Seite 3: Sexismus – wo Grenzen überschritten werden

DER ALBRECHT

UNABHÄNGIGE HOCHSCHULZEITUNG AN DER CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

Januar 2018 www.der-albrecht.net 19. Jahrgang, Kiel Nr. 131 | kostenlos

„Stark abhängig von der studentischen Beteiligung“

Die Evaluation der Lehrveranstaltung als Chance

Judith Rödger

QUELLE: MA

Was haben wir nicht alle schon in schlechten Vorlesungen oder Seminaren gesessen und uns Gedanken gemacht über die verschwendete Lebenszeit und Dinge, die wir stattdessen viel lieber getan hätten. Die Wohnung aufräumen, das Geschirr abwaschen, so einiges wäre manches Mal spannender gewesen. Doch wem berichten wir von unseren schlechten Erfahrungen oder auch dem Seminar, das uns tatsächlich gut gefallen hat, bei dem wir wirklich das Gefühl hatten, etwas gelernt zu haben? Meistens wohl nur unseren Kommiliton*innen. Das ist natürlich schön und gut, so unsere Erfahrungen weiterzugeben, damit andere wissen, bei wem eine gute Lehre erwartet werden kann und bei wem eher das Gegenteil der Fall ist. Wäre es da nicht trotzdem klüger, dem betreffenden Dozenten oder der Dozentin eine Rückmeldung darüber zu geben, was gut und was schlecht war? Wie gut, dass es an der CAU eine Möglichkeit gibt, mit dem dies getan werden kann – die Evaluation. Doch wie sinnbringend ist dieses System der Evaluation mit dem immer gleichen Fragebogen, der oft nicht recht zu der Lehrveranstaltung zu passen scheint, und wird es überhaupt genutzt? In den meisten Studiengängen, jedoch nicht in allen, wird regelmäßig nach einer Lehrveranstaltung eine Evaluation durchgeführt, manchmal sogar mit besonderen Anreizen für die Studierenden, die zum Beispiel mit einem Frühstück gelockt werden, wie es aus dem Studiengang Psychologie berichtet wird. Bei einer Umfrage des ALBRECHT auf *facebook* kam heraus, dass in anderen Fächern, wie Anglistik, Germanistik, Romanistik oder Geschichte zwar auch Evaluationen durchgeführt werden, jedoch kommen Zweifel auf, ob diese tatsächlich gewinnbringend sind. Denn auf der einen Seite nutzen nur sehr wenige Studierende die Möglichkeit, um ihre Rückmeldung zu geben, auf der anderen Seite kommt wiederum von den jeweiligen Dozierenden keine Rückmeldung zum Ergebnis der Evaluation, sodass das Gefühl entsteht, dass Feedback und Anregungen im Nichts verpuffen. Dabei sollte angemerkt werden, dass es keinen Nutzen hat, alles mit schlechten Noten zu bewerten, ohne zu sagen, was genau schlecht oder verbesserungswürdig ist. Der Satz: „Nicht beurteilbar, da nicht anwesend“, bringt da leider auch nicht sehr viel.

Die Lehrkräfte haben ebenso unterschiedliche Erfahrungen gemacht. So berichtet Professor Volker Seresse aus dem Fachbereich Geschichte, dass er in kleineren Lehrveranstaltungen einen auf den Kurs zugeschnittenen Fragebogen als Möglichkeit für die Rückmeldung anbietet. In größeren Vorlesungen kommt dann der vorgefertigte Evaluationsbogen zum Einsatz. Dort ist die Aussagekraft der Evaluation „jedoch stark abhängig von der studentischen Beteiligung“, berichtet Seresse.

Professor Andreas Mühling aus dem Fachbereich Informatik berichtet von seinen Kursen, in denen das Angebot der Evaluation gut genutzt wird. Jedoch seien dies eher kleine Lehrveranstaltungen und er werbe zusätzlich für die Teilnahme. „Ich versuche darüberhinaus auch andere Formen der Evaluation einzusetzen, die mir bereits früher im Semester ein Feedback zu den Veranstaltungen geben, sodass man gegebenenfalls rechtzeitig reagieren kann und nicht erst Verbesserungen im nächsten Durchlauf möglich sind“, so Mühling.

