(3) Java Advanced

Nico Ludwig (@ersatzteilchen)

TOC

- (3) Java Advanced
  - Mathematics vs Programming
  - Functions in Mathematics vs Functions in Programming
  - Mathematical Procedures
  - "Functional Programming Tools" in imperative Programming Languages
  - Data Containers for functional Programming and linked Lists
  - Higher Order Functions
  - Partial Function Application
  - Currying

- Cited Literature:
  - Functional Programming for the real World, Tomas Petricek with Jon Skeet
  - https://www.ps.uni-saarland.de/courses/cl-ss03/skript/ett.pdf
  - https://www.vavr.io/vavr-docs/#_values

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Functions in Maths and Functions in Programming

- Before we understand why functional programming is an important concept we have to review functions/methods.
  - We'll stress the relations between <u>functions in programming</u> and <u>functions in maths</u>.

- Let's discuss following implementation of *sum()* in Java:

```java
public static double sum(List<Double> numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += number.get(i);
    }
    return sum;
}
```

- <u>Actually, the method *sum()* cannot be expressed as mathematical function!</u>

- The restriction is mathematics is, that we <u>cannot have side effects</u>.
  - Following innocent looking Java expressions are basically <u>not allowed in at all mathematics</u>: `sum += number` `++i`
  - There is more, also something like <u>a loop can at least not be expressed in a mathematical function</u>.
  - Mind, that a loop only works, <u>because of side effects on variables</u>: *i* in the case above, or the <u>stateful</u> iterator in a for each loop.

- But, <u>in school</u> we used maths differently, somehow <u>side effects were possible</u>! – Let's clarify this aspect!   4

Mathematical Procedures – Part I

- What we mostly do, when we solve mathematical problems is using a <u>mathematical procedure</u> or <u>mathematical algorithm</u>.

```java
public static double sum(List<Double> numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += number.get(i);
    }
    return sum;
}
```

- The mathematical procedure of calculating the sum is just summing the individual numbers into an <u>intermediate result</u>.
    - The intermediate result is an <u>actual variable</u> in its sense, because we have a side effect on it during each iteration <u>"it varies"</u>.
    - Another way to understand its: we use the intermediate result <u>to rescue intermediate data into the next iteration</u>.

- In literature about the comparison of maths and imperative programming you'll read, you can't have something like this:
    - An expression like $\boxed{++i}$, which translates to $\boxed{i = i + 1}$ cannot exist, because i cannot be equal to $i + 1$.
    - <u>But this is a wrong way to understand imperative programming!</u> $\boxed{i = i + 1}$ is an <u>assignment</u>, <u>no comparison</u>, it must be read differently:
    - "After $\boxed{i = i + 1}$ was executed, the new value of i will be the old value of i plus 1!"

- The last cite underscores, how imperative programming is different from maths: <u>in maths there is no "after"</u>, <u>a formula can't "remember"</u>.
    - Can't remember means, that <u>a formula cannot have a state</u>!

5

Mathematical Procedures – Part II

- "Can't remember state in maths" bears a lot of <u>surprising</u> facts about maths, that are crucial to understand fp.

- (1) The following mathematical expression does not directly compare *a* with 0, <u>it instead asserts, that *a* is equal to 0</u>:

  $$a = 0$$

  /* corresponds to */ a == 0

- (2) <u>Maths can only initialize variables.</u> The correct mathematical notation is the <u>definition expression</u>:

  $$a := 0 \text{ or alternatively } a \stackrel{\text{def}}{=} 0$$

  /* corresponds to */ double a = 0;

- (3) <u>Maths has no variables as we know them from programming.</u>
  - But something like this is not allowed in "strict" maths:

  $$a := 0$$
  $$a := \pi \text{ illegal!}$$

  /* corresponds to */ double a = 0;
  /* would correspond to */ double a = 3.14; // Ok!

- In imperative programming, the variables known from maths rather work like <u>run time constants</u>!
  - The above shown restrictions on maths are the same for, e.g. Java, if we define *a* as run time constant:

  final double a = 0; // define a as run time constant

  a = 3.14; // Invalid! Cannot assign a value to final variable a

- Many fp languages also have <u>no concept of variables</u>! Instead they have <u>run time constants</u>, which they call <u>values</u>.

- We come back to how variables work in maths, when we compare mathematical functions with functions in programming.

6

- In maths, variables often apply identifiers following some rules:
  - In common many identifies are reduced to only one letter of the alphabet.
  - The first letters of the alphabet are often used to name known values. E.g. *a*, *b*, *c* and *d* to name the sides of a 4-polygon.
  - The last letters of the alphabet are often used to name unknown values. E.g. *x*, *y* and *z* to name parameters of a function.
  - The letters "in between" are often used to name natural numbers, esp. *n* and *m*, but also *i* and *j* to name indexes.
  - Angles are named using greek lower case letters.

From Maths to functional Programming – Part I

- In maths <u>there is no "run time"</u>, there is only <u>"instantaneousness"</u>, which means <u>"the moment"</u>.

- <u>Maths deals with problems of infinite nature</u>, <u>computer science deals with problems of finite nature</u>.

- All right, but if maths deals with infinite problems, how can it describe those with its <u>"finite" notation</u>?

- In strict maths formulas can only be expressed as <u>closed form</u> or <u>recursion</u>.

- Consider the definition of the series of natural numbers as sum:
$$S_n = 1 + 2 + 3 + \ldots + n$$
  - But this definition is very … infinite! There is a way to calculate this sum with this formula:
$$S_n = \frac{n^2 + n}{2}$$
  - <u>This is the closed form of the series of natural numbers!</u> Its definition is finite, there is no more "…".
  - A closed form of a calculation means expressing a calculation <u>as a deterministic count of steps independent from the input</u>.

- In math we can also use the <u>summation symbol</u>: $\sum_{k=1}^{n} k$

7

From Maths to functional Programming – Part II

- But <u>not all mathematical calculations have a closed form</u>! Consider the faculty defined as product of natural numbers:
$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$$

  - This definition is also very ... infinite! Is there a way to calculate this product with a "handy" formula?

  - <u>No (2020)!</u> There is no way to write this calculation without the need to calculate <u>each sub-product of n</u>.

  - The calculation is dependent from the full input, so there is no closed form.

