1 Java 8 features
2 Streams
3 Parallel streams

PARALLEL STREAM PROGRAMMING

① JAVA 8 FEATURES

* OOP ↔ FUNCTIONAL PROGRAMMING

evolves around objects and instances, a method cannot exist outside an object

able to define functions, give them reference variables, pass them as method arguments, ..

* lambda expressions

```
(Car volvo) → {
    S.O.P(volvo.getName());
    RETURN volvo.getName();
}
```

* event listeners

→ via anonymous classes

```
private JButton button = new JButton("Click me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        S.O.P. ("__");
    }
});
```

→ via lambda expressions

```
button.addActionListener(e →
    S.O.P. ("__"));
```

of

```
button.addActionListener(e → {
    S.O.P ("__");
    S.O.P ("__");
});
```

* functional interfaces

= single abstract method interface

= an interface with 1 single abstract method

e.g. Runnable, Comparator, Callable, ...

Java 8 annotation @FunctionalInterface

```
@FunctionalInterface
public interface Converter<F, T> {
    T convert(F from);
}

Converter<String, String> capitalizer = (from) -> from.toUpperCase();
String toUp = capitalizer.convert("hanne");
s.o.p(toUp);          // "HANNE"
```

built-in:

public interface Function<T, R>	→	R apply(T)
public interface Predicate<T>	→	boolean test(T)
public interface Supplier<T>	→	T get()
public interface Consumer<T>	→	void accept(T)

* default methods

enable us to add new functionality to interfaces without breaking the classes that implement the interface

e.g: forEach in interface Iterable<>

default void forEach(Consumer<? super T> action)

* closures

= the fact, for a function, to be able to access something in the enclosing context

```
private Function<Integer, Integer> calculate(Integer a) {
    return t -> t * a + b;
}
```

* optionals = container for a value which may be null or non-null ⇒ NullPointerExc

```
Optional<String> opt = Optional.of("Hanne");
```

② STREAM PROGRAMMING

e.g.
```
List<Integer> transactionIds =
    transactions.filter(t -> t.getType() == Transaction.ZALANDO)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

* streams:
  - are intermediate or terminal
  - executed sequential or parallel
  - no side-effects, static (deterministic)
  - accept kind of lambda expr.
  - processed lazily - on demand ⇒ do not store elements ! computed on-demand

* streams vs collections (data structures):

  both provide interfaces to a sequence of elements BUT

  ⚠ collections are about data, streams are about functions / computations

  streams let you process data in a declarative way: express what you expect, not how to calculate it

  collections use external iteration (forEach)
  streams use internal iteration

  streams do multi-core architectures with no need of multithread code

  streams are created on a source:
  - collection.stream()
  - Stream.of()
  - Files.lines()
  - intStream, longStream, doubleStream

* printing streams of strings:

```
namesStream.forEach(System.out::print);
```

* streams from file

```
Stream<String> namesFromFile = Files.lines(Paths.get("file.txt"), Charset.defaultCharset());
```

* numeric streams

→ `List<Integer> numbers = ...`
`numbers.stream();`

→ `IntStream numbers = IntStream.range(1, 8);`
`" numbers2 = IntStream.rangeClosed(1, 8);`

→ `DoubleStream doubles = DoubleStream.of(list, doubles);`

* filtering a stream

→ filter (predicate)
→ distinct
→ limit (n)
→ skip (n)

* stream operations

- intermediate → return a stream LAZY
- terminal → return any non-stream type

* Comparator on the fly

```
names.stream()
        .sorted((Name n1, Name n2) → {
            S.O.pf("sort: ...", n1.name, n2.name);
            return n1.name.compareTo(n2.name);
        })
        .forEach(System.out::print);
```

* map, collect

map : converts

collect : transforms into another datatype (list, set or map): `.collect(Collectors.toList())`

* Stream supplier

for every terminal operation you have to execute, you have to build a new stream ⇒ create a stream supplier

```
Supplier < Stream < Stream < Name >>> manyNamesMaker = () →
        Stream. of (names. stream (), moreNames. stream (), againMoreNames. stream ());
Stream < Stream < Name >> manyNames = manyNamesMaker. get ();
```

* flatmap

can convert a stream of streams to a flat stream

```
List < String > allNames = manyNamesMaker. get ()
                          · flatMap (s → s.map (m → m.name))
                          · collect (Collectors. toList ());
```

* infinite streams

- `static <T> Stream <T> iterate (T seed, UnaryOperator <T> f)`
- `static <T> Stream < T> generate (Supplier <T> s)`

e.g. all numbers : `Stream < Integer > numbers = Stream. iterate (0, n → n+1);`

e.g. Randoms : `Stream< Double > randoms = Stream. generate (new Random():: nextDouble);`

of DoubleStream `randoms 2 = DoubleStream. generate (() → Math. Random())`
`· map (d → Map. Round (d * 100)/100.0)`
`· forEach (System. out :: print);`

③ PARALLEL STREAMS

* parallel streams

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
s.o.p(commonPool.getParallelism());
```

