Programmierpraktikum

Claudius Gros, SS2012

Institut für theoretische Physik
Goethe-University Frankfurt a.M.

Collections and Maps

interfaces for collections and maps

there are more data types then just arrays

- **`java.util.Collection`** for lists, sets, queues
- **`java.util.Map`** for associative memory
- basic operations:
  - *request number of elements*
  - *add, delete, select, find elements*
  - *set operations*
  - *array conversion*

ArrayLists

- flexible array of objects

```
 1. import java.util.*;
 2.
 3. public class ArrayListDemo {
 4.
 5. public static void main(String args[]) {
 6.
 7.   ArrayList<String> aL = new ArrayList<String>();
 8.       // create an empty array list of Strings
 9.   System.out.printf("%25s: %d\n","initial size of aL",aL.size());
10. // ***
11. // *** add objects to the array list
12. // ***
13.   aL.add("This");
14.   aL.add("is");
15.   aL.add("an");
16.   aL.add("arraylist");
17.   aL.add("of");
```

```
18.   aL.add("String objects");
19.   aL.add(1, "is really");
20.   System.out.printf("%25s: %d\n","size of aL after adding",aL.size());
21. // display the array list
22.   System.out.printf("%25s: %s\n\n","contents of aL: ",aL);
23. // Remove elements from the array list
24.   aL.remove("of");
25.   aL.remove(2);
26.   System.out.printf("%25s: %d\n","size of aL after deleting",aL.size());
27.
28.   System.out.printf("%25s: ","contents of aL: ");
29.   for (int i=0;i<aL.size();i++)   // .size() and not .length as for arrays
30.     System.out.printf("%s ",aL.get(i));
31.   System.out.printf("\n\n");
32.   aL.clear();                        // remove all elements
33.   System.out.printf("%25s? %b\n","is aL empty",aL.isEmpty());
34.   }  // end of ArrayListDemo.main()
35. }    // end of ArrayListDemo
```

list operations

- **`boolean add(`***`element`***`)`**: adds *element* to the container; returns success
- **`boolean remove(`***`element`***`)`**: deletes *element* from the container; returns success
- **`void clear()`**: deletes all elements of the container
- **`int size()`**: returns the number of elements in the container

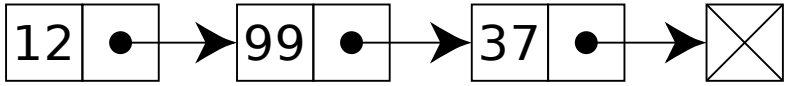
- **`boolean contains(`***`element`***`)`**: returns true if *element* $\in$ container; false otherwise
- **`int indexOf(`***`element`***`)`**: returns the index of *element*; -1 if not present
- **`boolean isEmpty()`**: returns true if container contains no element; false otherwise

- **`toArray()`**: returns an array with all the elements of the container

ArrayList vs LinkedList

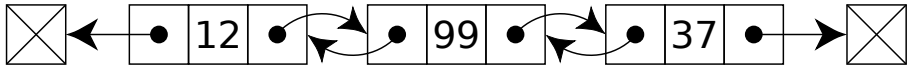
ArrayList

- list functionality by mapping to array
- implements **`interface List`**



LinkedList

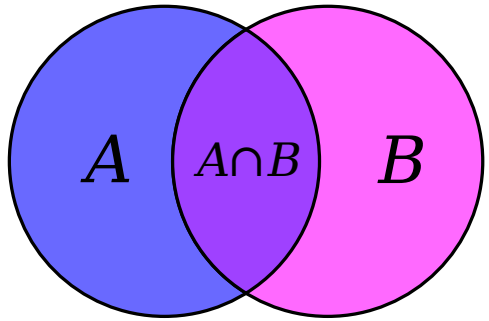
- double linked list
- with entries to the successor and predecessor
- helpful for adding and deleting elements at specific positions



linked lists are a central structure for data representation and handling in informatics,
they are however not often employed for storaged of scientific data and in numerical computing

collection of data: sets

`java.util.Set`



- mathematical set
- no double entries, `e1.equals(e2)` always false

HashSet

- implementation of `interface Set` using a hash algorithm

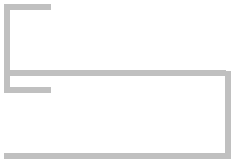
TreeSet

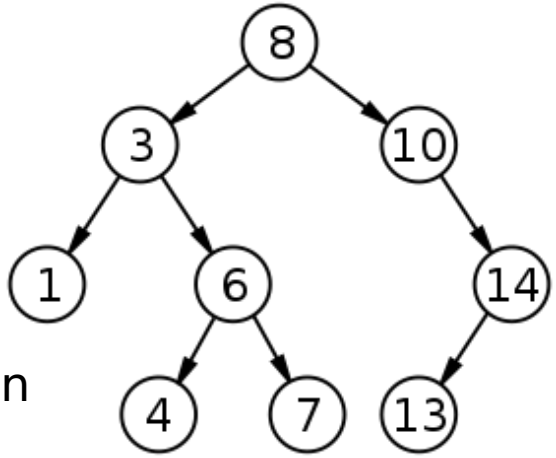
- implementation of `interface Set` using a sorted tree

