	Wydział Politechniczny **Katedra Informatyki**
Prowadzący	Tomasz Gądek
Kurs	Narzędzia i środowiska programistyczne
Rok / Semestr	1 / Letni
Temat	Lab03: Podstawy programowania w języku Kotlin cz. 3.

Data ostatniej modyfikacji: **01-10-2024**

Tablica

Tablica jest to struktura danych, która może zawierać wiele elementów tego samego typu. Tablica w języku Kotlin jest reprezentowana przez jedną zmienną. Do elementów możemy dostać się za pomocą indeksu, czyli każdy element umieszczony w tablicy posiada unikalny identyfikator. Poniżej prezentujemy przykład utworzenia pustej tablicy oraz tablicy, która posiada już elementy.

```
fun main() {
   var emptyArray:Array<Int> = arrayOf()
   var arrayOfDigits = arrayOf(1, 2, 3)
}
```

Zmienna **emptyArray** przechowuje pustą tablicę. Podczas tworzenia takiej tablicy wymagane jest deklarowanie typu elementów jakie będzie przechowywała. Druga linijka prezentuje inicjalizację tablicy, która już posiada elementy. W takim przypadku nie musimy deklarować konkretnego typu.

Na tablicach możemy wykonywać kilka podstawowych operacji: odczytać konkretny element, aktualizować element tablicy, dodawać element do tablicy.

Spójrz, jak możemy odczytać element z tablicy.

```
fun main() {
   var digits = arrayOf(1, 2, 3)
   var result = digits[1]

   println(result)
}
```

Rezultat działania programu jest następujący:

```
2
```

To nie jest błąd! Elementy w tablicach numerowane są od 0. Odwołując się do elementu o indeksie 1 tak naprawdę odwołujemy się do drugiego elementu.

Aktualizacja elementu tablicy polega na odwołaniu się do konkretnego elementu i przypisaniu mu nowej wartości za pomocą operatora **=**. Spójrz, jak taka operacja wygląda w praktyce.

```
fun main() {
   var result = arrayOf(1, 2, 3)
   result[1] = 5

   println(result.joinToString())
}
```

Rezultat działania kodu jest następujący.

```
1, 5, 3
```

Operacja przebiegła pomyślnie. Drugi element został zaktualizowany. Zwróć uwagę, że do wyświetlenia zawartości tablicy result wykorzystaliśmy następującą konstrukcję: **result.joinToString()**. Dzięki takiemu zapisowi mogliśmy ładnie sformatować wyniki (wywołaliśmy metodę **joinToString()** na tablicy **result**).

Kolejną operacją, którą omówimy będzie operacja dodawania. Spójrz na poniższy kod.

```
fun main() {
    var result = arrayOf(1, 2, 3)
    result = result.plus(10)
    println("Size = ${result.size}")
    println("Array = ${result.joinToString()}")
}
```

Rezultat działania programu jest następujący.

```
Size = 4
Array = 1, 2, 3, 10
```

Dodawanie elementu na koniec tablicy odbywa się przy pomocy funkcji **plus()**. Od tego momentu nasza tablica posiada nowy rozmiar. Rozmiar tablicy możemy zweryfikować za pomocą następującej konstrukcji **result.size**, czego dowodem są nasze wyniki.

Pętla for

Pętla **for** to troszkę inny rodzaj pętli. Jest to pętla, która potrafi iterować po kolekcjach. Teraz możemy przeglądać elementy tablicy w bardzo zgrabny sposób. Spójrz na poniższy przykład zastosowania pętli programowej **for** w praktyce.

```
fun main() {
    val digits = arrayOf(1, 2, 3)

    for (digit in digits) {
     println("Element: ${digit}")
    }
}
```

Rezultat jest następujący.

```
Element: 1
Element: 2
Element: 3
```

Pętla programowa **for** posiada ciekawą konstrukcję. Po prawej stronie operatora in podajemy kolekcję, a po lewej stronie zmienną tymczasową, która będzie reprezentowała w iteracjach kolejne elementy tablicy **digits**. Nie potrzebujemy tutaj żadnych dodatkowych liczników. Pętla jest bardzo wygodna w użyciu.

