	65
100	**264**	75	74
1 000	**410**	110	111
10 000	**552**	215	256

Taken from „Thinking in Java”, Bruce Eckel

Sets – efficiency comparison

- iteration (time in ns)

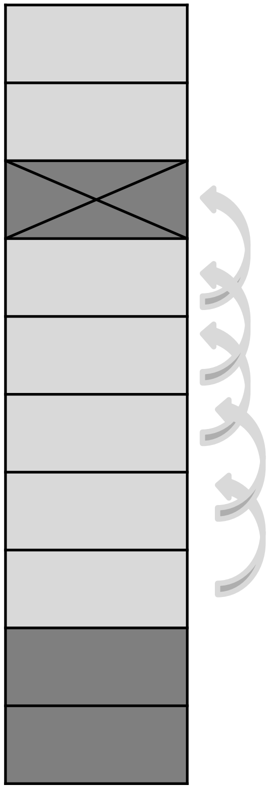
Size	Implementation		
	TreeSet	HashSet	LinkedHashSet
10	89	**94**	83
100	68	**73**	55
1 000	69	**72**	54
10 000	69	**100**	58

Taken from „Thinking in Java”, Bruce Eckel

Lists – standard implementation

- ArrayList<E>
  - based on array
  - get(int), set(int, E), add(E) – O(1)
  - add(int, E), remove(Object) – O(N)
    - has to move every element
- LinkedList<E>
  - based on bidirectional list
  - add(E) – O(1)
  - get(int), set(int, E) – O(N)
    - iterates over every element. If at the end, reverse iterator
  - add(int, E), remove(Object) – O(N)
    - still better than ArrayList<E>

ArrayList<E> - removing element



beginning

element to be removed

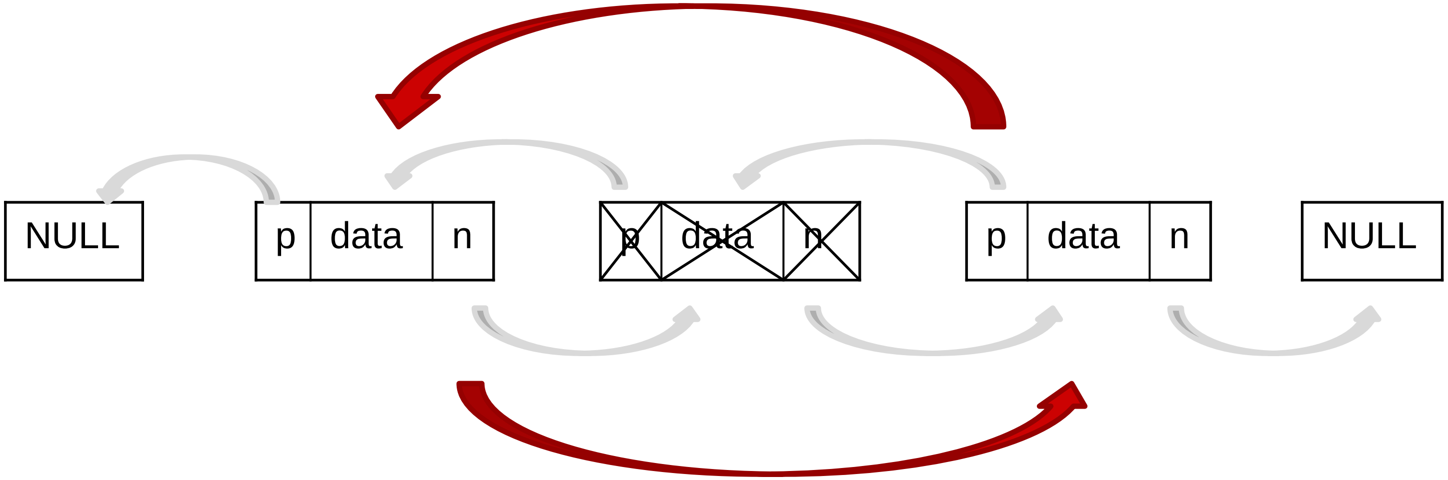
size of list

end

ArrayList<E> - optimization

- list has initial capacity
  - set in constructor
  - when array grows
    - a new is allocated, elements are copied
- bigger initial size
  - higher memory usage
  - rare rewriting
- smaller initial size – the opposite
  - ensureCapacity() – changes capacity
    - use always before adding much data
    - time saving