LinkedHashSet

- implementation of `interface Set`
- remembers the order of elements

binary / ordered / sorted trees

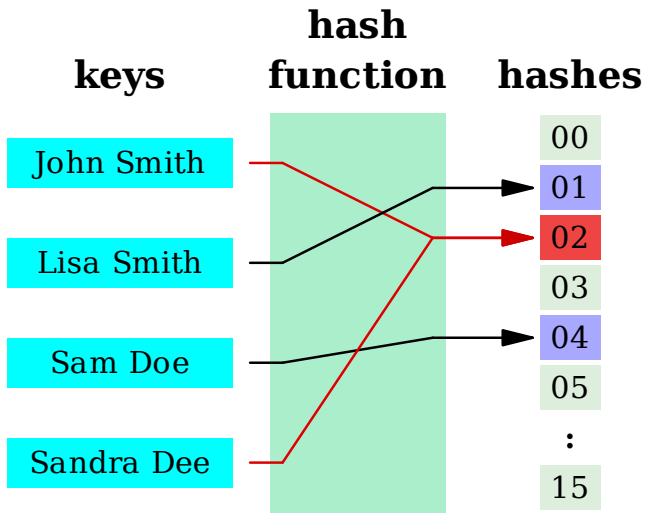
- 
- objects need to be **`Comparable`**
- fast access/search: $\log(size)$
- absolutely necessary for repeated manipulation of very large data sets
- **`TreeSet`** (only *keys*)
  `TreeMap` (map *key* $\rightarrow$ *value*)



hash table and functions

hash function

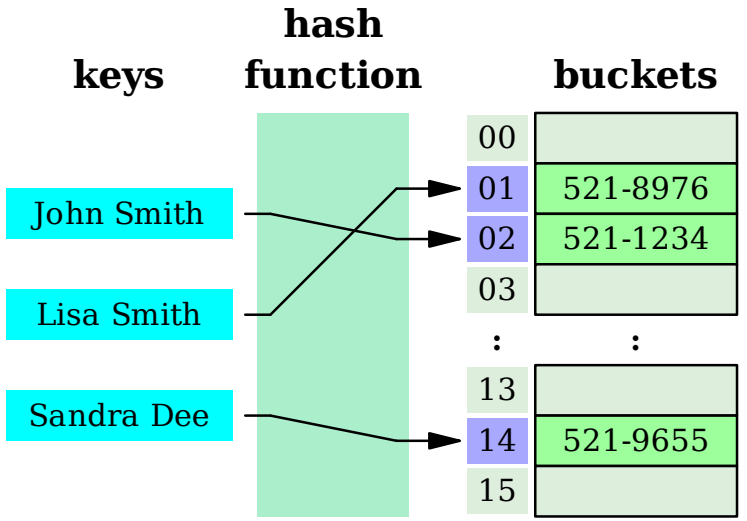
-
- integer: hash value
  hash code
  hash sum
  checksum
- password encription
  widely used alogrithm: md5
- identification of identical content
  objects
  entries
  data
- collision avoidance algorithm necessary



associative data storage: maps

`java.util.Map`

- associative memory
  map *key* $\longrightarrow$ *value*
- mathematical map/function
  combines a *key* with a corresponding *value*
  via intermediate *index* (hash value)
- dictionary, lexicon, telephone book
- no order of elements
- no implementation of **`interface Collection`**
  because of the coherence key – data



