Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 5 – List, Set and Map

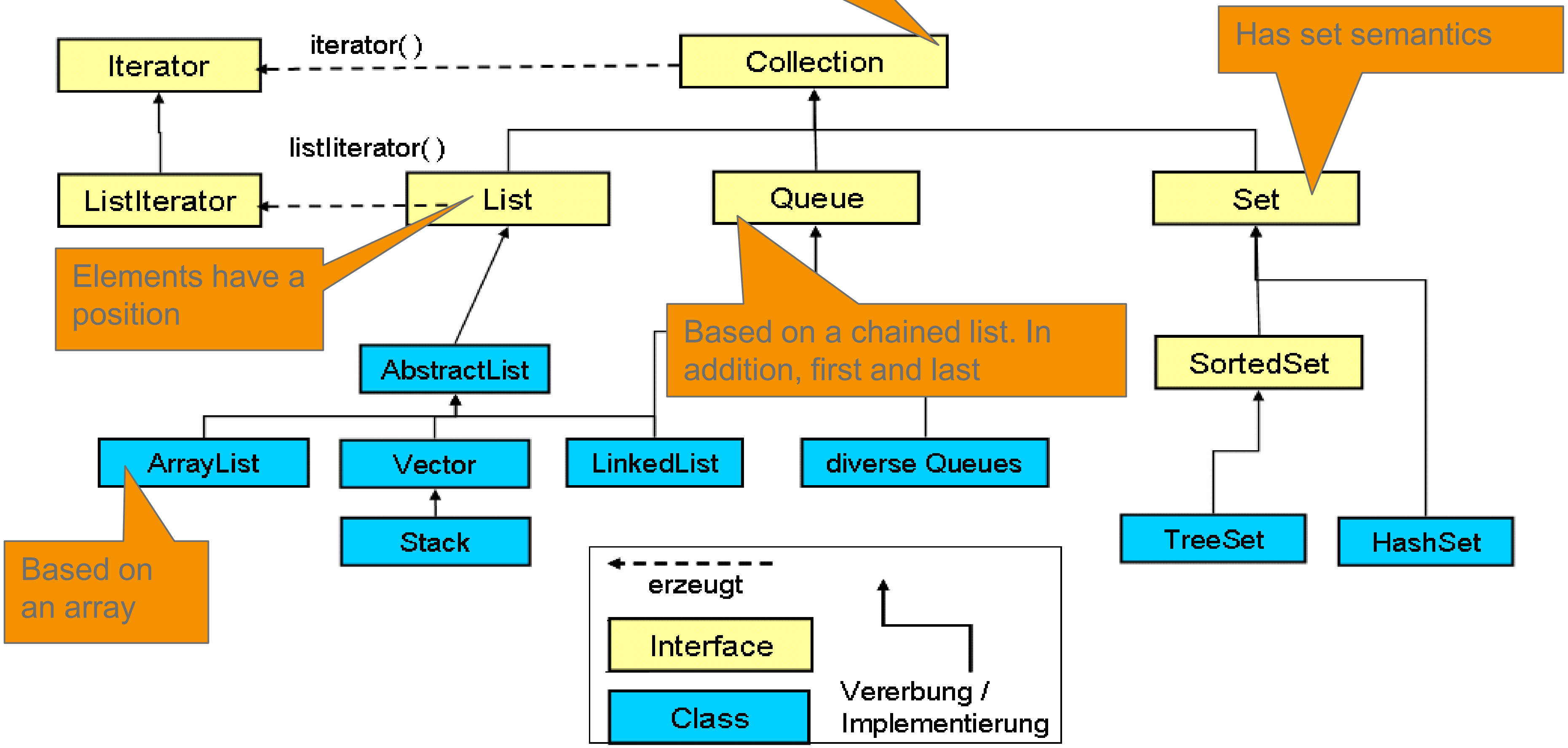
Prof. Dr Kai Höfig

Motivation

- **So far, we have**
  - Seen what the advantage is of developing implementations using an interface.
  - Seen own implementations of data structures in different variants:
    - using an array
    - using a block array
    - using a chained list
    - LIFO, FIFO stack
    - (binary) trees
  - Got to know type safe generic implementations of such data structures using generics
  - Extended the potential applications of generic implementations using `Object`, `Comparable`, `Comparator`, `Iterable` and `Iterator`
- **Now we will learn about the existing generic data structures of the Java Collection Framework**