Der vorgefertigte Evaluationsbogen mit den immer gleichen Fragen ist also nicht die einzige Möglichkeit für die Dozierenden, sich Rückmeldungen von den Studierenden zu holen. Wichtig ist jedoch, dass möglichst viele an den Befragungen, in welcher Form auch immer, teilnehmen und ihre Kritik äußern. Allerdings sind viele Dozierende selbst kein Paradebeispiel. Von den fünf für die Recherche dieses Artikels angeschriebenen Dozierenden, antworteten nur zwei. Nur durch rege Teilnahme, von allen Seiten, kann die Lehre für die Zukunft optimiert werden. Also erzählt es nicht nur euren Kommiliton*innen, wenn eine Veranstaltung besonders gut oder schlecht war. Gebt das Feedback an die entsprechenden Lehrbeauftragten weiter, denn nur durch Lob oder konstruktive Kritik der Studierenden wissen sie, was besonders gut, oder was eher ein Schuss in den Ofen war. Wenn also wieder die unzähligen Aufforderungen zur Evaluationsteilnahme kommen, klickt sie dieses Mal nicht einfach weg, sondern nutzt eure Chance, die Lehre zu verbessern.

EvaSys wants YOUR input NOW!

[Wikimedia Commons]

Gemeinsames Ziel:
Studierbarkeit verbessern

In EvaSys-Freitext, bitte folgende Fragen [Perle-TAP] beantworten:

1. Wodurch lernen Sie in dieser Veranstaltung am meisten?
2. Was erschwert Ihr Lernen?
3. Welche Verbesserungs-vorschläge haben Sie für die hinderlichen Punkte?

Five-Minute Review

1. What *debugging strategies* do you know?
2. When should you use *assertions*?
3. What is a *Hoare Triple*?
4. What is *multiple inheritance*? Does Java support it?
5. What is an *interface* in Java?

Programming – Lecture 11

Collection Classes (Chapter 13)

- **`HashMap`** class
- Aside: Computational complexity, Big-O
- Bucket hashing
- Java Collections Framework – lists, sets, maps
- Iteration in Collections
- Map hierarchy
- **`Collections`** class
- Principles of OO design

HashMap

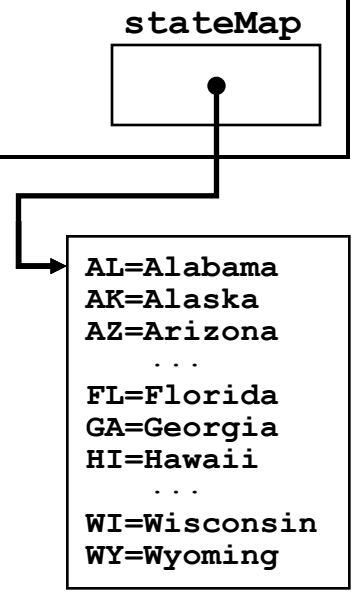
Map: mapping from a set of unique *keys* to *values*

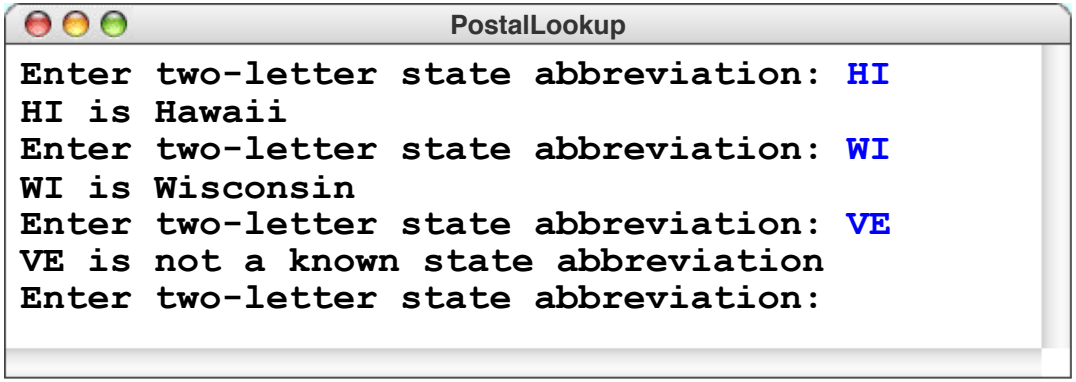
`new HashMap<>()`	Creates a new **`HashMap`** object that is initially empty.
map`.`**`put`**`(`*key*`,` *value*`)`	Sets the association for *key* in the map to *value*.
map`.`**`get`**`(`*key*`)`	Returns the value associated with *key*, or **`null`** if none.