  - Although there is no closed form, we can use the <u>capital pi</u> notation, which is derived from the summation symbol: $\prod_{k=1}^{n} k$

- However, programming n! with a function of an imperative programming language is easy:

```
public static int faculty(int n) { // if n >= 0
    int result = 1;
    for (; 0 != n; --n) {
        result *= n;
    }
    return result;
}
```

  - But we cannot have such a function in maths: the expressions $\boxed{--n}$ and $\boxed{result\ *=\ n}$ bear side effects, we don't have in maths!

  - Put differently: <u>we cannot have loops in maths!</u>

- A mathematical calculation like the one for faculty can instead be done by a <u>recursion</u>:
$$n! = \begin{cases} 1; & n=0 \\ n((n-1)!); & n>0 \end{cases} \quad n \in \mathbf{N}$$

8

- Technically, recursion means, that <u>a formula calls itself or is expressed in terms of itself</u>.

- In the beginning, recursion is hard to understand. Let's recapitulate three recursion steps for n!:

$$n! = \begin{cases} 1; & n=0 \\ n(\boxed{(n-1)}!); & n>0 \end{cases} \quad n \in \mathbf{N}$$

$$\boxed{(n-1)}! = \begin{cases} 1; & n=0 \\ n(\boxed{(n-2)}!); & n>0 \end{cases} \quad n \in \mathbf{N}$$

$$\boxed{(n-2)}! = \begin{cases} 1; & n=0 \\ n(\boxed{(n-3)}!); & n>0 \end{cases} \quad n \in \mathbf{N}$$

- Thinking about <u>each recursion step following the recursion doesn't make a lot of sense</u>. The <u>nature of recursion</u> is key.
    - The nature of the recursion for n! is: "to get n! I have to know (n-1)!"

- The idea of recursion is to <u>solve a tiny part of the problem</u> and <u>pass the remainder of the problem to the next recursion</u>.

- A basic recursive formula has only two cases:
    - (1) <u>Base case (or cases)</u>: the problem is solved when <u>the input is empty</u> and <u>the result can be set</u> (for n! the result is 1 if n is 0). 9
    - (2) <u>Recursive step</u>: <u>a part of the problem can be solved</u>, <u>the remaining part is passed to the next recursion</u> (for n! the result is n * (n -1)!)

- Another popular problem, which can be solved with recursion is the "Towers of Hanoi"-problem. – The basic task is not difficult, but the more discs we have to handle, the more moves are required, the more steps must be thought about in advance recursively.

From Maths to functional Programming – Part IV

```java
public static double sum(List<Double> numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += number.get(i);
    }
    return sum;
}
```

- Back to *sum()*: the programmed function *sum()* as we defined it, accepts a list of numbers, which is summed up in a loop.

- For $\sum_{k=1}^{n} k$ we know how the sum progresses: <u>each natural number is summed up, until n is reached</u>: $S_n = 1+2+3+\ldots+n$
  - $\sum_{k=1}^{n} k$ expresses this progress in maths as: <u>"sum each k when k is increased until n is reached"</u>
  - When we have this building-up rule, we can formulate the closed form $S_n = \dfrac{n^2+n}{2}$.

- But *sum()* isn't the same as $\sum_{k=1}^{n} k$ because we cannot predict each number in numbers:
  - *sum()* processes a list of <u>arbitrary numbers</u>, i.e. a set of <u>arbitrary constants</u>, that has <u>no building-up rule known to us</u>!
  - I.e. when we have a certain number, <u>we don't know the value of its successor</u>, <u>instead we must look into the list</u>.
    - In opposite to $\sum_{k=1}^{n} k$, where we exactly know, that the successor of a certain k is k + 1 based on the <u>building-up rule</u>.
  - (A basic restriction is that *numbers* in *sum()* are <u>real</u>: the successor of a real number cannot be predicted by building-up rules.)

From Maths to functional Programming – Part V

```java
public static double sum(List<Double> numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += number.get(i);
    }
    return sum;
}
```

- In maths a variable like, e.g., *sum* cannot exist in a function, because we can't express intermediate states.
  - Maths demands <u>referential transparency</u> for functions: values of expressions <u>only depend on the context</u>, <u>not the time of evaluation</u>.
  - <u>Referential transparency means, that we cannot have a loop, in which *sum* changes in between loop iterations.</u>

- The mathematical definition of a function, that sums up a set of *numbers* without a building-up rule can look like so:
$$sum(number) = number_n + number_{n-1} + ... + number_1 + number_0 \, ; number_k \in numbers \, ; n = |numbers|$$
  - The mathematical notation |numbers| denotes the <u>cardinality</u> of the set numbers, which is the <u>count</u> of elements in numbers.
  - The notation has no iterative aspects, i.e. no side effects are present. – It just <u>declares</u>, how the sum comes to happen.
  - It looks like a formula, which has no "moving parts".

- In maths, <u>the sum of arbitrary sets of numbers cannot be expressed as a closed form</u>, <u>because we've no building-up rule</u>.
  - <u>But we can create a recursive definition of such a sum!</u>

11
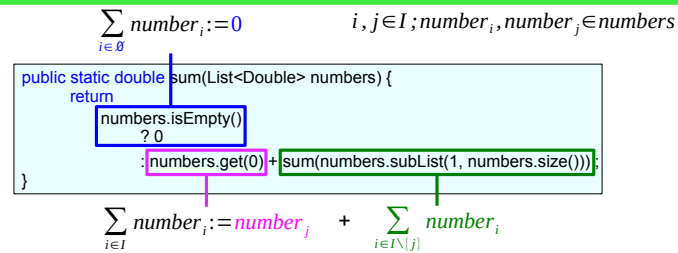
$$\sum_{i \in I} number_i \in numbers := \begin{cases} \sum_{i \in \emptyset} number_i := 0 \in numbers \\ \sum_{i \in I} number_i := number_j + \sum_{i \in I \setminus \{j\}} number_i \in numbers \end{cases} \qquad i, j \in I; number_i, number_j \in numbers$$

- <u>We still have no building-up rule of each *number* in *numbers*.</u>

- But we have other <u>information</u> about *numbers*!
  - We know, that <u>each *number* in *numbers* can be addressed via an index</u>.
  - We know, that <u>the greatest index is the count of elements in *numbers*</u>.

- Here we even go one step further:
  - Actually, we also know, that <u>a certain index has a specific successor</u>, which is <u>index + 1 or the count of elements in numbers</u>.
  - But this recursion abstracts this fact away: we just state there is an <u>index set</u>, which we call *I*.

- Next, let us try to express this recursive formula as Java code, i.e. create a <u>recursive implementation of *sum()*</u>.

