codecentric Blog (https://blog.codecentric.de/)

Overview (https://blog.codecentric.de/en/category/architecture/)

Perl – The Ultimate Programming Language :-) (https://blog.codecentric.de/en/2015/09/perl-the-ultimate-programming-language/)

The Essence of Object-Functional Programming and the Practical Potential of Scala (https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/)

08/31/15 by Martin Lau

1 Comment (https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/#comments)

The terms “object-functional” and “object-functional programming” are heard time and again in the context of software development. But what does the object-functional approach look like and what advantages does it have? Isn’t object-orientation or the functional approach good enough all by itself? And what does Scala have to do with all of this?

Object-Functional

Nontrivial software has to realize two types of behavior: It has to compute stuff and to effect stuff. For instance, a software system could compute what should be shown next on the display and then show it on the display. The latter can be called the production of an "effect". Often, no distinction is made between different types of effects, calling all effects "side effects". Yet, it is sensible to make a distinction between "essential effects" and "side effects". Essential effects are part of the software's requirements and can't be eliminated: If something is to be displayed on screen, there's no way around displaying it there. In contrast, side effects are effects that aren't really needed, aren't interesting or are even undesirable and only appear as an artifact of the behavior really wanted; for instance, imperative programming makes intensive use of such side effects as a means of computation.

Object-oriented programming is traditionally a rather imperative endeavor and the mutation of objects and object references, and thus side effects, are there with us all the time. Functional programming, though, emphasizes the transformation of values: Values are normally used to compute other values, but values and value references aren't mutated in doing so.

What do we get, if the transformed values are objects? This brings us to one possible definition of object-functional programming:

> ***Object-functional programming*** *is a programming approach emphasizing the transformation of objects and using object-oriented as well as functional means and principles to this end. The production of effects in general as well the mutation of objects and object references in particular happen only sparingly, as transparent as possible to other parts of the code and separated as much as possible from transformational code.*

With object-functional programming, object-oriented means are used to put software into a proper shape, fostering the code's local changeability and understandability; data and behavior are brought together; access modifiers are used to allow sensible dependencies only, if at all possible. Functional features and many features traditionally provided more often by functional languages are used to forgo mutation of object and objects references as extensively as possible and for having an easy time staying modular even in the small, e. g. with functions as a lightweight alternative to the Strategy Pattern. As customary for functional programming, software is separated into code mostly free of effects and code mostly producing effects, with effects being as transparent to other parts of the code as possible; that way, large parts of the code aren't affected at all by the disadvantages that effects involve, e. g. for local understandability and modular composability.

Unlike with object-orientation, objects are normally, at least looking from the outside, immutable; methods usually compute, but don't produce observable effects. Unlike with functional programming, functions work less on passive data structures but often occur in the form of methods belonging to objects or object types and having access to their non-public data and behavior.

Appropriate mechanisms are used to realize essential effects, e. g. to display data on screen; these mechanisms can turn out very differently.

Why Object-Functional?

Software development isn't about implementing wanted behavior in some "arbitrary" way. The software's development and further enhancement should also be efficient and the results should be of high quality. Both object-oriented programming as well as functional programming support this goal to a certain degree, each in its own way.

With object-orientation, its traditionally imperative focus is problematic, though. Positive attributes this approach fosters on the one hand are thwarted on the other hand by its traditionally imperative focus, especially local understandability and modular composability. To add, mutable objects aren't readily usable in a safe way in a parallel context; and pure object-orientation is often too heavyweight in the small. The functional approach, in contrast, shines with lightweight features as well as its limitation and isolation of effects.

The functional approach again isn't as strongly equipped with means for structuring software systems, abstracting data and inhibiting unwanted dependencies. In contrast, the object-oriented approach with its objects and related features offers effective responses.

Hence, strengths of one approach are weaknesses of the other. Combining both approaches also leads to a combination of their strengths and compensates for their respective weaknesses. We get, potentially, the advantages of both without their disadvantages. To what extent this potential can be realized also depends on the language used.

