Marcin Moskała

Efektywny Kotlin

NAJLEPSZE PRAKTYKI

Helion

Kt. Academy

Tytuł oryginału: Effective Kotlin: Best Practices

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-6799-9

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/efkonp
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Rozdział 2. Czytelność

> *Każdy głupi potrafi napisać kod zrozumiały dla komputerów.*
> *Dobrzy programiści piszą kod zrozumiały dla ludzi.*
>
> — Martin Fowler, *Refactoring: Improving the Design of Existing Code*[1], s. 15

Powszechnie przyjmowane jest błędne przekonanie, że Kotlin został zaprojektowany z myślą o zwięzłości. To nieprawda. Istnieją dużo bardziej zwięzłe języki. Najbardziej zwięzłym językiem, jaki znam, jest APL. Poniżej przedstawiona jest „gra w życie" Johna Conwaya zaimplementowana w tym języku:

```
Life←{ 1 ωV.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω}
```

Twoją pierwszą myślą może być: „Nieźle, to naprawdę zwięzłe". Potem możesz zdać sobie sprawę, że niektóre z tych znaków nie są dostępne na klawiaturze. Istnieje więcej takich języków. Na przykład oto ten sam program w języku J:

```
1 life=:[:+/(3 4=/[:+/(,/,"0/~i:1)|.])*.1,:]
```

Te dwa języki są naprawdę zwięzłe. Ta cecha sprawia, że są bardzo przydatne w konkursach dla programistów. Jednocześnie powoduje to, że są absurdalnie nieczytelne. Szczerze mówiąc, nawet doświadczonym programistom używającym APL-a (a zapewne jest ich na świecie tylko kilku) trudno jest zrozumieć, co ten program robi i jak działa.

Twórcy Kotlina nigdy nie dążyli do tego, by był on bardzo zwięzły. Język ten został zaprojektowany tak, aby był **czytelny**. Wprawdzie jest zwięzły w porównaniu z innymi popularnymi językami, ale wynika to z tego, że wyeliminowano w nim dużo szumu: szablonowego kodu i powtarzających się struktur. Opracowano go tak, aby pomagał programistom skupić się na tym, co ważne. Dzięki temu Kotlin jest bardziej czytelny.

[1] Wydanie polskie: *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, Helion, 2017 — *przyp. tłum.*

Kotlin umożliwia programistom projektowanie przejrzystego i sensownego kodu oraz API. Mechanizmy tego języka pozwalają ukrywać i uwydatniać to, na czym nam zależy. Ten rozdział dotyczy mądrego korzystania z takich narzędzi. Stanowi on wprowadzenie do tego tematu i zawiera zestaw ogólnych wskazówek. Jest też wprowadzeniem do kwestii czytelności, do której będę wracać w dalszych fragmentach książki, a przede wszystkim w części II „Projektowanie kodu", gdzie omawiam zagadnienia związane z projektowaniem klas i funkcji.

Zaczynam tu od nieco abstrakcyjnego zagadnienia związanego z czytelnością, które posłuży jako wprowadzenie do ogólnego problemu.

Temat 11. Projektuj z myślą o czytelności

W świecie programowania zaobserwowano, że programiści znacznie dłużej czytają kod, niż go piszą. Szacuje się, że każdej minucie przeznaczonej na pisanie kodu odpowiada dziesięć minut spędzonych na jego czytaniu (ten stosunek spopularyzował Robert C. Martin w książce *Clean Code*[2]). Jeśli wydaje Ci się to niewiarygodne, pomyśl tylko o tym, ile czasu zajmuje Ci lektura kodu, gdy próbujesz znaleźć błąd. Podejrzewam, że każdemu przynajmniej raz w ciągu kariery zdarzyło się przez tygodnie szukać błędu, a następnie naprawić go, zmieniając jeden wiersz kodu. Gdy uczysz się używać nowego API, często w tym celu czytasz kod. Zwykle czytamy kod, aby zrozumieć logikę lub działanie implementacji. **Programowanie polega przede wszystkim na czytaniu, a nie na pisaniu**. Dlatego powinno być jasne, że należy pisać kod z myślą o jego czytelności.

Zmniejszanie obciążenia poznawczego

Czytelność dla każdego może oznaczać coś innego. Istnieją jednak pewne reguły wynikające z doświadczenia i nauk poznawczych. Porównaj dwie poniższe implementacje:

```
 1 // Implementacja A
 2 if (person != null && person.isAdult) {
 3     view.showPerson(person)
 4 } else {
 5     view.showError()
 6 }
 7
 8 // Implementacja B
 9 person?.takeIf { it.isAdult }
10     ?.let(view::showPerson)
11     ?: view.showError()
```

Która z nich jest lepsza, A czy B? Naiwne stwierdzenie, że wersja o mniejszej liczbie wierszy jest lepsza, to zła odpowiedź. Równie dobrze można byłoby usunąć podziały wiersza z pierwszej implementacji, a nie stałaby się ona dzięki temu bardziej czytelna.

Czytelność obu rozwiązań wynika z tego, jak szybko można je zrozumieć. To z kolei zależy od tego, w jakim stopniu mózg jest wyćwiczony w rozpoznawaniu poszczególnych idiomów (struktur, funkcji, wzorców). Dla nowicjusza uczącego się Kotlina bardziej czytelna będzie implementacja A, gdzie używane są standardowe idiomy (`if/else`, `&&`, wywołania metod). W implementacji B znajdują się idiomy typowe dla Kotlina (bezpieczne wywołanie `?.`, `takeIf`, `let`, operator Elvis `?:`, powiązana referencja do funkcji `view::showPerson`).

[2] Wydanie polskie: *Czysty kod*, Helion, 2015 — *przyp. tłum.*

Wszystkie te idiomy są powszechnie stosowane w Kotlinie, dlatego są dobrze znane większości doświadczonych programistów używających tego języka. Mimo to trudno porównać je z ogólnymi idiomami. Dla większości programistów Kotlin nie jest pierwszym językiem. Mamy znacznie większe ogólne doświadczenie w programowaniu niż w używaniu Kotlina. Ponadto nie piszemy kodu tylko dla doświadczonych programistów. Możliwe, że nowicjusz, którego zatrudniłeś (po miesiącach bezowocnego szukania doświadczonego programisty), nie wie, czym są wywołania `let` i `takeIf` lub powiązane referencje, a także nigdy nie widział operatora Elvis zastosowanego w taki sposób. Taka osoba może spędzić cały dzień, starając się zrozumieć ten jeden blok kodu. Ponadto nawet doświadczeni programiści używający Kotlina posługują się nie tylko tym językiem. Wielu programistów czytających pisany przez Ciebie kod nie będzie biegłych w Kotlinie. Mózg zawsze będzie potrzebował trochę czasu na rozpoznanie idiomów specyficznych dla tego języka. Nawet po latach spędzonych na programowaniu w Kotlinie zrozumienie pierwszej implementacji zajmuje mi znacznie mniej czasu. Każdy mało znany idiom zwiększa złożoność kodu, a gdy analizujesz kilka takich idiomów w jednej instrukcji, którą musisz zrozumieć w całości, ta złożoność szybko rośnie.

Zauważ też, że implementację A łatwiej jest zmodyfikować. Załóżmy, że musisz dodać operację w bloku `if`. W implementacji A jest to proste. W implementacji B nie można wtedy użyć referencji do funkcji. Dodanie czegoś do bloku `else` w implementacji B jest jeszcze trudniejsze — trzeba zastosować funkcję, aby można było zapisać więcej niż jedno wyrażenie po prawej stronie operatora Elvis:

```
if (person != null && person.isAdult) {
    view.showPerson(person)
    view.hideProgressWithSuccess()
} else {
    view.showError()
    view.hideProgress()
}

person?.takeIf { it.isAdult }
    ?.let {
        view.showPerson(it)
        view.hideProgressWithSuccess()
    } ?: run {
        view.showError()
        view.hideProgress()
    }
```

Również debugowanie jest znacznie łatwiejsze w implementacji A. Nic dziwnego — narzędzia do debugowania opracowano z myślą o takich prostych strukturach.