Saving individual elements: `Collection`

Technische Hochschule **Rosenheim**
Technical University of Applied Sciences



Basic operations such as add and remove

Has set semantics

Elements have a position

Based on a chained list. In addition, first and last

Based on an array

https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Interface `Collection<E>` (1)

`boolean add(E e)`

Ensures that this collection contains the specified element (optional operation).

`boolean addAll(Collection<? extends E> c)`

Adds all of the elements in the specified collection to this collection (optional operation).

`void clear()`

Removes all of the elements from this collection (optional operation).

`boolean isEmpty()`

Returns true if this collection contains no elements.

`boolean contains(Object o)`

Returns true if this collection contains the specified element.

`boolean containsAll(Collection<?> c)`

Returns true if this collection contains all of the elements in the specified collection.

`boolean equals(Object o)`

Compares the specified object with this collection for equality.

`int hashCode()`

Returns the hash code value for this collection.

Interface `Collection<E>` (2)

`Iterator iterator()`

Returns an iterator over the elements in this collection

`boolean remove(Object o)`

Removes a single instance of the specified element from this collection, if it is present (optional operation).

`boolean removeAll(Collection<?> c)`

Removes all this collection's elements that are also contained in the specified collection (optional operation).

`boolean retainAll(Collection<?> c)`

Retains only the elements in this collection that are contained in the specified collection (optional operation).

`int size()`

Returns the number of elements in this collection .

`Object[] toArray()`

Returns an array containing all of the elements in this collection.

`T[] toArray(T[] a)`

Returns an array containing all of the elements in this collection whose runtime type is that of the specified array.

Revision: List and Set

- We know
  - List as a *sequential* container that can grow and shrink dynamically.
  - Set (or binary tree) as *a duplicate-free* container that stores unique elements.

```java
List<String> entries = new LinkedList<>();
while (true) {
    System.out.print(“Input: ");
    String line = br.readLine();

    if (line == null)
        break;

    entries.add(line);
}

System.out.println("\nInput: " + entries);
```

```java
Set<String> set = new TreeSet<>();
while (true) {
  System.out.print("Input: ");
    String line = br.readLine();

  if (line == null)
    break;

  if (set.contains(line)) {
    System.out.println("allocated.");
    continue;
    } else {
    entries.add(line);
    set.add(line);
    }
}
```

- *Always the right choice if the number of elements is not known at the time of development and/or if set semantics are required.*

Associative data field: Map

- For a key object *K*, the associative data field (map) stores exactly one value object *V (K -> V).* In Java, this data structure is defined as a generic interface:

```
interface Map<K, V> {
  void put(K key, V value);
  V get(K key);
  boolean containsKey(K key);
}
```

- What is noteworthy is that the map has two type variables:
  - **K** for the key type (key) (= like *Set*, stores each value exactly once)
  - **V** for the value type (value) (multiple times)

- Very common data structure in computer science, in order to quickly calculate the associated *value* for a *key*:

```
Map<Integer,String> pcodes = new TreeMap<Integer,String>();
pcodes.put(83101,"Thansau");
pcodes.put(83026,"Rosenheim");
pcodes.put(83022,"Rosenheim");
```

Interface `Map<K,V>` (1)

`Boolean                containsKey(Object key)`

Returns true if this map contains a mapping for the specified key.

`Boolean                containsValue(Object value)`

Returns true if this map maps one or more keys to the specified value.

`V                      get(Object key)`

If this map contains a mapping from a key k to a value v such that `Objects.equals(key, k)`, then this method returns v; otherwise it returns null.

`Boolean                isEmpty()`

Returns true if this map contains no key-value mappings.

`Set<K>                 keySet()`

Returns a Set view of the keys contained in this map.

`V                      put(K key, V value)`

Associates the specified value with the specified key in this map (optional operation).