HashMap

- association **String** → **int[]**
  only in between objects,
  not in between primitive data types like **int**, **double**
- objects need to be instantiated

```
 1. import java.util.*;
 2.
 3. public class HashMapDemo {
 4.
 5. public static void main(String args[]) {
 6.
 7.   HashMap<String,int[]> hM = new HashMap<String,int[]>();
 8.      // only objects, no primitive data types like int, double
 9. // ***
10. // *** put objects to the HashMap (not add)
11. // ***
12.   hM.put("Lara",new int[2]);    // create new String, int[] objects
13.   hM.get("Lara")[0] = 18;       // set age of Lara
```

```
14.   hM.get("Lara")[1] = 168;      // set height of Lara
15.
16.   hM.put("John",new int[3]);
17.   hM.get("John")[0] = 19;       // set age of John
18.   hM.get("John")[1] = 178;      // set height of John
19.   hM.get("John")[2] = 81;       // set weigth of John
20.
21.   hM.put("Greg",new int[2]);
22.   hM.get("Greg")[0] = 17;       // set age of Greg
23.   hM.get("Greg")[1] = 188;      // set height of Greg
24.
25. // ***
26. // *** iterate over HashMap (ordering not retained)
27. // ***
28.   for (String name : hM.keySet())  // loop over all keys
29.     {
30.     System.out.printf("name: %5s, age: %3d years, height: %3d cm",
31.            name,hM.get(name)[0],hM.get(name)[1]);
32.     if (hM.get(name).length==3)   // check for number or data entries
33.       System.out.printf(", weight: %3d kg\n", hM.get(name)[2]);
34.     System.out.printf("\n");
35.     }
36. // ***
```

```
37. // *** various testing
38. // ***
39.   System.out.printf("      any Clara there? %b \n", hM.containsKey("Clara"));
40.   System.out.printf("how many folks arount? %d \n", hM.size());
41.   System.out.printf("trying to remove Jonas %b \n", hM.remove("Jonas"));
42.   System.out.printf("trying to remove Greg  %b \n", hM.remove("Greg"));
43.   System.out.printf("   all remaining folks %s \n", hM.keySet());
44.
45.   }  // end of HashMapDemo.main()
46. }    // end of HashMapDemo
```

Map member functions

- identical for **`HashMap`**
  `TreeMap`
  `...`
  just exchange definitions in program
- **`.put(`***`key,value`***`)`**: creates a new *key*-*value* entry to **`HashMap`**
  not **`.add()`** like for collections since position is not determined
- **`.get(`***`key`***`)`**: returns value-object for given *key*
- **`.remove(`***`key`***`)`**: removes value-object for given *key*

- **`.keySet()`**: set of all keys
- **`.values()`**: collection of all values
- **`.entrySet()`**: set of all *key*-*value* entries

- **`.containsKey(`***`key`***`)`**: checks for the presence of given *key*
- **`.containsValue(`***`value`***`)`**: checks for the presence of given *value*

HashMap of user defined objects

- **Integer**/**Double** objects
  for primitive data types, **int**/**double**;
  note capitalization
- needed for **HashMap<String,Integer>**
- attention, objects need to be instantiated!

```
 1. import java.util.*;
 2.
 3. public class HashMapMapDemo {
 4.
 5. public static void main(String args[]) {
 6.
 7.   HashMap<String,HashMap<String,Integer>> hMM =    // a HashMap of
 8.      new HashMap<String,HashMap<String,Integer>>();    // a HashMap
 9. // ***
10. // *** put objects to the HashMapMap
11. // ***
12.   hMM.put("Lara",new HashMap<String,Integer>());            // create key-value entry
```

```
13.
14.   hMM.get("Lara").put("age",new Integer(18));          // create new key-value entry
15.                                                       // for the age of Lara
16.   hMM.get("Lara").put("height",new Integer(168));     // create new key-value entry
17.                                                       // for the height of Lara
18.
19.   hMM.put("John",new HashMap<String,Integer>()); // now its John's turn
20.   hMM.get("John").put("age",new Integer(19));
21.   hMM.get("John").put("height",new Integer(178));
22.   hMM.get("John").put("weight",new Integer(81));
23.
24.   HashMap<String,Integer> strInt = new HashMap<String,Integer>();
25.   strInt.put("age",new Integer(17));                  // for Greg
26.   strInt.put("height",new Integer(188));              // for Greg
27.   hMM.put("Greg",strInt);
28.
29.   strInt.put("age",new Integer(33));                  // replaces
30.   strInt.put("height",new Integer(199));              // old values
31.   hMM.put("Anna",strInt);                             // what is happing here?
32.
33. // ***
34. // *** looping over keySet()
35. // ***
```

```
36.   HashMap<String,Integer> tempMap = new HashMap<String,Integer>();
37.   for (String name : hMM.keySet())                    // loop over all keys
38.     {
39.     tempMap = hMM.get(name);                          // just for convenience
40.     System.out.printf("name: %5s",name);
41.     if (tempMap.containsKey("age"))
42.       System.out.printf(", age: %3d",(int)tempMap.get("age"));
43.     if (tempMap.containsKey("height"))
44.       System.out.printf(", height: %3d",(int)tempMap.get("height"));
45.     if (tempMap.containsKey("weight"))
46.       System.out.printf(", weight: %3d",(int)tempMap.get("weight"));
47.     System.out.printf("\n");
48.     }
49.
50.   }  // end of HashMapMapDemo.main()
51. }    // end of HashMapMapDemo
```

exercise: maps

Write a program for manipulating user defined object with a **HashMap**.

- use your planet program and the **class Planet** for storing the properties of the planets
- when reading the planet data from file store now the data in an appropriate **HashMap**
- pass the **HashMap** with the planet data to a **static** member function of the **Planet** class where the printing of all **Planet** objects should be done.