Generic types for keys and values:
`HashMap<String,Integer>`
uses strings as keys and int's as values

PostalLookup

```
public void run() {
   HashMap<String,String> stateMap = new HashMap<>();
   initStateMap(stateMap);
   while (true) {
      String code = readLine("Enter two-letter state abbreviation: ");
      if (code.length() == 0) break;
      String state = stateMap.get(code);
      if (state == null) {
         println(code + " is not a known state abbreviation");
      } else {
         println(code + " is " + state);
      }
   }
}
```





skip simulation

Implementing `put` and `get`

map`.put(`*key*`, `*value*`)`	Sets the association for *key* in the map to *value*.
map`.get(`*key*`)`	Returns the value associated with *key*, or `null` if none.

First approach: linear search in parallel arrays

Efficiency?

I.e., how costly is each get/put in terms of computation time?

Aside: Computational Complexity

What is a good cost measure?

- **Not:** seconds or clock cycles
- **Instead:** count *operations*, relative to *problem size*
- An operation may be arbitrarily complex/abstract, but is assumed to take constant time
- Thus, ignore constant factors
- Are (at least here) not interested in absolute number of operations
- Instead, want to know how the operation count increases if problem size increases, and asymptotically approaches ∞
- Typically, are interested in *worst case*
- May also be interested in *average case*

Aside: Computational Complexity

For maps:

- Problem size *N* = number of elements

Implementing maps with parallel arrays:

- put takes a constant number of operations; thus, may consider put as 1 operation
- put “takes constant time”; denote this as ***O*(*1*)**
- get takes at most *N* (on average *N*/2) operations
- get "takes linear time", "has linear (asymptotic) complexity”; denote this as ***O*(*N*)**

Bachmann-Landau Notation – Big-O

Algorithm has complexity $O(f(N))$:
For large N, the growth rate of its run time
behaves as the growth rate of $f(N)$

Formally: let $t(N)$ denote run time for problem size N.
$t(N) \in O(f(N))$ iff $\exists c \in \mathbb{R}, N_0 \in \mathbb{N} \,.\, \forall N > N_0 \,.\, t(N) < c\, f(N)$

Simplify $f(N)$ as much as possible:

- Eliminate parts that disappear as N becomes large
- Eliminate constant factors

Thus don't write $O(2N + 3N^2 + \log N^3)$, but simply $O(N^2)$.
(Mathematically, both would be equivalent, but we always ask for the simplified form **only**.)

But don't forget: in practice, constants matter as well.
What is really best algorithm depends on actual problem sizes.

Assume problem size N = `n`, loop body $O(1)$

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < i; j++) {
       . . . loop body . . .
   }
}
```

