USAL1J: Java Collections

S. Rosmorduc

A simple collection: ArrayList

- A list, implemented as an Array
- ArrayList<Student> l= new ArrayList<Student>()
    - l.add(x): adds x at the end of the list
    - l.add(i,x): adds x in position i
    - l.remove(x): removes the first occurrence of s
    - l.remove(i): removes the element at position i
    - x= l.get(i): returns element at position i.
    - l.contains(x): returns true if l contains an element equal to x (more on this later)
    - l.size() : number of elements in x.

Excursus: hash tables

- We want to quickly store and retrieve data in a table (e.g. Strings)
- idea: an integer value is computed from the data. This value gives the index of the cell in the table where the data should be stored
- This integer value is called « hash function » or « hash code ».

Simplified example

- We want to represent a set of strings in an array of size1,000
- possible hash function: sum of Unicode character codes of the String

«le»

code of «l» = 108
+ code of «e» = 101
= 209

0
1
2
....
209
999

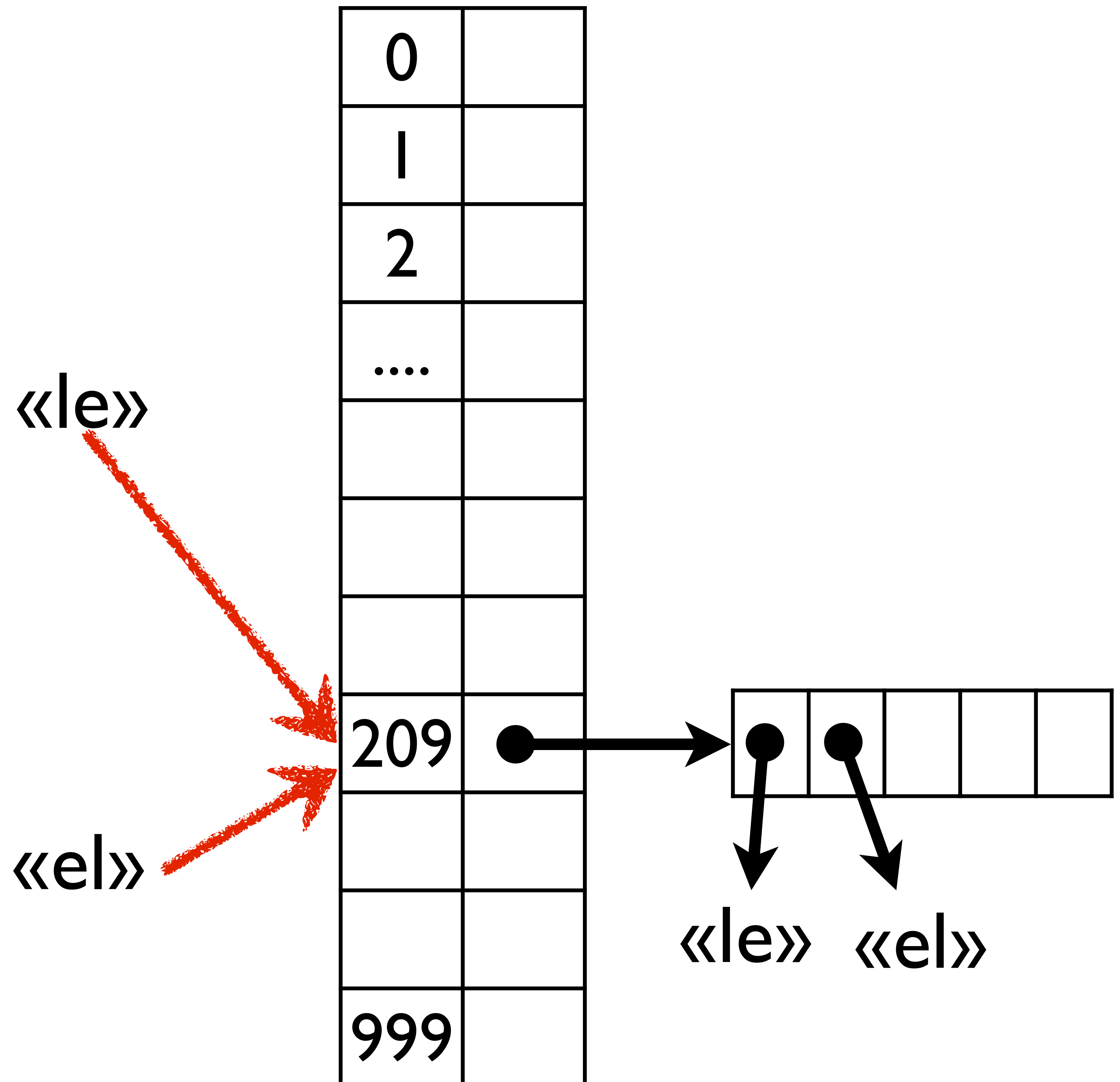
«le»