`void                   putAll(Map<? extends K,? extends V> m)`

Copies all of the mappings from the specified map to this map (optional operation).

Interface `Map<K,V>` (2)

```
Set<Map.Entry<K,V>>  entrySet()
```
Returns a Set view of the mappings contained in this map.

```
V                 remove(Object key)
```
Removes the mapping for a key from this map if it is present (optional operation).

```
default boolean   remove(Object key, Object value)
```
Removes the entry for the specified key only if it is currently mapped to the specified value.

```
default V         replace(K key, V value)
```
Replaces the entry for the specified key only if it is currently mapped to some value.

```
default boolean   replace(K key, V oldValue, V newValue)
```
Replaces the entry for the specified key only if currently mapped to the specified value.

```
int               size()
```
Returns the number of key-value mappings in this map.

```
Collection<V>     values()
```
Returns a Collection view of the values contained in this map.

Example using supervisor

```java
Map<Person,Person> supervisor = new HashMap<>();

Person boss = new Person("Maria");
Person a1 = new Person("Klaus");
Person a2 = new Person("Kathrin");
Person al = new Person("Michael");

supervisor.put(boss,null);
supervisor.put(al,chef);
supervisor.put(a1,al);
supervisor.put(a2,al);

System.out.println(supervisor);

System.out.println(supervisor.get(new Person("Michael")));
```

```
{Chapter05.Person@17c68925=Chapter05.Person@19dfb72a, Chapter05.Person@19dfb72a=null, Chapter05.Person@7e0ea639=Chapter05.Person@17c68925, Chapter05.Person@3d24753a=Chapter05.Person@17c68925}
```

```
{Michael=Maria, Maria=null, Klaus=Michael, Kathrin=Michael}
```

With overridden `toString` method

```
null
```

```
Maria
```

If `equals` and `hashCode` are implemented correctly

Object equality using `equals`

- Check equality using `equals`
- In Java, two objects can be compared for content equality using equals:

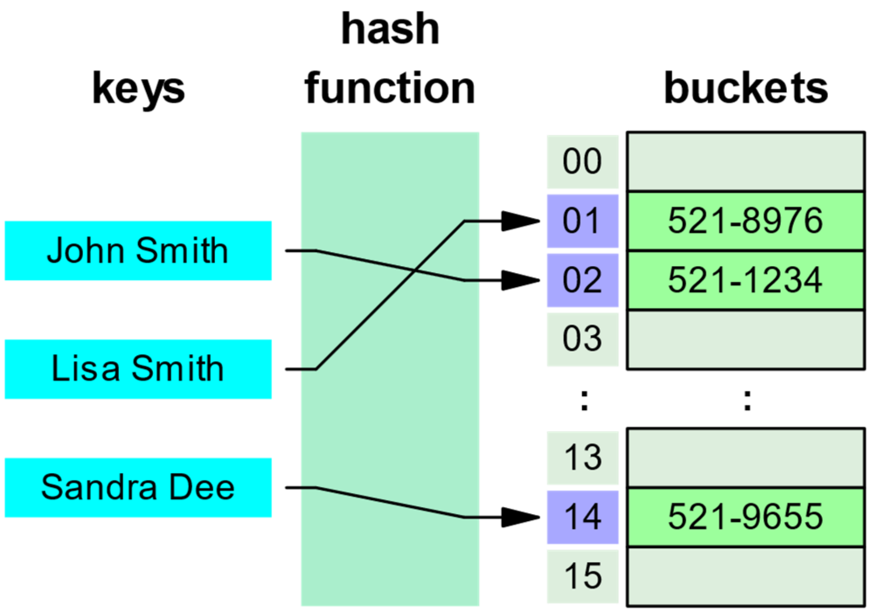
```java
public class MyClass {
  int attribute;
  public boolean equals(Object o) {
    // 1. The same object?
    if (o == this)
      return true;
    // 2. Does the class match?
    if (!(o instanceof MyClass))
      return false;
    // convert...
    MyClass other = (MyClass) o;
    // 3. Compare attributes
    if (this.attribute != other.attribute)
      return false;
    return true;
    }
}
```