Java runtime partitions the stream into substreams
↝ aggregate functions iterate over the substreams & combine the results

```
String elements = Arrays.asList("Harre", "Mats", "Lotte")
        .parallelStream()
        .map(...)
        ...
```

DATASTRUCTUREN & ZOEKALGORITMES

Datastructuren : Collections framework van Java

- Set
- List
- Map
- Stack & Queue

Zoekalgoritmes

- grootteordebepalingen
- insertion sort
- bubble sort
- merge sort
- quick sort

1

Collectie / Datastructuur = interface + implementatie

doorlopen ve. collectie :
* aggregate functies
* foreach
* iterators

1. aggregate functies

```
myCollection.stream()
    .filter(e → e.getColor() = Color.Red)
    .forEach(e → System.out.println(e.getName());
```

OF

```
myCollection.parallelstream()
    .filter(e → e.getColor() = Color.Red)
    .forEach(e → System.out.println(e.getName());
```

2. foreach

```
for (Object o : collection)
    System.out.println(o);
```

3. iterators

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();      // optional
}
```

↳ gebruiken wanneer je huidige element wil verwijderen (gaat niet met foreach) of als je parallel over meerdere collecties wil itereren

voorbeeld

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();
        if (!cond(it.next()))
            it.remove();
}
```

INTERMEZZO : GENERICS

Generics laten toe om types (= klassen & interfaces) te gebruiken als parameter bij het definiëren van klassen, interfaces, methoden.

⇒ zelfde code hergebruiken met andere input

input = types ! geen waarden

⊕ voordelen

- typecontrole tijdens compileren
- casts

```
List namenlijst = new ARRAYLIST ().
namenlijst.Add ("Hanne");
String naam = (String) namenlijst.get (0);
```

↔

```
List < String > namenlijst = new ARRAYLIST < String > ();

String naam = list.get (0);        ↳ geen cast.
```

Generic type = generic klasse of interface met (types) als parameters

```
public class Box {
    private Object obj ,
    public void set (Object obj) {
        this obj = obj ;
    }
    public Object get () {
        Return obj ;
    }
}
```

↳ generic versie :

```
public class Box < T > {
    private T type ;
    public void set (T type) {
        this type = type ;
    }
    public T get () ;
        Return type ;
    }
}
```

generic type invocation :

```
Box<Integer> intBox = new Box<Integer>();
Pair<String, Integer> par = new Pair<String, Integer>("Harry", 8);
```

Wildcard : '?' = type ve. parameter / veld / lokale variabele

↳ upper bound wildcard : List<? extends Number>
laat lijsten van Number en al zijn subklassen (Int, Double, ..) toe

↳ unbounded wildcard : lijst van onbekend type
printlijst die list van elk type kan afprinten :

```
public static void printlijst (List<?> lijst) {
    for (Object elem : lijst)
        System.out.println(elem);
}
```

↳ lower bounded wildcard : onbekende type = specifiek opgelegd type of supertype ervan
List<? super Integer>

EINDE INTERMEZZO

⊛ INTERFACE LIST<E>

```
E get (int index)
E set (int index, E element)
void add (int index, E element)
E remove (int index)
boolean addAll (int index, Collection<? extends E> c)
```
} positional access

```
int indexOf (Object o)
int lastIndexOf (Object o)
```
} search

```
ListIterator<E> listIterator()
ListIterator<E> listIterator (int index)
```
} iteration

```
List<E> subList (int from, int to)
```
} range view

`list1.addAll(list2);`

↪ list1 wordt vernietigd !
^
oorspronkelijke

niet-destructieve vorm:

```
List<T> list3 = new ArrayList<T>(list1);
list3.addAll(list2);
```

List : 2 implementaties in Java < ArrayList / LinkedList

opm. : Stack klasse : (LIFO) (↔ queue : FIFO)

INTERFACE SET <E>

basic operations:
- int size()
- boolean isEmpty()
- boolean contains(Object elem)
- boolean add(E elem)
- boolean remove(Object elem)
- Iterator<E> iterator()

bulk operations:
- boolean containsAll(Collection<?> c)
- boolean addAll(Collection<? extends E> c)
- boolean removeAll(Collection<?> c)
- boolean retainAll( )
- void clear()

array operations:
- Object[] toArray()
- <T> T[] toArray(T[] a)

Set = collection ZONDER DUBBELS (≈ verzameling → wiskunde)

2 implementaties in Java <
- HashSet (volgorde vd elementen kan over time veranderen)
- TreeSet (sorted!)

