Call by reference

Call by reference bedeutet, dass der Funktion / Methode die Parameter beim Aufruf als Referenz auf die Daten übergeben werden. D. h. im Umkehrschluss: wenn der Inhalt der Parameter verändert wird, dann ändern die nicht nur innerhalb des Kontexts der Funktion, sondern dauerhaft (auch ausserhalb der Funktion). Dies kann zu schwer zu lokalisierenden Seiteneffekten führen. Deshalb wird in der funktionalen Programmierung "call by value" bevorzugt.

Call by value

Call by value bedeutet, dass der Funktion / Methode die Parameter beim Aufruf als Kopie der Originaldaten übergeben werden. D. h. wenn innerhalb der Funktion / Methode die Daten der Parameter angepasst werden, hat dies keine Auswirkungen / Seiteneffekte auf Daten die Ausserhalb der Funktion / Methode liegen. Deshalb sollten in der funktionalen Programmierung die Funktionen / Methoden immer mit "call by value" aufgerufen werden.

Eager evaluation

Eager evaluation ist das Gegenteil von lazy evaluation. Ausdrücke werden während dem Sie definiert werden auch sofort ausgewertet (und nicht erst dann, wenn sie tatsächlich gebraucht werden).

Impure functions

Ist das Gegenteil von pure functions - d. h. wenn eine Funktion eine oder mehrere der drei Regeln für pure functions nicht erfüllt.

Lazy evaluation

Bei lazy evaluation geht es darum, dass Ausdrücke im Code erst dann ausgewertet werden, wenn sie wirklich gebraucht werden. Dies kann helfen die Performance zu steigern, weil nicht immer alle Teile eines Programms durchlaufen werden und so rechenarbeit gespart werden kann. Das Gegenteil von lazy evaluation ist eager evaluation.

Weitere Details

Pure functions

Pure functions sind Funktionen die drei Regeln erfüllen:

- Nur ein Rückgabewert
- Rückgabewert nur abhängig von den Aufrufparametern
- Verändert keine existierenden Werte

Referenzielle Transparenz

Man spricht dann von referenzieller Transparenz einer Funktion, wenn anstelle des Funktionsaufrufs auch nur der Return-Wert im Code angegeben werden könnte und sich das Programm dennoch exakt gleich verhält. Ein Beispiel:

```
val mySum = (x: Int, y: Int) => x + y
println("Sum of 4+5: " + mySum(4, 5))  // Ausgabe: Sum of 4+5: 9
println("Sum of 4+5: " + 9)  // Ausgabe: Sum of 4+5: 9
```

Eine Funktion, die **nicht** referenziell transparent ist, wäre beispielsweise today() die das Datum von heute zurückgibt. Wenn nun morgen ist, dann würde die Funktion beim gleichen Aufruf einen anderen Rückgabewert haben. D. h. der Rückgabewert hängt nicht alleine von den Input-Parametern ab.

Rekursion

Von Rekursion oder rekursiver Funktion wird gesprochen, wenn eine Funktion sich selbst aufruft. Damit das nicht in Endlosschleifen endet, braucht jede rekursive Funktion ein Abbruchkriterium - also ein Fall in dem die Funktion sich selbst nicht mehr weiter aufruft.

Imperativ vs Deklarativ Programmieren

In diesem Modul geht es darum, das imperative Programmieren vom deklarativem Programmieren zu unterscheiden. Wir sprechen in diesem Zusammenhang auch von einem Paradigmenwechsel: Wenn wir imperativ programmieren, dann geben wir die einzelnen Schritte im Programm vor. Wir beschreiben somit das WIE. Beispiel: Wir wollen ein Programm schreiben, welches uns eine Punktzahl aus einem Wort ermittelt. Wenn wir imperativ programmieren, würde das so aussehen:

```
public static int calculateScore(String word){
        int score = 0;
        for (char c : word.toCharArray()) {
                score++;
        }
        return score;
}
```

Wir initialisieren score mit dem Wert 0. Dann wird mittels Schleife jedes Zeichen im String durchgezählt und zu score addiert. Wir haben somit einzelne Schritte definiert – **das WIE im Programm.**

Ganz anders, wenn wir deklarativ programmieren. Uns interessiert nicht das WIE, sondern das WAS:

```
public static int wordScore(String word){
        return word.length();
}
```

Wir wollen die Länge des Strings erhalten und kümmern uns nicht um die einzelnen Schritte, um dies zu erreichen. Wir können die *length-Methode* in Java verwenden und müssen uns nicht um einzelne Schritte kümmern, um das zu berechnen. Ein Detail: wir haben auch den Namen der Funktion geändert – also kein Verb, sondern nur ein Nomen. Uns interessiert nur das WAS – also den «wordScore» und nicht das WIE («calculate» hat schon eher den Geschmack des WIE programmieren).