12

$$\sum_{i \in I} a_i \in A := \begin{cases} \sum_{i \in \emptyset} a_i := 0 \in A \\ \sum_{i \in I} a_i := a_j + \sum_{i \in I \setminus \{j\}} a_i \in A \end{cases} \qquad i, j \in I; a_i, a_j \in A$$

From Maths to functional Programming – Part VII

$$\sum_{i \in \emptyset} number_i := 0 \qquad i, j \in I \, ; number_i , number_j \in numbers$$

```
public static double sum(List<Double> numbers) {
    return
        numbers.isEmpty()
            ? 0
            : numbers.get(0) + sum(numbers.subList(1, numbers.size()));
}
```

$$\sum_{i \in I} number_i := number_j \quad + \quad \sum_{i \in I \setminus \{j\}} number_i$$

- In Java: We pass *numbers*.
  - In maths: *numbers* is a set, the elements of the set are either at index *i* or *j*; *i* and *j* are valid indexes (elements of set *I*) for elements in *numbers*.
- If *numbers* is empty the result is 0.
  - In maths: if there is no element at index *i* the result is 0.
- Else get the first element of *numbers* and add it to the sum of *numbers*' elements except the first element of *numbers*.
  - In maths: else get the element at index j and add it to the sum of all numbers, which are not at index j.

- The else part is required for recursion to work: it transports the remaining part of the problem to the next method call.

- Recursive programming is a little bit strange in the beginning, however, it also shows no side effects in its definition.
  - The recursive function seems to have no "moving parts": no loops and no variable is updated.

Functions revisited – Part I

- At this stage we will have another look at how mathematical functions differ from programmed functions <u>ideologically</u>.
  - Let's discuss this function: $f(x) = x^2 + 5 ; x \in \mathbf{R}$

```java
public static double f(double x) {
    return x*x + 5;
}
```

- In maths, we're interested in <u>the specific x for a specific $f(x)$</u>, in programming we are interested in the <u>$f(x)$ for a specific x</u>.
  - In maths, we are often observing functions <u>analytically</u>, i.e. we want to find the x for a certain result, consider:

$$
\begin{aligned}
f(x) &= x^2 + 5 \\
30 &= x^2 + 5 \\
x_1 &= 5 \\
x_2 &= -5
\end{aligned}
$$

  - We even have two xs, when we analyze the function f.
  - <u>=> Maths' analytical aim is to solve an equation involving the function somehow, i.e. finding xs for a certain $f(x)$.</u>

- In programming, <u>we are interested in the result</u>, because we know the argument, well, we are passing it to the function.
  - <u>In programming, we are only interested in a result of a function call, consider:</u>

```java
double result1 = f(5);
// result1 = 30
double result2 = f(-5);
// result2 = 30
```

  - <u>=> A programming language's aim is to call a function as part of the work of a bigger problem or to reuse code.</u>   14

Functions revisited – Part II

- The different understanding of functions in maths and programming also reflects in the usage of terminology.

$$f(x)=x^2+5\,;x\in \boldsymbol{R}$$

- In maths, the x in the definition of f <u>is not a variable</u>, <u>instead it is a placeholder</u>.
  - This placeholder is replaced by any element of the function's domain, which is **R** in this case.
  - It is <u>"called"</u> variable, because <u>there will be different xs passed to $f(x)$</u>. <u>But the definition of f does never modify the passed x!</u>
  - The passing of different xs to f describes <u>the mapping of x to $f(x)$</u>.
  - Usually x is called <u>parameter</u>: during f is "executed", <u>it will be executed for exactly one x</u>, <u>which gets not modified in the function</u>.
  - Formally, maths clearly distinguishes between (1) <u>$f(x)$</u>, which is the <u>value of the function at x</u> and (2) <u>f</u>, which <u>the function's name</u>.

- In programming a function is more like a <u>mathematical procedure</u> as <u>it can have states</u>.
  - Actually, *f(double)* has an implicit state, when it intermediately stores the result. Virtually, it looks similar to this:

```
public static double f(double x) {
    double ret;
    ret = x*x + 5
    return ret;
}
```

  - It is looking really imperative, right?

15

- We can say, that <u>a mathematical function is declarative</u> (no moving parts), whereas a <u>programmed function is imperative</u>.

Which Tools do we already have in imperative Programming Languages? – Part I

- Computer science vs mathematics:
    - <u>Computer science deals with problems of finite nature</u>, <u>mathematics deals with problems of infinite nature</u>.
    - In maths there is no formal way to define a calculation, <u>which is dependent on another calculation through side effects</u>.
        - The only way to have a calculation depend on another one is <u>to pass new data to it</u>.
        - <u>When a calculation passes new data to itself, we call this recursion.</u> Recursion can be used to <u>mimic loops</u> known from computer science.
    - In "school maths", we use <u>mathematical procedures</u> and <u>informal intermediate results</u>.

- From a "mechanical" standpoint, fp languages go some steps farther than imperative programming languages:
    - Many fp languages have <u>no concept of variables</u>! Instead they have <u>run time constants</u>, which they call <u>values</u>.
    - <u>Pure fp languages</u> have <u>no loops</u>, but only <u>recursion to express repetition</u>!
    - Pure fp languages have <u>no "real" statements</u>, but rather have <u>expressions</u>.

- Lisp, a representation of a not-so-pure fp language uses <u>macros</u> to <u>mimic imperative syntax</u>.
    - E.g. Lisp's *loop* macro was specifically created to enable inexperienced fp programmers writing imperative-iterative code in Lisp.

16

Which Tools do we already have in imperative Programming Languages? – Part II

- Non-fp programming languages (this includes C/C++, Java and C#) often have some support for fp "mechanics".
  - "Mechanical" support mainly means syntactical aspects, esp. support to write code to avoid side effects.

- (1) To express conditionals we can use the ternary operator ?: instead of if/else statements:

```
public static double sum(List<Double> numbers) {
    if (numbers.isEmpty()) {
        return 0;
    } else {
        return numbers.get(0) + sum(numbers.subList(1, numbers.size()));
    }
}
```

```
public static double sum(List<Double> numbers) {
    return
        numbers.isEmpty()
            ? 0
            : numbers.get(0) + sum(numbers.subList(1, numbers.size()));
}
```

  - As can be seen, the variant using if/else looks clearly like it has some moving parts, where the ?:-variant doesn't.

  - ?: expresses the full algorithm as a single statement, well, even as a single expression.