```
for (int k = 1; k <= n; k *= 2) {
   . . . loop body . . .
}
```

```
for (int i = 0; i < 100; i++) {
   for (int j = 0; j < i; j++) {
       . . . loop body . . .
   }
}
```

Please visit pingo.upb.de/643250

https://xkcd.com/2098

Implementing `put` and `get`

map`.put(`*key*`, `*value*`)`	Sets the association for *key* in the map to *value*.
map`.get(`*key*`)`	Returns the value associated with *key*, or **`null`** if none.

1) Linear search in parallel arrays
   $O(N)$
2) Binary search in parallel arrays
   $O(\log N)$
3) Table lookup in a two-dimensional array
   $O(1)$
   But requires space $O(A^2)$, for A = # letters
4) Bucket hashing
   $O(1)$ on average if "enough" buckets
   (i.e., if linked lists have average length $O(1)$)

Hashing

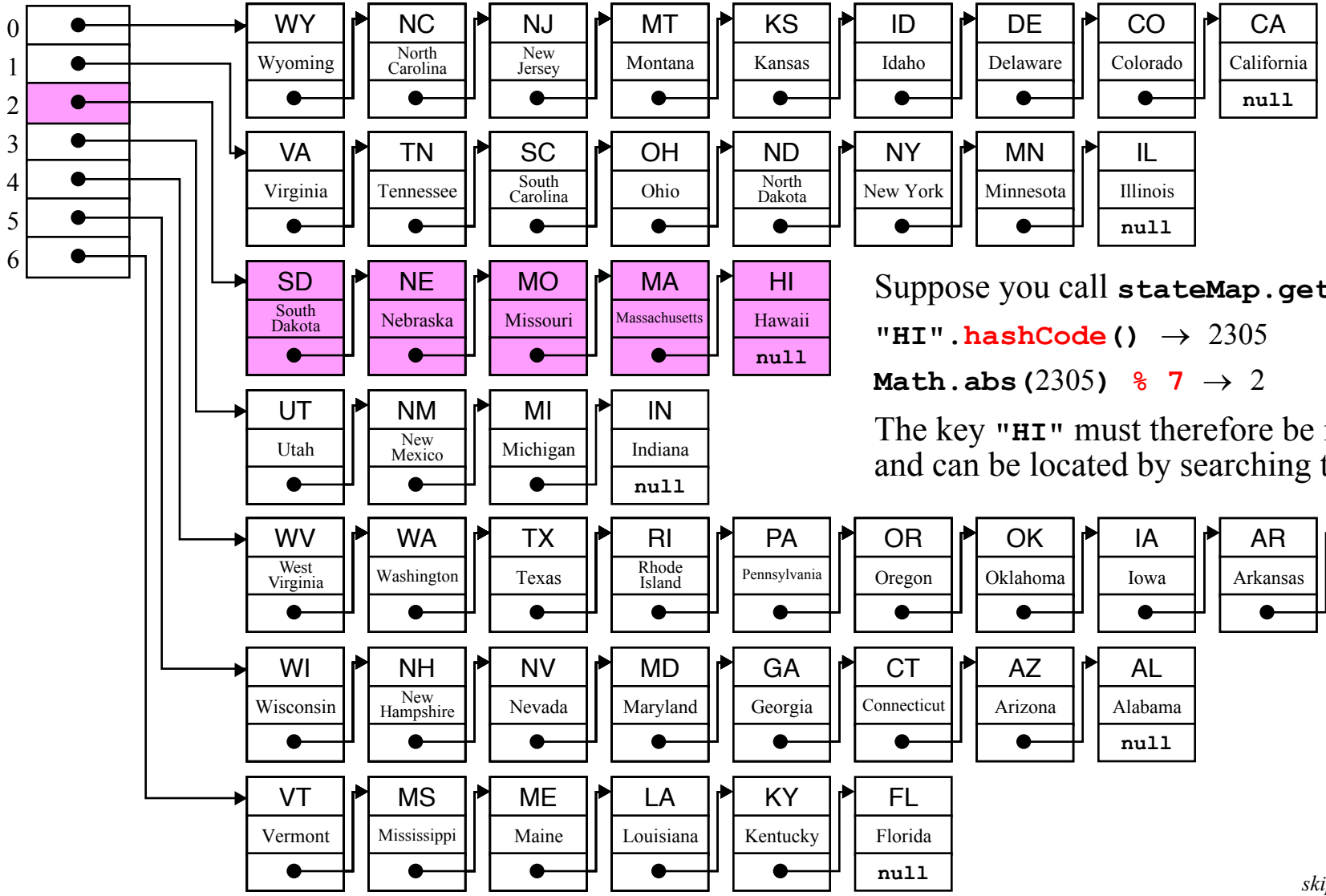
Each object has *hash code*

- To get it: **`hashCode()`**, returns **`int`**
- If objects have equal values, the objects have same hash codes
- Hash codes must be unique for given object
- *Collision*: different values are mapped to same hash code
- Want hash function that minimizes collisions

Bucket hashing:

- Map hash codes (of keys) to buckets
- In bucket, linked list of (key, value) pairs

Bucket Hashing



Suppose you call **`stateMap.get("HI")`**

`"HI".hashCode()` → 2305

`Math.abs`(2305) **`%`** **`7`** → 2

The key **`"HI"`** must therefore be in bucket 2 and can be located by searching the chain.

skip simulation

Bucket Hashing

```
public class SimpleStringMap {
   private static final int N_BUCKETS = 7;
   private HashEntry[] bucketArray;

   /** Creates a new SimpleStringMap with no key/value pairs */
   public SimpleStringMap() {
      bucketArray = new HashEntry[N_BUCKETS];
   }

   /**
    * Sets the value associated with key.  Any previous value for key is lost.
    * @param key The key used to refer to this value
    * @param value The new value to be associated with key
    */
   public void put(String key, String value) {
      int bucket = Math.abs(key.hashCode()) % N_BUCKETS;
      HashEntry entry = findEntry(bucketArray[bucket], key);
      if (entry == null) {
         entry = new HashEntry(key, value);
         entry.setLink(bucketArray[bucket]);
         bucketArray[bucket] = entry;
      } else {
         entry.setValue(value);
      }
   }
```


skip code

Bucket Hashing

```
   /**
    * Retrieves the value associated with key, or null if no such value exists.
    * @param key The key used to look up the value
    * @return The value associated with key, or null if no such value exists
    */
    public String get(String key) {
       int bucket = Math.abs(key.hashCode()) % N_BUCKETS;
       HashEntry entry = findEntry(bucketArray[bucket], key);
       if (entry == null) {
          return null;
       } else {
          return entry.getValue();
       }
    }