Digression: Efficiency through hashing

- Use array because of quick direct access
- Hash function to get from key to "drawer"
- Object.hashCode
  https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html#hashCode()
- Defined for all API classes (String, Double, etc.)
- For your own classes: implement hashCode:

```
public class MyClass implements Comparable<MyClass> {
  int attribute;
    String s;
…

  public int hashCode() {
    return s.hashCode() + attribute;  // for example...

     }
}
```

- Characteristics of a good hash function
  - Distributes all possible inputs evenly across buckets
  - Easy to calculate



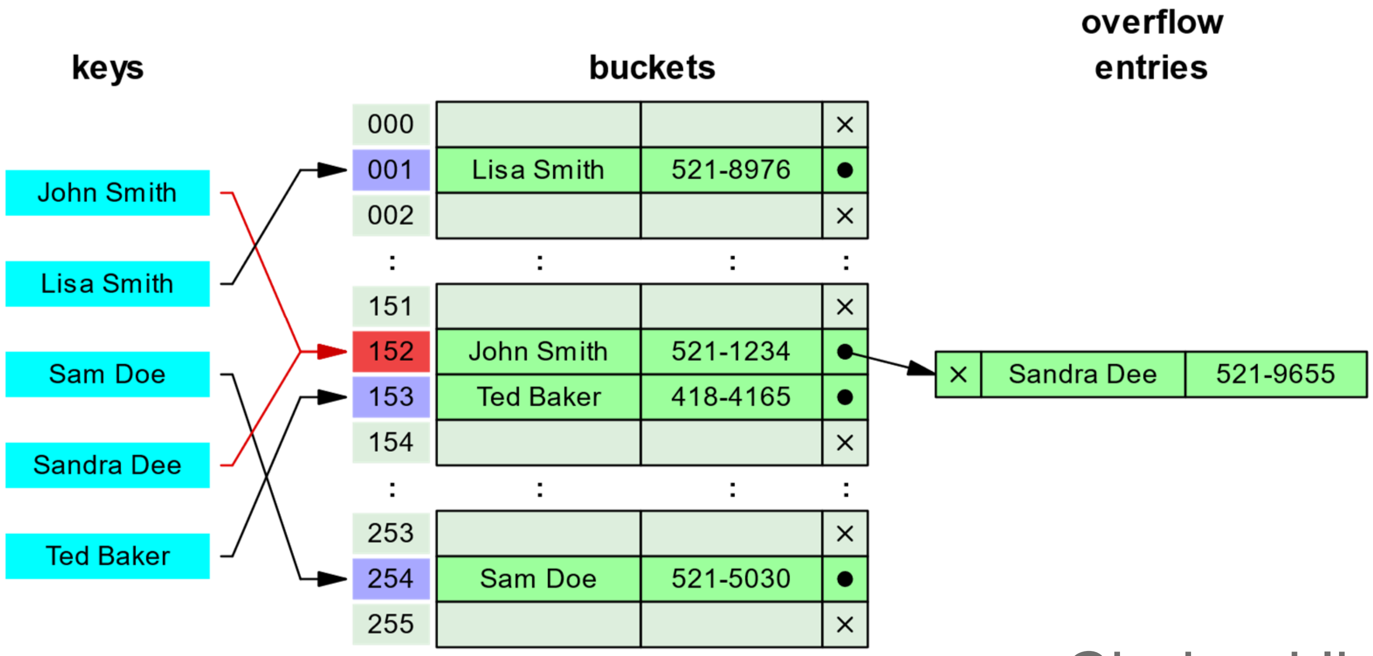
Hash function from open source software library

- In practice: Using the open source software library
- e.g. `org.apache.commons.lang3.builder.HashCodeBuilder`

```
import org.apache.commons.lang3.builder.HashCodeBuilder;

public class MyClass implements Comparable<MyClass> {
  int attribute;
    String s;
…
  public int hashCode() {
    // choose any two odd numbers
    HashCodeBuilder b = new HashCodeBuilder(17, 19);

    // append all important elements
    b.append(attribute).append(s);

    return b.hashCode();
    }
}
```