LinkedList<E> - removing element



Lists – RandomAccess interface

- marker interface telling, that list has constant access time
- list should implement this interface if index-access is faster than iterating
- algorithms should adapt to access time of list
  - constant – iterate over indexes
  - linear – use iterator
    - iterator has pointer to next element in list
    - this way we achieve constant access time

Lists - optimization

- adjust implementation to most often used operation
- check if list implements RandomAccess
  - if yes

```
for (int i = 0, n = list.size(); i < n; i++) {
  list.get(i);
}
```

  - if no

```
for (Iterator i = list.iterator(); i.hasNext(); ) {
  i.next();
}
```

Lists - optimization

- do not calculate size in every iteration

```
for (int i = 0; i < list.size(); i++) {
  list.get(i);
}
// better
for (int i = 0, n = list.size(); i < n; i++) {
  list.get(i);
}
```

- avoid often realocation

```
ArrayList<String> list = new ArrayList<String>();
list.ensureCapaticy(1024);
// better
ArrayList<String> list = new ArrayList<String>(1024);
```

Lists – efficency comparison

- adding elements at the end – add (time in ns)

Size	Implementation	
	ArrayList	LinkedList
10	121	**182**
100	72	**106**
1 000	98	**133**
10 000	122	**172**

Integer objects were added

Taken from „Thinking in Java”, Bruce Eckel

Lists – efficency comparison

- getting elements from random positions – get (time in ns)

Size	Implementation	
	ArrayList	LinkedList
10	139	**164**
100	141	**202**
1 000	141	**1 289**
10 000	144	**13 648**

Integer objects were used

Taken from „Thinking in Java”, Bruce Eckel

Lists – efficency comparison

- changing elements at random positions – set (time in ns)

Size	Implementation	
	ArrayList	LinkedList
10	191	**198**
100	191	**230**
1 000	194	**1 353**
10 000	190	**13 187**

Integer objects were used

Taken from „Thinking in Java”, Bruce Eckel

Lists – efficency comparison

- adding in the middle of the list (time in ns)

Size	Implementation	
	ArrayList	LinkedList
10	435	**658**
100	247	**457**
1 000	**839**	430
10 000	**6 880**	435

Integer objects were used

Taken from „Thinking in Java”, Bruce Eckel

Lists – efficency comparison

- adding at the beginning of the list – add(int, E) (time in ns)

Size	Implementation	
	ArrayList	LinkedList
10	**3 952**	366
100	**3 934**	108
1 000	**2 202**	136
10 000	**14 042**	255

Integer objects were used
Elements were added at 5th position.
Added 5000 elements for first three sizes, 500 for fourth

Taken from „Thinking in Java”, Bruce Eckel

Lists – efficency comparison

- removing from beginnig – remove(int, E) (time in ns)

Size	Implementation	
	ArrayList	LinkedList
10	**466**	262
100	**296**	201
1 000	**923**	239
10 000	**7 333**	239

Integer objects were used
Repeated till size of list greater than 5

Taken from „Thinking in Java”, Bruce Eckel

Maps – standard implementations

- HashMap<K,V>
  - based on hashtable (for keys)
  - optimization through capacity and load factor
  - get(Object), put(K, V) – O(1)
- TreeMap<K,V>
  - based on binary tree
  - ascending order based on natural order or comparator
  - get(Object), put(K, V), remove(Object), containsKey(Object) – O(log(N))
  - can return sections of tree subMap(K, K)

Maps – standard implementations

- LinkedHashMap<K,V>
  - like HashMap<K,V> (organization, optimization)
  - additionally viewing in insertion order or access order
    - contains additional bidirectional list – faster viewing
    - longer time for add and delete
    - same search time, still a hash map

Maps – how to build a cache

- LinkedHashMap<K,V> is a ready implementation
- to build a cache
  - create a class extending LinkedHashMap<K,V>
  - override removeEldestEntry(Map.Entry)
    - called automatically on every add
    - provides oldest element *
    - return true if element should be removed from cache
    - deletion is automatic

```
private static final int MAX_ENTRIES = 100;
protected boolean removeOldestEntry(Map.Entry oldest) {
  return size() > MAX_ENTRIES
}
```

Maps - optimization

- load factor and capacity (or expectedMaxSize) for maps based on hashtable – set in constructor
- map copying techniques
  - sometimes a module copies a map only to view it and discard later changes
    - before copying a TreeMap<K,V> change to LinkedHashMap<K,V>!
    - constructor allows copying
    - ensured order, without time overhead of binary tree (O(1) instead of O(log(N)))

