	Wydział Politechniczny **Katedra Informatyki**
Prowadzący	Tomasz Gądek
Kurs	Narzędzia i środowiska programistyczne
Rok / Semestr	1 / Letni
Temat	Lab02: Podstawy programowania w języku Kotlin cz. 2.

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

Operatory porównania

Operatory porównania służą do porównania (jak sama nazwa wskazuje) dwóch wartości lub zmiennych. W zależności od postawionego warunku zwracają wartość logiczną - **true** lub **false**.

Spójrzmy, jakie mamy dostępne operatory porównania?

- Operator równy: **==**,
- Operator różny: **!=**,
- Operator większy: **>**,
- Operator mniejszy: **<**,
- Operator większy lub równy: **>=**,
- Operator mniejszy lub równy: **<=**.

Zwróć uwagę na pierwszy operator, to nie pomyłka. W języku Kotlin operator równości to **==**, ponieważ znak **=** jest zarezerwowany dla zmiennych (operacje przypisania).

Spójrz na poniższy kod, który prezentuje w działaniu kilku operatorów:

```
fun main() {

   var a = 2;
   var b = 4;
   var result1: Boolean = 2 >= 3
   var result2 = a != b

   println(result1)
   println(result2)
}
```

Rezultat działania programu jest następujący:

```
false
true
```

Operatory logiczne

Operatory logiczne w języku Kotlin służą do porównywania warunków logicznych i zwracania wartości typu **Boolean** (**true** lub **false**) na podstawie wyniku porównania. Mamy do dyspozycji 3 operatory logiczne: AND, OR i NOT.

Jak działa operator **AND** (w kodzie operator jest reprezentowany przez znaki **&&**)?

- Zwraca true, jeżeli oba warunki są prawdziwe (np: 2 > 1 && 2 == 2),

- Zwraca false, jeżeli przynajmniej jeden z warunków jest fałszywy (np: 1 > 2 && 2 == 2).

Jak działa operator **OR** (w kodzie operator jest reprezentowany przez znaki **||**)?

- Zwraca true, jeżeli przynajmniej jeden z warunków jest prawdziwy (np: 2 > 1 || 0 > 5),
- Zwraca false, jeśli oba warunki są fałszywe (np: 1 != 1 || 0 > 5).

Jak działa operator **NOT** (w kodzie operator jest reprezentowany przez znak **!**)?

- Zwraca **true**, jeżeli warunek jest fałszywy (np: !(1 != 1)),
- Zwraca **false**, jeśli warunek jest prawdziwy (np: !(1 != 2)).

Poniższy kod prezentuje użycie operatorów logicznych w praktyce.

```
fun main() {
   var a = 1
   var b = 2
   var c = 3

   var result1 = (a > b && a == 1) // false
   var result2 = (c > a || c == 3) // true
   var result3 = (!(a == b)) // true

   println("${result1}, ${result2}, ${result3}")
}
```

Rezultat:

```
false, true, true
```

Zatrzymajmy się na chwilę i przeanalizujmy kluczowe linijki zaprezentowanego kodu. Zweryfikujmy dlaczego zmienna **result1** posiada wartość **false**. Wyrażenie (a > b && a == 1) ewoluuje do postaci (1 > 2 && 1 == 1), a następnie do (false && true). W końcowym rozrachunku otrzymamy **false**.

Pozostałe przypadki postaraj przeanalizować we własnym zakresie.

Instrukcje warunkowe

Instrukcja warunkowa **if(){}** to blok kodu, który zawiera warunek. Spełnienie warunku (**true**) spowoduje wykonanie bloku kodu. Jeżeli warunek będzie fałszywy to blok kodu się nie wykona. Spójrz na poniższy fragment.

```
fun main() {
   var a = 10

   if(a > 5 && a > 0) {
       println("${a} greater 5 and ${a} is positive.")
   }

   println("End..")
}
```

Rezultat:

```
10 greater 5 and 10 is positive.
End..
```

Warunek jest prawdziwy, więc wykona się ciało instrukcji warunkowej **if**. Zweryfikuj co takiego się wydarzy, gdy warunek nie zostanie spełniony. Zmodyfikuj kod i przypisz do zmiennej **a** wartość **5**.

