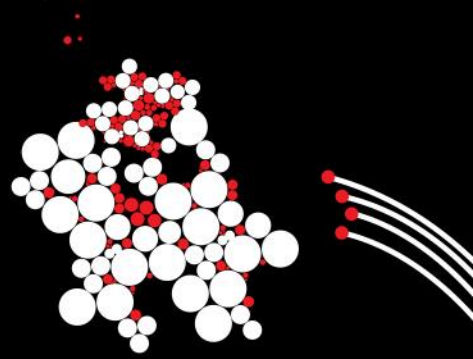
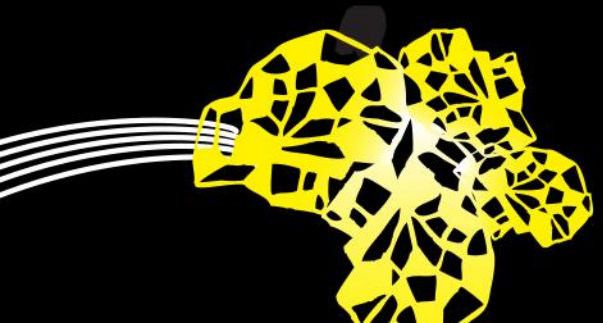
UNIVERSITY OF TWENTE.



Java Collections

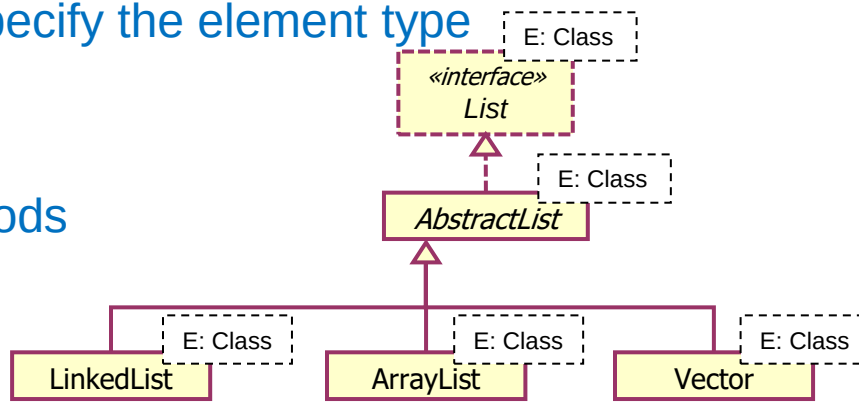
Topic of Software Systems (TCS module 2)

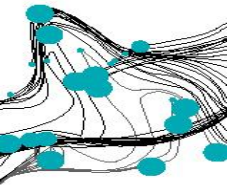
Lecturer: Faizan Ahmed



JAVA STANDARD INTERFACE: JAVA.UTIL.LIST

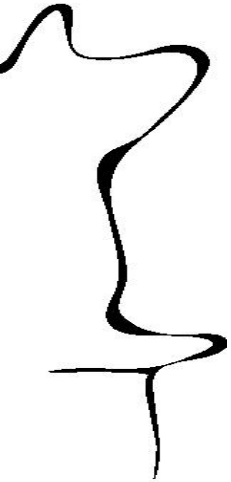
- A list implements the mathematical concept of a *sequence*
  - Elements of the list have an *index* and are therefore *ordered*
- Lists are *generic*
  - The type of the elements in a list does not make a difference
  - When using a List, you have to specify the element type
- Class hierarchy (incomplete, there is more!)
  - `List` interface: no functionality
  - `AbstractList`: some basic methods
  - Implementations:
    - `ArrayList`: efficient indexing
    - `LinkedList`: efficient addition & removal





EXAMPLE: LIST OF STUDENTS

- A student is identified by a name and a student number



```
public class Student {
    private final String nr;
    private final String name;

    public Student(String nr, String name) {
        …
    }

    // getters and setters
}
```

Number	Name
s0123	Mary
m0246	John
s1345	Kim

USING LISTS (AND OTHER COLLECTIONS)

- When declaring a variable
  - Declared type: as abstract as possible
  - Instantiated type: choose appropriate concrete implementation
  - This improves maintainability
- Example:

abstract declaration

concrete implementation

```
List<Student> slist = new ArrayList<>();
…
slist.remove(…);
```

