ROZDZIAŁ 3

KOTLIN

3 Kotlin

3.1 Hello World/Operacje wejścia i wyjścia

3.1.1 Wyjście

Operacje wyjścia są fundamentalną funkcjonalnością każdego języka programowania, bez nich nie możliwe by było komunikowanie się komputera z użytkownikiem. Dzięki nim możemy wyświetlać kluczowe informacje lub na przykład zapisać dane do pliku.

Najprostszą i najbardziej podstawową jest **print(),** który wypisuje umieszczone w nim informacje do konsoli.

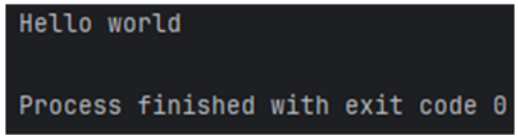
```
print()
```

Listing 3.1: Funkcja wyjścia konsoli

W środku nawiasów tej funkcji umieszczamy informacje, którą chcemy, aby program wyświetlił użytkownikowi. Np. Hello World.

```
print("Hello world")
```

Listing 3.2: Zastosowanie funkcji print()



Grafika 3.1: Wynik działania print() w konsoli

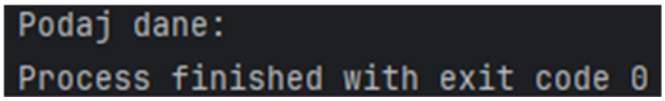
Po uruchomieniu aplikacji z taką linijką w kodzie program wyświetli w konsoli komunikat o treści zawartej w tej funkcji.

Istnieje jeszcze jeden wariant tej funkcji, która na końcu każdego komunikatu dodaje znak nowej linii(\n), który sprawi, że kolejny komunikat nie zostanie umieszczony jeden obok drugiego tylko pod sobą.

Jak na grafikach poniżej.

```
print("Podaj ")
print("dane: ")
```

Listing 3.3: Działanie print()

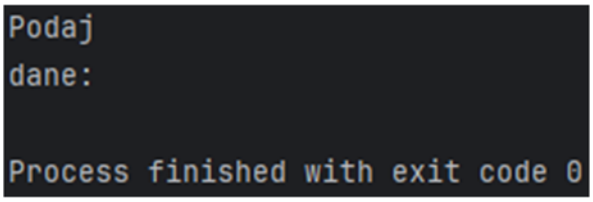


Grafika 3.2: Wynik działania programu

Gdy zamienimy **print()** na **println()** to komunikat w konsoli będzie wyglądał inaczej:

```
println("Podaj ")
println("dane: ")
```

Listing 3.4: Wariant funkcji wyjścia



Grafika 3.3: Wynik w konsoli

Komunikat w konsoli będzie inny, ponieważ pomiędzy słowami "Podaj" i "dane" zostanie wstawiona nowa linia. Teraz możemy wprowadzić jeszcze jedną istotną operację, czyli zapis do pliku. Używamy funkcji **File()**, w której podajemy nazwę pliku, w którym chcemy umieścić nasze dane. Funkcja ta sprawdzi czy plik o podanej nazwie i rozszerzeniu istnieje i go użyje. Jeżeli plik takowy nie istnieje to funkcja **File()** automatycznie go utworzy.

```
File("wyjscie.txt").writeText("Informacje")
```

Listing 3.5: Zapis wyjścia do pliku txt

Ta linijka stworzy w katalogu projektu plik txt o nazwie **wyjscie** i zapisze w nim wartość zawartą w metodzie **writeText**(). Po zakończeniu działania programu możemy znaleźć ten plik w panelu z lewej strony i po dwukrotnym naciśnięciu go otworzyć.