- (2) To express repetition of activity we use recursion instead of loops:

```
public static double sum(List<Double> numbers) {
    double sum = 0;
    for (int i = 0; i < numbers.size(); ++i) {
        sum += number.get(i);
    }
    return sum;
}
```

```
public static double sum(List<Double> numbers) {
    return
        numbers.isEmpty()
            ? 0
            : numbers.get(0) + sum(numbers.subList(1, numbers.size()));
}
```

17

Functional Data Containers – Part I

- Actually, there is a third aspect we need for functional programming: <u>data-containers that work free of side effects</u>.

- Let's discuss the method *collectPersons()*:

```java
public static Map<String, List<Person>> collectPersons(List<Person> source) {
    Map<String, List<Person>> collected = new HashMap<>();
    for (Person person : source) {
        if (collected.containsKey(person.getName())) {
            collected.get(person.getName()).add(person);
        } else {
            collected.put(person.getName(), new ArrayList<>(List.of(person)));
        }
    }
    return collected;
}
```

  – The method groups the *Person*s in the passed *List* by their name and puts these groups into a *Map*.

  – The returned *Map<String, List<Person>>* groups each name to a *List* of *Person*s with the same name:

```java
List<Person> persons = List.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
Map<String, List<Person>> result = collectPersons(persons);
```

Functional Data Containers – Part II

- The question: can we implement *collectPerson()* in a functional way?

```java
public static Map<String, List<Person>> collectPersons(List<Person> source) {
    Map<String, List<Person>> collected = new HashMap<>();

    for (Person person : source) {
        if (collected.containsKey(person.getName())) {
            collected.get(person.getName()).add(person);
        } else {
            collected.put(person.getName(), new ArrayList<>(List.of(person)));
        }
    }
    return collected;
}
```

- Maybe the better question is: What makes *collectPerson()* "non-functional" style?
    - (1) We have a for each loop, which uses an *Iterator* to store the state of iteration. -> Side effect and state for the loop.
    - (2) We also use the "ordinary" if/else statements instead of ?:, but this is only a syntactic detail.

- The more serious thing is, that *collectPersons()* uses *List* and *Map* and performs side effects on those!
    - In the if branch, *Person* objects are added to a *List*, in the else branch *List*s are added to a *Map*. And this all is done in a loop.

- Actually, the problem lies in the collections we are using! – Their working paradigm bases on side effect!

19

- So, there exist "side effect free collections"? – Yes! There exist functional-style collections!

- Each time we add a new *Person* to *persons*, the *List persons* is modified as a side effect:

```
List<Person> persons = new ArrayList<>();
persons.add(new Person("Ashley", 67));
persons.add(new Person("Lisa", 17));
persons.add(new Person("Ashley", 23));
```

- Apart from the JDK, 3rd party libraries like *vavr* provide object-functional-style collections:

```
List<Person> persons = List.empty();
persons = persons.append(new Person("Ashley", 67));
persons = persons.append(new Person("Lisa", 17));
persons = persons.append(new Person("Ashley", 23));
```
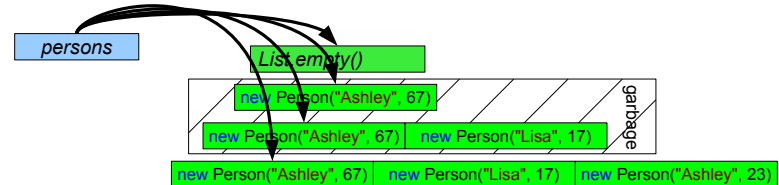

- – Object-functional collections work fundamentally different from "classical" collections.
- – The methods on object-functional collections do never modify a present collection, but rather create a new different one.
- – E.g. *append()* returns a copy of the original *List* having the passed *Person* object appended to it.
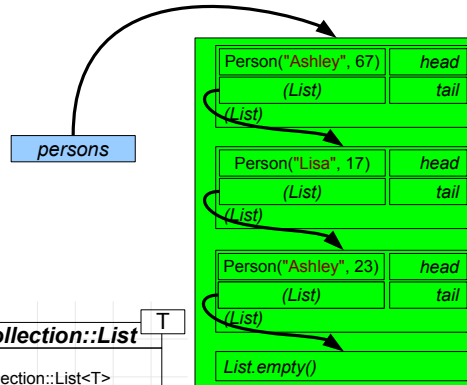
20

- *vavr* is a (3rd party) Java-library, which provides types for object-functional programming, esp. non-modifiable collection types preserving referential transparency and side effect free programming. It adds some Scala-features into Java.

Linked Lists – Part I

- The internal representation of *vavr*'s *List*s is no longer array-based!
  - Its elements are not stored in a contiguous block of memory, but as a bunch of linked lists.

```
List<Person> persons = List.empty();
persons = persons.append(new Person("Ashley", 67));
persons = persons.append(new Person("Lisa", 17));
persons = persons.append(new Person("Ashley", 23));
```

Person("Ashley", 67)	head
(List)	tail
(List)	

persons

Person("Lisa", 17)	head
(List)	tail
(List)	

Person("Ashley", 23)	head
(List)	tail
(List)	

| List.empty() |

- Some facts about *vavr*'s *List*:

  io::vavr::collection::List   T
  - head : T
  - tail : io::vavr::collection::List<T>

  - It is an abstract class, not an interface.
  - A *List* refers to the head element of the list.
  - And it refers to yet another *List*, the tail, which contains the remaining elements.
  - The last tail *List* is the empty *List*.
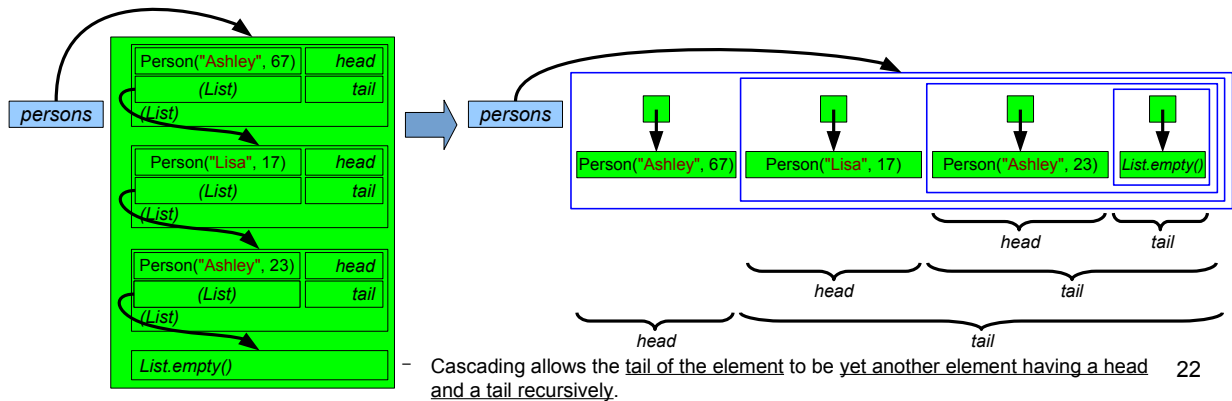 - => *vavr*'s *List* is a recursive UDT.