Czasami będziemy chcieli obsłużyć sytuację, gdy warunek nie będzie spełniony - wykonać blok kodu, który odnosi się do przeciwnej sytuacji. Z pomocą przyjdzie nam blok **else**, który może działać w parze z blokiem **if**. Blok **else** zostaje wykonany, gdy warunek w instrukcji **if** nie będzie spełniony. Poniższy kod zawiera opisany przypadek.

```
fun main() {
   var a = -5

   if(a > 0) {
       println("${a} is positive")
   } else {
       println("${a} is negative or zero")
   }
   println("End..")
}
```

Rezultat działania kodu jest następujący:

```
-5 is negative or zero
End..
```

W języku Kotlin występują mechanizmy, które mogą uprościć kod. Możemy cały blok **if-else** przypisać do zmiennej. Spójrz, jak zgrabnie możemy to zrobić.

```
fun main() {
   var a = -5

   var result = if(a > 0) {
       "${a} is positive"
   } else {
       "${a} is negative or zero"
   }

   println(result)
}
```

W zależności od spełnionego warunku wartość występująca w bloku zostanie przypisana do zmiennej **result**. Rezultat działania programu jest następujący.

```
-5 is negative or zero
```

Elvis

Podczas pracy z kodem może się zdarzyć, że niektóre zmienną mogą posiadać wartość **null**. Zmienna, która może przyjmować **null** musimy ją specjalnie oznaczyć. Spójrz na poniższy kod.

```
fun main() {
   var number:Int? = null

   println(number)
}
```

Rezultat:

```
null
```

Spójrz. Zmienna, która może przyjmować wartość **null** została wyposażona w znak **?**. Jeżeli tego nie zakomunikujesz, wówczas program się nie skompiluje. Czasami taki **null** może być problematyczny. Spójrz w jaki sposób możemy zweryfikować i przypisać domyślną wartość do zmiennej number. Oto przykładowe rozwiązanie problemu.

```
fun main() {
   var number:Int? = null

   if(number == null) {
       number = 0
   }

   println(number)
}
```

Rezultat:

```
0
```

Wykorzystując wiedzę z poprzedniego etapu możemy skorzystać z instrukcji warunkowej. Weryfikujemy, czy zmienna posiada **null**. Jeżeli warunek będzie prawdziwy, wówczas zaktualizujemy jej wartość na liczbę całkowitą.

Ten rodzaj rozwiązania jest bardzo rozwlekły, więc z pomocą przyjdzie nam operator **Elvis**. Spójrz na analogiczny kod, ale z wykorzystaniem nowego operatora.

```
fun main() {
   var number:Int? = null
   number = number ?: 0

   println(number)
}
```

W powyższym kodzie, pomiędzy zmienną **number**, a **0** znajduje się operator **Elvis**, czyli **?:** Czy widzisz charakterystyczną czuprynę? Teraz zastanówmy się, jak działa nowy operator. W powyższym kodzie, jeżeli zmienna number posiada **null** wówczas zostaje przypisane **0** (wartość występująca po prawej stronie operatora Elvis), w przeciwnym wypadku wartość number pozostaje niezmieniona (wartość występująca po lewej stronie operatora **Elvis**).

when

Instrukcja **when** jest odpowiednikiem instrukcji **switch** występującej w innych języka programowania. Instrukcja pozwala nam na wybór z dostępnych opcji. Znacznie ułatwia nam pracę, ponieważ musielibyśmy stosować konstrukcję wielu instrukcji **if**. Spójrz na praktyczny przykład zastosowania **when** w języku Kotlin.

```
fun main() {
   var day = 3

   when (day) {
       1 -> println("Monday")
       2, 3 -> println("Tuesday or Wednesday")
       4 -> println("Thursday")
       5 -> println("Friday")
       else -> println("Weekend")
   }
}
```

Rezultat:

```
Tuesday or Wednesday
```

Zatrzymajmy się na chwilę i zastanówmy się, jak działa **when**. Instrukcja sprawdza wartość zmiennej **day** i próbuje ją dopasować do dostępnych opcji. W naszym przypadku wartość zmiennej pasuje do liczby 3. Po dopasowaniu zostaje wyprowadzony tekst na wyjście **Tuesday or Wednesday**. Jeżeli wśród opcji nie uzyskamy dopasowania to wykona się sekcja **else**.