Collisions

- Indexes from hash can *collide*
- Use a list for a bucket instead of individual elements



keys

John Smith

Lisa Smith

Sam Doe

Sandra Dee

Ted Baker

buckets

000			×
001	Lisa Smith	521-8976	●
002			×
:	:	:	:
151			×
152	John Smith	521-1234	●
153	Ted Baker	418-4165	●
154			×
:	:	:	:
253			×
254	Sam Doe	521-5030	●
255			×

overflow entries

×	Sandra Dee	521-9655

Chained list

Key Value

Implementing a HashMap using an array index (outlined)

- How do we now represent a hash value of a key in an array index? After all, the value range of `int` is quite large -- on the one hand, the storage would not be adequate for such a large array, and on the other hand, it would probably then be mostly empty.

- We solve this in a different way: First, we select array sizes that represent a power of two, e.g. 4, 8, 32, 256, etc. There are thus corresponding array indices of 0..3, 0..8, 0..31, etc. A mathematical trick (and knowledge of the binary system) are also helpful here: The index in the array is calculated as `(array.length - 1) & key.hashCode()`, where & is the bitwise AND operator.

- An example: An object has the hashCode of 42, the array has a length of 16:

- If collisions occur, one `List` can be used per array entry.

```
Person[] array = new Person[16];
Person p = new Person("Klaus");
int hash = p.hashCode();  // z.B. 42


int index = (array.length - 1) & hash;


// array.length - 1 in binary:  001111
// hash             in binary:  101010
// 15 & 42       bitwise AND:  001010
//                      ...between 0..16


System.out.println(index);
```

Example HashMap

Person Klaus
Name=“Klaus“
Einkommen=200
hashCode=72578462
hashCode als Binärzahl=100010100110111010110011110

`buckets.length()-1` als Binärzahl **11**

`buckets.length()-1 & „Klaus“.hashCode` als Binärzahl **10**

→ *Klaus kommt in Bucket No.2*

Generic Array creation

```
// Hinzufügen von Klaus
buckets[2].add(new Entry<String,Integer>("Klaus",200));
```

```
// Array von Generics
@SuppressWarnings("unchecked")
List<Entry<String, Integer>>[] buckets = new List[4];
for(int i=0;i<4;i++)
    buckets[i]=new ListImpl<Entry<String, Integer>>();
```

0 *null*

1 *null*

2 (Maria,400)→(Klaus,200)→*null*

3 *null*

Implementation using `TreeMap`

- A hash map is therefore another data structure with both advantages and disadvantages.
  - Buckets can be found quickly via hashing
  - Adding elements is easy
  - Finding elements is easier, if the hashing works well
  - Deleting is easy (find and skip)

- Implementation using TreeMap is also possible.
  - Finding elements is even better

```
  class Entry<K extends Comparable<K>, V> implements Map.Entry<K, V>, Comparable<Entry<K, V>> {
  K key;
  V value;

    Entry<K, V> left, right;


…
  }
```

- *Why not create buckets with trees instead of lists?*

For a hash map we required `hashCode` and `equals` to insert and find. Now we also need `compareTo` so that we can compare for our tree.

Object comparison using `compareTo`

- For use in a tree map, an entry must be comparable via its key.

```java
public class MyClass implements Comparable<MyClass> {
  int attribute;
…
  public int compareTo(MyClass other) {
    // 0 for equality, negative if smaller than other
    if (this.attribute == other.attribute)
      return 0;
    else if (this.attribute < other.attribute)
      return -1;
    else
            return 1;

    // `alternative very much shorter:`
        // return this.attribute - other.attribute;
  }
}
```