Maps - optimization

- If map does not contain null values or you ignore them, retrieve value immediately

```
if (map.containsKey(key)) {
  System.out.println(„key: ” + key + „ value: ” + map.get(key));
}

// better
Integer value = map.get(key);
if (value != null) {
 System.out.println(„key: ” + key + „ value: ” + value);
}
```

Maps - optimization

- Never iterate over keys, always over entrySet()

```
for (Integer key : map.keySet()) {
  System.out.println(„key: ” + key + „ value: ” + map.get(key));
}

// better
for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
  System.out.println(„key: ” + entry.getKey() + „ value: ”
        + entry.getValue());
}
```

Maps – NavigableMap<K,V>(Java 6)

- extends SortedMap<K,V>
  - allows iterating in both directions
    - descendingMap() returns reversed map
    - iterating in reverse is slower
  - additional methods
    - return submaps with indication if upper or lower boundary is included
    - return neares element in map to given key
    - return and remove biggest and smallest element
- implemented by TreeMap<K,V>

Maps – efficiency comparison

- adding elements – put(K,V) (time in ns)

Size	Implementation				
	TreeMap	HashMap	Linked-HashMap	Identity-HashMap	Week-HashMap
10	**748**	281	354	290	146
100	**506**	179	273	204	126
1 000	**771**	267	385	508	136
10 000	**2 962**	1 305	2 787	767	138

Integer objects were used
IdentityHashMap in not of wide use
WeakHashMap – unrealistically good results. Map did not contain references to elements, was constantly cleared and small

Taken from „Thinking in Java”, Bruce Eckel

Maps – efficiency comparison

- getting elements – get(Object) (time in ns)

Size	Implementation				
	TreeMap	HashMap	Linked-HashMap	Identity-HashMap	Week-HashMap
10	**168**	76	100	144	146
100	**264**	70	89	287	126
1 000	**450**	102	222	336	136
10 000	**561**	265	341	266	138

Integer objects were used

Taken from „Thinking in Java”, Bruce Eckel

Maps – efficiency comparison

- iterating over entrySet() (time in ns)

Size	Implementation				
	TreeMap	HashMap	Linked-HashMap	Identity-HashMap	Week-HashMap
10	**100**	**93**	72	101	151
100	**76**	**73**	50	132	117
1 000	**78**	**72**	56	77	152
10 000	**83**	**97**	56	56	555

Integer objects were used

Taken from „Thinking in Java”, Bruce Eckel

Collections - views

- many collections have methods returning subranges
  - List<E> - subList(from, to)
  - SortedSet<E> - subSet(from,to), headSet(to), tailSet(from)
  - SortedMap<K,V> - subMap(from,to), headMap(to), tailMap(from)
- those collections are only **views**
  - underneath the parent collection
  - changes in view visible in parent and vice-versa

Collections – views - optimization

- removing elements in range

```
for (int i = from; i < to; i++) {
  list.remove(i);
}
// better
list.subList(from, to).clear();
```

- much faster solution
  - the faster, the more elements are removed
  - critically faster for ArrayList<E>

Arrays

- Array provides fastest data access
  - allocates continous amount of memory
  - every cell is of same size (simple type or reference)
  - cell address = start address + index * cell size
- cannot be resized
  - can be copied to bigger array
  - ArrayList<E> uses this mechanism underneath
  - System.arraycopy – efficient array copying
    - copies whole memory fragment
    - better than copying in loops
- can contain simple types!

Arrays – helper class *Arrays*

- similar to collections, array have a helper class
  - asList – packs array into List<E>
    - constant size!
    - changes in list visible in array
  - binarySearch
    - array has to be sorted
    - array will be search in sort order
  - copyOf, copyOfRange – copies array or range creating new array of given size
    - System.arraycopy underneath
    - *Arrays* provides the second array to System.arraycopy
    - for arrays of objects only references will be copied, not objects

Arrays – helper class *Arrays*

- Arrays
  - equals – are the arrays equal
    - same elements in same order
  - fill – an array with given value
  - deepEquals/hashCode/toString – for multidimensional arrays
  - hashCode – calculated over content of array
  - sort
    - for simple types in natural order
    - for objects in natural order or using comparator
  - toString – content of table as string

Conclusions

- What collection implementations do you know?
- Which collection is best for general purpose?
- What list implementations do you know?
- What map implementations do you know?
- Which map is best for general purpose?
- What are the advantages of arrays over lists?
- What are the advantages of lists over arrays?