Problems

- What will happen if the hash function h is larger than the size s of the table:
- we take the modulo: (h % s)
- What happens if (when!) two strings have the same hash code ("le" and "el" in our example?)
- collision. Several possible solutions ...

Collision

Simplest solution: the cells contain lists of values, not the values directly



Search Algorithm

- let us search a value s in a hash table t.
- let *h* be the hash code of s
- let *i* be (h % t.length)
- s is supposed to be in cell *i*
- we search s in the list which is in cell number *i*, using equals to compare s and the list elements

Insertion algorithm

- let us add *s* to hash table t.
- let h be the hash code of s.
- let i be (h % t.length)
- we check if s is in the list in t[i]
- if not, we add s to the list in t[i]

Conclusion on hash codes

- for 100000 values in an array of size 1000 : mean length of sub-lists: 100
- interesting if the hash function is fair (the hash code values are « dispersed » on the whole range of integers).
- Object has hashCode() and equals() methods: hash tables can be used with all java types!
- In Object: hashCode depends only on the *address* of the object.

Back to collections…

Definition of equality

- lots of methods in collections need a definition of equality: contains(), remove(), add() for sets…
- Except for TreeSet and TreeMap, they use the equals() method

Equals

- For two objects a and b, a == b is true iff a and b have the **same address**
- The object class defines an equals method, which can be used to compare two objects:
  - boolean ok= a.equals(b)
- In the **Object** class, **a.equals(b) compares the addresses**; i.e. is equivalent to a == b
- Equals can be redefined in sub-classes. E.g. if s1 and s2 are Strings, s1.equals(s2) means s1 and s2 have the same **content**.

Redefining equals

- For a new class, the equals method is inherited from the parent class, usually Object
- So equals compares the addresses
- If one wants another result, one should redefine equals.
- When redefining equals, one **must** redefine also **hashCode**(), and ensure that:
    - if a.equals(b), then a.hashCode() == b.hashCode()
    - i.e. if two objects are equals, they have the same hashCode

Equals

```
public boolean equals(Object arg0)
```

```
public class MarkOnM
	private int x,y;
	String label;

	public MarkOnM
		super();
		this.x = x;
		this.y = y;
		this.label =
	}

	@Override
	public int hashC
		return x+
	}

	@Override
	public boolean e
		if (! arg0.g
			retu
		} else {
			Mark
			retu


		}
	}
}
```

- The signature of the equals method **always** takes an **Object** as argument.
- any other signature won't work correctly

Equals: example

```
public class MarkOnMap {
    private int x,y;
    String label;

    @Override
    public boolean equals(Object arg0) {
    if (arg0 instanceof MarkOnMap) {
            MarkOnMap other= (MarkOnMap) arg0;
            return this.x == other.x &&
                    this.y== other.y &&
                    this.label.equals(other.label);
        } else {
            return false;
        }
    }
```