Zgodnie z ogólną zasadą rzadziej stosowane i „pomysłowe” struktury dają zwykle mniej swobody i nie są dobrze obsługiwane. Załóżmy na przykład, że chcesz dodać trzecie odgałęzienie, aby wyświetlać inny błąd, gdy obiekt `person` to `null`, a inny, gdy obiekt `person` nie reprezentuje dorosłej osoby. W implementacji A używaną w niej strukturę `if/else` można łatwo zmienić w strukturę `when` za pomocą refaktoryzacji ze środowiska IntelliJ, a następnie dodać odgałęzienie. W implementacji B ta sama zmiana jest trudna i zapewne będzie wymagać napisania kodu od nowa.

Zauważyłeś, że implementacje A i B nie działają tak samo? Potrafisz dostrzec różnicę? Wróć teraz do kodu i zastanów się nad tym.

Różnica polega na tym, że wywołanie `let` zwraca wynik z wyrażenia lambda. To oznacza, że jeśli wywołanie `showPerson` zwróci `null`, druga implementacja wywoła `showError`! Jest to nieoczywiste, a wniosek z tego taki, że gdy używasz mniej popularnych struktur, łatwiej jest przeoczyć nieoczekiwane działanie kodu.

Ogólna reguła jest taka, aby ograniczać obciążenie poznawcze. Nasze mózgi rozpoznają wzorce i na podstawie tych wzorców budują wizję działania programów. Czytelny kod ma to ułatwiać. Preferowany jest krótszy kod, ale też lepiej znane struktury. Rozpoznajemy znajome wzorce, jeśli widzimy je wystarczająco często. W innych dziedzinach też zwykle preferujemy struktury, które znamy.

Nie posuwaj się do skrajności

W poprzednim przykładzie pokazałem, że wywołanie `let` może zostać użyte w niewłaściwy sposób. Nie oznacza to jednak, że nigdy nie należy go stosować. Jest to popularny idiom, sensownie używany w różnych kontekstach do poprawy jakości kodu. Przykładem poprawnego zastosowania `let` są modyfikowalne właściwości z dopuszczalną wartością `null`, gdzie operację należy wykonać tylko wtedy, gdy wartość jest różna od `null`. Nie można zastosować tu inteligentnego rzutowania, ponieważ modyfikowalna właściwość może zostać zmieniona przez inny wątek. Doskonałym rozwiązaniem tego problemu jest użycie bezpiecznego wywołania `let`:

```
1 class Person(val name: String)
2 var person: Person? = null
3
4 fun printName() {
5    person?.let {
6       print(it.name)
7    }
8 }
```

Ten idiom jest popularny i powszechnie rozpoznawalny. Istnieje też wiele innych sensownych zastosowań wywołań `let`. Oto przykład:

- przenoszenie operacji w miejsce, w którym jej argument jest już obliczony,
- opakowywanie obiektu za pomocą wzorca dekorator.

Oto przykłady ilustrujące te dwa scenariusze (w obu dodatkowo używane są referencje do funkcji):

```
 1 students
 2    .filter { it.result >= 50 }
 3    .joinToString(separator = "\n") {
 4       "${it.name} ${it.surname}, ${it.result}"
 5    }
 6    .let(::print)
 7
 8 var obj = FileInputStream("/file.gz")
 9    .let(::BufferedInputStream)
10    .let(::ZipInputStream)
11    .let(::ObjectInputStream)
12    .readObject() as SomeObject
```

We wszystkich tych sytuacjach musimy ponieść koszty — kod jest trudniejszy do debugowania i zrozumienia przez mniej doświadczonych programistów używających Kotlina. Jednak otrzymujemy coś w zamian i wydaje się, że płacimy za to uczciwą cenę. Problem pojawia się, gdy znacznie zwiększamy złożoność bez dobrego powodu.

Zawsze będą toczyły się dyskusje na temat tego, czy jakieś rozwiązanie ma sens. Znajdowanie równowagi jest sztuką. Warto jednak wiedzieć, w jaki sposób różne struktury zwiększają złożoność i jak pozwalają zwiększać przejrzystość kodu. Jest to ważne przede wszystkim dlatego, że gdy dwie struktury są używane razem, ich złożoność jest znacznie większa niż ich sumaryczna złożoność, gdy są stosowane niezależnie od siebie.

Konwencje

Wiesz już, że różne osoby mogą w inny sposób rozumieć, czym jest czytelność. Nieustannie spieramy się o nazwy funkcji, dyskutujemy, co należy zapisywać bezpośrednio, a co można wywnioskować, jakie idiomy zastosować itd. Programowanie jest sztuką wyrażania koncepcji. Istnieją jednak konwencje, które trzeba zrozumieć i zapamiętać.

Gdy uczestnik jednej z moich grup warsztatowych w San Francisco zapytał mnie o najgorszą rzecz, jaką można zrobić w Kotlinie, pokazałem następujący kod:

```
1 val abc = "A" { "B" } and "C"
2 print(abc) // ABC
```

Aby ta okropna składnia była dozwolona, wystarczy napisać taki kod:

```
1 operator fun String.invoke(f: ()->String): String =
2    this + f()
3
4 infix fun String.and(s: String) = this + s
```

Ten kod narusza wiele reguł, które omawiam dalej:

- Narusza znaczenie operatora. Operatora `invoke` nie należy stosować w ten sposób. Nie można wywoływać tak wartości typu `String`.
- Zastosowanie konwencji „lambda jako ostatni argument" jest tu mylące. Nie ma nic złego w używaniu lambdy po funkcjach, ale należy zachować ostrożność, gdy stosuje się tę technikę z operatorem `invoke`.
- Słowo `and` jest oczywiście złą nazwą dla tej infiksowej metody. Znacznie lepsza byłaby nazwa `append` lub `plus`.
- Istnieją już mechanizmy języka do złączania wartości typu `String`, dlatego należy z nich korzystać, zamiast po raz drugi wymyślać koło.

Z każdą z tych wskazówek związana jest bardziej ogólna reguła pomagająca zachować dobry styl w kodzie w Kotlinie. Najważniejsze z tych reguł omawiam w tym rozdziale. Zaczynam od tej, która dotyczy przesłaniania operatorów.

Temat 12. Znaczenie operatora powinno być zgodne z jego nazwą

Przeciążanie operatorów to mechanizm dający duże możliwości i — podobnie jak większość takich mechanizmów — związany jest z dużym ryzykiem. W programowaniu wraz z dużymi możliwościami przychodzi duża odpowiedzialność. Pracując jako szkoleniowiec, często widzę, jak uczestników ponosi fantazja, gdy po raz pierwszy odkrywają przeciążanie operatorów. Na przykład jedno z ćwiczeń polega na napisaniu funkcji do obliczania silni liczb:

```
1 fun Int.factorial(): Int = (1..this).product()
2
3 fun Iterable<Int>.product(): Int =
4    fold(1) { acc, i -> acc * i }
```

Ponieważ ta funkcja jest zdefiniowana jako funkcja rozszerzająca typ `Int`, łatwo jest z niej korzystać:

```
1 print(10 * 6.factorial()) // 7200
```

Matematycy wiedzą, że istnieje specjalna notacja do zapisywania silni. Ma ona postać wykrzyknika po liczbie:

```
1 10 * 6!
```

