Język Kotlin – przyszłość Javy?

Java jest już językiem mocno leciwym i coraz częściej słyszy się o tym, że potrzebuje ona następcy. Nowoczesne języki, takie jak Python czy Swype, są znacznie wygodniejsze i bardziej eleganckie, dzięki czemu są chętniej wybierane przez programistów. Ale przecież istnieją alternatywy dla Javy. Już od wielu lat rozwijana Scala czy Groovy, nieco tylko młodszy Clojure czy pod wieloma względami zaskakujący Cejlon. Nawet powstał Ruby oraz Python na JVM znane odpowiednio jako JRuby i Jython. Mimo tylu możliwości wciąż jednak żaden z nich nie jest uznawany za godnego następcę Javy. Ostatnio jednak szybko zyskuje popularność nowy język: Kotlin. Co on prezentuje i czy jest godny bycia następcą Javy?

KOTLIN

Kotlin jest tworzony i rozwijany przez JetBrains oraz społeczność Open Source. JetBrains to najpotężniejszy aktualnie dostawca środowisk programistycznych na świecie i twórca Idei Intellij, PyCharm, PhpStorm i wielu innych. Sam język nie jest jednak uzależniony od Intellij'a, ponieważ wydawana jest bardzo dobra wtyczka do Eclipse, dzięki której tam też można z niego korzystać.

Celem tego języka jest maksymalne uproszczenie programowania na JVM przy zachowaniu szybkości działania i kompilacji Javy. Środowisko wzorowało się głównie na takich językach jak Scala, Swift, Groovy i Python. Twórcy skupili się jednak na tym, by wyciągnąć wnioski nie tylko z ich sukcesów, ale i z błędów. Tak właśnie zrodził się Kotlin.

Pierwsza stabilna wersja tego języka została wydana w lutym 2016 roku. Jest więc on bardzo młody, ale jednocześnie dostatecznie dojrzały, by był wykorzystywany komercyjnie. Nawet w polskich firmach. Na przykład aplikacja doktorska ZnanegoLekarza została napisana w Kotlinie. Sam Kotlin ma już wiele bibliotek oraz wtyczek napisanych specjalnie pod niego. W końcu jest on rozwijany do 2011 roku i liczne grono fanów korzysta z niego już od kilku lat.

Kotlin jest w pełni kompatybilny z Javą – można zamienić tylko jedną klasę w projekcie z Javy na Kotlina i wszystko będzie działało. Jest też bardzo lekki. A jaką wartość wnosi? Przede wszystkim niesamowitą ilość uproszczeń i cukru składniowego (syntactic sugar). Także wiele zabezpieczeń, które zachęcają programistę do pisania lepszego kodu (null-safety, powszechne stałe i obiekty immutable). Z własnego doświadczenia wiem, że przejście w projekcie na Kotlina zapewnia ok. 50% redukcję kodu oraz dużo większą jego czytelność. Dzięki pewnym mechanizmom dostajemy też „za darmo" wykrycie i przeciwdziałanie przynajmniej kilku trudnym do wykrycia błędom. W tym miejscu chciałbym zaprezentować kilka udogodnień wprowadzonych przez Kotlina.

ZWIĘZŁA STRUKTURA

Weźmy typową klasę napisaną w Javie zawierającą pola typu String przedstawioną na Listingu 1.

Listing 1. Przykładowa klasa przechowująca dane osoby napisana w Javie

```
class Person {
  String name;
  String surname;

  Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
  }

  String getName () {
    return name;
  }

  String getSurname () {
    return surname;
  }

  void setName (String name) {
    this.name = name;
  }

  void setSurname (String surname) {
    this.surname = surname;
  }
}
```

Co właściwie ta klasa robi? Wyłącznie przechowuje i udostępnia swoje pola. Czemu więc zajmuje tyle linijek? Jest to jeden z częstszych wzorców wykorzystywanych w Javie, znany jako Java Bean. Warto jednak zauważyć, że ta klasa nic nie robi. Przechowuje tylko pola. To niesamowite, że wymaga tyle boilerplate'u. Kotlin ma inne podejście – rzeczy częste i proste powinny być krótkie i proste. W Kotinie ten sam kod będzie wyglądał jak na Listingu 2.