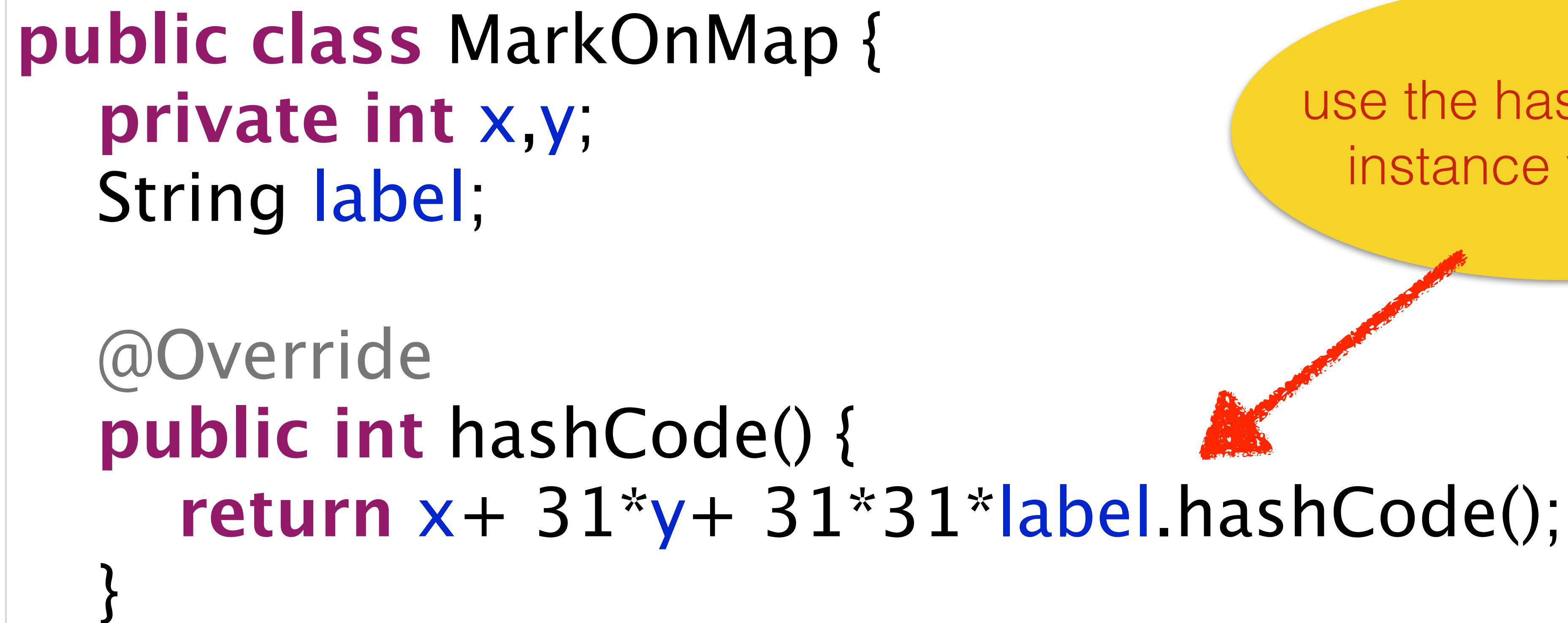
checks if arg0 has the right type

cast

compare the values of the relevant fields

returns false if type is wrong

Writing hashCode



- returns an int computed from some of the fields used in equals (to ensure a.equals(b) ⇒ a.hashCode()== b.hashCode()

- a good hashCode spreads the values over the whole set of integer

If an object is in a collection which uses hashCode:

- the values of the variables used for computing equals should not change
- because the identity of the object would change
- and the position of the object in the collection will be incorrect
- simple solution: don’t write setters for those variables

Entity semantics

- When a reality is represented **once** in memory
- some of its properties can change without changing its identity
- it is said to have *entity semantics.*
- Example: a Student object.
    - The address of the student can change without changing the student identity
    - it's probably a bad idea to have two objects for the same student in memory