Kotlin nie udostępnia takiego operatora, jednak — jak zauważył jeden z uczestników moich warsztatów — można zastosować przeciążanie do operatora `not`:

```
1 operator fun Int.not() = factorial()
2
3 print(10 * !6) // 7200
```

Można zastosować takie rozwiązanie, ale czy należy? Najprostsza odpowiedź brzmi: NIE. Wystarczy przeczytać deklarację funkcji, aby zauważyć, że jej nazwa to `not`. Zgodnie z nazwą **nie** należy jej używać w pokazany sposób. Reprezentuje ona operację logiczną, a nie silnię liczb. Zastosowanie jej do obliczania silni byłoby mylące. W Kotlinie wszystkie operatory są lukrem składniowym i reprezentują przedstawione w tabeli 2.1 funkcje o konkretnych nazwach. Każdy operator można wywołać jako funkcję zamiast za pomocą składni operatora. Jak wygląda następujący zapis?

```
1 print(10 * 6.not()) // 7200
```

Tabela 2.1. Funkcje odpowiadające operatorom w Kotlinie

Operator	Powiązana funkcja
`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`
`++a`	`a.inc()`
`--a`	`a.dec()`
`a+b`	`a.plus(b)`
`a-b`	`a.minus(b)`
`a*b`	`a.times(b)`
`a/b`	`a.div(b)`
`a..b`	`a.rangeTo(b)`
`a in b`	`b.contains(a)`
`a+=b`	`a.plusAssign(b)`
`a-=b`	`a.minusAssign(b)`
`a*=b`	`a.timesAssign(b)`
`a/=b`	`a.divAssign(b)`
`a==b`	`a.equals(b)`
`a>b`	`a.compareTo(b) > 0`
`a<b`	`a.compareTo(b) < 0`
`a>=b`	`a.compareTo(b) >= 0`
`a<=b`	`a.compareTo(b) <= 0`

Znaczenie każdego operatora w Kotlinie zawsze pozostaje takie samo. Jest to bardzo ważna decyzja projektowa. W niektórych językach, na przykład w Scali, operatory można przeciążać bez żadnych ograniczeń. Niektórzy programiści nadużywają tej swobody. Pierwsza lektura kodu używającego nieznanej biblioteki może być trudna nawet wtedy, gdy nazwy funkcji i klas są opisowe. Teraz wyobraź sobie, że operatory mają niestandardowe znaczenie, znane tylko programistom obeznanym z *teorią kategorii*. Wtedy kod znacznie trudniej byłoby zrozumieć. Trzeba byłoby zrozumieć każdy operator z osobna, zapamiętać jego znaczenie w określonym kontekście, a także utrzymywać to wszystko w pamięci, aby móc połączyć ze sobą elementy układanki i zrozumieć całą instrukcję. W Kotlinie taki problem nie istnieje, ponieważ każdy operator ma określone znaczenie. Przyjrzyj się na przykład następującemu wyrażeniu:

```
x + y == z
```

Wiesz, że oznacza ono to samo co:

```
x.plus(y).equal(z)
```

Może też być odpowiednikiem następującego kodu, jeśli typ zwracanej wartości w operatorze `plus` dopuszcza wartości `null`:

```
(x.plus(y))?.equal(z) ?: (z == null)
```

Używane są tu funkcje o określonych nazwach i oczekujemy, że wszystkie funkcje działają zgodnie ze znaczeniem tych nazw. Mocno ogranicza to zakres zastosowań każdego operatora. Używanie `not` do zwracania silni jest oczywistym naruszeniem konwencji i nie powinno mieć miejsca.

Niejasne przypadki

Największy problem pojawia się wtedy, gdy nie jest jasne, czy dana struktura jest zgodna z konwencjami. Na przykład co oznacza potrojenie funkcji? Dla niektórych osób oczywiste jest, że oznacza to utworzenie innej funkcji, która trzykrotnie powtarza daną funkcję:

```
1 operator fun Int.times(operation: () -> Unit): ()->Unit =
2    { repeat(this) { operation() } }
3
4 val tripledHello = 3 * { print("Witaj") }
5
6 tripledHello() // Wyświetla „WitajWitajWitaj”
```

Z kolei dla innych może być oczywiste, że chcesz wywołać daną funkcję trzy razy:[3]

```
1 operator fun Int.times(operation: ()->Unit) {
2    repeat(this) { operation() }
3 }
4
5 3 * { print("Witaj") } // Wyświetla „WitajWitajWitaj”
```

Gdy znaczenie jest nieoczywiste, lepiej jest stosować opisowe funkcje rozszerzające. Jeśli chcesz używać ich jak operatorów, możesz utworzyć je w wersji infiksowej:

```
1 infix fun Int.timesRepeated(operation: ()->Unit) = {
2    repeat(this) { operation() }
3 }
```

[3] Różnica polega na tym, że w pierwszym przypadku tworzona jest nowa funkcja, natomiast w drugim tylko wywoływana jest istniejąca funkcja. Tak więc w pierwszej sytuacji wynikiem mnożenia jest `()->Unit`, a w drugim — `Unit`.

```
4
5 val tripledHello = 3 timesRepeated { print("Witaj") }
6 tripledHello() // Wyświetla „WitajWitajWitaj”
```

Czasem lepiej jest używać funkcji najwyższego poziomu. Powtarzanie funkcji trzy razy jest już zaimplementowane i dostępne w bibliotece standardowej:

```
1 repeat(3) { print("Witaj") } // Wyświetla „WitajWitajWitaj”
```

Kiedy łamanie reguł jest akceptowalne?

W jednym ważnym scenariuszu dozwolone jest przeciążanie operatorów w dziwny sposób. Chodzi tu o projektowanie języka dziedzinowego (ang. *Domain Specific Language* — DSL). Pomyśl o klasycznym przykładowym języku dziedzinowym dla HTML-a:

```
1 body {
2    div {
3       +"Jakiś tekst"
4    }
5 }
```

Aby dodać tekst do elementu, używany jest operator `String.unaryPlus`. Jest to akceptowalne, ponieważ operator ten jest częścią języka dziedzinowego. W tym konkretnym kontekście dla czytelników kodu nie jest zaskoczeniem, że obowiązują w nim inne reguły.

Podsumowanie

Rozważnie stosuj przeciążanie operatorów. Nazwa zawsze powinna odzwierciedlać działanie funkcji. Unikaj sytuacji, w których znaczenie operatora jest niejasne. Doprecyzuj je, używając standardowej funkcji z opisową nazwą. Jeśli chcesz uzyskać składnię typową dla operatorów, zastosuj modyfikator `infix` lub funkcję najwyższego poziomu.

Temat 13. Unikaj zwracania wartości Unit? lub operowania nimi

W procesie rekrutacji mój bliski przyjaciel otrzymał następujące pytanie: „Dlaczego ktoś mógłby chcieć zwracać wartość `Unit?` z funkcji?”. No cóż, `Unit?` przyjmuje tylko dwie możliwe wartości: `Unit` lub `null`. Podobnie wartość typu `Boolean` może być równa `true` lub `false`. Tak więc typy te są izomorficzne, dlatego można je stosować wymiennie. Dlaczego ktoś miałby chcieć reprezentować coś za pomocą wartości `Unit?` zamiast przy użyciu typu logicznego? Nie mam innego pomysłu jak to, że pozwala to użyć operatora Elvis lub bezpiecznego wywołania. Tak więc zamiast:

```
1 fun keyIsCorrect(key: String): Boolean = //...
2
3 if(!keyIsCorrect(key)) return
```

można zastosować następujący kod:

```
1 fun verifyKey(key: String): Unit? = //...
2
3 verifyKey(key) ?: return
```