Warto wiedzieć, że po uruchomieniu programu jeszcze raz i wpisaniu nowych danych, nie zostaną one dodane do pliku tylko zastąpią poprzednie. Aby to zmienić na działanie, w którym dodajemy nowe informacje do pliku musimy zmienić metodę z **writeText()** na **appendText()**.

```
File("wyjscie.txt").appendText("Nowe dane")
```

Listing 3.6: Kod dodawania wpisanych danych do pliku

Teraz każde kolejne uruchomienie programu i wpisanie nowych danych nie usunie poprzednich tylko je doda do już istniejących w pliku. Jeżeli chcemy jeszcze, aby dane były dodawane do pliku w nowej linii musimy do podanych przez nas danych dopisać znak nowego wiersza, czyli \n.

```
File("wyjscie.txt").appendText("\n Nowe dane")
```

Listing 3.7: Dodanie znaku nowej linii przy każdym podaniu nowych danych

Od teraz w pliku wyjscie.txt każde dopisane dane są w nowej linii dzięki czemu plik jest czytelny.

3.1.2 Wejście

Operacje wejścia są totalnym przeciwieństwem do wyjścia, dzięki nim program pozyskuje dane od użytkownika lub z pliku zawierającymi dane. Dzięki nim możemy tworzyć programy takie jak np. kalkulator, który przyjmuje dane na wejściu następnie je przetwarza i wyświetla w wyjściu programu.

Podstawową funkcją realizującą te założenia jest **readLine()** lub **readln()**, który przyjmuje dane wpisane do konsoli z klawiatury.

```
readLine()
readln()
```

Listing 3.8: Funkcja wejścia konsoli

Bardziej zaawansowanym odpowiednikiem jest **readlnOrNull()**, który dodatkowo zabezpiecza przed niewpisaniem żadnej wartości, w takim przypadku wstawia tam wartość null, dzięki czemu, gdy chemy na takiej zmiennej przeprowadzić niektóre operacje program się zbuntuje i nie wykona operacji na takiej zmiennej.

```
readlnOrNull()
```

Listing 3.9: Bardziej zaawansowana funkcja wejścia

Mamy jeszcze wczytywanie danych z pliku, lecz do ich zastosowania konieczna jest znajomość pętli wiec o tym opowiemy sobie w dalszej części książki. Operacje wejścia mają również swoje inne warianty, które konkretyzują zastosowanie danego wejścia przykładem takim jest metoda do funkcji **readLine(),** która pozwala określić przyjmowane dane oraz dodatkowo tą funkcję zabezpieczyć. Jedną z takich metod jest **.toIntOrNull()**, który przyjmuje tylko wartości liczb całkowitych lub w przypadku niepodania żadnych danych zapisuje w zmiennej wartość **Null**.

```
readLine()?.toIntOrNull()
```

Listing 3.10: Metody funkcji readLine()

Zastanawiające w tej linii kodu jest znak zapytania. Pełni on dosyć konkretną rolę, mianowicie co, jeżeli użytkownik nie poda liczby całkowitej a zamiast tego wpisze tekst, właśnie ten problem ma rozwiązać ten symbol. Gdy wpiszemy tekst do takiego wejścia wtedy wartość zostanie podmieniona na **Null**. Jak omawialiśmy język Kotlin na początku książki była tam informacja o tym, że język ten skupia się na bezpieczeństwie danych, tak więc kompilator nawet nie daje nam zbytniego wyboru, bo gdy usuniemy znak zapytania z powyższej linii, program się nie uruchomi i zgłosi błąd dotyczący braku zabezpieczeń przed przypisaniem nieprawidłowych wartości.

3.2 Zmienne, stałe oraz typy danych

3.2.1 Typy danych

Pora wprowadzić kolejną fundamentalną wiedzę o Kotlinie – typy danych.

Typy danych - to klasyfikacja określająca rodzaj informacji. Przykłady: **int** - to typ danych przechowujący liczby całkowite (56 lub -87), **float** - przechowuje liczby zmiennoprzecinkowe, czyli np. 1.45 lub -1.98. Mamy również typ **bool**, który przechowuje wartości logiczne **true** lub **false**.

Typ danych	Przechowywane wartości	Zakres przechowywanych wartości
Byte	8-bitowa liczba całkowita	-128 do 127
Short	16-bitowa liczba całkowita	-32 768 do 32 767
Int	32-bitowa liczba całkowita	-2 147 483 648 do 2 147 483 647
Long	64-bitowa liczba całkowita	-9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
Float	32-bitowa liczba zmiennoprzecinkowa	1.4E-45 do 3.4028235E38
Double	64-bitowa liczba zmiennoprzecinkowa	4.9E-324 do 1.7976931348623157E308
Boolean	Typ logiczny	true lub false
Char	Pojedynczy znak Unicode	np. ‘h’
String	łańcuch znaków	Nieograniczona długość

Tabela 3.1 Podstawowe typów danych

3.2.2 Zmienne oraz stałe

Do przechowywania i zarządzania wartościami wykorzystujemy zmienne oraz stałe.