Instrukcja **when** może być przypisana do zmiennej - podobna sytuacja miała miejsce w etapie związanym z instrukcjami warunkowymi.

```
fun main() {
    var numberName = "two"

    var result = when (numberName) {
        "one" -> 1
        "two" -> 2
        "three" -> 3
        else -> null
    }

    println(result)
}
```

Rezultat:

```
2
```

Ta konstrukcja wydaje się bardziej elastyczna, ale oczywiście wybór należy do Ciebie. Zwróć uwagę, że instrukcja **when** może przyjmować również ciąg tekstowy. To nie jest instrukcja jedynie dla typów całkowitych. Może obsługiwać wiele innych typów, np: znakowy, zmiennoprzecinkowy.

while

Wyobraź sobie, że twoje zadanie polega na wyprowadzeniu na wyjście następującego tekstu.

```
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

Prawdopodobnie twoje rozwiązanie będzie wyglądało następująco.

```
fun main() {
   println("Counter: 1")
   println("Counter: 2")
   println("Counter: 3")
   println("Counter: 4")
   println("Counter: 5")
}
```

Powtarzanie fragmentów kodu to nie jest dobra praktyka. W języku kotlin możemy użyć konstrukcji, która umożliwi nam zmniejszenie nakładów pracy w implementacje. Spójrz na praktyczne użycie pętli **while**.

```
fun main() {
   var counter = 1

   while(counter <= 5) {
       println("Counter: ${counter}")
       counter++; // counter = counter + 1
   }
}
```

Nasz kod możemy podzielić na 3 sekcje. 1 sekcja to przypisanie wartości początkowej do naszego licznika (**var counter = 1**). Następnie mamy naszą pętle programową **when**. W niej znajduje się warunek, który jest sprawdzany z każdym obiegiem pętli. Pętla wykonuje się dopóki warunek będzie prawdziwy. Ciało pętli jest ograniczone znakami **{** i **}**. W ciele pętli, oprócz wyprowadzenia wartości licznika na wyjście znajduje się aktualizacja zmiennej licznikowej (**counter++**). Taki zapis to tzw. inkrementacja, czyli zwiększenie wartości zmiennej o 1. Jest to operacja równoważna operacji **counter = counter + 1**. Jeżeli aktualizacja zmiennej nie wystąpi, wówczas możemy doprowadzić do sytuacji, że pętla będzie się wykonywała w nieskończoność, ponieważ warunek nigdy nie stanie się fałszywy. Pętla wykonuje się do momentu, gdy warunek stanie się fałszywy, pętla zostanie przerwana i zostaną wykonane dalsze instrukcje.

Z pętlami programowymi związane są instrukcje **break** i **continue**.

- **break** - natychmiast przerywa działanie pętli programowej,
- **continue** - przerywa bierzącą iteracje pętli programowej, działanie pętli jest kontynuowane.

Zastosowanie **break**:

```
fun main() {
   var counter = 0

   while(counter <= 5) {
       counter++

       if(counter == 3) {
           break
       }

       println("Counter: ${counter}")
   }

   println("End.")
}
```

Rezultat:

```
Counter: 1
Counter: 2
End.
```

Zastosowanie **continue** zweryfikuj we własnym zakresie. W miejsce **break** wstaw słowo kluczowe **continue** i zaobserwuj wyniki.

Odczyt danych z klawiatury

W języku Kotlin odczyt danych odbywa się przy pomocy funkcji **readln()**. Wystarczy wywołanie funkcji przypisać do zmiennej.