Wydaje się, że była to oczekiwana odpowiedź. Jednak w rozmowie rekrutacyjnej mojego przyjaciela zabrakło znacznie ważniejszego pytania: „Czy należy stosować takie rozwiązanie?”. Ta sztuczka wygląda dobrze, gdy piszemy kod, ale już niekoniecznie w trakcie jego czytania. Używanie wartości `Unit?` do reprezentowania wartości logicznych jest mylące i może prowadzić do trudnych do wykrycia błędów. Wyjaśniłem już, że poniższe wyrażenie może dawać zaskakujące wyniki:

```
1 getData()?.let{ view.showData(it) } ?: view.showError()
```

Gdy `showData` zwraca `null` i `getData` zwraca „nie `null`”, wywoływane są zarówno `showData`, jak i `showError`. Użycie standardowej struktury `if-else` jest mniej skomplikowane i bardziej czytelne:

```
1 if (person != null && person.isAdult) {
2    view.showPerson(person)
3 } else {
4    view.showError()
5 }
```

Porównaj dwa następujące zapisy:

```
1 if(!keyIsCorrect(key)) return
2
3 verifyKey(key) ?: return
```

Nie natrafiłem na żadną sytuację, w której wersja z wartością `Unit?` jest bardziej czytelna. Ta składnia jest myląca i niezrozumiała. Prawie zawsze zamiast niej należy stosować typ `Boolean`. To dlatego zgodnie z ogólną regułą należy unikać zwracania wartości `Unit?` i operowania nimi.

Temat 14. Podawaj typ zmiennej, gdy nie jest on oczywisty

W Kotlinie używany jest świetny system wnioskowania typów, który umożliwia ich pomijanie, jeśli są oczywiste dla programistów:

```
1 val num = 10
2 val name = "Marcin"
3 val ids = listOf(12, 112, 554, 997)
```

Nie tylko przyspiesza to pracę, ale też poprawia czytelność, gdy typ w oczywisty sposób wynika z kontekstu, a dodatkowa specyfikacja jest nadmiarowa i komplikuje kod. Nie należy jednak nadużywać tej techniki, gdy typ nie jest jasny:

```
1 val data = getSomeData()
```

Kod trzeba projektować z myślą o czytelności. Dlatego nie należy ukrywać istotnych informacji, które mogą być ważne dla czytelników. Błędne jest argumentowanie, że można pomijać typy zwracanych wartości, ponieważ czytelnik zawsze może sprawdzić je w specyfikacji funkcji. Także w tej specyfikacji może być zastosowane wnioskowanie funkcji, przez co użytkownik jest zmuszony przechodzić do kolejnych specyfikacji. Ponadto użytkownik może czytać kod w serwisie GitHub lub w innym środowisku, które nie obsługuje przechodzenia do implementacji. Nawet jeśli przeskakiwanie do implementacji jest możliwe, wszyscy dysponujemy ograniczoną ilością pamięci roboczej i marnowanie jej nie jest dobrym pomysłem. Typ to ważna informacja i jeśli nie jest oczywisty, należy go podać:

```
1 val data: UserData = getSomeData()
```

Zwiększanie czytelności nie jest jedynym argumentem na rzecz podawania typów. Określenie typu zwiększa też bezpieczeństwo, co wyjaśniłem w temacie 3. „Jak najszybciej wyeliminuj typy z zewnętrznych platform” i temacie 4. „Nie udostępniaj wywnioskowanych typów” w rozdziale 1. „Bezpieczeństwo”. **Typ może być ważną informacją zarówno dla programisty, jak i dla kompilatora. W takiej sytuacji nie obawiaj się podawać typu. Nie kosztuje to wiele, a może okazać się bardzo pomocne.**

Temat 15. Rozważ bezpośrednie podawanie odbiorców

Jedną z sytuacji, w której często stosowana jest dłuższa struktura w celu doprecyzowania operacji, jest używanie funkcji lub właściwości z odbiorcy (zamiast zmiennej lokalnej lub zmiennej najwyższego poziomu). W najprostszym scenariuszu oznacza to wskazanie klasy, z którą dana metoda jest powiązana:

```
1 class User: Person() {
2    private var beersDrunk: Int = 0
3
4    fun drinkBeers(num: Int) {
5       // ...
6       this.beersDrunk += num
7       // ...
8    }
9 }
```

Podobnie można bezpośrednio podać odbiorcę rozszerzenia (`this` w metodzie rozszerzającej), aby jego użycie było bardziej oczywiste. Porównaj poniższą implementację szybkiego sortowania napisaną bez bezpośrednio podanych odbiorców:

```
1 fun <T : Comparable<T>> List<T>.quickSort(): List<T> {
2    if (size < 2) return this
3    val pivot = first()
4    val (smaller, bigger) = drop(1)
5       .partition { it < pivot }
6    return smaller.quickSort() + pivot + bigger.quickSort()
7 }
```

z wersją z podanymi odbiorcami:

```
1 fun <T : Comparable<T>> List<T>.quickSort(): List<T> {
2    if (this.size < 2) return this
3    val pivot = this.first()
4    val (smaller, bigger) = this.drop(1)
5       .partition { it < pivot }
6    return smaller.quickSort() + pivot + bigger.quickSort()
7 }
```

Sposób używania obu tych funkcji jest identyczny:

```
1 listOf(3, 2, 5, 1, 6).quickSort()       // [1, 2, 3, 5, 6]
2 listOf("C", "D", "A", "B").quickSort() // [A, B, C, D]
```

Wielu odbiorców

Bezpośrednie podawanie odbiorców jest pomocne zwłaszcza wtedy, gdy w zasięgu znajduje się więcej niż jeden odbiorca. Często dzieje się tak, gdy używamy funkcji `apply`, `with` lub `run`. Takie sytuacje są niebezpieczne i należy ich unikać. Bezpieczniej jest używać obiektu, bezpośrednio podając odbiorcę. Aby zrozumieć ten problem, przyjrzyj się poniższemu kodowi:[4]

```
 1 class Node(val name: String) {
 2
 3    fun makeChild(childName: String) =
 4       create("$name.$childName")
 5          .apply { print("Utworzono ${name}") }
 6
 7    fun create(name: String): Node? = Node(name)
 8 }
 9
10 fun main() {
11    val node = Node("rodzic")
12    node.makeChild("dziecko")
13 }
```

Jaki wynik otrzymasz? Odłóż na chwilę książkę i spróbuj samodzielnie znaleźć odpowiedź przed jej sprawdzeniem.

Prawdopodobnie spodziewasz się, że wynik to „Utworzono rodzic.dziecko”. Jednak w rzeczywistości rezultat to „Utworzono rodzic”. Dlaczego tak się dzieje? Aby to zbadać, można jawnie podać odbiorcę przed `name`:

```
1 class Node(val name: String) {
2
3    fun makeChild(childName: String) =
4       create("$name.$childName")
5          .apply { print("Created ${this.name}") }
6          // Błąd kompilacji
7
8    fun create(name: String): Node? = Node(name)
9 }
```

Problem polega na tym, że typem `this` w `apply` jest `Node?`, dlatego metod nie można używać bezpośrednio. Trzeba je najpierw wypakować, na przykład za pomocą bezpiecznego wywołania. Wtedy wynik stanie się prawidłowy:

[4] Inspiracją była zagadka dodana pierwotnie przez Romana Dawydkina do kolekcji Dmitrija Kandalova i zaprezentowana później na konferencji KotlinConf przez Antona Keksa.

```
class Node(val name: String) {

    fun makeChild(childName: String) =
       create("$name.$childName")
          .apply { print("Utworzono ${this?.name}") }

    fun create(name: String): Node? = Node(name)
}

fun main() {
    val node = Node("rodzic")
    node.makeChild("dziecko")
    // Wyświetla „Utworzono rodzic.dziecko”
}
```