    /*
     * Scans the entry chain looking for an entry that matches the specified key.
     * If no such entry exists, findEntry returns null.
     */
    private HashEntry findEntry(HashEntry entry, String key) {
       while (entry != null) {
          if (entry.getKey().equals(key)) return entry;
          entry = entry.getLink();
       }
       return null;
    }
}
```

Bucket Hashing

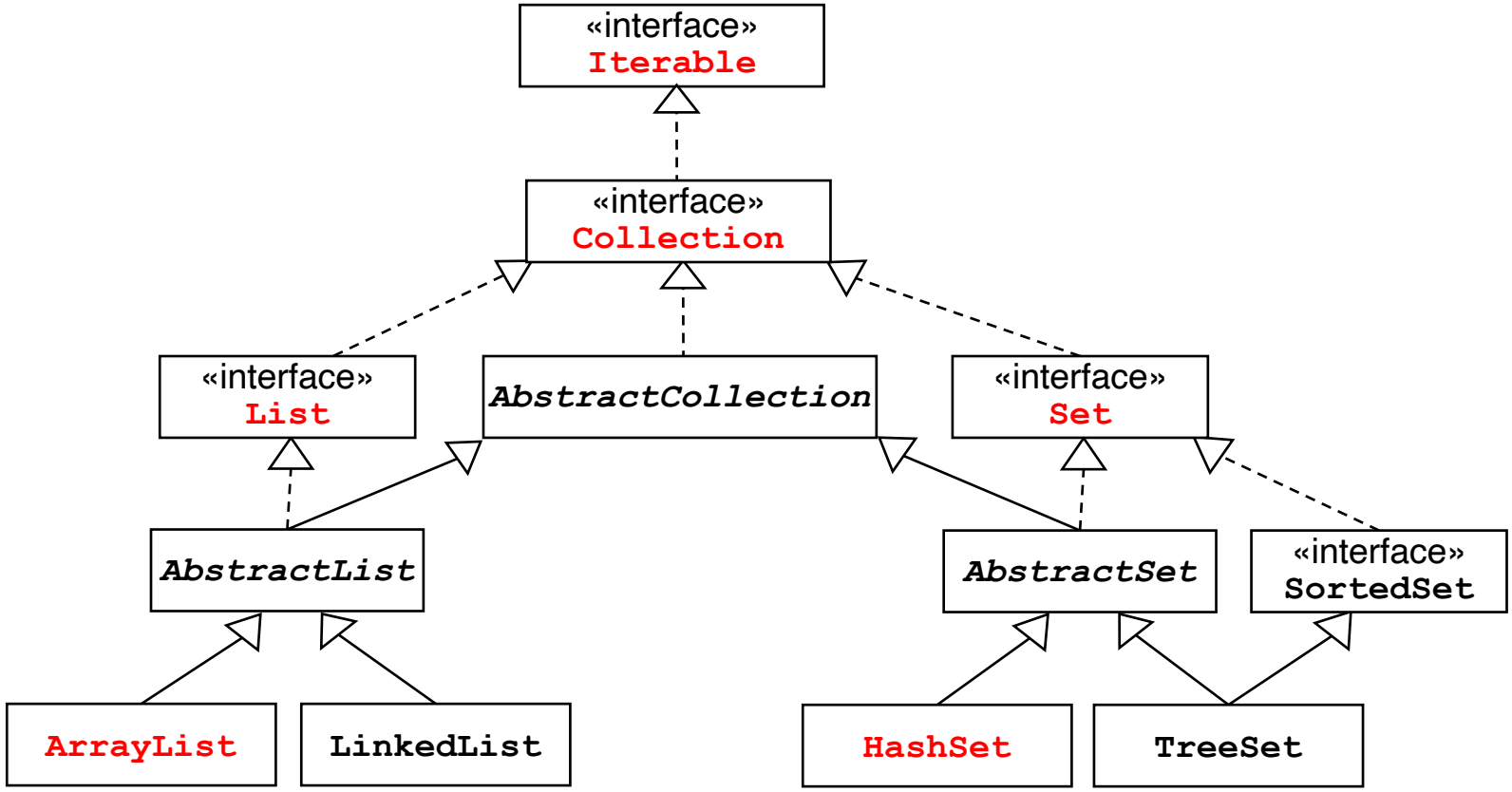
```
/* Package-private class: HashEntry */
/*
 * This class represents a pair of a key and a value, along with a reference
 * to the next HashEntry in the chain.  The methods exported by the class
 * consist only of getters and setters.
 */
class HashEntry {

   /* Private instance variables */
   private String key;     /* The key component for this HashEntry   */
   private String value;   /* The value component for this HashEntry */
   private HashEntry link; /* The next entry in the chain            */


