Warsztat 2

1. Obiekty singleton
2. Zmienne
3. Instrukcje i wyrażenia
4. Funkcje jako zmienne
5. Testy i asercje w ScalaTest

Programowanie w języku Scala

Mateusz Walczak 239723

Konrad Kajszczak 239668

Bartłomiej Jencz 239665

Obiekty Singleton

- Scala umożliwia użycie słowa kluczowego *Object* zamiast słowa Class. W ten sposób automatycznie tworzony jest pojedynczy obiekt, który może posiadać tylko jedną instancję.
- Ze względu na brak możliwości tworzenia funkcji statycznych w Scala, wszystkie funkcje i pola zadeklarowane wewnątrz obiektu Singleton są dostępne globalnie.
- Obiekt Singleton może rozszerzać klasy.

Przykładowy obiekt Singleton

```
object Main {
    def main(args: Array[String]): Unit = {
        printMessage("Hello world")
    }

    def printMessage(message: String): Unit = {
        println(message)
    }
}
```

Rozszerzanie klasy przez obiekt Singleton

```
class Tool{}

object Hammer extends Tool{
    var price = 45
    def getPrice() = price
}

object Hello extends App {
    println(Hammer.getPrice())
}
```

Obiekty towarzyszące

- Companion object (obiekt towarzyszący).
- Obiekt towarzyszący to obiekt, którego nazwa jest taka sama jak nazwa klasy, którą nazywamy wtedy klasą towarzyszącą (ang. companion class).
- Klasa i obiekt towarzyszące mogą uzyskać dostęp do swoich prywatnych pól i metod.

- ScalaBook: https://docs.scala-lang.org/overviews/scala-book/companion-objects.html

Przykładowy obiekt towarzyszący

```
class Person {
    var name: String = ""
    var age: Int = 0
    override def toString = s"$name, $age"
}

object Person {
    def apply(name: String): Person = {
        val p = new Person
        p.name = name
        p
    }
}
```

Zmienne

W Scala wyróżniamy dwa rodzaje deklaracji:

- **Var** – zmienna, której wartość można dowolnie modyfikować.
- **Val** – stała, której wartości nie można zmieniać po ustawieniu (odpowiednik final w Java).

Pola i zmienne mogą być deklarowane jako var i val, natomiast parametry funkcji zawsze są stałe (val).

Zmienne – wnioskowanie typów

W języku Scala nie ma konieczności deklaracji typu zmiennej. W przypadku nieokreślonego typu kompilator wykrywa typ zmiennej na podstawie jej wartości. Oczywiście istnieje możliwość ręcznego określenia typu zmiennej.

```
var name = "Tomasz" // Kompilator wnioskuje typ String dla zmiennej Name

var secondName = "Adam" : String //  Typ dla zmiennej secondName jest
określony (String)
```

Instrukcje

W większości instrukcje sterujące w języku Scala działają analogicznie do tych znanych z Java. Różnice występują głównie w przypadku słowa *return,* pętli *for* oraz konstrukcji *switch*. W niektórych przypadkach możemy pominąć słowo *return*. Wówczas zwracana jest ostatnia używana wartość. Dla przykładu, poniższa funkcja zwróci wartość 3:

```
def getThree() : Integer = 3
```

Instrukcje - pętla for

Standardowa konstrukcja pętli w Scala wygląda następująco:

```
for(i <- 1 to 20){
    println(i);
}
```

Pętla *for* w języku Scala daje nam o wiele więcej możliwości niż w Java. Możemy na przykład filtrować wyniki zwracane w pętli. Poniższa pętla wypisuje cyfry parzyste z zakresu 1 do 10.

```
for(x <- 1 to 10 if x%2==0) {
    println(x);
}
```

Instrukcje - pętla for - słowa kluczowe

Do sterowania pętlą for w języku Scala wykorzystuje się słowa kluczowe:

"**To**" - określa przedział (np. 1 to 10) i bierze pod uwagę jego krańce.

"**Until**" - działa podobnie do "To", ale nie bierze pod uwagę górnego krańca przedziału.

"**By**" - pozwala ominąć podaną liczbę iteracji, np. Pętla `for(i<-1 to 10 by 2)` wykona tylko nieparzyste iteracje.