Zakres

Zakres reprezentuje sekwencję wartości. Wystarczy podać zakres, które rozdzielimy kropkami ".." i mamy gotowy zbiór wygenerowanych wartości. Nie musimy już mozolnie generować zakresów, wystarczy jedna linijka.

Budując nasz generator zakresu wartości możemy posługiwać się konkretnymi słowami kluczowymi, które świadczą o sposobie generowania danych.

- **step** - słowo kluczowe pozwala określić krok pomiędzy kolejnymi wartościami w zakresie.
- **downTo** - słowo kluczowe pozwala na tworzenie zakresu w odwrotnej kolejności, np. gdy chcemy iterować po liczbach malejąco.
- **until** - słowo kluczowe, który pozwala na utworzenie zakresu, który nie bierze pod uwagę ostatniej wartości.

Zakresami nie muszą być liczby całkowite. Mogą być również generowane zakresy znaków. Spójrz na poniższy kod, który prezentuje praktyczne użycie zakresów. Zakresy idealnie nadają się do współpracy z pętlami programowymi.

```
fun main() {
    var range1 = 1..5 // 1 2 3 4 5
    var range2 = 5 downTo 1 // 5 4 3 2 1
    var range3 = 1 until 5 // 1 2 3 4
    var range4 = 0..5 step 2 // 0 2 4
    var range5 = 'a'..'c' // a b c

    for(result in range1) {
        print("${result} ") // 1 2 3 4 5
    }
    println()

    for(result in range2) {
        print("${result} ") // 5 4 3 2 1
    }
    println()

    for(result in range3) {
        print("${result} ") // 1 2 3 4
    }
    println()

    for(result in range4) {
        print("${result} ") // 0 2 4
    }

    println()

    for(result in range5) {
        print("${result} ") // a b c
    }
}
```

Rezultat jest następujący.

```
1 2 3 4 5
5 4 3 2 1
1 2 3 4
0 2 4
a b c
```

Funkcja

Funkcja jest to nazwany blok kodu, który wykonuje określone zadanie. Funkcje umożliwiają zorganizowanie kodu na mniejsze bloki. Utworzone bloki mogą być wielokrotnie używane (wywoływane), co ułatwia ponowne wykorzystanie kodu. Dodatkowo użycie funkcji ma wpływ na czytelność kodu.

Pierwszy przykład prezentuje funkcję, która nie przyjmuje parametrów oraz nie zwraca żadnej wartości. Spójrz na przykładowy kod.

```
fun add() {
   var a = 1
   var b = 2

   println("${a} + ${b} = ${a + b}")
}

fun main() {
   add()
}
```

Rezultat jest następujący.

```
1 + 2 = 3
```

Funkcję w kodzie oznaczamy słowem kluczowym **fun**. Po słowie występuje nazwa funkcji, w naszym przykładzie jest to **add**. Następnie występują nawiasy, za pomocą których możemy przekazywać dane z zewnątrz, ale o tym za chwilę. Pomiędzy klamrami **{** i **}** znajduje się ciało funkcji. W ciele umieszczamy operacje, które będą reprezentowane przez nazwę naszej funkcji. W funkcji **main()** znajduje się wywołanie funkcji **add()**. Polega ono na podaniu nazwy funkcji i nawiasów. Wywołując funkcję powodujemy, że wykona się kod widoczny w ciele funkcji **add()** (czyli pomiędzy nawiasami **{** i **}**).

Kolejny przykład funkcji to funkcja, która przyjmuje parametry. Spójrz na poniższy kod.

```
fun add(a: Int, b: Int) {
   println("${a} + ${b} = ${a + b}")
}

fun main() {
   add(1, 2)
   add(2, 3)
}
```

Rezultat jest następujący.

```
1 + 2 = 3
2 + 3 = 5
```

Teraz nasz funkcja działa bardziej uniwersalnie. Wywołaliśmy ją 2 razy i wykonuje obliczenia dla różnych wartości wejściowych. Zmodyfikowaliśmy naszą funkcję dodaliśmy parametry pomiędzy nawiasami **(** i **)**. Teraz mamy fragment kodu, który możemy używać w naszej aplikacji w wielu miejscach.