Anforderungen sollen das WAS beschreiben

Wir haben kennengelernt, dass in der funktionalen Programmierung nicht das WIE im Vordergrund steht, sondern das WAS. Nochmals zur Wiederholung: beim WIE beschreiben wir, wie wir ein Problem in Code programmieren. Wir beschreiben die einzelnen Schritte, um das Problem zu lösen. Wenn wir nur das WAS beschreiben, dann ist das deklarativ. Wir beschreiben das Endergebnis. Wir können dann daraus die entsprechenden Funktionen ableiten.

Dazu ein einfaches Beispiel:

Ich möchte ein Programm, welches 5% Rabatt vom Totalbetrag berechnet.

Dies ist eine Anforderung, die nicht beschreibt, WIE wir etwas umsetzen sollen, sondern nur das WAS. Wir können aus dieser Beschreibung eine Funktion ableiten:

```
f(x) = x * 95 / 100
```

Für jeden Preis x wird uns diese Funktion den korrekten Preis abzgl. Rabatt berechnen.

Merkmale einer pure function

Wenn alle die folgenden Eigenschaften erfüllt sind, ist eine Funktion eine sogenannte "pure function":

1. Gibt nur einen Wert zurück (Wenn die Funktion einen Array zurückgibt, gilt das dennoch als ein Wert - nämlich ein Array)
2. Berechnet den Rückgabewert nur aufgrund der ihr übergebenen Parameter
3. Verändert keine existierenden Werte (neue in der Funktion selber definierte Werte dürfen verändert werden)

Weil sich eine pure function an diese drei Regeln hält, ist sie referenziell Transparent - d. h. Sie könnte auch durch Ihren Return-Wert ersetzt werden, ohne dass sich die Funktionalität des Programmes ändert.

Beim 2. Punkt ist wichtig zu beachten: Wenn Sie beispielsweise die random()-Funktion verwenden, und die Zufallszahl mit dem Parameter verrechnen, dann ist die Regel bereits verletzt, weil das Resultat nicht nur von den Parametern abhängt. Sogar wenn innerhalb der Funktion "nur" mit println eine Nachricht ausgegeben wird, macht dies die Funktion "impure" weil der globale Zustand des Output-Buffers dadurch verändert wird. D. h. im Umkehrschluss: in einer pure function sind Ihnen gewisse Begrenzungen auferlegt - Sie können also nicht jede beliebige Logik mit pure functions umsetzen. Das Ziel von functional programming ist es aber so viel wie möglich pure umzusetzen, weil die Vorteile die pure functions bieten, auf der Hand liegen und die Nachteile überwiegen.

Beim 3. Punkt (keine existierenden Werte verändern) existieren die folgenden Ansätze, um diese Forderung zu erfüllen:

- **Kopie der Daten:** Die ursprünglichen Daten (Original) wird unverändert gelassen. Es wird eine Kopie der Daten erstellt, die dann verändert wird.
- **Rekursion:** Die Methode wird rekursiv aufgerufen und mit jedem neuen Aufruf werden neue "Kopien" der lokalen Variablen angelegt.

Die Anforderung keine existierenden Werte zu verändern ist eines der Kernthemen bei der funktionalen Programmierung und wird im Unterkapitel Immutable Values vertieft angeschaut.

Vorteile

- **Keine Seiteneffekte:** Bei der Verwendung einer pure function treten keine Seiteneffekte auf. Sie könnten - ohne die Programmlogik zu verändern - auch den Rückgabewert anstelle des Funktionsaufrufs in den Code packen (referenziell Transparent). D. h. der Funktionsaufruf löst nichts Unerwartetes bzw. keine Statusänderung aus.
- **Einfach zu testen:** Weil die pure function nur von den Parametern abhängt, reicht der Funktionsaufruf, um diese zu testen. Bei Funktionen die auf globale Werte zugreifen und diese verwenden müssen diese globalen Werte für einen Test der Funktion zuerst noch mit sinnvollen Werten instanziiert werden.
- **Code wird klarer:** Weil keine Seiteneffekte und globalen Werte für das Codeverständnis wichtig sind, wird der Code viel einfacher zu lesen und ist sehr viel verständlicher.