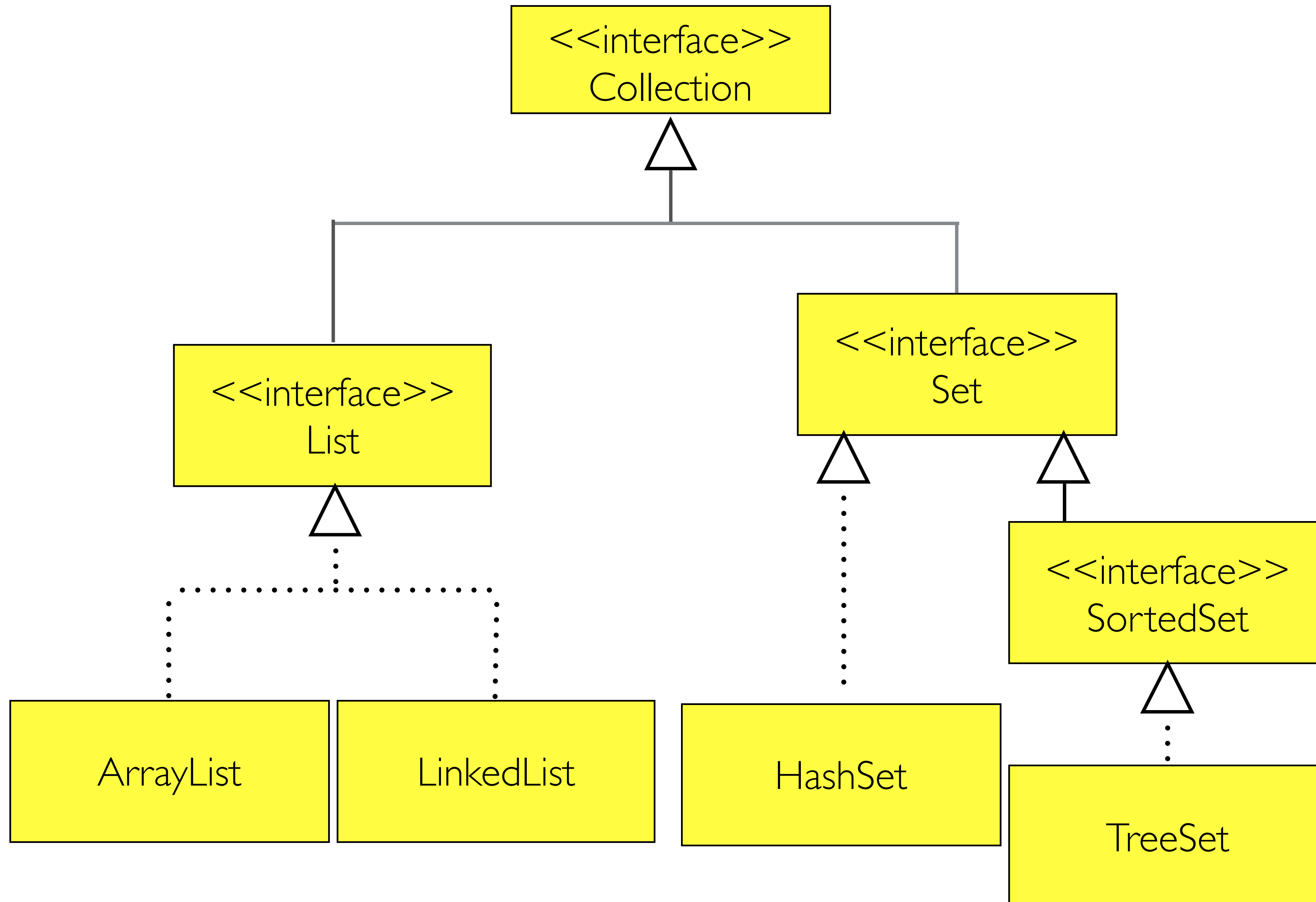
Value semantics

- A class whose objects represent *values*
- e.g. : String, Integer…
- typically immutable: no setters at all
- Two different objects in memory can represent the same value

Entity, Values and Equals

- Entity classes use address comparison
  - no need to redefine equals and hashCode !
  - the implementation from Object is o.k.
- Value classes use content comparison
  - redefine equals and hashCode
- Not all classes are like that. Example: collections themselves: mutable, but often with equals redefined.