Ostatnim rodzajem funkcji jakim się zajmiemy to funkcja, która przyjmuje i zwraca wartość. Funkcja, która zwraca wartość może być przydatna, gdy chcemy jej rezultat przypisać do zmiennej. Spójrz na zmodyfikowaną funkcję **add()**.

```
fun add(a: Int, b: Int): Int {
   return a + b
}

fun main() {

   var a = 1
   var b = 2
   var c = 3

   var result1 = add(a, b)
   var result2 = add(b, c)

   println("${a} + ${b} = ${result1}")
   println("${b} + ${c} = ${result2}")
}
```

Rezultat działania programu jest następujący.

```
1 + 2 = 3
2 + 3 = 5
```

Rezultat jest podobny, ale skorzystaliśmy z funkcji, która zwraca wartość. To że funkcja może zwracać wartość świadczy o tym zadeklarowany typ po nawiasach okrągłych. Taki zapis pozwoli zwracać wartość z wnętrza funkcji. Aby zwrócić wartość z funkcji możemy skorzystać z instrukcji powrotu **return**.

Łańcuchy znaków - podstawowe operacje

Jak zamienić fragment ciągu w łańcuchu tekstowym?

Metoda **replace()** w Kotlinie służy do zamiany wszystkich wystąpień określonego ciągu znaków na inny ciąg znaków wewnątrz łańcucha. Oto prosty przykład:

```
fun main() {
    val quote = "I'll be back."
    val modifiedQuote = quote.replace("I'll", "He'll")

    println("Original quote: $quote")
    println("Modified quote: $modifiedQuote")
}
```

Rezultat jest następujący.

```
Original quote: I'll be back.
Modified quote: He'll be back.
```

Jak wyszukać pierwsze wystąpienie ciągu w łańcuchu tekstowym?

Metoda **indexOf()** w Kotlinie służy do znajdowania pozycji pierwszego wystąpienia określonego ciągu znaków w łańcuchu tekstowym. Oto prosty przykład:

```
fun main() {
   val quote = "May the Force be with you."
   val searchPhrase = "Force"
   val index = quote.indexOf(searchPhrase)

   if (index != -1) {
       print("The phrase '$searchPhrase' was ")
       println("found at position: $index")
   } else {
       println("The phrase '$searchPhrase' was not found.")
   }
}
```

Rezultat jest następujący.

```
The phrase 'Force' was found at position: 8
```

Jak wyodrębnić fragment ciągu z łańcucha tekstowego?

Funkcja **substring()** w Kotlinie służy do pobierania fragmentów łańcuchów znaków (String). Pozwala ona na wyodrębnienie części łańcucha na podstawie indeksów znaków, które chcesz pobrać.

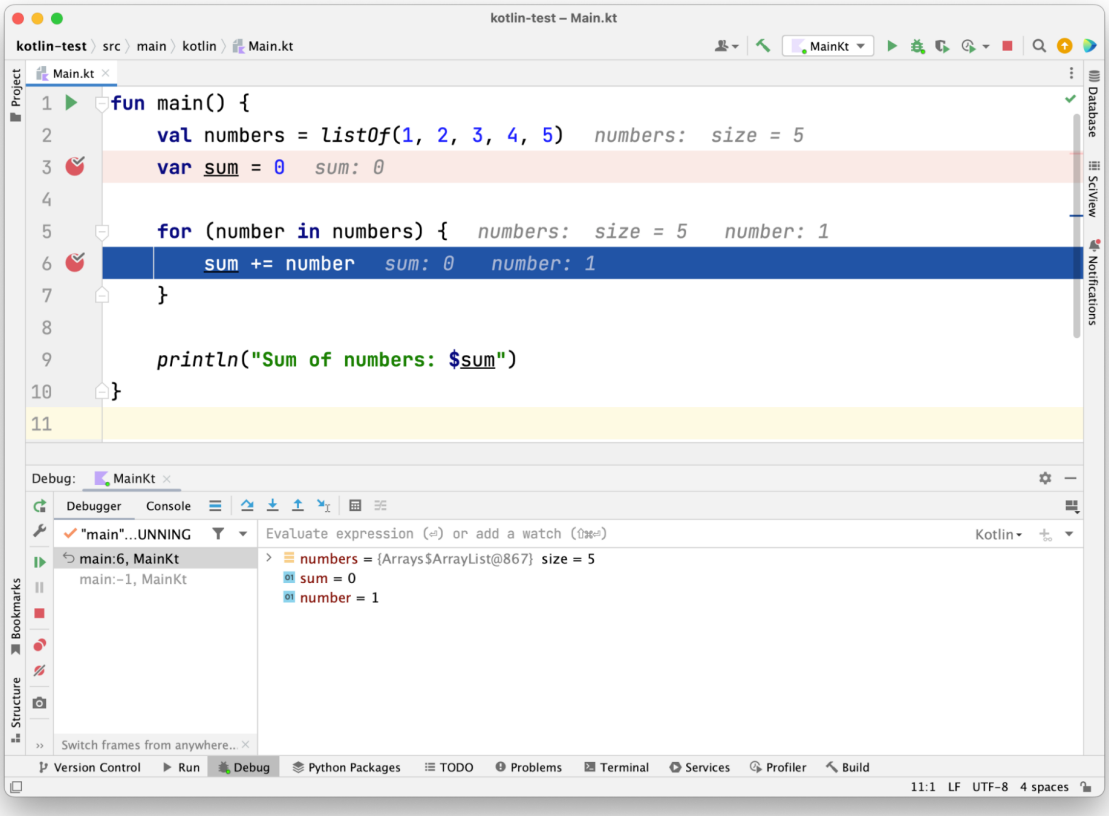
```
fun main() {
    val sentence = "To be or not to be, that is the question."

    // Wycinanie od indeksu 13 do końca
    val substring1 = sentence.substring(13)
    println("Substring 1: $substring1")

    // Wycinanie od indeksu 0 do 5 (bez indeksu 5)
    val substring2 = sentence.substring(0, 5)
    println("Substring 2: $substring2")

    // Wycinanie od indeksu 14 do 17 (bez indeksu 17)
    val substring3 = sentence.substring(14, 17)
    println("Substring 3: $substring3")
}
```