21

- In the programming language Lisp the concept of linked lists is basically the only idiom used to write programs. – Lisp programs themselves are linked lists.

Linked Lists – Part II

- Esp. in functional programming languages, lists do often play an important role.
  - As a matter of fact immutable (linked) lists are often first class citizens in fp languages, i.e. often they have direct syntactical support.
  - In opposite to imperative languages, where (mutable) arrays are often first class citizens with direct syntactical support.

- *vavr*'s *List*s can be understood as a chain of cascaded head and tail elements:

- Cascading allows the tail of the element to be yet another element having a head and a tail recursively.

22

Linked Lists – Part III

- *vavr*'s *List* could be iterated using indexes and a *List*'s length, but <u>this is not the way to use head/tail-based lists</u>.

```
// Traditional imperative loop to enumerate a list/array:
for (int i = 0; i < persons.length(); ++i) {
    System.out.prinln("name: "+persons.get(i).getName());
}
```


logical indexes

| 0 | 1 | 2 | 3 |
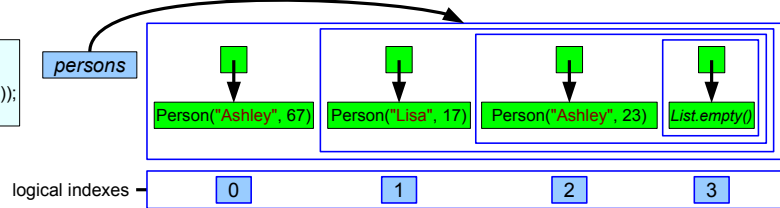
- Usually, head/tail-based lists are iterated by visiting all heads, until there is an empty tail:

```
// Still a traditional imperative loop:
while (!persons.isEmpty()) {
    System.out.println("name: ", persons.head().getName());
    persons = persons.tail();
}
```

- But in fp, we don't want state (mind, that the *persons* reference is overwritten in each loop), so we go a <u>recursive</u> approach:

```
// The usual way to enumerate a head/tail-based list:
public static void recurseList(List<Person> persons) {
    if (!persons.isEmpty()) {
        System.out.println("name: " + persons.head().getName());
        recurseList(persons.tail());
    }
}
```

23

Linked Lists – Part IV

- In many fp languages head/tail-based lists have syntactical support to break lists into head and tail with special operators.
  - E.g. let's sum the numbers contained in a (linked) list, by chopping its head and tail recursively:

```fsharp
// F#
let rec sumList list =
    match list with
    | head :: tail -> head + sumList tail
    | [] -> 0

printfn "%A" (sumList [1; 2; 3; 4; 5; 6; 7; 8; 9])
// >45
```

```scala
// Scala
def sumList(list: List[Int]) : Int =
    list match {
        case head :: tail => head + sumList(tail)
        case Nil => 0
    }

sumList(List(1, 2, 3, 4, 5, 6, 7, 8, 9))
// =45
```

```haskell
-- Haskell
import System.Environment

sumList (x:xs)  = x + sumList xs
sumList []       = 0

main = print(sumList [1, 2, 3, 4, 5, 6, 7, 8, 9])
-- >45
```

  - In each snippet the passed list is matched against a given pattern:
    - If the list is not empty, it'll be broken into its head and tail. The head will be added to the result of a recursive call with the tail.
    - If the list is empty the result is 0 and the recursion ends.
  - This idiom of F#, Haskell and Scala is called (structural) pattern matching. Pattern matching is a kind of control structure.

- In Lisp we don't have syntactical support like pattern matching, but there are functions to get the head and the tail of a list:
  - *car* (Contents of the Address part of Register number) and *cdr* (Contents of the Decrement part of Register number).

```lisp
; Lisp
(defun sum-list(l)
  (if (not (null l))
      (+ (car l)(sum-list (cdr l)))
      0))

(sum-list '(1 2 3 4 5 6 7 8 9))
; =45
```

24

- *car* and *cdr* have the aliases *first* and *rest* meanwhile :).
- There exist more functions to address the contents of a list apart from the operators ::/: and *car*/*cdr*, e.g.: *nth* or handy combinations of *car* and *cdr* like *caar* and *cadr*.
- It should be mentioned that memory management of fp languages is very sophisticated. Esp. during processing lists, many new lists are generated, because lists are immutable; all the remaining superfluous lists need to be freed from memory. Therefor fp languages/runtimes were the first systems introducing garbage collection. Mind that the general ways of processing lists and also the memory considerations are equivalent to those of processing strings in Java and .NET, where strings are immutable (lists of characters).

Linked Lists – Part V

- Actually, object-functional collections are very much similar to Java's *String*s:
    - Operations on *String* never change the *String*, but produce new *String*s.
    - In opposite to *StringBuilder*, whose calls change the state of the *StringBuilder*.
    - *StringBuilder*s work more like the JDK's side effect based *List*.

- *vavr*'s *List* has a so called fluent interface: most of its methods return new *List*s, on which yet more methods can be called.
    - Instead of this:

    ```
    List<Person> persons = List.empty(); // Traditional style:
    persons = persons.append(new Person("Ashley", 67));
    persons = persons.append(new Person("Lisa", 17));
    persons = persons.append(new Person("Ashley", 23));
    ```

    - we can use a fluent programming style:

    ```
    List<Person> persons = List.empty() // Fluent style:
            .append(new Person("Ashley", 67))
            .append(new Person("Lisa", 17))
            .append(new Person("Ashley", 23));
    ```

- We can also use *vavr*'s simple factory *List.of()* to directly create a *List* from some *Person*s:

    ```
    List<Person> persons = List.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
    ```

    - Java 9 also supports *java.util.List.of()*, which works the way as *vavr*'s.