Structure of the Collection Libraries



Collection

- Defines methods **common** to all collections
- The exact semantics of methods can be specified in sub-interfaces
- collections use **generic types** to describe the type of their element
- the element type is indicated between <…>
    - Collection<Student> s;
- The helper class Collection**s** contains static methods to manipulate collections

Collection operations

- c.add(x): adds x to the collection c
- c.remove(x): removes x from c
- c.contains(x): returns true if c contains element x (more on this later)
- c.size() : number of elements in s.
- c.toArray() : returns an array containing the elements of c.

Operations *between* collections

- Let a and b be Collection<T>
- a.addAll(b) ⇒a ← a ∪ b
- a.removeAll(b) ⇒a ← a - b
- a.retainAll(b) ⇒a ← a ∩ b
- a.containsAll(b) ⇒ b ⊆ A ?

Sets

- A Set is the representation a mathematical set.
- Important operations on sets are $\subset$,$\in$,=, $\cup$, $\cap$
- Two sets are equals iff they contain the same elements.
- Thus, if $x \in A$, $A \cup \{X\} = A$
- The methods of the Set interface are the same as the one of the Collection interface
- The semantics is the set semantics. e.g.:
    - s.add(x) has no effect if $x \in s$.

Implementations

- HashSet:
  - when the element order is not important.
  - works for all java types
  - uses hashCode() and equals()
- TreeSet:
  - sorted sets : compare the elements to sort them
  - only possible if there is an order on the elements

Lists

- In a list, elements can be repeated
- In a list, the position of an element can be explicitly manipulated through its index
- hence, it's possible to add a new element anywhere in the list (possible positions from 0 to l.size()).

List implementations

- ArrayList: usually used, based on arrays
  - Fast access to an element with its index.
  - Addition or removal of an element can be O(n)
  - … but amortized addition of n elements is also O(n), which is good
- LinkedList : based on linked lists
  - efficient addition and removal
  - slow access to an element from its index (O(n)).

Iterators

- Problem: how to browse a collection ?
    - for list: we have the index (but it's not efficient for LinkedLists)
    - for sets ???
- need to know the inner representation of the class ?
- no: the iterator encapsulates the needed information
- Collection has an c.iterator() methods which returns an iterator

Iterator

- creation: by the collection iterator() method
- test to know if there are still elements to see: **it.hasNext()**
- method to get the current value **and** advance the iterator: **it.next()**

```
Iterator<Student> it= c.iterator();
while (it.hasNext()) {
    Student s= it.next();
    …
}
```

Important note on iterators

- The iterator store data about its current position
- if the collection contents changes, the iterator may contain bad data
- if elements are added or removed to a collection, access to existing iterators will cause a ConcurrentModificationException

Iterators and modifications

```
List<Double> l= new ArrayList<Double>();
...
Iterator<Double> it= l.iterator();
while (it.hasNext()) {
  Double d= it.next();
  if (d > 100) {
    l.remove(d);
  }
}
```

Forbidden !
will cause an exception
no modification while using the iterator.

Solution 1: use the remove method of the iterator

```
List<Double> l= new List<Integer>();
...
Iterator<Double> it= l.iterator();
while (it.hasNext()) {
  Double d= it.next();
  if (d > 100) {
     it.remove();
  }
}
```

ok.
removes the element sent by next().
The iterator is aware of the removal

Solution 2: first look, then remove

```
List<Double> l= new List<Double>();
......
Set<Double> toRemove= new HashSet<Double>();
Iterator<Double> it= l.iterator();
while (it.hasNext()) {
    Double d= it.next();
    if (d > 100) {
        toRemove.add(d);
    }
}
l.removeAll(toRemove);
```

Iterable

- If a class implements Iterable (has an iterator()) method, then one can use the « for each » loop:

```
Set<Student> s= ....;
for (Student e: s) {

}
```

- Also works with arrays.