Jest to przykład niewłaściwego zastosowania funkcji `apply`. Taki problem nie wystąpiłby, gdyby zastosować funkcję `also` i wywołać `name` dla argumentu. Używanie `also` wymaga bezpośredniego podania odbiorcy funkcji. Gdy wykonujesz dodatkowe operacje lub przetwarzasz elementy z dopuszczalną wartością `null`, funkcje `also` i `let` są zwykle dużo lepszym wyborem:

```
class Node(val name: String) {

    fun makeChild(childName: String) =
       create("$name.$childName")
          .also { print("Utworzono ${it?.name}") }

    fun create(name: String): Node? = Node(name)
}
```

Gdy odbiorca nie jest oczywisty, można albo go pominąć, albo doprecyzować kod, bezpośrednio go podając. Jeśli odbiorca jest używany bez doprecyzowania, wybierany jest najbliższy odbiorca. Gdy chcesz zastosować zewnętrznego odbiorcę, musisz go dookreślić. To w takich sytuacjach najbardziej przydatne jest bezpośrednie podawanie odbiorcy. Oto przykład ilustrujący obie opisane techniki:

```
class Node(val name: String) {

    fun makeChild(childName: String) =
       create("$name.$childName").apply {
          print("Utworzono ${this?.name} w "+
             " ${this@Node.name}")
       }

    fun create(name: String): Node? = Node(name)
}
```

```
11
12 fun main() {
13    val node = Node("rodzic")
14    node.makeChild("dziecko")
15    // Utworzono rodzic.dziecko w rodzic
16 }
```

W ten sposób bezpośrednie wskazanie odbiorcy pozwala określić, o jakiego odbiorcę nam chodzi. Może to być ważna informacja, która nie tylko chroni przed błędami, ale też poprawia czytelność.

Adnotacja DslMarker

W pewnym kontekście często używamy różnych odbiorców w mocno zagnieżdżonym zasięgu i w ogóle nie musimy bezpośrednio ich podawać. Mam tu na myśli języki dziedzinowe w Kotlinie. Nie trzeba w nich bezpośrednio podawać odbiorców, ponieważ wynika to z projektu takich języków. Jednak w językach DSL jesteśmy szczególnie narażeni na przypadkowe użycie funkcji z zewnętrznego zasięgu. Wyobraź sobie prosty kod w języku dziedzinowym dla HTML-a, służący do tworzenia tabel HTML-owych:

```
 1 table {
 2    tr {
 3       td { +"Kolumna 1" }
 4       td { +"Kolumna 2" }
 5    }
 6    tr {
 7       td { +"Wartość 1" }
 8       td { +"Wartość 2" }
 9    }
10 }
```

Zauważ, że domyślnie w każdym zasięgu można także używać metod odbiorców z zasięgu zewnętrznego. Możesz wykorzystać ten fakt, aby nadużyć języka dziedzinowego:

```
 1 table {
 2    tr {
 3       td { +"Kolumna 1" }
 4       td { +"Kolumna 2" }
 5       tr {
 6          td { +"Wartość 1" }
 7          td { +"Wartość 2" }
 8       }
 9    }
10 }
```

Aby ograniczyć taki kod, dostępna jest specjalna metaadnotacja (czyli adnotacja dotycząca adnotacji), która nie pozwala na niejawne używanie odbiorców z zasięgu zewnętrznego. Ta metaadnotacja to `DslMarker`. Gdy dodasz ją do innej adnotacji, a następnie użyjesz tej innej adnotacji do klasy używanej jako budowniczy, nie będzie można korzystać w tej klasie z niejawnych odbiorców. Oto przykład pokazujący, jak używać metaadnotacji `DslMarker`:

```
1 @DslMarker
2 annotation class HtmlDsl
3
4 fun table(f: TableDsl.() -> Unit) { /*...*/ }
5
6 @HtmlDsl
7 class TableDsl { /*...*/ }
```

Nie można wtedy niejawnie używać odbiorców z zasięgu zewnętrznego:

```
 1 table {
 2    tr {
 3       td { +"Kolumna 1" }
 4       td { +"Kolumna 2" }
 5       tr { // BŁĄD KOMPILACJI
 6          td { +"Wartość 1" }
 7          td { +"Wartość 2" }
 8       }
 9    }
10 }
```

Stosowanie funkcji z odbiorcy z zasięgu zewnętrznego wymaga teraz bezpośredniego podania odbiorcy:

```
 1 table {
 2    tr {
 3       td { +"Kolumna 1" }
 4       td { +"Kolumna 2" }
 5       this@table.tr {
 6          td { +"Wartość 1" }
 7          td { +"Wartość 2" }
 8       }
 9    }
10 }
```

Metaadnotacja `DslMarker` to bardzo ważny mechanizm, który stosuje się, aby wymusić użycie najbliższego odbiorcy lub jego jawne podanie. Jednak zwykle i tak lepiej jest nie używać w językach dziedzinowych jawnie podawanych odbiorców. Respektuj projekt języków dziedzinowych i stosuj je zgodnie z nim.

Podsumowanie

Nie zmieniaj zasięgu odbiorców tylko dlatego, że jest to możliwe. Zbyt duża liczba odbiorców udostępniających metody może prowadzić do dezorientacji. Zwykle lepsze jest bezpośrednie podawanie argumentów lub referencji. Gdy już zmieniasz odbiorcę, bezpośrednie podanie go zwiększa czytelność kodu, ponieważ pozwala dookreślić źródło funkcji. Gdy dostępnych jest wielu odbiorców, można nawet posłużyć się etykietą, aby jednoznacznie określić, z którego z nich pochodzi wywoływana funkcja. Jeśli chcesz wymusić jawne podawanie odbiorców z zasięgu zewnętrznego, użyj metaadnotacji `DslMarker`.

Temat 16. Właściwości powinny reprezentować stan, a nie działanie

Właściwości w Kotlinie wyglądają podobnie jak pola w Javie, ale reprezentują inną koncepcję.

```
1 // Właściwość w Kotlinie
2 var name: String? = null
3
4 // Pole w Javie
5 String name = null;
```

Choć obu tych konstruktów można używać w ten sam sposób (do przechowywania danych), trzeba pamiętać, że właściwości oferują znacznie większe możliwości. Zacznijmy od tego, że zawsze mogą udostępniać niestandardowe settery i gettery:

```
1 var name: String? = null
2    get() = field?.toUpperCase()
3    set(value) {
4       if(!value.isNullOrBlank()) {
5          field = value
6       }
7    }
```

Widać tu, że używany jest identyfikator `field`. Jest to referencja do powiązanego pola, które pozwala przechowywać dane w tej właściwości. Takie powiązane pola są generowane domyślnie, ponieważ używają ich domyślne implementacje settera i gettera. Można też zaimplementować niestandardowe akcesory, które nie używają takich pól. W takiej sytuacji właściwość w ogóle nie zawiera elementu `field`. Na przykład właściwość w Kotlinie można zdefiniować, używając tylko gettera dla przeznaczonej jedynie do odczytu właściwości `val`:

```
1 val fullName: String
2    get() = "$name $surname"
```

Aby uzyskać właściwość `var` przeznaczoną do odczytu i zapisu, można zdefiniować getter i setter. Powstają w ten sposób tak zwane *właściwości pochodne*. Są one dość popularne i stanowią główny powód, dla którego w Kotlinie domyślnie wszystkich właściwości są hermetyzowane. Wyobraź sobie, że musisz przechowywać w obiekcie datę i używasz typu `Date` z biblioteki standardowej Javy. Później z jakiegoś powodu okazuje się, że obiekt nie może już zawierać właściwości tego typu. Może to wynikać z trudności z serializacją lub przeniesienia obiektu do ogólnego modułu. Problem polega na tym, że ta właściwość

jest już używana w projekcie. W Kotlinie nie stanowi to problemu, ponieważ można przenieść dane do odrębnej właściwości `millis` i zmodyfikować właściwość `date`, aby nie przechowywała danych, tylko pakowała i wypakowywała tę nową właściwość.

```
1 var date: Date
2    get() = Date(millis)
3    set(value) {
4       millis = value.time
5    }
```