Nachteile

- **Performance:** Die vielen Kopien der Daten brauchen einerseits Speicherplatz andererseits aber auch Zeit für den Kopiervorgang. Das führt dazu, dass die Ausführung des Programmcodes in der Regel weniger performant sein wird. Um dem entgegenzuwirken, gibt es das Konzept der lazy evaluation, wo Ausdrücke erst dann ausgewertet werden, wenn diese auch tatsächlich benötigt werden.
- **Rekursion:** Rekursive Funktionen können nicht ganz so intuitiv zu verstehen sein. Wenn der Punkt keine existierenden Werte zu veränder mit rekursiven Funktionen gelöst wird, kann das die Verständlichkeit des Codes auch wieder etwas erschweren. Zudem kann eine sehr tiefe Rekursion dazu führen, dass der Verbrauch des Zwischenspeichers massiv anwächst und erst wieder freigegeben werden kann, wenn das Abbruchkriterium für die rekursive Funktion erreicht ist. Rekursion sollte entsprechend nur dort eingesetzt werden, wo es auch wirklich sinnvoll ist (beispielsweise dort wo Sie in der klassischen Programmierung mit while- oder for-Loops durch eine Liste von Elementen iterieren).

Rekursion

Von Rekursion wird gesprochen, wenn eine Funktion sich selbst innerhalb des Funktionsbody aufruft. Damit keine Endlos-Loops entstehen, braucht jede rekursive Funktion ein Abbruchkriterium. Wenn die Rekursion zu tief verschachtelt ist, kann die Applikation zudem bei einer Speicherlimite anstossen, weil jede Stufe der Rekursion wieder ein komplettes Set an Daten enthält. Wenn die einzelnen Datensätze nun sehr gross sind und es sehr viele sind, dann sind schnell die dem Prozess zugewiesenen Ressourcen an Zwischenspeicher ausgeschöpft.

Bei der Verwendung von rekursiven Funktionen ist auf folgendes zu achten:

- **Abbruchkriterium:** Fehlt das Abbruchkriterium oder ist es vorhanden, aber kann niemals eintreten, dann führt dies zu einem Endlos-Loop.
- **Grösse der Daten:** Für ganz grosse Datenmengen ist Rekursion unter Umständen ungeeignet. Grosse Listen können allenfalls in Portionen aufgeteilt werden, um die Thematik etwas zu entschärfen.

Beispiele

Um Beispielsweise die Fakultät einer Zahl zu berechnen, ist eine rekursive Funktion geeignet.

Zur Erinnerung hier einige Beispiele einer Fakultät:

- 5! = 5 * 4! = 5 * 4 * 3 * 2 * 1 = 120
- 4! = 4 * 3! = 4 * 3 * 2 * 1 = 24
- 3! = 3 * 2! = 3 * 2 * 1 = 6
- 2! = 2 * 1! = 2 * 1 = 2
- 1! = 1 * 0! = 1
- 0! = 1 -> Abbruchkriterium

Eine Funktion, die die Fakultät berechnen kann, könnte folglich wie folgt aussehen:

In JavaScript:

```
// Rekursive Funktion zur Berechnung der Fakultät einer Zahl
function factorial(n) {
    // Abbruchkriterium: Bei n = 0, ist die Fakultät 1
    if (n === 0) { // sollte diese Bedingung nie wahr werden führt das zum Endlos-Loop
        // hier endet die Rekursion
        return 1;
    } else {
        // Rekursiver Fall: n! = n * (n-1)!
        return n * factorial(n - 1);
    }
}

// Beispielaufruf der Funktion
console.log(factorial(5)); // Ausgabe: 120 (5! = 5 * 4 * 3 * 2 * 1 = 120)
```

		richtig	falsch
a)	Die funktionale Programmierung hat den Vorteil, dass der Code performanter ist in der Ausführung, weil der Code tendenziell kürzer wird wie in anderen Programmierparadigmen.	( ) -0.25	(×) 0.5
b)	Unit-Tests lassen sich beim funktionalen Programmieren einfacher realisieren, weil bei der funktionalen Programmierung immutable Values verwendet werden, was dazu führt, dass keine zusätzlichen Umgebungsvariablen für einen Testcase gesetzt werden müssen.	(×) 0.5	( ) -0.25
c)	Ein Vorteil der funktionalen Programmierung ist, dass der Code sehr klar und einfach verständlich wird (natürlich nur dann wenn sich die Entwickler auch an die Regeln der funktionalen Programmierung konsequent halten).	(×) 0.5	( ) -0.25
d)	Die funktionale Programmierung bewirkt, dass der Grad der Abstraktion des Codes tendenziell grösser wird. Es wird auf einer "Metaebene" programmiert. Das WAS steht im Zentrum und nicht das WIE.	( ) 0.5	(×) -0.25

Welche Aussagen in Bezug auf die Fachbegriffe imperativ und deklarativ sind **richtig**, welche Aussagen sind **falsch**?