25

Linked Lists – Part VI

- Using functional collections bears another implication, we should be aware of.
  - Because <u>many temporary collections are created during processing</u>, <u>the garbage collector has a lot of garbage to clean up</u>:

```
List<Person> persons = List.empty();
persons = persons.append(new Person("Ashley", 67));
persons = persons.append(new Person("Lisa", 17));
persons = persons.append(new Person("Ashley", 23));
```
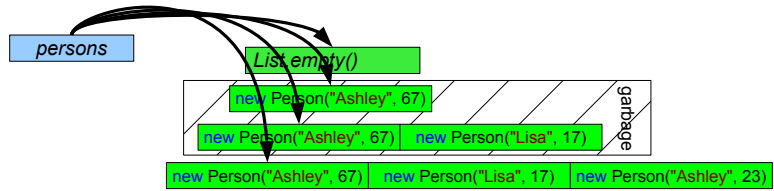

  - So, <u>there is a prize to pay</u>, when we want to have side effect free collections.
  - But what we get is rock solid collection, which allows esp. <u>parallel processing without any data races</u>.

- The <u>believe</u> is that fp (esp. in Java) <u>over-compensates its costs</u>, when processing runs in parallel.

- The oldest fp languages like Lisp already introduced garbage collection to <u>only deal with the many temporary copies</u>.

- To drive this point home, we'll now refactor *collectPersons()* using *vavr*'s collections, to have it fully fp-style.
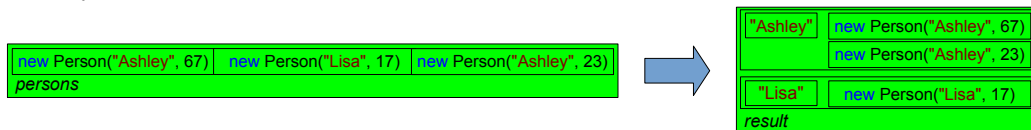
26

```
public static Map<String, List<Person>> collectPersons(List<Person> source, Map<String, List<Person>> collected) {
    return !source.isEmpty()
        ? collectPersons(source.tail(),
                collected.containsKey(source.head().getName())
                    ? collected.replaceValue(source.head().getName(), collected.get(source.head().getName()).get().append(source.head()))
                    : collected.put(source.head().getName(), List.of(source.head())))
        : collected;
}
```

- We have to call this implementation of *collectPersons()* a little bit differently from the former one:

```
List<Person> persons = List.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
Map<String, List<Person>> result = collectPersons(persons, HashMap.empty());
```

  – Effectively, the result is the same:

- In *vavr*, operations like *Map.replace()* do not modify the *Map*, but produce a new *Map*.

- The old and new implementation of *collectPersons()* look and work completely differently.

```java
// Traditional imperative implementation:
public static Map<String, List<Person>> collectPersons(List<Person> source) {
    Map<String, List<Person>> collected = new HashMap<>();
    for (Person person : source) {
        if (collected.containsKey(person.getName())) {
            collected.get(person.getName()).add(person);
        } else {
            collected.put(person.getName(), new ArrayList<>(List.of(person)));
        }
    }
    return collected;
}
```

```java
// Functional implementation:
public static Map<String, List<Person>> collectPersons(List<Person> source, Map<String, List<Person>> collected) {
    return !source.isEmpty()
        ? collectPersons(source.tail(),
            collected.containsKey(source.head().getName())
                ? collected.replaceValue(source.head().getName(), collected.get(source.head().getName()).get().append(source.head()))
                : collected.put(source.head().getName(), List.of(source.head())))
        : collected;
}
```

28

- Let's discuss how the fully fp-style version of *collectPersons()* work.

- Here we can see a visible effect of how code changes, when we go the fp-style path in languages of the C-family like Java: we see less semicolons (because we have less statements)! – The functional implementation of *collectPersons()* has only one semicolon/statement!

Linked Lists – Part IX

```
public static Map<String, List<Person>> collectPersons(List<Person> source, Map<String, List<Person>> collected) {
    return !source.isEmpty()
        ? collectPersons(source.tail(),
            collected.containsKey(source.head().getName())
                ? collected.replaceValue(source.head().getName(), collected.get(source.head().getName()).get().append(source.head()))
                : collected.put(source.head().getName(), List.of(source.head())))
        : collected;
}
```

- The processing of *source* makes use of all "fp-tools" we have discussed up to here:
    - If *source* is <u>not empty</u>, we call *collectPersons()* recursively and <u>we pass the tail of *source*</u>, <u>which is all its elements except the first</u>.
        - => This is the way we make the <u>"progression" through the *List*. *source.tail()* doesn't modify *source*, but creates a new *List*</u>.
        - If we already have collected *source*'s first element, which we get with *source.head()*. – The first *Person* in the *List* is the one we process now.
            - We replace the value of this key with a new *List*, consisting of the old *List* having the first *Person* appended.
                - <u>*Map.replaceValue()* also returns a new *Map* and doesn't modify the old *Map*</u>.
        - Otherwise, we add the *Person*'s name as key and a new *List* only containing this *Person* as value into the collected *Persons*' *Map*.
    - Otherwise, everything is assumed as processed and the collected *Person*-groups will be returned as *Map<String, List<Person>>*.

- Also mind, that <u>we have to pass an empty *Map* initially</u>, to satisfy the <u>recursive nature</u> of *collectPersons()*:

```
List<Person> persons = List.of(new Person("Ashley", 67), new Person("Lisa", 17), new Person("Ashley", 23));
Map<String, List<Person>> result = collectPersons(persons, HashMap.empty());
```

    - <u>While the *source*-arguments shrink, the *collected*-arguments grow as the recursion goes through the processing</u>.

29

Linked Lists – Part X

```java
public static Map<String, List<Person>> collectPersons(List<Person> source, Map<String, List<Person>> collected) {
    return !source.isEmpty()
        ? collectPersons(source.tail(),
            collected.containsKey(source.head().getName())
                ? collected.replaceValue(source.head().getName(), collected.get(source.head().getName()).get().append(source.head()))
                : collected.put(source.head().getName(), List.of(source.head())))
        : collected;
}
```

- Visually, this fp-code looks more like modeling a piece of clay, whereas imperative programming looks more angular.
    - The fp-style *collectPersons()* shows how Java's fp-tools come together: recursion, ?-expressions and side effect free collections.
    - It is required to mix all required calculations into a single big formula, or in Java-programming terms a single big expression.
    - => Programming even moderately complex fp-style algorithms needs some training.
    - We have no loops and we do even never request the length of the *source List* we pass around.
        - In fp-style algorithms the lengths of collections are usually not required, getting the length of such a collection could be very costly.
    - Actually, there is only one expression, without any side effects.