Rezultat.

```
Substring 1: to be, that is the question.
Substring 2: To be
Substring 3: o b
```

Debugger

Debugger to narzędzie używane przez programistów do znajdowania i naprawiania błędów w kodzie komputerowym. Działa on poprzez umożliwienie programiście śledzenia działania programu linia po linii oraz monitorowanie wartości zmiennych w trakcie jego wykonania.



Checkpointy (czasem nazywane także punktami kontrolnymi) są jedną z przydatnych funkcji debuggera. Są to punkty w kodzie, które programista może ustawić, aby zatrzymać działanie programu w konkretnym miejscu. Kiedy debugger napotka checkpoint, zatrzyma wykonanie programu, umożliwiając programiście zbadanie stanu programu w tym konkretnym punkcie. Dzięki checkpointom można dokładnie sprawdzać, co się dzieje w kodzie w różnych momentach jego wykonywania.

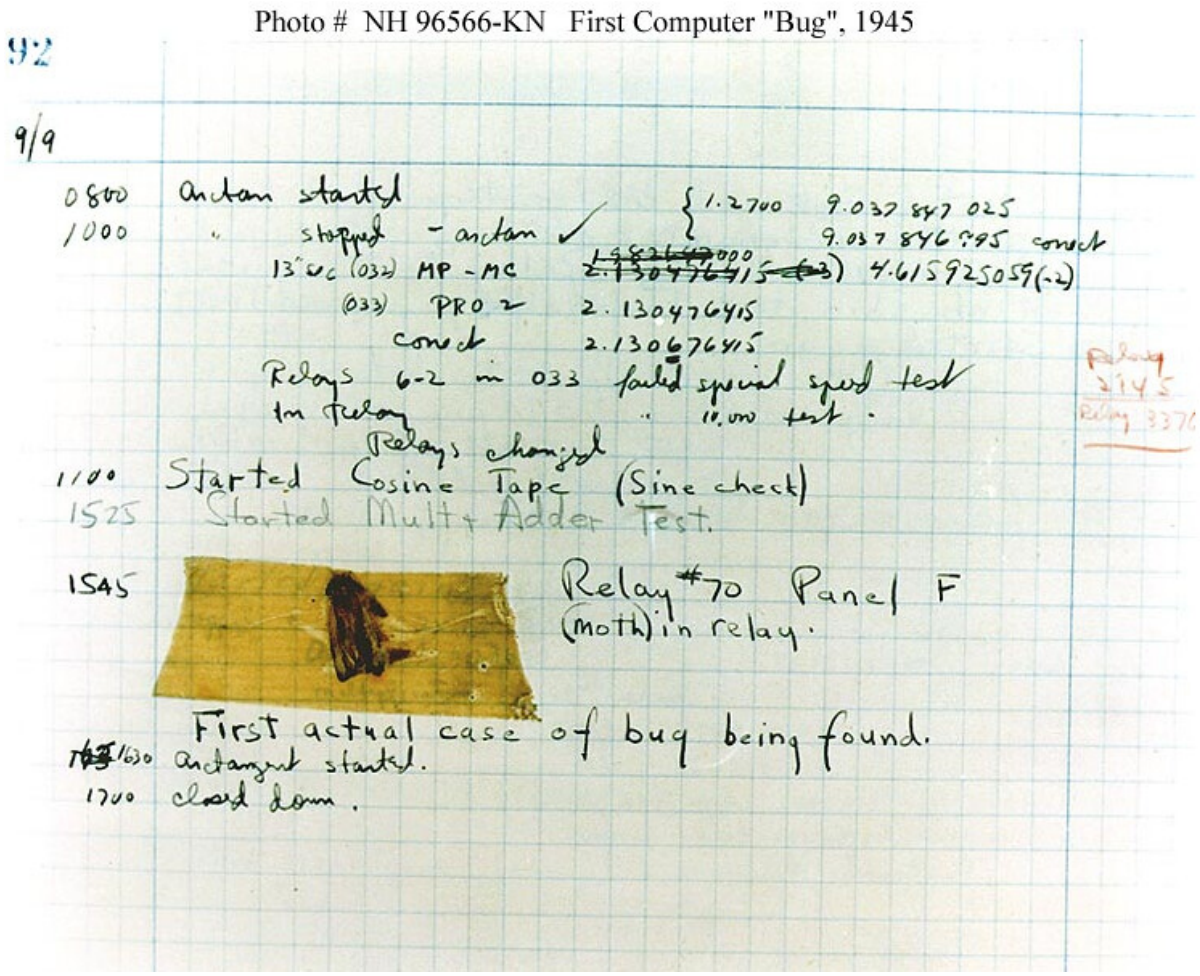
Obejrzyj filmik i przekonaj się w praktyce, jak działa debugger.

Zadania

```
fun calculateSum(n: Int): Int {
    var sum = 0


    for (i in 1 until n) {
        sum += i
    }


    return sum
}


fun main() {
    val n = 3
    val sum = calculateSum(n)
    println("Sum from 1 to $n: $sum")
}
```

- Funkcja **calculateSum()** działa nieprawidłowo. Przekazując do niej argument równy 3 funkcja powinna zwrócić 6. Zadaniem funkcji jest sumowanie elementów od 1 do n. Użyj debuggera do zlokalizowania problemu.