   /* Creates a new HashEntry for the specified key/value pair */
   public HashEntry(String key, String value) {
      this.key = key;
      this.value = value;
   }

   /* Returns the key component of a HashEntry */
   public String getKey() {
      return key;
   }
```

Bucket Hashing

```
    /* Returns the value component of a HashEntry */
    public String getValue() {
       return value;
    }

    /* Sets the value component of a HashEntry to a new value */
    public void setValue(String value) {
       this.value = value;
    }

    /* Returns the next link in the entry chain */
    public HashEntry getLink() {
       return link;
    }

    /* Sets the link to the next entry in the chain */
    public void setLink(HashEntry link) {
       this.link = link;
    }
}
```

Java Collections Framework

Maps + lists (ordered) + sets (unordered)



... and much more

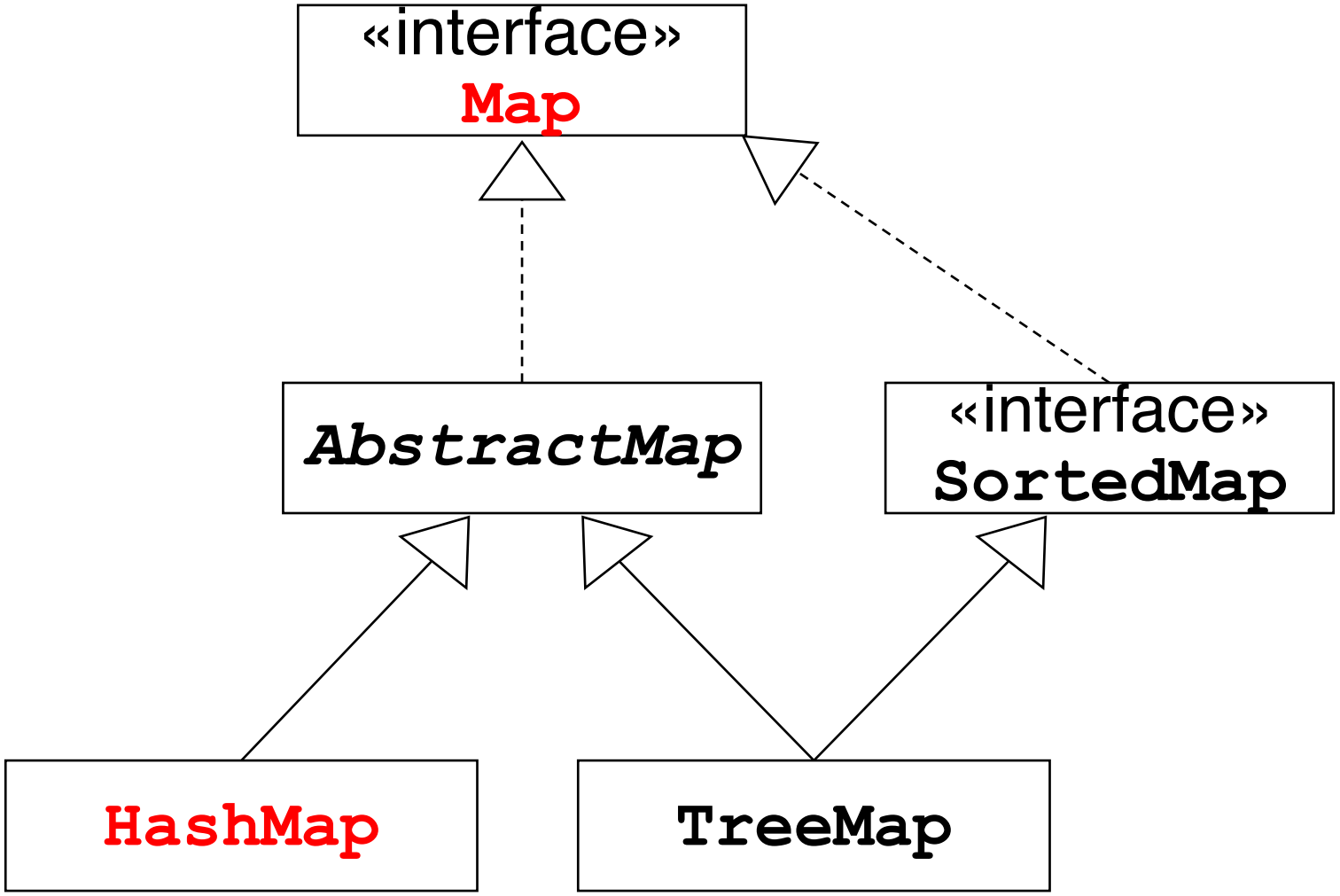
Iteration in Collections