We właściwościach pola nie są konieczne. Właściwości reprezentują akcesory (getter dla właściwości `val` oraz getter i setter dla właściwości `var`). To dlatego można definiować je w interfejsach:

```
1 interface Person {
2    val name: String
3 }
```

To oznacza, że ten interfejs deklaruje, iż ma getter. Właściwości można też przesłaniać:

```
1 open class Supercomputer {
2    open val theAnswer: Long = 42
3 }
4
5 class AppleComputer : Supercomputer() {
6    override val theAnswer: Long = 1_800_275_2273
7 }
```

Z tego samego powodu można delegować właściwości:

```
1 val db: Database by lazy { connectToDb() }
```

Delegowanie właściwości jest szczegółowo opisane w temacie 21. „Stosuj delegaty właściwości do wyodrębniania powtarzających się wzorców dotyczących właściwości". Ponieważ właściwości są w zasadzie funkcjami, można tworzyć właściwości rozszerzające:

```
 1 val Context.preferences: SharedPreferences
 2    get() = PreferenceManager
 3       .getDefaultSharedPreferences(this)
 4
 5 val Context.inflater: LayoutInflater
 6    get() = getSystemService(
 7       Context.LAYOUT_INFLATER_SERVICE) as LayoutInflater
 8
 9 val Context.notificationManager: NotificationManager
10    get() = getSystemService(Context.NOTIFICATION_SERVICE)
11       as NotificationManager
```

Tak więc **właściwości reprezentują akcesory, a nie pola**. Dlatego można ich używać zamiast niektórych funkcji. Należy jednak zachować ostrożność. Właściwości nie należy używać do reprezentowania operacji algorytmicznych takich jak w poniższym przykładzie:

```
1 // NIE STOSUJ TEGO ROZWIĄZANIA!
2 val Tree<Int>.sum: Int
3    get() = when (this) {
4       is Leaf -> value
5       is Node -> left.sum + right.sum
6    }
```

Tu właściwość `sum` iteracyjnie pobiera wszystkie elementy, dlatego reprezentuje operację algorytmiczną. Tak więc ta właściwość jest myląca. Znalezienie odpowiedzi może być obliczeniowo wymagające, dlatego nie jest oczekiwane w getterze. Ten kod powinien znajdować się nie we właściwości, ale w funkcji:

```
1 fun Tree<Int>.sum(): Int = when (this) {
2    is Leaf -> value
3    is Node -> left.sum() + right.sum()
4 }
```

Oto ogólna reguła: **właściwości należy stosować tylko do reprezentowania lub ustawiania stanu, bez używania żadnej innej logiki**. Następująca heurystyka pomaga zdecydować, czy należy utworzyć właściwość: „Gdybym zdefiniował tę właściwość jako funkcję, to czy dodałbym do jej nazwy przedrostek `get` czy `set`?”. Jeśli nie, zapewne nie należy tworzyć właściwości. Oto najbardziej typowe sytuacje, w których zamiast właściwości należy stosować funkcje:

- **Operacja jest obliczeniowo kosztowna lub ma złożoność obliczeniową wyższą niż O(1)**. Użytkownicy nie oczekują, że korzystanie z właściwości może być kosztowne. Lepiej jest wtedy utworzyć funkcję, ponieważ informuje ona, że operacja może być kosztowna. Dlatego użytkownik może unikać jej niepotrzebnego wywoływania, a programista — zapisać jej wyniki w pamięci podręcznej.
- **Właściwość obejmuje logikę biznesową (określającą działanie aplikacji)**. W trakcie czytania kodu nie oczekujemy, że właściwość może wykonywać jakieś operacje poza prostymi czynnościami takimi jak rejestrowanie działań, powiadamianie odbiorników lub aktualizowanie powiązanego elementu.
- **Właściwość nie jest deterministyczna**. Dwukrotne wywołanie składowej raz po razie daje różne wyniki.

- **Właściwość wykonuje konwersję, na przykład `Int.toDouble()`**. Zgodnie z konwencją konwersje wykonuje się w metodach lub w funkcjach rozszerzających. Użycie właściwości bardziej przypomina korzystanie z jakiegoś wewnętrznego elementu niż opakowywanie całego obiektu.
- **Gettery nie powinny zmieniać stanu właściwości**. Programiści oczekują, że można swobodnie korzystać z getterów bez obaw o zmianę stanu właściwości.

Na przykład obliczenie sumy elementów wymaga iteracyjnego pobrania wszystkich elementów (jest to działanie, a nie stan) i ma złożoność liniową. Dlatego nie należy wykonywać tej operacji we właściwości. Operacja ta jest zdefiniowana w bibliotece standardowej jako funkcja:

```
1 val s = (1..100).sum()
```

Z kolei do pobierania i ustawiania stanu w Kotlinie używane są właściwości. Nie należy stosować do tego funkcji, chyba że jest to uzasadnione. Właściwości służą do reprezentowania i ustawiania stanu, a jeśli później będziesz musiał go zmodyfikować, zastosuj niestandardowe gettery i settery:

```
 1 // NIE STOSUJ TEGO ROZWIĄZANIA!
 2 class UserIncorrect {
 3    private var name: String = ""
 4
 5    fun getName() = name
 6
 7    fun setName(name: String) {
 8       this.name = name
 9    }
10 }
11
12 class UserCorrect {
13    var name: String = ""
14 }
```

Zgodnie z prostą regułą **właściwość służy do opisywania i ustawiania stanu, natomiast funkcja — do opisywania działań**.

Temat 17. Rozważ stosowanie nazw dla argumentów

Gdy czytasz kod, znaczenie argumentów nie zawsze jest oczywiste. Przyjrzyj się poniższemu przykładowi:

```
1 val text = (1..10).joinToString("|")
```

Czym jest `"|"`? Jeśli dobrze znasz funkcję `joinToString`, wiesz, że jest to `separator`. Choć równie dobrze mógłby to być `prefix`. Nie jest to jasne[5]. Można jednak poprawić czytelność kodu, doprecyzowując przeznaczenie argumentów, których wartości nie informują jasno o ich znaczeniu. Najlepiej posłużyć się w tym celu argumentami nazwanymi:

```
1 val text = (1..10).joinToString(separator = "|")
```

Podobny efekt można uzyskać, ustawiając nazwę zmiennej:

```
1 val separator = "|"
2 val text = (1..10).joinToString(separator)
```

Nazywanie argumentów to jednak solidniejsze rozwiązanie. Nazwa zmiennej informuje o intencjach programisty, ale nie zapewnia poprawności. Co się stanie, jeśli programista popełni błąd i umieści zmienną na niewłaściwej pozycji? Co będzie, jeśli zmieni się kolejność parametrów? Argumenty nazwane chronią przed takimi sytuacjami, natomiast nazwane wartości tego nie robią. To dlatego warto stosować argumenty nazwane nawet wtedy, gdy używane są nazwane wartości:

```
1 val separator = "|"
2 val text = (1..10).joinToString(separator = separator)
```

Kiedy należy używać argumentów nazwanych?

Argumenty nazwane są oczywiście dłuższe, ale mają dwie ważne zalety:

- nazwa określa, jakie wartości są oczekiwane;
- są bardziej bezpieczne, ponieważ ich kolejność nie jest istotna.