Collections and primitive types

- Collections are always collections of objects.
- Thus, it's not possible to create a collection of int, of boolean, of double…
- Instead, one must use the corresponding classes:

int	Integer
char	Character
double	Double
boolean	Boolean
long	Long
float	Float
byte	Byte

TreeSets

- Sets of ordered values, using a red-black tree
- elements are sorted along a predefined order, which depends on their values.
- Example: a TreeSet<Integer> will sort its values in increasing order.

```
TreeSet<Integer> s= new TreeSet<Integer>();
s.add(15); s.add(2);
s.add(9); s.add(3);
System.out.println(s);
```

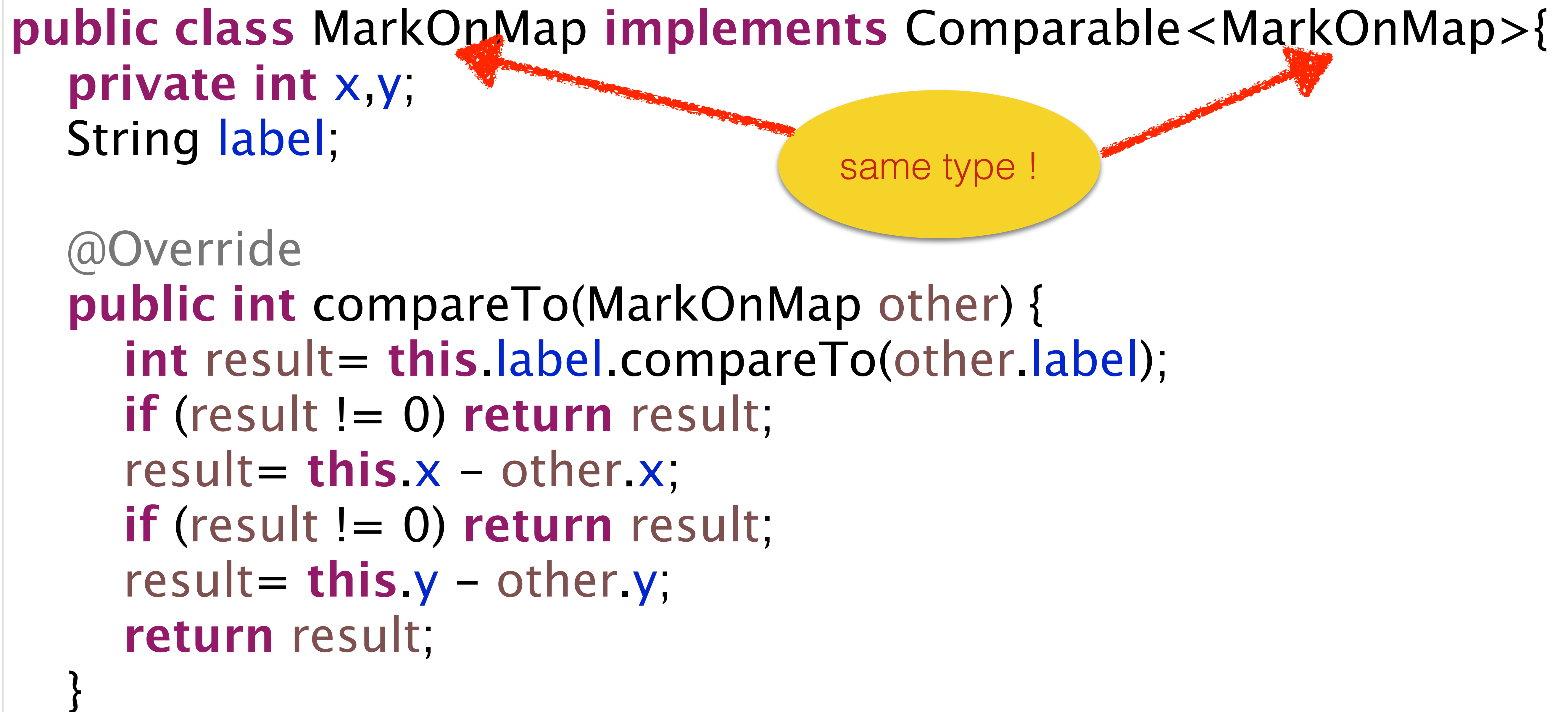
How to define this order?