HashSet codevoorbeeld : dubbels detecteren

```
public class findDups {
    public static void main (String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            s.add(a);
        System.out.println (s.size() + " distinct words : " + s);
    }
}
```

Java 8 way :

```
Set<String> distinctWords = Arrays.asList(args).stream()
        .collect(Collectors.toSet());
System.out.println (distinctWords.size() + " ___ " + distinctWords);
```

⑭ INTERFACE SORTEDSET

```
SortedSet<E> subSet (E from, E to)   }
SortedSet<E> headSet (E to)          }  range view
SortedSet<E> tailSet (E from)        }

E first()                            }
E last()                             }  endpoints

Comparator<? super E> comparator()   }  comparator access
```

SortedSet = Set die elementen in stijgende volgorde bijhoudt

1 implementatie in Java : TreeSet

⊛ INTERFACE QUEUE

E element()
boolean offer(E e)
E peek()
E poll()
E remove()

FIFO (↔ Stack : LIFO)

voorbeeld :

```
public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();
        for (int i = time; i >= 0; i--)
            queue.add(i);
        while (!queue.isEmpty()) {
            s.o.p.(queue.remove());
            thread.sleep(1000);
        }
    }
}
```

⑧ INTERFACE MAP

```
V put (K key, V value)
V get (Object key)
V remove ( " )
boolean containsKey (Object key)
boolean containsValue (Object value)
int size ()
boolean isEmpty ()
void putAll (Map < ? extends K, ? extends V > m)
void clear ()
Set < K > keySet ()
Collection < V > values ()
Set < Map.Entry < K,V >> entrySet ()
```

} basic operations

↳ **INTERFACE ENTRYSET**

```
K getKey ()
V getValue ()
V setValue (V value)
```

Map beeldt keys af op values

geen dubbele sleutels

3 implementaties in Java — HashMap, TreeMap, LinkedHashMap

voorbeelden zie ppt slide 49 tem 52

itereren over Map :

```
for (KeyType key : m.keySet ())
    S.o.p. (key) ;
```

INTERFACE SORTEDMAP

```
Comparator < ? super k > comparator ()
SortedMap < k, v > subMap (k from, k to)
SortedMap < k, v > headMap (k to)
SortedMap < k, v > tailMap (k from)
k firstkey ()
k lastkey ()
```

SortedMap - Map die zijn entries bijhoudt in stijgende volgorde

PERFORMANTIE

TreeSet sneller dan ArrayList ; gesorteerd

2 Zoekalgoritmes

Doelen :
* zo snel mogelijk sorteren
* zo weinig mogelijk geheugen gebruiken

hoe minder geneste loops, hoe sneller

indien lijst niet geordend : sequentieel zoeken
gem. $n/2$ vgl^n

indien lijst geordend : $n = 2^x$ met x = aantal vgl^n nodig
$\Leftrightarrow x = \log_2(n)$

slechtste algoritme : bogosort (= stupid sort, slowsort)
= kaarten id lucht gooien tot ze gesorteerd zijn

insertion sort : element 0 en 1 t.o.v. vergelijken en wisselen indien nodig v plaats
idem voor 1 en 2, daarna 0 en 2
idem voor 2 en 3, daarna 1 en 3, daarna 0 en 3

bubble sort : element 0 wordt met alle volgende vergeleken
element 1 " " " " "
...

code zie slide 69

RECURSIEVE SORTEERALGORITMES : MERGESORT & QUICKSORT

→ **MERGESORT**

DIVIDE if input length <= 2 → eindmethode gebruiken
anders → input in 2 verdelen

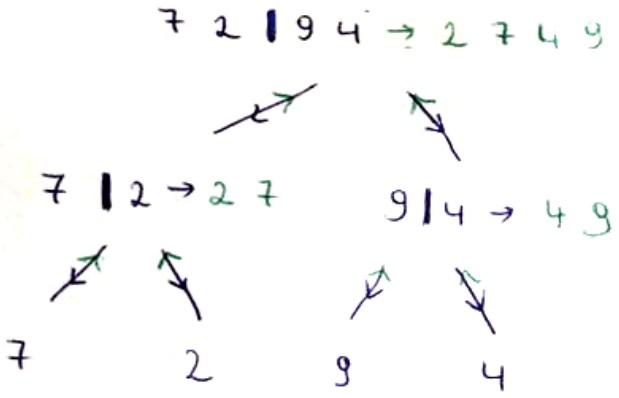
RECUR deelproblemen recursief oplossen

CONQUER oplossingen samenvoegen

⇒ input S wordt verdeeld in 2 lijsten S_1 en S_2, elk $\frac{n}{2}$ elementen

S_1 en S_2 $\bar{w}$ recursief gesorteerd

S_1 en S_2 $\bar{w}$ samengenomen



code : zie slide 90

→ QUICKSORT

S
"
Stap 1: voor lijst.length >= 2

laatste element v S selecteren (= pivot)

opsplitsen in 3 delen: L : el^ < x

E : el^ = x

R : el^ > x

Stap 2: L en R RECURSIEF SORTEREN

Stap 3: L + E + R (die volgorde)

code: slide 94 tem 98