```
import java.util.*;

Iterator<type> iterator = collection.iterator();
while (iterator.hasNext()) {
    type element = iterator.next();
    . . . statements that process this particular element . . .
}
```

Simplify to *for-each*:

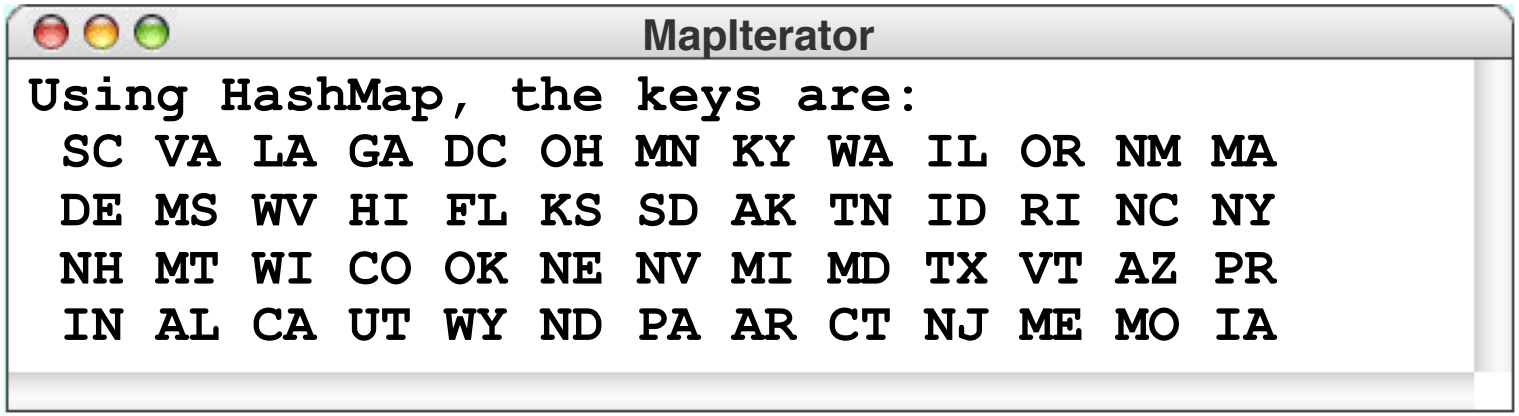
```
for (type element : collection) {
    . . . statements that process this particular element . . .
}
```

`Map` Hierarchy



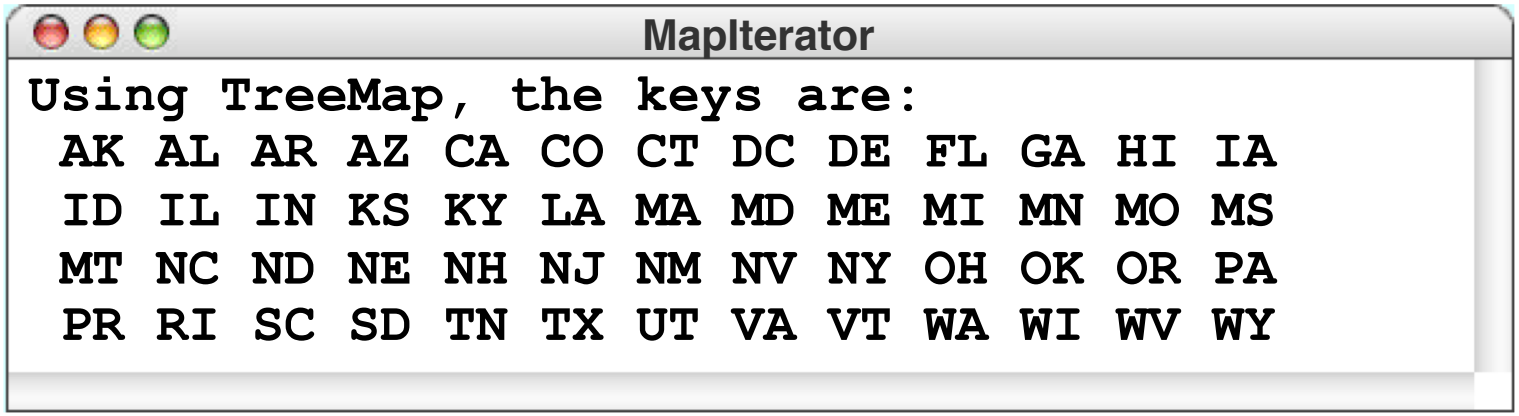
Iteration Order in **HashMap**