Nazwa argumentu jest ważną informacją nie tylko dla programisty używającego danej funkcji, ale też przy czytaniu kodu z jej wywołaniem. Przyjrzyj się temu wywołaniu:

```
1 sleep(100)
```

[5] Środowisko IntelliJ pomaga, wyświetlając wskazówki, gdy umieszczasz literały w wywołaniu funkcji. Jednak ten mechanizm może zostać wyłączony, a ponadto możesz korzystać z innego środowiska IDE.

Ile potrwa uśpienie? Sto milisekund? A może sto sekund? Można to doprecyzować, stosując argument nazwany:

```
1 sleep(timeMillis = 100)
```

W tym scenariuszu nie jest to możliwy sposób na doprecyzowanie kodu. W językach ze statyczną kontrolą typów (takich jak Kotlin) pierwszą ochroną programistów przy przekazywaniu argumentów jest typ parametru. Tu można posłużyć się nim do podania informacji o jednostkach czasu:

```
1 sleep(Millis(100))
```

Można też zastosować właściwość rozszerzającą, aby uzyskać składnię podobną jak w językach dziedzinowych:

```
1 sleep(100.ms)
```

Typy są dobrym sposobem przekazywania takich informacji. Jeśli martwisz się o wydajność, zastosuj klasy z modyfikatorem `inline` opisane w temacie 46. „Stosuj modyfikator inline dla funkcji z parametrami o typach funkcyjnych". Pomaga to zwiększyć bezpieczeństwo parametrów, ale nie rozwiązuje wszystkich problemów. Niektóre argumenty nadal mogą być niejasne, a także mogą zostać zapisane na błędnej pozycji. To dlatego proponuję stosowanie argumentów nazwanych. Dotyczy to przede wszystkim parametrów:

- z argumentami domyślnymi,
- tego samego typu co inne parametry,
- typu funkcyjnego (jeśli nie są ostatnim parametrem).

Parametry z argumentami domyślnymi

Gdy dla właściwości istnieje argument domyślny, prawie zawsze należy używać nazwy argumentu. Takie opcjonalne parametry są modyfikowane częściej niż wymagane parametry. Nie chcesz przecież przeoczyć takiej zmiany. Nazwa funkcji zwykle sugeruje, jakie nieopcjonalne parametry są używane, ale nie informuje o parametrach opcjonalnych. To dlatego bezpieczniejsze i bardziej przejrzyste jest stosowanie nazw dla argumentów opcjonalnych[6].

[6] W zalecanych praktykach dla innych języków czasem sugeruje się nawet, aby zawsze stosować nazwy dla argumentów opcjonalnych. Przykład znajdziesz w książce *Effective Python* Bretta Slatkina (wydanie polskie: *Efektywny Python*, Helion, 2015), w sposobie 19. „Zdefiniowanie zachowania opcjonalnego za pomocą argumentów w postaci słów kluczowych".

Kilka parametrów tego samego typu

Wspomniałem już, że gdy parametry mają różne typy, zwykle jesteśmy chronieni przed umieszczeniem argumentu w niewłaściwym miejscu. Jednak gdy parametry są tego samego typu, nie mamy takiej pomocy.

```
1 fun sendEmail(to: String, message: String) { /*...*/ }
```

W funkcjach tego rodzaju warto doprecyzowywać znaczenie argumentów za pomocą nazw:

```
1 sendEmail(
2    to = "kontakt@kt.akademia",
3    message = "Witaj, ..."
4 )
```

Parametry typu funkcyjnego

Ponadto w specjalny sposób należy traktować parametry typów funkcyjnych. W Kotlinie dla takich parametrów przeznaczona jest specjalna pozycja — ostatnia. Czasem nazwa funkcji opisuje argument typu funkcyjnego. Na przykład gdy w nazwie występuje słowo `repeat`, można oczekiwać, że podana lambda to blok kodu, który należy powtórzyć. Gdy widzisz słowo `thread`, możesz się domyślić, że podany blok to kod nowego wątku. Takie nazwy opisują tylko parametr z ostatniej pozycji.

```
1 thread {
2    // ...
3 }
```

Wszystkie pozostałe argumenty o typie funkcyjnym powinny być nazwane, ponieważ łatwo jest błędnie je zinterpretować. Przyjrzyj się na przykład temu prostemu językowi dziedzinowemu:

```
1 val view = linearLayout {
2    text("Kliknij poniżej")
3    button({ /* 1 */ }, { /* 2 */ })
4 }
```

Która funkcja jest częścią budowniczego, a która to odbiornik kliknięcia? Należy to doprecyzować, podając nazwę odbiornika i przenosząc budowniczego poza listę argumentów:

```
1 val view = linearLayout {
2    text("Kliknij poniżej")
3    button(onClick = { /* 1 */ }) {
```

```
4       /* 2 */
5    }
6 }
```

Trudna do zrozumienia jest zwłaszcza dłuższa lista argumentów typu funkcyjnego:

```
1 fun call(before: ()->Unit = {}, after: ()->Unit = {}){
2    before()
3    print("Środek")
4    after()
5 }
6
7 call({ print("WYWOŁANIE") }) // WYWOŁANIEŚrodek
8 call { print("WYWOŁANIE") }  // ŚrodekWYWOŁANIE
```

Aby zapobiec takim sytuacjom, gdy nie ma żadnego argumentu typu funkcyjnego o specjalnym znaczeniu, podaj nazwy wszystkich takich argumentów:

```
1 call(before = { print("WYWOŁANIE") }) // WYWOŁANIEŚrodek
2 call(after = { print("WYWOŁANIE") })  // ŚrodekWYWOŁANIE
```

Jest to prawdą przede wszystkim w bibliotekach do programowania reaktywnego. Na przykład w bibliotece RxJava, subskrybując zdarzenia obiektu typu `Observable`, można podać funkcje, które mają być wywoływane:

- dla każdego otrzymanego elementu,
- po wystąpieniu błędu,
- po zakończeniu pracy obserwowalnego elementu.

Często widziałem, że w Javie programiści określają te funkcje za pomocą wyrażeń lambda i podają w komentarzach, z którą metodą powiązane są poszczególne wyrażenia lambda:

```
1 // Java
2 observable.getUsers()
3    .subscribe((List<User> users) -> { // onNext
4       // ...
5    }, (Throwable throwable) -> { // onError
6       // ...
7    }, () -> { // onCompleted
8       // ...
9    });
```

W Kotlinie można zrobić coś więcej i użyć zamiast tego argumentów nazwanych:

```
observable.getUsers()
   .subscribeBy(
      onNext = { users: List<User> ->
         // ...
      },
      onError = { throwable: Throwable ->
         // ...
      },
      onCompleted = {
         // ...
      })
```

Zauważ, że zmieniłem tu nazwę funkcji z `subscribe` na `subscribeBy`. Wynika to z tego, że biblioteka `RxJava` jest napisana w Javie, a **nie można używać argumentów nazwanych w wywołaniach funkcji Javy**. Wynika to z tego, że Java nie zachowuje informacji o nazwach funkcji. Aby móc używać argumentów nazwanych, często trzeba tworzyć w Kotlinie własne nakładki na potrzebne funkcje (czyli funkcje rozszerzające stosowane zamiast pierwotnych funkcji).

Podsumowanie

Argumenty nazwane są przydatne nie tylko wtedy, gdy chcesz pomijać niektóre wartości domyślne. Takie argumenty są ważną informacją dla programistów czytających kod i mogą zwiększać jego bezpieczeństwo. Należy rozważyć ich stosowanie zwłaszcza wtedy, gdy używanych jest kilka parametrów tego samego typu (lub kilka parametrów typu funkcyjnego), a także dla argumentów opcjonalnych. Gdy używanych jest kilka parametrów typu funkcyjnego, prawie zawsze powinny być one nazwane. Wyjątkiem od tej reguły jest ostatni argument funkcyjny, gdy ma on specjalne znaczenie (na przykład w języku dziedzinowym).