- Another implication is that we can usually not easily debug fp-style code.
    - Debugging basically means to go through the states of an algorithm stepwise, but in fp we no states, we only have the declaration.

Higher Order Functions – Part I

- Next, we are going to refine our *sum()* method. This time, we only want to sum <u>only positive numbers</u>, no problem:

```java
public static double sumOfPositives(List<Double> numbers) {
    return
        numbers.isEmpty()
            ? 0
            : (0 < numbers.get(0) ? numbers.get(0) : 0) + sum(numbers.subList(1, numbers.size()));
}
```

  - We are using all the fp-tools in Java to have no side effects here, but pay the prize with a more complex code in one expression.

  - However, it works:

```java
List<Double> numbers = List.of(10., -10., 20., -20., 30., -30.);
double result = sumOfPositives(numbers);
// result = 60.
```

- A <u>predictable</u> requirement of another developer might be to only have the <u>negative numbers</u> be summed up, no problem:

```java
public static double sumOfNegatives(List<Double> numbers) {
    return
        numbers.isEmpty()
            ? 0
            : (0 > numbers.get(0) ? numbers.get(0) : 0) + sum(numbers.subList(1, numbers.size()));
}
```

  - Sure, it'll work fine:

```java
double result = sumOfNegatives(numbers);
// result = -60.
```
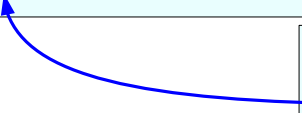
- But … "predictable" is just a good signaling term here: <u>Don't we have a DRY problem here?</u>

31

Higher Order Functions – Part II

- There <u>is</u> a DRY problem! *sumOfPositives()* and *sumOfNegatives()* <u>only differ in the condition under which numbers are summed up</u>!

```java
public static double sumOfPositives(List<Double> numbers) {
    return numbers.isEmpty()
        ? 0
        : ( 0 < numbers.get(0) ? numbers.get(0) : 0) + sum(numbers.subList(1, numbers.size()));
}
```

```java
public static double sumOfNegatives(List<Double> numbers) {
    return numbers.isEmpty()
        ? 0
        : ( 0 > numbers.get(0) ? numbers.get(0) : 0) + sum(numbers.subList(1, numbers.size()));
}
```

- In an fp language, we have the option to <u>formulate the condition as additional function parameter</u>:

```java
public static double conditionalSum(List<Double> numbers, Predicate<Double> condition ) {
    return numbers.isEmpty()
        ? 0
        : ( condition.test(numbers.get(0)) ? numbers.get(0) : 0) + conditionalSum(numbers.subList(1, numbers.size()), condition );
}
```

   - As can be seen, we pass the condition as type *Predicate<Double>*, which is a functional interface.

   - Let's understand, how we can use *conditionalSum()*.

32

Higher Order Functions – Part III

```java
public static double conditionalSum(List<Double> numbers, Predicate<Double> condition) {
    return numbers.isEmpty()
        ? 0
        : (condition.test(numbers.get(0)) ? numbers.get(0) : 0) + conditionalSum(numbers.subList(1, numbers.size()), condition);
}
```

- To call *conditionalSum()* we have to pass a *List<Double>* and <u>we have to pass another function</u>!
  - First let's define a static method *Algorithms.isPositive()*, which acts as predicate for *conditionalSum()*:

```java
public class Algorithms {
    public static boolean isPositive(double d) {
        return 0 <= d;
    }
}
```

  - Then we pass the *List<Double> numbers* as argument and a method reference to *Algorithms.isPositive()* as argument:

```java
List<Double> numbers = List.of(10., -10., 20., -20., 30., -30.);
double result = conditionalSum(numbers, Algorithms::isPositive);
// result = 60.
```

- <u>Functions</u>, which <u>accept other functions as arguments</u> or <u>return other functions</u> are called <u>higher order functions</u>.
  - <u>I.e. the method *conditionalSum()* is a higher order function.</u>
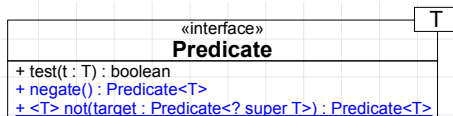
33

Higher Order Functions – Negating Predicates

- If we want to have only negative values of the numbers summed up, we can just pass a predicate testing for negativity.
  - Therefor we introduce a new method *Algorithms.isNegative()*:

```java
public class Algorithms {
    public static boolean isPositive(double d) {
        return 0 <= d;
    }
    public static boolean isNegative(double d) {
        return !isPositive(d);
    }
}
```

  - … and pass a reference to *Algorithms.isNegative()* to *conditionalSum()*:

```java
List<Double> numbers = List.of(10., -10., 20., -20., 30., -30.);
double result = conditionalSum(numbers, Algorithms::isNegative);
// result = -60.
```

- Alternatively, the target type *Predicate<T>* offers the default method *Predicate<T>.negate()* to reverse a predicate:

```
              T
┌─────────────────────────────┐
│         «interface»         │
│         Predicate           │
├─────────────────────────────┤
│ + test(t : T) : boolean     │
│ + negate() : Predicate<T>   │
│ + <T> not(target : Predicate<? super T>) : Predicate<T> │
└─────────────────────────────┘
```

```java
Predicate<Double> thePredicate = Algorithms::isPositive;
double result2 = conditionalSum(numbers, thePredicate.negate());
// result2 = -60.
```

  - And if we have no *Predicate* object at hand, but the target method is "predicative", we can use the simple factory *Predicate.not()*:

```java
double result3 = conditionalSum(numbers, Predicate.not(Algorithms::isPositive));
// result3 = -60.
```

34

Higher Order Functions with Lambdas

- When we apply higher order functions, and <u>with Java's *Stream* API we'll apply them very often</u>, <u>usually lambdas are used</u>:

```
List<Double> numbers = List.of(10., -10., 20., -20., 30., -30.);
double positiveResult = conditionalSum(numbers, d -> 0 <= d);
// positiveResult = 60.
double negativeResult = conditionalSum(numbers, d -> 0 > d);
// negativeResult = -60.
```

 - Mind, that the methods *Algorithms.isPositive()* and *Algorithms.isNegative()* are really simple, <u>writing them "in place" is "ok"</u>.

 - Thus, <u>using lambdas actually negates the point of the presence of methods as means to support the DRY principle</u>!