```
private void listKeys(Map<String,String> map, int n) {
    String className = map.getClass().getName();
    int lastDot = className.lastIndexOf(".");
    String shortName = className.substring(lastDot + 1);
    println("Using " + shortName + ", the keys are:");
    int i = 0;
    for (String key : map.keySet()) {
       print(" " + key);
       if (++i % n == 0) println();
    }
}
```



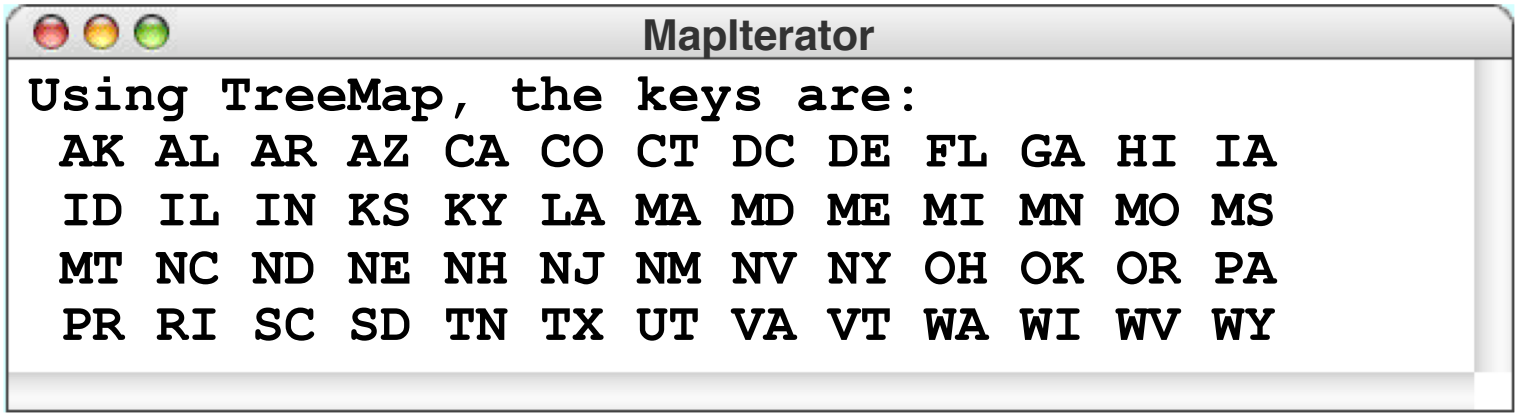
Iteration Order in `TreeMap`

```
private void listKeys(Map<String,String> map, int n) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   int i = 0;
   for (String key : map.keySet()) {
      print(" " + key);
      if (++i % n == 0) println();
   }
}
```



Iteration Order in `TreeMap`

```
private void listKeys(Map<String,String> map, int n) {
   String className = map.getClass().getName();
   int lastDot = className.lastIndexOf(".");
   String shortName = className.substring(lastDot + 1);
   println("Using " + shortName + ", the keys are:");
   Iterator<String> iterator = map.keySet().iterator();
   for (int i = 1; iterator.hasNext(); i++) {
      print(" " + iterator.next());
      if (i % n == 0) println();
   }
}
```



List Methods in `Collections` Class

`binarySearch(` *list*`,` *key*`)`	Finds *key* in a sorted list using binary search.
`sort(`*list*`)`	Sorts a list into ascending order.
`min(`*list*`)`	Returns the smallest value in a list.
`max(`*list*`)`	Returns the largest value in a list.
`reverse(`*list*`)`	Reverses the order of elements in a list.
`shuffle(`*list*`)`	Randomly rearranges the elements in a list.
`swap(`*list*`,` p_1`,` p_2`)`	Exchanges the elements at index positions p_1 and p_2.
`replaceAll(` *list*`,` x_1`,` x_2`)`	Replaces all elements matching x_1 with x_2.

`java.util` provides similar **`Arrays`** class

Principles of OO Design

Packages should be

- Unified
- Simple
- Sufficient
- Flexible
- Stable
- Well-documented

Summary

- Java Collections Framework is unified architecture for representing and manipulations collections
- Important interfaces are *lists* (indexed), *sets* (not indexed) and *maps* (map keys to values)
- **`ArrayList`** is more efficient for selecting particular element or for searching an element; **`LinkedList`** is more efficient for adding/removing elements
- **`HashSet`** builds on *hashing* (fast, unordered), **`TreeSet`** builds on *binary trees* (slower, ordered)
- Lists and sets allow *iteration* (*for-each*)
- Packages should be *unified*, *simple*, *sufficient*, *flexible*, and *stable*