Listing 2. Ten sam objekt, co na Listingu 1, ale w Kotlinie

```
class Person (var name: String, var surname: String)
```

Co, jeśli chcemy, by wartości nie były zmienne? Wystarczy zamienić zmienne na stałe jak na Listingu 3.

Listing 3. Obiekt o stałych wartościach. Tworzy się go przez zamianę definicji wartości z var (od variable) na val (od value)

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

To, co znajduje się w nawiasie, to parametry głównego konstruktora. Te pola równie dobrze można by umieścić w ciele klasy jak na Listingu 4.

Listing 4. Inna forma definicji obiektu z Listingu 3

```
class Person (name: String, surname: String) {
  val name = name
  val surname = surname
}
```

Zastanawiać może, czemu publiczne pola name i surname nie mają zdefiniowanego typu. Typ ten jest wnioskowany z przypisywanej wartości. Tutaj środowisko widzi, że name w konstruktorze ma typ String, więc w czasie przypisania tak samo typuje pole name w ciele klasy. Jak widać, dzięki inteligencji środowiska Kotlin pozostaje silnie typowany, mimo iż bardzo często nie wymaga typowania.

Wiele osób mając po raz pierwszy kontakt z takim rozwiązaniem, może się niepokoić, co z getterami i setterami. Wciąż można zrobić z nimi wszystko, co chcemy. Dla przykładu zmienię tę klasę tak, by pole name można było zmienić tylko lokalnie oraz by zwracane w getterze name i surname zawsze zaczynało się z dużej litery. Tak zdefiniowana klasa została przedstawiona na Listingu 5.

Listing 5. Przykład dodania getterów i setterów

```
class Person (name: String, surname: String) {
  var name = name
    get() = surname.capitalize()
    private set
  var surname = surname
    get() = surname.capitalize()
}
```

ZABEZPIECZENIE PRZED NULLAMI

Złośliwi mówią, że „Java jest zoptymalizowana do rzucania null pointer exception". Faktycznie, nie ma ona właściwie żadnego mechanizmu, który zabezpieczałby przed taką sytuacją, i najprawdopodobniej to jest właśnie najczęstszy błąd, z jakim się spotykamy w Javie. W Kotlinie zmienna zadeklarowana dowolnego typu nie ma prawa mieć wartości null. Aby mogła, musi być to jasno określone poprzez dodanie „?" na końcu jej deklaracji. Wtedy jednak nie będzie można z niej skorzystać bez rozpakowania jej do zmiennej nienullowalnej. Zamiast jednak to opisywać, na Listingu 6 przedstawiłem przykłady operacji na zmiennych nullowalnych i nienullowalnych.

Listing 6. Przykłady operacji na zmiennych nullowalnych i nienullowalnych

```
// var str: String = null nie działa,
// bo String nie może mieć wartości null
val str2: String = "Some string"
var str3: String? = null
println (str2?.length) //11
println (str3?.length) //null
println (str3 ?: "Some str") //Some str
println ((str3 ?: str2).length) //11

val str: String? = "Nullable"
// println( str.length ) nie działa bo wartość
// nullowalną trzeba najpierw odpakować
if (str != null)
  println( str.length )

if (str == null)
  return
//Ten if odpakowuje str i zamienia ją na String
println( str.length ) //11
```

Wszystko coś zwraca

W Kotlinie właściwie wszystko zwraca coś jako swój rezultat. To pozwala na przyjemną, zwięzłą składnię typu jak w przypadku funkcji przedstawionych na Listingu 7.