Scala

Scala is a programming language that was designed as an object-functional composition from the start. Both in respect to object-orientation and to the functional approach it goes far beyond providing just the basics:

- **Object-oriented:** Besides classes and (single) inheritance that are obligatory for a class based language, there are many more language features and language details that make it easier to modularize software. For instance, with traits there is a powerful type of class composition. Not only do traits allow some kind of classical mixins, they also allow to build up classes by a kind of nesting related to the Decorator Pattern.

Implicit classes enable objects to be automatically wrapped on demand at runtime for using methods and other features the original objects doesn't even have. This allows the ad hoc realization of different extended views on objects; totally transparent to the original class and their instantiation and without the danger of global conflicts caused by contradicting views. And Scala provides a lot more regarding object-orientation.

- **Functional:** Besides providing the obligatory first class functions and closures, Scala is characterized by many more language features and details that make it easy to extensively do without mutating objects and object references. These include, for instance, pattern matching and "for comprehensions". With functions, closures and tuples we also have, in the small, a lightweight alternative to defining custom classes. As with any functional language, to be of practical use, there has to be some way to directly or indirectly produce essential effects. For this purpose, Scala pragmatically provides direct means well known from imperative programming, e. g. assignments.

In doing so, Scala's object-oriented and functional features are crafted in a way compatible with the original approaches and making their integrated use practical and worthwhile. For instance, not only can objects be pattern matched functionally; this can even be done in an object-oriented way without breaking the principle of Information Hiding; and, for instance, object-oriented methods can not only be interpreted as a special case of functions conceptually, but can they can often also easily be used as such technically. Scala's imperative parts can also be used to, in some cases, make code more efficient and clearer; used sparingly, as transparent as possible to other parts of the code and separated from non-transformative code as much as possible, the imperative parts enhance the practicability of the object-functional approach instead of running counter to it. All of this is completed by compact syntax and advanced static typing that allow object-functional code to normally be formulated in a very short, clear and yet statically type safe way.

But what about the standard library? All of this would be nice, but worthy of improvement, if the standard library lacked fundamental object-functional abstractions, in particular immutable collections. Even here, we won't be disappointed. The standard library provides important object-functional base abstractions, in particular immutable collections; and other Scala APIs usually expect and return instances of these base abstractions.

Conclusion

Object-functional programming is the natural combination of two compatible approaches to programming. This combined approach brings together the advantages of both original approaches, avoiding their disadvantages. In the end, this leads to potentially higher productivity and software quality.

To what extent this can be realized in practice depends, amongst other things, on the language used. Scala is a programming language where the fusion of the object-oriented and functional approach was the guiding principle. Accordingly well, the language is suited for object-functional programming.

It is therefore worthwhile to go more deeply into object-functional programming in general and Scala in particular. And with all details and temporary imponderabilities this entails, to keep in mind what the object-functional approach is all about: Realizing higher productivity and quality by focussing on object transformations.

Tags

SCALA (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/SCALA-EN/)

Martin Lau (https://blog.codecentric.de/en/author/martin-lau/)

(http://www.facebook.com/sharer.php?u=https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/)

(http://twitter.com/share?url=https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/&text=The+Essence+of+Object-Functional+Programming+and+the+Practical+Potential+of+Scala)


(https://www.linkedin.com/shareArticle?mini=true&url=https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/)


(http://reddit.com/submit?url=https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/&title=The+Essence+of+Object-Functional+Programming+and+the+Practical+Potential+of+Scala)

Post by **Martin Lau**

SCALA

Die Essenz objektfunktionaler Programmierung und das praktische Potential von Scala (https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/)

More content about **Architecture**

ARCHITECTURE

Offloading and more from Reedelk Data Integration Services through Kong Enterprise (https://blog.codecentric.de/en/2020/09/offloading-and-more-from-reedelk-data-integration-services-through-kong-enterprise/)

ARCHITECTURE

API as a Product insights (https://blog.codecentric.de/en/2020/09/api-as-a-product-insights/)

Kommentare

9. September 2015 (https://blog.codecentric.de/en/2015/08/essence-of-object-functional-programming-practical-potential-of-scala/#comment-374900) von Javin Paul

Great article, also new blog template looks quite nice 🙂 cheers

Reply

Comment

Nachricht

Name

E-Mail

Save my name, email, and website in this browser for the next time I comment.

SUBMIT

(https://www.facebook.com/codecentric) (https://twitter.com/codecentric)