```
fun main() {

   print("Your name: ")
   val name = readln()

   var size = name.length

   println("My name is ${name}! ${name} contains ${size}
characters")
}
```

Rezultat:

```
Your name: Tom
My name is Tom! Tom contains 3 characters
```

Funkcja **readln()** zwraca ciąg tekstowy. Jak zaimplementować program, który wczyta dwie liczby całkowite oraz wyznaczy ich sumę. Nie możemy operować na ciągach tekstowych ponieważ wynik operacji będzie nieprawidłowy. Można wykonać rzutowanie do konkretnego typu. Oto przykład.

```
fun main() {
   print("Digit a = ")
   val aAsString = readln()
   print("Digit b = ")
   val bAsString = readln()

   val aAsInt = aAsString.toInt()
   val bAsInt = bAsString.toInt()

   val sum = aAsInt + bAsInt

   println("${aAsInt} + ${bAsInt} = ${sum}")
}
```

Rezultat:

```
Digit a = 1
Digit b = 2
1 + 2 = 3
```

Program nie jest doskonały. Zakładamy, że użytkownik poda prawidłowe dane. W przypadku podania innych danych niż liczby program zgłosi problemy. O sytuacjach wyjątkowych opowiemy sobie na kolejnych laboratoriach.

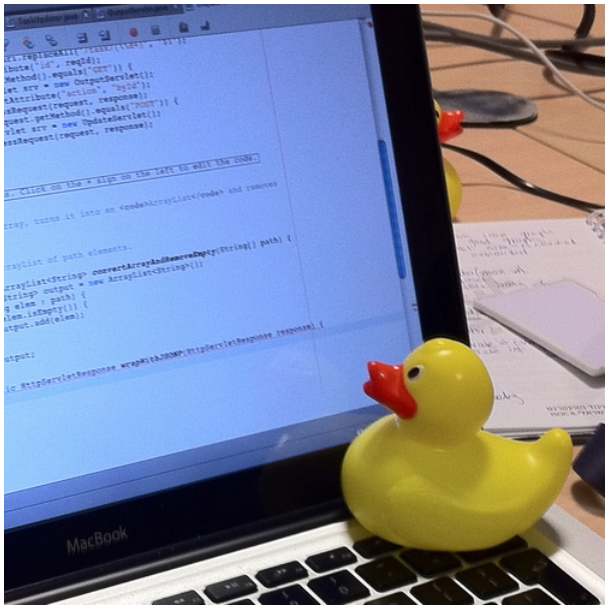
Zadania

- Napisz program, który obliczy znaki (pomijamy białe znaki: tabulatory, spacje) wprowadzone przez użytkownika z klawiatury.
- Napisz program, który wyznaczy potęgę (wykładnik oraz podstawa powinny być podawane z klawiatury). Nie używaj gotowych bibliotek - użyj pętli. Zakładamy, że wykładnik musi być większy lub równy 0.

- Do zmiennej **message** przypisz dowolny ciąg tekstowy. Następnie wyświetl, co drugi znak z ciągu tekstowego.
- Zaimplementuj program, który narysuje kwadrat bez wypełnienia. Bok kwadratu powinien zostać odczytany z klawiatury. W kodzie można użyć tylko jeden raz następujących funkcji **print("#")**, **print(" ")** oraz **println()**.

```
#####
#   #
#   #
#   #
#####
```

Czy wiesz, że...



Debugowanie na gumową kaczuszkę jest techniką stosowaną podczas debugowania oprogramowania. Polega ona na wyjaśnieniu problemu lub zrozumieniu kodu poprzez wyjaśnienie go komuś innemu, nawet jeśli ta osoba nie jest specjalistą z dziedziny.

W skrócie, wyobraź sobie, że masz gumową kaczkę na biurku. Kiedy napotykasz problem w swoim kodzie, opowiadasz o tym problemie gumowej kaczce. Podczas próby wyjaśnienia problemu kaczce, często zaczynasz rozumieć go lepiej.

Gumowa kaczuszka nie odpowie na twoje pytania, ale samo wyjaśnianie problemu może pomóc w zidentyfikowaniu przyczyny problemu lub znalezieniu sposobu na jego rozwiązanie. Ta technika pomaga programistom zyskać nową perspektywę i lepiej zrozumieć kod, który piszą.