Listing 7. Krótkie definicje funkcji

```
fun sum(i: Int, j: Int) = i + j
fun bigger(i: Int, j: Int) = if (i > j) i else j
```

Nieobytych może zaskoczyć, że nawet blok try catch zwraca wynik ostatniej operacji. Przykład zastosowania tej właściwości można zobaczyć na Listingu 8. Jak bardzo jest to wygodne, przekonamy się, gdy uświadomimy sobie, że aby zapisać to samo, nie wykorzystując tej właściwości, kod musiałby wyglądać jak na Listingu 9. Byłby on więc nie tylko dłuższy, ale i mniej czytelny.

Listing 8. Przykład wykorzystania faktu, że struktura try-catch zwraca ostatnią wartość

```
val num = try {
  count()
} catch (e: Exception) {
  -1
}
```

Listing 9. Kod dający identyczny efekt, co kod z Listingu 8, ale nie korzystający z faktu zwracania ostatniej wartości przez try-catch

```
var tempNum: Int? = null
try {
  tempNum = count()
} catch (e: Exception) {
  tempNum = -1
}
val num = tempNum!!
```

WHEN

Jednym z moich ulubionych uproszczeń wprowadzonych w Kotlinie jest struktura when. Jest ona podobna do switch-case, ale ma znacznie większe możliwości. Jest to struktura zaczerpnięta ze Scali (match-case), gdzie jest intensywnie wykorzystywana i mocno doceniana. Przykład jej wykorzystania przedstawiam na Listingu 10.

Listing 10. Przykładowa funkcja zawierająca when

```
fun description(a: Any?) = when(a) {
  null -> "There is nothing there"
  is Int -> "It is number $a"
  is Float, Double -> "It is float or double $a"
  is String -> "It is string \"$a\""
  !is Object -> "It is not object"
  else -> "OMG. Have no idea what is it"
}
```

FUNKCJE ROZSZERZAJĄCE

Wreszcie czas na coś, z czego Kotlin stał się znany także w kręgach, którym obcy jest JVM, a mianowicie na funkcje rozszerzające. Brakuje w Stringu jakiejś metody? Ponieważ w Javie nie ma funkcji rozszerzających, to kończyło się to zwykle długimi StringUtills, IntUtills, ListUtills itp. Przykład, jak to się robi w Kotlinie, przedstawiam na Listingu 11.

Listing 11. Przykład definicji i wykorzystania funkcji rozszerzającej

```
fun String.startWithUpperCase() = this[0].isUpperCase()
"Marcin".startWithUpperCase() // true
fun Context.getInflanter() = LayoutInflater.from(this)
// Wtedy gdy piszemy kod wewnętrz klasy dziedziczącej
// po context, możemy ją wywołać
getInflanter()
// Lub po prostu
Inflater
```

Innym ciekawym przykładem funkcji rozszerzającej jest bardzo przydatna funkcja dołączona do biblioteki standardowej apply. Jej definicja pokazana jest na Listingu 12.

Listing 12. Definicja funkcji apply z biblioteki standardowej. Jest generycznym rozszerzeniem funkcji i zawiera funkcję lambda w kontekście tego rozszerzenia

```
public inline fun <T> T.apply(block: T.() -> Unit): T {
  block();
  return this
}
```

Jest ona nie tylko funkcją rozszerzającą, ale również ma w parametrze metodę lambda rozszerzającą klasę. Cała metoda jest też generyczna. Aby zrozumieć, do czego służy, wystarczy porównać Listingi 13 i 14.