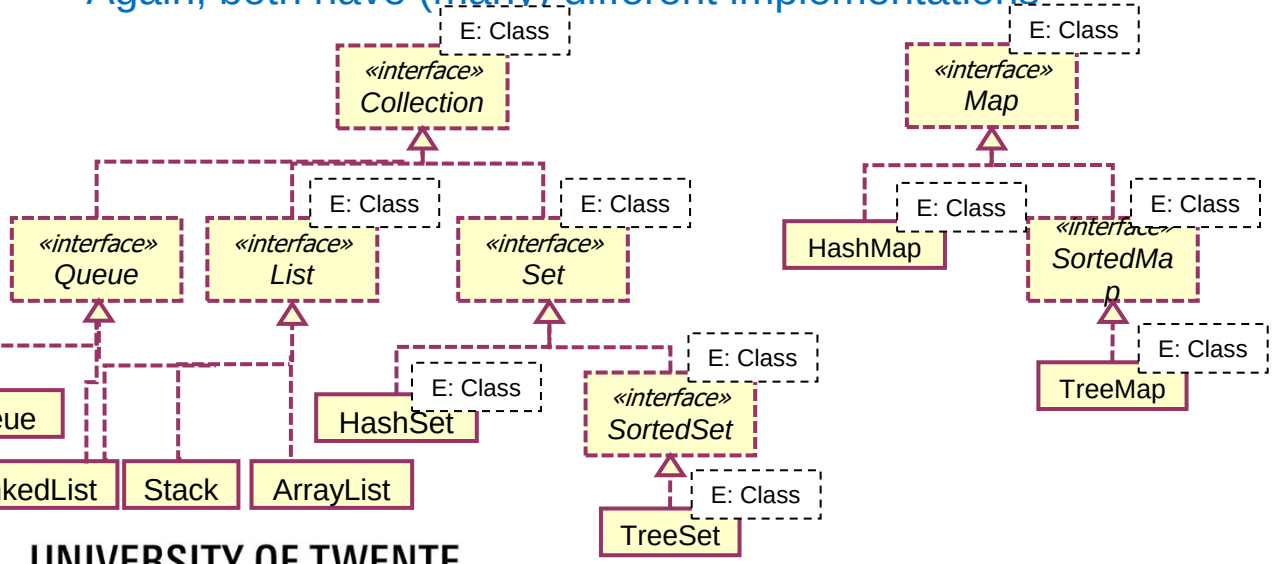
  - `remove` is inefficient for `ArrayList`
  - easily changed into

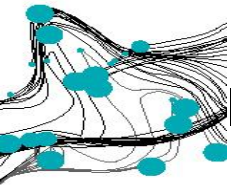
Only change required!

```
List<Student> slist = new LinkedList<>();
…
slist.remove(…);
```

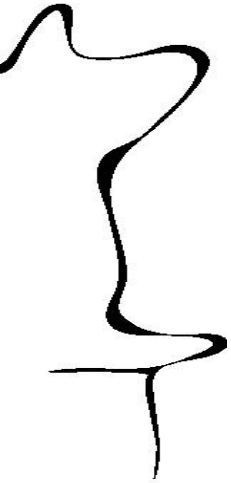
JAVA COLLECTION HIERARCHY

- Besides `List`, there are other fundamental data structures
  - `Set` implements the mathematical concept of a set (surprise…)
  - `Map` implements the mathematical concept of a function
  - Again, both have (many) different implementations





ITERATOR



- Each collection comes with an iterator
  - `Collection<E>` has a method `iterator()`
  - returns an object of type `Iterator<E>`
- Key iterator methods:
  - **`boolean`** `hasNext()`: returns true if the iteration has more elements
  - `E next()`: returns the next element in the iteration
  - **`void`** `remove()`:
    - removes from underlying collection last element returned by this iterator
    - can be called only once per call to `next()`

ITERATOR TYPICAL USAGE PATTERN

- Iterate through a list and print the context

```
Iterator<Student> i  = scoll.iterator();
while (i.hasNext()) {
  Student s = i.next();
  System.out.printf("Nr: %s, name: %s%n", s.getNr(), s.getName());
}
```

- Alternative: Use a "**for**" loop:

```
for (Student s: scoll) {
  System.out.printf("Nr: %s, name: %s%n", s.getNr(), s.getName());
}
```

- This is the preferred alternative, if possible

ITERATOR TYPICAL USAGE PATTERN

- Sometime you need the iterator. For example, when the collection is manipulated.

```
public static void removeInvalidNr(Collection<Student> scoll)
{
      Iterator<Student> i = scoll.iterator();
      while (i.hasNext()) {
            if (!i.next().getNr().matches("s[0-9]*")) {
                  i.remove();
            }
      }
}
```

regex

Do *not* use scoll.remove(i)!

Behaviour of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling remove