- Two solutions:
  - in the class to sort itself
    - define a « natural order » for the elements of the class
    - implements the interface **Comparable**
    - should be compatible with **equals**
  - define an **external Comparator** class
    - can be used to sort objects of classes which don't implements Comparable
    - can be used for alternative orders.

Comparable

- defines : **public int** compareTo(MyClass other)
- compares **this** to **other**
- **this** and **other** have the same type
- must return an integer
  - negative if **this < other**
  - **0** if **this is equals to other**
  - positive if **this > other**
  - tip think of it as « **this minus other »**

Comparable : example



How to write compareTo ?

- Chose how to sort:
    - here: first by label,
    - then, by x
    - then, by y
- if comparison result is not 0, you can stop

```
public int compareTo(MarkOnMap other) {
        int result= this.label.compareTo(
                            other.label);
        if (result != 0) return result;
        result= this.x - other.x;
        if (result != 0) return result;
        result= this.y - other.y;
        return result;
}
```

Comparator

- to propose an other sort order.
- here: sort first by coordinates, then by label

```
public class ByPositionComparator
            implements Comparator<MarkOnMap>{

    @Override
    public int compare(MarkOnMap o1, MarkOnMap o2) {
        int result= o1.getX() - o2.getX();
        if (result != 0) return result;
        result= o2.getY() - o1.getY();
        if (result != 0) return result;
        result= o1.getLabel().compareTo(o2.getLabel());
        return result;
    }
}
```

Use of comparators

- The comparator is passed to the TreeSet constructor:

```
TreeSet<MarkOnMap> s=
    new TreeSet<MarkOnMap>(new ByPositionComparator());
```

Maps

- « Associative arrays »
- associate values to keys
- e.g. ***caches***, dictionaries

```java
TreeMap<String, String> dictionary= new TreeMap<String, String>();
dictionary.put("java", "Object oriented programming language");
dictionary.put("C", "Imperative language");
String def= dictionary.get("java");
```

Maps

- Two implementations: **TreeMap**, and **HashMap** (first uses **keys** orders, second uses **keys** hashCodes).
- Important operations:
  - map.put(k,v) : stores the value v for the key k, replacing possible existing value
  - v= map.get(k) : retrieves value for key k, or null if none
  - map.size() : number of entries

Maps and collections

- map.keySet() : returns the set of keys
- map.values() : returns the collection of values, with possible duplicates
- map.entrySet() : returns couples (key,value), using the class Map.Entry.

Browsing maps

- Using keySet (easy)

```
for (String k: dictionary.keySet()) {
   System.out.println(k + ":" + dictionary.get(k));
}
```

- Using entrySet (more efficient)

```
for (Map.Entry<String, String> e: dictionary.entrySet()) {
   System.out.println(e.getKey() + ":" + e.getValue());
}
```

A note about java 8

- define a new way to use collections: Stream

```
List<MarkOnMap> l = s.stream()
        .filter(x -> x.getLabel().startsWith("a"))
        .collect(Collectors.toList());
```

- provide an easy way to build comparators:

```
Comparator<MarkOnMap> c= Comparator
            .comparingInt(MarkOnMap::getX)
            .thenComparingInt(MarkOnMap::getY)
            .thenComparing(MarkOnMap::getLabel);
```