		richtig	falsch
a)	Die Wahl des Funktionsnamens sollte in der deklarativen Programmierung vor allem Verben enthalten (der Funktionsname soll die Aktionen die die Funktion ausführt erklären).	( ) -0.25	( ) 0.5
b)	Bei der imperativen Programmierung wird der Fokus darauf gelegt, was getan werden muss (und nicht wie es getan werden muss).	(×) -0.25	( ) 0.5
c)	In der funktionalen Programmierung soll der Code deklarativ umgesetzt werden.	( ) 0.5	( ) -0.25
d)	Die Anforderung "Berechne den Wordscore, indem du Buchstaben für Buchstaben im Wort durchgehst und den Zähler für den Wordscore bei jedem Buchstaben der kein a ist um eines erhöhst" ist eine deklarative Anforderung.	( ) -0.25	( ) 0.5
e)	Die Anforderung "Der Wordscore ist die Länge eines Wortes ohne die Buchstaben a mit zu berücksichtigen" ist eine deklarative Anforderung.	( ) 0.5	( ) -0.25

		richtig	falsch
a)	Die Regel, dass eine pure function immer nur einen Rückgabewert haben darf bedeutet, wenn ein Array mit mehreren unterschiedlichen Werten (int, String, Objekt) zurück gegeben wird, dann ist diese Regel verletzt.	(×) -0.25	( ) 0.5
b)	Wenn eine Funktion neben dem Normalen Return-Wert zusätzlich eine Exception werfen kann, dann ist diese Funktion impure weil die Regel "Nur ein Rückgabewert" verletzt wird.	( ) 0.5	(×) -0.25
c)	Eine pure function ist immer auch referenziell transparent.	( ) 0.5	(×) -0.25
d)	Innerhalb des Funktionsbodies dürfen Variablen definiert und auch verändert werden und dennoch gilt die Funktion als pure.	(×) 0.5	( ) -0.25
e)	Eine pure function verhindert nicht in jedem Fall Seiteneffekte. Beispielsweise könnte mit einer debugging-Ausgabe innerhalb der pure function ein Seiteneffekt erzeugt weden.	( ) -0.25	(×) 0.5
f)	Dank der eager evaluation kann die Performance von pure functions etwas optimiert werden.	( ) -0.25	( ) 0.5
g)	Mit call by reference kann dafür gesorgt werden, dass eine Kopie der Daten der Methode übergeben werden. D. h. die Parameter dürfen innerhalb der Methode direkt verändert werden ohne dass eine der Regeln für pure functions verletzt wäre.	( ) -0.25	(×) 0.5

-

		richtig	falsch
a)	Bei der Funktion getBiggestScaleFactor handelt es sich um eine pure function, weil alle Regeln für pure functions eingehalten werden.	( ) -0.25	(×) 0.5
b)	Die Funktion getBiggestScaleFactor verletzt die Regel, dass nur ein Return-Value vorhanden sein darf und ist deshalb impure	(×) 0.5	( ) -0.25
c)	Die Funktion ist nur pure weil die beiden Variablen widthScaleFactor und heightScaleFactor als const definiert wurden.	( ) -0.25	(×) 0.5

		richtig	falsch
a)	Call by value bedeutet, dass bei einem Funktionsaufruf der Parameter beim Aufruf kopiert wird und die Kopie der Funktion als Parameter übergeben wird.	(×) 0.5	( ) -0.25
b)	Hinter dem Fachbegriff immutable value steht das Konzept, dass keine bestehenden Daten verändert werden dürfen. Dies kann Seiteneffekte im Code vermeiden.	(×) 0.5	( ) -0.25
c)	Call by value ist ausreichend um dafür zu sorgen, dass innerhalb der Funktion keine bestehenden Werte verändert werden.	( ) -0.25	(×) 0.5
d)	Wenn eine rekursive Funktion eine globale Variable verändert, dann ist das Konzept von immutable values verletzt obwohl Rekursion verwendet werden kann, um das Konzept von immutable values umzusetzen.	(×) 0.5	( ) -0.25

		richtig	falsch
a)	Diese Funktion ist impure, weil sie zwei return-Statements beinhaltet.	( ) -0.25	(×) 0.5
b)	Diese Funktion berechnet beim Beispielsaufruf 6*4*2*0.	( ) -0.25	(×) 0.5
c)	Das Abbruch-Kriterium der rekursiven Funktion befindet sich auf Zeile 3 und der Abbruch der Rekursion auf Zeile 4.	(×) 0.5	( ) -0.25
d)	Zeile 8 macht aus der Funktion eine rekursive Funktion.	(×) 0.5	( ) -0.25
e)	Diese Funktion ist impure, weil das Argument auf Zeile 7 verändert wird.	( ) -0.25	(×) 0.5
f)	Diese Funktion bewirkt beim Aufruf factorial(5) einen Endlosloop.	( ) 0.5	(×) -0.25