Partial Function Application – Part I

- Assume the method *Algorithms.add(int, int)*, it just adds two ints:

```
public class Algorithms {
    public static int add(int x, int y) {
        return x + y;
    }
}
```

- Often, not only in fp, it is required, that a function must be applied again and again with only some constant arguments.
  - E.g. assume we need to add 10 to other numbers over and over:

```
int forty = add(10, 30); int result1 = add(10, fromUser());
int result2 = add(10, x); int result3 = add(10, fromDatabase());
// ...
```

- Partial (function) application means to slash a function having multiple parameters into a function having less parameters.
  - This is done by filling some parameters with constant arguments and returning a new function with those bound arguments.

- We can create a new function binding a constant to calling *add()* offering only one argument to be added to the constant.
  - Then we can create a new function, that binds the 10 to a new function offering only one argument to be added to 10:

```
Function<Integer, Integer> adder10 = y -> Algorithms.add(10, y); // Creates the function f(y) = 10 + y.
int forty = adder10.apply(30); // ... and called with the argument 30.
```

  - Sure we can create any other adder with different constants, e.g. an 1-incrementer:

```
Function<Integer, Integer> incrementer = y -> Algorithms.add(1, y); // Creates the function f(y) = 1 + y.
int five = incrementer.apply(4); // ... and called with the argument 4.
```

36

Partial Function Application – Part II

- Partial application is an example of higher order functions, which <u>accept and also return other functions</u>.
  - The function, which we have generated for partial application can also be written as "ordinary" method, instead as lambda:

```java
Function<Integer, Integer> adder10 = y -> Algorithms.add(10, y);
int forty = adder10.apply(30);
```

```java
public static <T> Function<T, T> applyFirstToFunction(T x, BiFunction<T, T, T> fun) {
    return y -> fun.apply(x, y);
}
```

```java
Function<Integer, Integer> adder10 = applyFirstToFunction(10, Algorithms::add);
int forty = adder10.apply(30);
```

  - The lambda variant calls *Algorithms.add()* explicitly, whereas *applyFirstToFunctions()* accepts a *BiFunction* as argument.
  - Both variants, only apply the argument 30 to the created *adder10*-function, the first argument is bound to the constant 10.
  - Therefor, *adder10* only <u>partially applies an open parameter to be specified by the caller,</u> <u>but the first argument is already applied</u>.

- Of course we can also partially apply functions with <u>more than one parameter</u>:

```java
public class Algorithms {
    public static int add3(int x, int y, int z) {
        return x + y + z;
    }
}
```

  - We can partially apply the constant 12, but leave the two remaining arguments for the caller to specify:

```java
BiFunction<Integer, Integer, Integer> biAdder = (x, y) -> Algorithms.add3(x, y, 12);
int result = biAdder.apply(34, 56);
```

  - We can partially apply an unknown int from the user and the constant 12, but leave only one argument for the caller to specify:

```java
Function<Integer, Integer> adder = x -> Algorithms.add3(x, fromUser(), 12);
int result2 = adder.apply(72);
```

37

Currying – Part I

- Fp also offers another way to handle <u>functions with many parameters</u>, so called <u>currying</u>.

- Currying means to <u>slash a single function having 2+ parameters</u> <u>into multiple functions having only one parameter each</u>.
  - E.g. we can slash the binary function *Algorithms.add()* into two unary functions like so:

    ```
    Function<Integer, Function<Integer, Integer>> curriedAdder = x -> y -> Algorithms.add(x, y); // (1)
    Function<Integer, Integer> adder10 = curriedAdder.apply(10); // (2)
    int forty = adder10.apply(30); // (3)
    ```

  - (1) The first function *curriedAdder* accepts an int and returns a unary function, that binds the accepted int to *Algorithms.add()*.

  - (2) The second function *adder10* is a result of calling the first function *curriedAdder*, having bound 10 as first argument.

  - In (3) the second function is called specifying the 30 as second argument, which yields the result of the call chain.

- We can use currying on any function with 2+ parameters, <u>it'll just result in more slashed functions</u>:

    ```
    Function<Integer, Function<Integer, Function<Integer, Integer>>> curriedAdder = x -> y -> z -> Algorithms.add3(x, y, z);
    Function<Integer, Function<Integer, Integer>> adder12 = curriedAdder.apply(12);
    Function<Integer, Integer> adder12_34 = adder12.apply(34);
    int hunderedTwo = adder12_34.apply(56);
    ```

Currying – Part II

- Java has no dedicated currying-syntax. It is simulated by creating cascaded lambdas from a call to the original function:

  `x -> y -> z -> Algorithms.add3(x, y, z)`

  - The complexity isn't in the syntax of how such lambdas are created, but the cascaded target types get complex:

    `x -> y -> z -> Algorithms.add3(x, y, z)`

    `Function<Integer, Function<Integer, Function<Integer, Integer> > >`

- The notation of the lambda calculus directly supports currying and needs no special syntax:

  $$\lambda\,x.\lambda\,y.\lambda\,z.(f(x,y,z))$$

- Currying was named after the American mathematician Haskell Brooks Curry.

  - The fp language Haskell is also named after Mr. Curry.

  - Sometimes currying is also called "schönfinkeln" named after the Russian mathematician MosesI Iyich Schönfinkel.

- <u>Effectively, partial application means to use a curried function without the need to specify all the arguments.</u>

- Fp languages often provide extra syntax for partial function application and currying.

  - Some fp languages call functions returning other (created) functions <u>function generators</u>.

Applicability of functional Programming

- Programming is per se no mathematics! Do not try to find mathematical constructs by misinterpreting syntax!
  - The classic: Java's =-operator does an assignment and is no comparison or assertion.

- But functional programming is near to mathematics, it can be used to cast complex mathematical tools into code:
  - Definition of grammars
  - Definition of heuristics
  - Pattern matching
  - Nowadays, those tools gain more and more importance, e.g. to implement computer linguistics and artificial intelligence.

- Besides more or less scientific applications, fp is also interesting in other areas, esp. parallel programming:
  - Traditionally, it is hard to parallelize algorithms, because:
    - (1) They often access and modify data concurrently.
    - (2) Their inner workings is mostly based on side effects, i.e. they must be executed in a certain order.
  - Pure functional algorithms can be parallelized much simpler, because:
    - (1) They have no side effects, but only produce data, so there cannot be any concurrent modification effect.
    - (2) If no side effects occur, the execution order of "steps" in the algorithm does not matter.

- Modern languages are getting extended with syntax elements for functional programming, esp. lambdas.

- Other fp tools, which are getting added into modern languages: tuples and pattern matching.

Thank you!