Temat 18. Przestrzegaj konwencji programistycznych

Podstawy

W Kotlinie używane są ugruntowane konwencje programistyczne opisane w dokumentacji w trafnie nazwanej sekcji „Coding Conventions”[7]. **Wprawdzie nie we wszystkich projektach są one optymalne, ale dla społeczności optymalne jest używanie konwencji przestrzeganych we wszystkich projektach**. Dzięki konwencjom:

- łatwiej jest przechodzić między projektami;
- kod jest bardziej czytelny nawet dla zewnętrznych programistów;
- łatwiej jest domyślić się działania kodu;
- łatwiej jest później scalać kod ze wspólnym repozytorium lub przenosić fragmenty kodu między projektami.

Programiści powinni zapoznać się z konwencjami opisanymi w dokumentacji. Należy także przestrzegać ich nowych wersji, jeśli — co może się zdarzyć w przyszłości — zostaną wprowadzone w nich zmiany. Ponieważ trudno jest pogodzić obie te rzeczy, dostępne są dwa pomocne narzędzia:

- Formater ze środowiska IntelliJ można tak skonfigurować, aby automatycznie formatował kod zgodnie z oficjalnym stylem z sekcji „Coding Conventions”. W tym celu wybierz opcję *Settings/Editor/Code Style/Kotlin*, kliknij odsyłacz *Set from…* w prawym górnym rogu i wybierz z menu opcję *Predefined style/Kotlin style guide*.
- Narzędzie ktlint[8] to popularny linter, który analizuje kod i informuje o wszystkich naruszeniach konwencji programistycznych.

Gdy przeglądam projekty pisane w Kotlinie, zauważam, że zwykle są one spójne z większością konwencji. Prawdopodobnie wynika to z tego, że konwencje w Kotlinie w dużej części wzorowane są na konwencjach z Javy, a obecnie większość programistów używających Kotlina wcześniej programowała w Javie. Reguła, która często jest naruszana, dotyczy formatowania klas i funkcji. Zgodnie z konwencjami klasy z krótkim konstruktorem podstawowym można definiować w jednym wierszu:

```
class FullName(val name: String, val surname: String)
```

[7] Odsyłacz: *https://kotlinlang.org/docs/reference/coding-conventions.html*.

[8] Odsyłacz: *https://github.com/pinterest/ktlint*.

Jednak klasy z wieloma parametrami należy formatować w taki sposób, aby każdy parametr znajdował się w odrębnym wierszu[9]. Ponadto w pierwszym wierszu nie należy podawać parametrów:

```
1 class Person(
2    val id: Int = 0,
3    val name: String = "",
4    val surname: String = ""
5 ) : Human(id, name) {
6    // Ciało klasy
7 }
```

W podobny sposób należy formatować długie funkcje:

```
 1 public fun <T> Iterable<T>.joinToString(
 2    separator: CharSequence = ", ",
 3    prefix: CharSequence = "",
 4    postfix: CharSequence = "",
 5    limit: Int = -1,
 6    truncated: CharSequence = "...",
 7    transform: ((T) -> CharSequence)? = null
 8 ): String {
 9    // ...
10 }
```

Zauważ, że bardzo różni się to od konwencji, zgodnie z którą pierwszy parametr należy podać w pierwszym wierszu, a kolejne parametry wyrównać względem pierwszego.

```
1 // Nie stosuj tego rozwiązania
2 class Person(val id: Int = 0,
3              val name: String = "",
4              val surname: String = "") : Human(id, name){
5    // Ciało klasy
6 }
```

Ten zapis może powodować dwa rodzaje problemów:

- Wcięcia dla argumentów każdej klasy wyglądają inaczej i zależą od nazwy klasy. Ponadto po zmianie nazwy klasy trzeba dostosować wcięcia dla wszystkich parametrów konstruktora podstawowego.
- Klasy definiowane w ten sposób często zajmują za dużo miejsca. Szerokość klasy tak definiowanej to nazwa klasy plus słowo kluczowe `class` plus najdłuższy parametr konstruktora podstawowego (lub ostatni parametr plus nadklasy i interfejsy).

[9] Parametry ściśle powiązane ze sobą, na przykład x i y, można umieścić w tym samym wierszu, choć nie jestem zwolennikiem tego podejścia.

Niektóre zespoły mogą zdecydować się na zastosowanie nieco innych konwencji. Nie ma w tym nic złego, jednak wtedy takich konwencji trzeba przestrzegać we wszystkich miejscach projektu. **Każdy projekt powinien wyglądać tak, jakby został napisany przez jedną osobę, a nie przez grupę walczących ze sobą programistów**.

Programiści często w niewystarczającym stopniu stosują się do konwencji. Są one jednak ważne, a rozdział poświęcony czytelności byłby niekompletny bez poświęcenia im choćby krótkiej sekcji. Zapoznaj się z nimi, używaj statycznych narzędzi kontrolnych wspomagających zachowanie zgodności z nimi i stosuj je w swoich projektach. Dzięki przestrzeganiu konwencji programistycznych sprawiamy, że projekty w Kotlinie są lepsze dla nas wszystkich.

Kotlin: wydajność, bezpieczeństwo, radość z programowania!

Projektanci Kotlina postawili na pragmatyzm. Oznacza to, że do dziś wszystkie decyzje związane z rozwojem języka są podejmowane z uwzględnieniem potrzeb biznesowych: produktywności, skalowalności, prostej konserwacji, niezawodności i wydajności. Od początku szczególną uwagę zwraca się również na bezpieczeństwo, czytelność, wielokrotne używanie kodu, łatwość użycia narzędzi i możliwości współdziałania z innymi językami. Bardzo ważna okazała się też wydajność zarówno działania kodu, jak i pracy programisty. Konsekwentne przestrzeganie tych założeń dało znakomite efekty. Dziś Kotlin jest świetnym wyborem dla programisty — pod jednym warunkiem: że zna specyfikę języka i korzysta z niego we właściwy sposób.

To książka przeznaczona dla doświadczonych programistów Kotlina, którzy chcą opanować ten język na wysokim poziomie. Wyjaśniono w niej sposoby korzystania ze szczególnych funkcji Kotlina, aby możliwe było uzyskanie bezpiecznego, czytelnego, skalowalnego i wydajnego kodu. Znalazły się w niej informacje o najlepszych praktykach pisania kodu w Kotlinie ze szczególnym uwzględnieniem praktyk wysokiego poziomu, zalecanych przez autorytety i twórców języka. Wyczerpująco omówiono też kwestie związane z właściwościami i typami specyficznymi dla platformy oraz tworzenie języków dziedzinowych, a także klasy i funkcje wewnątrzwierszowe. Jest to praktyczny przewodnik zawierający zbiór wytycznych, których uwzględnienie pozwoli pisać dobry i idiomatyczny kod w Kotlinie.

W tej książce między innymi:

- reguły pisania kodu o wysokiej jakości
- programowanie dla różnych platform oraz aplikacje mobilne
- wzorce i konwencje programistyczne
- konwencje i kontrakty dla programistów wspólnie tworzących projekt
- zapewnianie wysokiej wydajności aplikacji i bibliotek

Marcin Moskała jest doświadczonym programistą i szkoleniowcem. Językiem Kotlin zajmował się już na bardzo wczesnym etapie jego rozwoju. Założył firmę Kt. Academy, która została oficjalnym partnerem firmy JetBrains w zakresie nauczania Kotlina, i napisał książkę *Android Development with Kotlin*. Często występuje na międzynarodowych konferencjach, prowadzi też warsztaty programowania w Kotlinie. Jest twórcą i współtwórcą licznych bibliotek programistycznych.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-6799-9

9 788328 367999

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 69,00 zł