UNIVERSITY OF TWENTE.

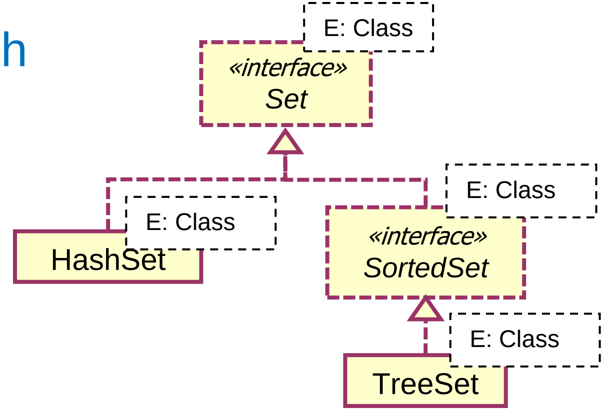
Set and Maps

Topic of Software Systems (TCS module 2)

Lecturer: Faizan Ahmed

COLLECTIONS: SET

- A `Set` has no duplicates, no indexing (`get`, `set`), no predetermined ordering
- `HashSet` is the preferred “default” implementation
- Hashing principle
  - Calculate pseudo-random numbers, called hash codes, for Objects that you want to store.
  - Store an object using its hash code
    - If you know the hash code, you can find the element in a small (almost) constant amount of time.
    - Unlike when you store Objects in a (long) List.
  - Purpose: **fast** way to find data



UNIVERSITY OF TWENTE.

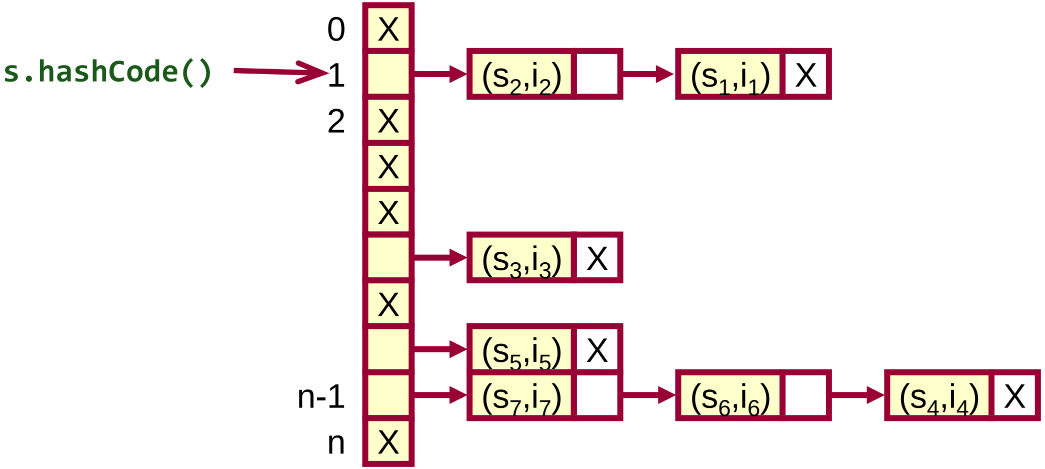
COLLECTIONS: HASHSETS

- Typical implementation as `HashSet`
  - Uses a (fixed) array of lists. Each of these is called a bucket.
  - Each bucket contains a linked list.
  - Method `hashCode` assigns elements into a bucket.
  - Finding a method looks up the method to find the bucket, then searches only the bucket.
  - Uses `equals` to compare elements in the same bucket

- Collision: two different objects have same hash code
  - Good hash function avoids too many collisions
  - Hash codes should be distinct in as many bits as possible

COLLECTIONS: HASHSETS

- Within a bucket
  - all hash codes are equal
  - but the objects are different when compared with `equals(Object)`



HASHCODE SUPPORT IN JAVA

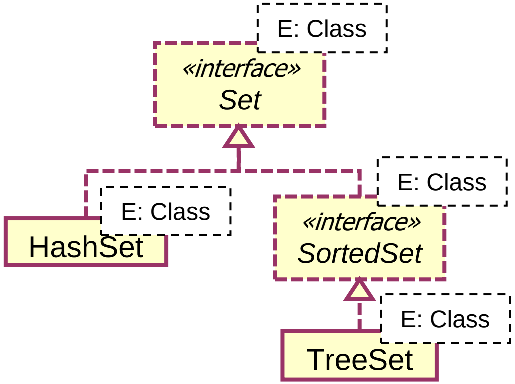
- Each class has a method `int hashCode()` returning an integer
  - In order to use `HashSet`, *you have to overwrite `hashCode()`*!
  - As well as `equals(Object)`
- Equality of objects **implies** equality of hash codes
  - If `o1.equals(o2)` returns **`true`**,
    then `o1.hashCode() == o2.hashCode()` ***must** hold.*
  - It is *your* responsibility to ensure this.
  - Otherwise `HashSet` methods like `add`, `contains,` will fail unpredictably
- The inverse typically does not hold, so:
  - *Distinct* objects may still have the *same* hash codes
  - Hence the need for linked lists in buckets

COLLECTIONS: SET

- A `Set` has no duplicates, no indexing (`get`, `set`), no predetermined ordering
- `TreeSet` is and alternative implementation
- It sorts the elements, using a binary tree
- The class of element has to implement `Comparable`
  - And the default method

```
  public int compareTo(E o)
```

  - It returns a positive number if the object is considered larger than **o**
  - It returns a negative number if the object is considered smaller than **o**
  - 0 otherwise, i.e. if they are considered the same.