"**Yield**" - zwraca wartość po wykonaniu wszystkich iteracji pętli. Zmienną *result* będzie więc wektor zawierającą cyfry od -1 do 8.

```
var result = for(a <- 1 to 10) yield a-2
```

Instrukcje - switch

W języku Scala zastąpiono konstrukcję *switch* bardziej funkcjonalną konstrukcją *match*. Konstrukcja *match* pozwala na dopasowanie wzorca do obiektu. Ponadto umożliwia również porównywanie zmiennych różnych typów.

```
val x = 0
val ret = x match
{
case 0 => "zero"
case 1 => "one"
case 2 => "two"
case _ => "other"
}
print(ret)
```

```
val x:Any = 3
val ret = x match
{
case 1 => "one"
case 2 => "two"
case "three" => "three"
case _ => "other"
}
print(ret)
```

Wyrażenia (Funkcje jako zmienne)

Wyrażeniem w języku Scala nazywamy kombinację zmiennych, stałych, operatorów i funkcji. Wartości wyrażenia mogą być przypisywane do zmiennych lub stałych. Dla przykładu, poniższy blok kodu zapisze w zmiennej result wartość 97.

```
def fun() : Integer = 10
var x = 9
var result = fun()*x + 7
```

Testing and Assertions

ScalaTest udostępnia następujące klasy (sposoby) do tworzenia testów:

- FunSuite
- FlatSpec
- FunSpec
- FeatureSpec
- WordSpec
- FreeSpec
- PropSpec

FunSuite

- Testy jednostkowe przeznaczone dla osób zapoznanych z **JUnit**
- Z dokumentacji Scala Test:

  *"For teams coming from xUnit, the **FunSuite** style feels comfortable and familiar while still giving some of the benefits of BDD: the **FunSuite** style makes it easy to write descriptive test names, natural to write focused tests, and generates specification-like output that can facilitate communication among stakeholders."*

  Przykładowy test typu Fun Suite

FlatSpec

- Dobry pierwszy krok na przeniesienie się z testów typu **JUnit** w stronę testów typu **BDD**
- Z dokumentacji Scala Test:

  *"A good first step for teams wishing to move from xUnit to BDD, the **FlatSpec** style's structure is flat like xUnit, so simple and familiar, but the test names must be written in a specification style: "X should Y," "A must B," etc."*

Przykładowy test typu Flat Spec

Prop Spec

- Bardzo wygodny do testowania pojedynczych funkcji dla wielu różnych parametrów wejściowych.
- Z dokumentacji Scala Test:

> "*The **AnyPropSpec** style is perfect for teams that want to write tests exclusively in terms of property checks; also a good choice for writing the occasional test matrix when a different style trait is chosen as the main unit testing style.*"

Przykładowy test typu Prop Spec

Feature Spec

- Styl **FeatureSpec** jest przeznaczony do testów akceptacyjnych.
- Ułatwia zrozumienie kodu dla osób którzy nie są programistami i ułatwia zdefiniowanie wymagań akceptacyjnych.
- Celem testów akceptacyjnych jest nabranie zaufania do systemu, jego części lub pewnych atrybutów niefunkcjonalnych. (więcej informacji: https://testerzy.pl/baza-wiedzy/poziomy-testowania)
- Z dokumentacji Scala Test:

  *"The **FeatureSpec** style is primarily intended for acceptance testing, including facilitating the process of programmers working alongside non-programmers to define the acceptance requirements."*

  Przykładowy test typu Feature Spec

Fun Spec

- Rozwinięcie testów **FlatSpec** o zagnieżdżanie
- Z dokumentacji Scala Test:

  *"For teams coming from Ruby's RSpec tool, the **FunSpec** style will feel very familiar; More generally, for any team that prefers BDD, FunSpec 's nesting and gentle guide to structuring text (with describe and it) provides an excellent general-purpose choice for writing specification-style tests."*

  Przykładowy test typu Fun Spec

Word Spec

- Z dokumentacji Scala Test:

  *"For teams coming from specs or specs2, the **WordSpec** style will feel familiar, and is often the most natural way to port specsN tests to ScalaTest. AnyWordSpec is very prescriptive in how text must be written, so a good fit for teams who want a high degree of discipline enforced upon their specification text."*

  Przykładowy test typu Word Spec

Free Spec

- Pozwala no kompletnie dowolne podejście do tworzenia testów.
- Z dokumentacji Scala Test:

  *"Because it gives absolute freedom (and no guidance) on how specification text should be written, the **FreeSpec** style is a good choice for teams experienced with BDD and able to agree on how to structure the specification text."*

  Przykładowy test typu Free Spec

Ćwiczenia

Napisz klasę testującą X, która za pomocą wybranego przez ciebie rodzaju testu:

- Testuje funkcję `cube()` klasy `CubeCalculator` obliczającej sześcian z danej liczby.
- Testuje funkcję `factorial()` klasy `FactorialCalculator` obliczającej silnię z danej liczby.
- Testuje funkcję `scalperBuy()` klasy `GraphicsCardShop`.

Dziękujemy za uwagę

- Mateusz Walczak 239723
- Konrad Kajszczak 239668
- Bartłomiej Jencz 239665

Repozytorium GitHub

Dokumentacja ScalaTest