Technische Hochschule Rosenheim

Modul- Fortgeschrittene Programmierkonzepte

Bachelor Informatik

12- Functional Porgramming

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Technische Hochschule Rosenheim

Functional Programming

Immutable Objects

Functions as First-Class Citizens

Immutable Objects

Technische Hochschule Rosenheim

If objects cannot be changed after their creation, parallelization becomes much easier.

`java.lang.String`

- no methods to change instance
- always returns *new* instance

`final` modifier for attributes and variable, sort of:

- only prevents overwriting of primitive type or reference
- object may still be mutated

  No mutation means no `for`/`while`!

Functions as First-Class Citizens

```
@FunctionalInterface
interface Function<A, B> {
    B apply(A obj);
}
```

```
Function<Integer, Integer> square1 = new Function<Integer, Integer>() {
    @Override
    public Integer apply(Integer i) {
        return i * i;
    }
}
```

Or shorter as lambda expression `(arglist) -> { block; }`

```
Function<Integer, Integer> square2 = (Integer i) -> { return i * i };
```

Or even shorter, for single instructions

```
Function<Integer, Integer> square3 = i -> i * i;
```

Why Functional Programming?

Immutability simplifies parallelization

Separation of Concerns

Functions as first-class citizens help to separate the iteration logic from the actual business logic.

Example

Technische Hochschule Rosenheim

Say you want to

- retrieve all students from a database,
- filter out those who took *Programmieren 3*,
- load their transcript of records from another database
- print all class names

Iterative Solution

```
for (Student s : getStudents()) {
    if (s.getClasses().contains("Programmieren 3")) {
        ToR tor = db.getToR(s.getMatrikel());
        for (Record r : tor) {
            System.out.println(r.getName());
        }
    }
}
```

A Simple Immutable List

`head` stores the data, `tail` links to the next element.

The end of the list is explicitly modeled.

```
class List<T> {
    final T head;
    final List<T> tail;

    private List(T el, List<T> tail) {
        this.head = el;
        this.tail = tail;
    }

    boolean isEmpty() {
        return head == null;
    }

    // ...
}
```

Some Helper Functions

Some factory functions for convenience:

```
class List<T> {
    // ...

    static <T> List<T> empty() {
        return new List<T>(null, null);
    }

    static <T> List<T> list(T elem, List<T> xs) {
        return new List<>(elem, xs);
    }

    static <T> List<T> list(T... elements) {
        if (elements.length == 0)
            return empty();
        int i = elements.length - 1;
        List<T> xs = list(elements[i], empty());
        while (--i >= 0)
            xs = list(elements[i], xs);
        return xs;
    }
}
```

Recursive Sort Algorithms

Technische Hochschule Rosenheim

Recursive Sort Algorithms

Technische Hochschule Rosenheim

Insertion Sort

```
static <T extends Comparable<T>> List<T> isort(List<T> xs) {
    if (xs.isEmpty()) return xs;
    else return insert(xs.head, isort(xs.tail));
}

private static <T extends Comparable<T>> List<T> insert(T x, List<T> xs) {
    if (xs.isEmpty()) return list(x, empty());
    else {
        if (x.compareTo(xs.head) < 0) return list(x, xs);
        else return list(xs.head, insert(x, xs.tail));
    }
}
```

Recursive Sort Algorithms

Technische Hochschule Rosenheim

Merge Sort

```
static <T extends Comparable<T>> List<T> msort(List<T> xs) {
    if (xs.isEmpty()) return xs;              // no element at all
    else if (xs.tail.isEmpty()) return xs;    // only single element
    else {
        int n = length(xs);
        List<T> a = take(xs, n/2);
        List<T> b = drop(xs, n/2);

        return merge(msort(a), msort(b));
    }
}

private static <T extends Comparable<T>> List<T> merge(List<T> xs, List<T> y
    if (xs.isEmpty()) return ys;
    else if (ys.isEmpty()) return xs;
    else {
        if (xs.head.compareTo(ys.head) < 0)
            return list(xs.head, merge(xs.tail, ys));
        else
```

Anonymous Classes, Lambda, References

```
static <A> void forEach(List<A> xs, Consumer<A> c) {
    if (xs.isEmpty()) return;
    else {
        c.accept(xs.head);
        forEach(xs.tail, c);
    }
}
```

And here's a `Consumer` that prints elements to `System.out`:

```
List<Integer> xs = list(1, 2, 3, 4);
forEach(xs, new Consumer<Integer>() {
    @Override
    public void accept(Integer i) {
        System.out.println(i);
    }
});

// or shorter with lambda
forEach(xs, i -> System.out.println(i));

// or even shorter with method references
```

Example

Technische Hochschule Rosenheim

Iterative Solution (see earlier slide)

```
for (Student s : Database.getStudents()) {
    if (s.getClasses().contains("Programmieren 3")) {
        Transcript tr = Database.getToR(s.getMatrikel());
        for (Record r : tr)
            System.out.println(r);
    }
}
```

Example

Technische Hochschule Rosenheim

Functional Solution

```
Database.getStudents().stream()
    .filter(s -> s.getClasses().contains("Programmieren 3"))
    .map(Student::getMatrikel)
    .map(Database::getToR)
    .flatMap(t -> t.records.stream())  // stream of lists to single list
    .forEach(System.out::println);
```


wow