UNIVERSITY OF TWENTE.

EXAMPLE: LIST OF STUDENTS

- Usage

Don’t forget the type parameter

```
public class Student implements Comparable<Student>{
  ...
}
```

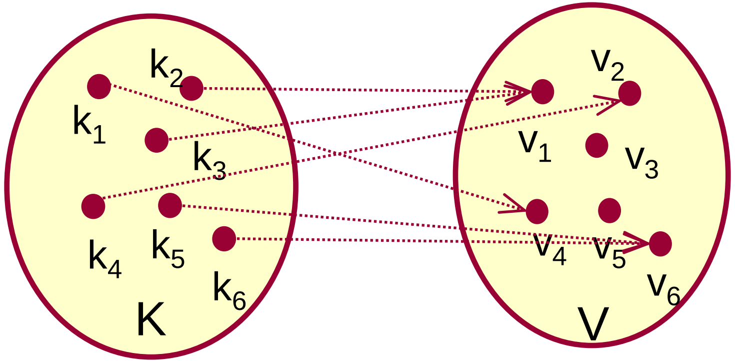
- Override `compareTo`

```
@Override
public int compareTo(Student o) {
    return nr.compareTo(o.getNr());
}
```

Sort the students by student number

MAPS

- Map based on mathematical concept of a function
  - Map: Keys → Values
  - For every key, there is at most one corresponding value



SOME `MAP<K,V>` OPERATIONS

- Basic methods:
  - `get(K key)`: returns the value associated with key, which could be **null**!
  - `put(K key, V value)`: associate key with value.
  - `remove(K key)`: removes the mapping for key.
- Boolean Testing methods:
  - `containsKey(K key)`: “true” if the key is used in the map.
  - `containsValue(V value)`: “true” if this value is associated with some key.
- From Maps to Sets:
  - `keySet()`: returns a **set** containing all keys used by this map
  - `values()`: returns all the values associated with at least one key in this map (returns a **`Collection`**)
  - `entrySet()`: returns a **`Set`** view of the mappings contained in this map
  - “Set **view**” means: *you* can simply use it like any “Set”, but the implementation “behind the scenes” might be rather “clever”.

EXAMPLE: STUDENTS GRADES

- Set up a map of student grades

```
Map<Student, Integer> grades = new HashMap<>();
// Provide some input
grades.put(new Student("s0124", "Adrian"), 7);
Student chris = new Student("s1102", "Chris");
grades.put(chris, 4);
//resit
grades.put(chris, 6);
if (grades.get(chris) >= 6) {
      chris.makeCertificate();
}
```

- There is only one student **chris**. The second **put** overwrites the first

EXAMPLE: STUDENTS GRADES

- Accessing parts of the map

```
Collection<Student> enrolled= grades.keySet();
for(Student s:enrolled){
    System.out.println(s.toString());
}

Collection <Integer> given= grades.values();
for(Integer g:given){
    System.out.println(g);
}

Set<Entry<Student, Integer>> a = grades.entrySet();
for(Entry<Student, Integer> e:a) {
           System.out.println(e.toString());
}
```

EXAMPLE: STUDENTS GRADES

- Exercise
  - Compute the average grade (method `double avg(Map<Student,Integer> grades)`)

```
double avg(Map<Student,Integer> grades) {
    int sum = 0;
        for (Integer c: grades.values()) {
            sum += c;
        }
    return (double)sum / grades.size();
}
```

JAVA COLLECTION SUMMARY

- `Collection`: general methods (`add`, `remove`, `contains`, `iterator`, …)
- `List`: see above (implementations: `ArrayList`, `LinkedList`)
- `Set`: no duplicates, no indexing (`get`, `set`), no predetermined ordering
  - `HashSet`: fast implementation based on hash codes
  - Requires element type to have overwritten `equals` and `hashcode`
- `SortedSet`: set with predetermined ordering (still no indexing)
  - Requires element type to be subtype of (interface) `Comparable`
  - `TreeSet`: `SortedSet` implementation based on binary trees
  - Slightly less efficient than `HashSet`
- `Map`: implements the mathematical concept of a function
  - `HashMap`: fast implementation based on hash codes
- `SortedMap`: map with fixed ordering, key type should be `Comparable`
  - `TreeMap`: `SortedMap` implementation based on binary trees