- Napisz funkcję, która obliczy obwód koła. Funkcja powinna przyjmować jeden parametr **r** oraz nie powinna zwracać żadnej wartości.
- Zaproponuj funkcję, która zaprezentuje trójkąt prostokątny przy pomocy znaku **#**. Możesz użyć w kodzie tylko jeden raz następujące funkcje: **println()** oraz **print("#")**. Funkcja powinna przyjmować 1 parametr - długość boku.
- Gra w **FizzBuzz**. Wypisz wszystkie liczby od 1 do 100, jeżeli liczba jest podzielna przez: 3 to wypisz **Fizz**, 5 to wypisz **Buzz**, 3 i 5 to wypisz FizzBuzz. W programie wykorzystaj pętlę **for** oraz **zakres**.
- Napisz funkcję rekurencyjną, która przyjmuje jeden parametr n. Funkcja powinna sumować liczby całkowite z przedziału **<0, n>** (np. <0, 5>, <0, -5>).

Czy wiesz, że...



Termin **bug** w kontekście informatyki został po raz pierwszy użyty przez Grace Hopper, jedną z pionierek programowania komputerów? W roku 1947, podczas pracy nad komputerem Mark II w laboratoriach Harvard University, Grace Hopper napotkała problem w pracy komputera. Okazało się, że przyczyną problemu był owad – dosłownie. Wewnątrz komputera Mark II znaleziono martwą ćmę, która utknęła między przekaźnikami, powodując awarię systemu.