Listing 13. Przykład ustawiania wartości przykładowego obiektu

```
holder.input.inputType = inputType
holder.input.setText(defaultValue)
holder.inputLayout.hint = hint
holder.input.isEnabled = isActive
this.holder = holder
```

Listing 14. Prezentacja działania funkcji apply – kod dający taki sam efekt jak ten z Listingu 13

```
this.holder = holder.apply {
  input.inputType = inputType
  input.setText(defaultValue)
  input.isEnabled = isActive
  inputLayout.hint = hint
}
```

Kiedy przewidywano to rozwiązanie, nikt chyba nie przypuszczał, jakie będą jego dalsze konsekwencje. Otworzyło to drogę dla takich bibliotek jak Kotson, która pozwala na zapis plików JSON w Kotlinie (Listing 15); Anko, która pozwala pozbyć się paskudnych XMLi opisujących ekrany w Androidzie (Listing 16), oraz wreszcie Kobalta, który pozwala na zamianę Gradle na plik Kotlina (Listing 17). Jak widać – dzięki temu Kotlin pozwala na wygodny zapis struktur hierarchicznych. A ponieważ jest silnie typowany, to zapewnia korekcję i podpowiedzi na każdym z tych poziomów. Większość plików towarzyszących projektom programistycznym to takie właśnie struktury (XML, Maven i Gradle, JSON), więc wprowadza to zupełnie nową jakość. Ponieważ Kotlin może być także kompilowany do JavaScript (na razie beta, ale ma wyjść pierwsza oficjalna wersja na przestrzeni najbliższych miesięcy), to w społeczności Kotlina mówi się, że przyszłością są aplikacje All in Kotlin. Jest to jak najbardziej osiągalna wizja, tym bardziej że Kotlin pozwala także na pisanie skryptów.

Listing 15. Przykładowy Json w Kotsonie. Przykład pochodzi z https://github.com/SalomonBrys/Kotson

```
val obj: JsonObject = jsonObject(
  "property" to "value",
  "array" to jsonArray(
    21,
    "fourty-two",
    jsonObject("go" to "hell")
  )
)
```

Listing 16. Anko – struktura przedstawiająca element zawierający 2 pola do wpisania imienia i hasła oraz przycisk logowania. Przykład pochodzi z https://github.com/Kotlin/anko

```
verticalLayout {
  padding = dip(30)
  editText {
    hint = "Name"
    textSize = 24f
  }
  editText {
    hint = "Password"
    textSize = 24f
  }
  button("Login") {
    textSize = 26f
  }
}
```

Listing 17. Kobalt – cały plik Gradle. Przykład pochodzi z http://beust.com/kobalt/home/index.html

```
val jcommander = project {
  name = "jcommander"
  group = "com.beust"
  artifactId = name
  version = "1.52"

  dependenciesTest {
    compile("org.testng:testng:6.9.5")
  }

  assemble {
    mavenJars {
    }
  }

  bintray {
    publish = false
  }
}
```

PODSUMOWANIE

Kotlin jest bardzo przyjemnym i efektywnym językiem. Dzięki silnemu typowaniu oraz dobremu mechanizmowi zabezpieczenia przed nullami zapewnia dużą odporność na błędy. Dzięki licznym udogodnieniom, takim jak wartości domyślne, odwołania po nazwie w funkcjach, delegaty, klasy typu `data` i wiele innych, jest on wygodny i nowoczesny. Jednocześnie nie ustępuje on Javie szybkością kompilacji ani działań. Można go również stosować w projektach komercyjnych.

Kotlin jest jednym z wielu języków, które pretendują do miana następcy Javy. Pracuje nad nim jednak JetBrains i robi to naprawdę dobrze. Język rozwija się sprawnie i w sposób bardzo przemyślany. W swojej praktyce zdążyłem przetestować właściwie wszystkich pretendentów do bycia następcą Javy i muszę powiedzieć, że moim zdaniem to Kotlin rokuje największe nadzieje. Trudno przewidzieć, co przyszłość przyniesie. Gdybym jednak miał stawiać na zwycięzcę, to postawiłbym właśnie na Kotlina.

MARCIN MOSKAŁA
marcin.moskala@docplanner.com
programista Kotlin i Groovy na platformę Android pracujący dla ZnanegoLekarza, autor bloga KotlinDev.pl oraz współzałożyciel Nootro (www.nootro.pl). Pasjonat samorozwoju oraz tworzenia pięknego i prostego kodu.