Otherwise once again as usual, only this time as key/value

No root, the tree is empty

The key already exists, we overwrite the value

The key to be inserted is "smaller" than the current node

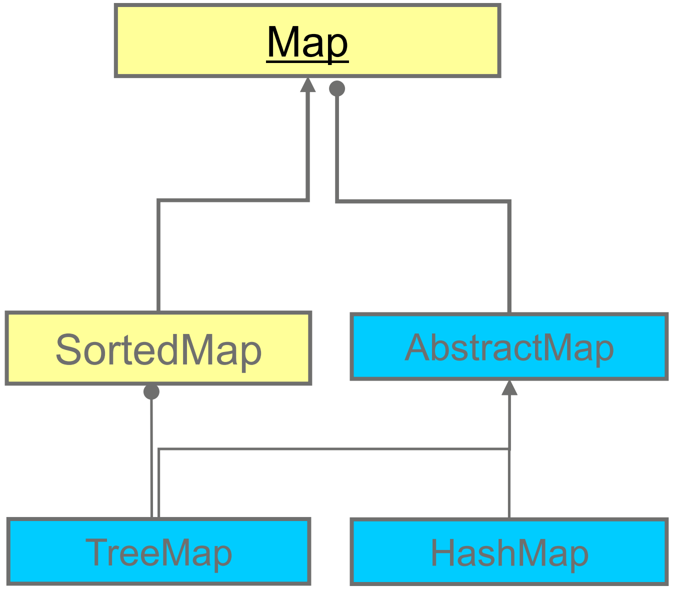
→ insert

→ or go left

otherwise go right

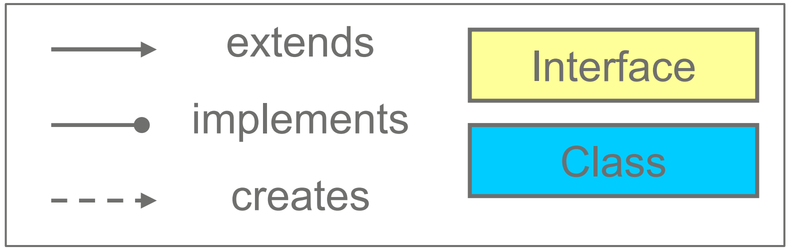
```
@Override
public void put(K key, V value) {
  if (root == null) {
    root = new Entry<>(key, value);
    return;
    }

    Entry<K, V> it = root;
  while (it != null) {
    int c = it.key.compareTo(key);
    if (c == 0) {
      // update!
      it.value = value;
      return;
        } else if (c < 0) {
      if (it.left == null) {
        it.left = new Entry<>(key, value);
        return;
            } else {
                it = it.left;
            }
        } else {
      if (it.right == null) {
        it.right = new Entry<>(key, value);
        return;
            } else {
                it = it.right;
            }
        }
    }
}
```

Associative memory: `Interface <Map>`



Implementation using a binary tree

Implementation using a hash process



Lists and sets store elements, whereas associative memory stores (key value) pairs!

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Java Collection Framework

- `List` as sequential data structure
- `Set` as duplicate-free (unordered) data structure
- `List` and `Set` extend `Collection`, which in turn extends `Iterable`, i.e. all are iterable via `Iterator`
- `Map` as associative data structure
- Realisations in the Java API:
  - `ArrayList` and `LinkedList`
  - `TreeSet` and `HashSet`
  - `TreeMap` and `HashMap`

Summary

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- *The fundamental data structures in computer science are*
  - **`List`** is a sequential data structure, realised for example as **`ArrayList`** or **`LinkedList`**
  - **`Set`** is a duplicate-free (unordered) set, realised for example as **`HashSet`** or **`TreeSet`**
  - **`Map`** is an associative data structure that maps keys to values, realised for example as **`HashMap`** or **`TreeMap`**

- *When programming:*
  - define variables as interfaces
  - initialise the variables from classes of the Java API
  - e.g. **`Set<String> s = new TreeSet<>()`**
  - avoid using raw types (unparameterised generic classes), so for example always use **`List<...>`** instead of **`List`**.

- *If data structures with their own classes are used, then it’s essential that*
- **`equals`** is implemented to check for value equality
- **`hashCode`** is implemented, if hashing is used
- **`Comparable<T>`** is implemented, if objects should be comparable.
- Collections are **`Iterable`**, so we can use them in **`for-each`** loops, or receive an **`Iterator`** for traversal.