Zmienna - Zmienna to konstrukcja programistyczna, która posiada trzy podstawowe atrybuty:

- **Nazwę**: Jest to identyfikator, który pozwala nam odwoływać się do danej zmiennej w kodzie źródłowym.
- **Miejsce przechowywania:** Zmienna jest przechowywana w pamięci komputera pod określonym adresem.
- **Wartość**: To zawartość miejsca przechowywania, czyli dane, które zmienna reprezentuje.

W kotlinie możemy stworzyć zmienną na dwa sposoby pierwszy to słowo kluczowe **var** od słowa z języka angielskiego **variable** drugie to **val** od **value**. Istnieje pomiędzy nimi istotna różnica:

- **var** - Możemy zmieniać wartość w trakcie działania programu. Możemy dowolnie zmieniać przechowywane dane.
- **val -** Możemy nadać wartość tylko raz, jest to zmienna tylko do odczytu.
- **const val** - stała, podobnie jak w val nadajemy wartość tylko raz, lecz musi ona być nadana przed uruchomieniem programu.

3.2.2.1 Zmienne Var

Jeżeli chcemy stworzyć zmienną typu **int** musimy użyć słowa kluczowego **var**, następnie nadać nazwę, ja nadałem nazwę **number**. Kolejnym krokiem jest nadanie typu, Kotlin z automatu przydziela typ danych **string,** czyli gdy nie zdefiniujemy typu to nasza zmienna **number** będzie miała typ **string**. Jeżeli chcemy nadać typ danych z góry, należy użyć dwukropka i po nim wpisać nazwę typu danych, czyli dla typu **int** linijka kodu powinna być następująca:

```
var number: Int
```

Listing 3.11: Tworzenie zmiennej typu Int

Teraz przydało by się nadać tej zmiennej wartość do tego użyjemy operatora przypisania(=), powiedzmy, że chcemy by nasza zmienna przechowywała liczbę 4.

```
var number: Int = 4
```

Listing 3.12: Tworzenie zmiennej typu Int o wartości równej 4

Co, gdy chcemy, aby nasza zmienna miała wartość podaną przez użytkownika? Logika podpowiada nam, że powinniśmy skorzystać z funkcji **readLine()** lub **readlnOrNull()**, gdy spróbujemy takiego rozwiązania program nie uruchomi się dlatego, że zmienna nie jest zabezpieczona przed wpisaniem niepoprawnych wartości. Warto to wyjaśnić na przykładzie, aby łatwiej zrozumieć nam było zasady Kotlina.

```
var number: Int? = readLine()?.toInt()
var number2: Int = 3
print("number + number2 = "+number?.plus(number2))
```

Listing 3.13: Przykładowy program

```
3
number + number2 = 6
Process finished with exit code 0
```

Grafika 3.4: Wynik działania po podaniu liczby 3 w wejściu

W podanym przykładzie przy nadanym typie dla wartości number widzimy znak zapytania, pełni on tą samą rolę co przy funkcji readLine(), w readLine() zabezpiecza przed podaniem do wejścia niepoprawnych danych a przy : Int zabezpiecza przed przypisaniem niepoprawnych danych do zmiennej.

W funkcji print widzimy jak można sumować dane w wyjściu przy użyciu symbolu plusa oraz mamy dedykowaną funkcję dodawania do siebie liczb. Teraz warto przedstawić program zapisany niepoprawnie w stosunku do funkcji jaką chcemy aby spełniał czyli dodawał do podanej przez użytkownika liczby wartość 3 oraz wypisywał ją po dodaniu.

Jeżeli zapisalibyśmy program w ten sposób:

```
var number = readLine()
var number2: Int = 3
print("number + number2 = "+number+number2)
```

Listing 3.14: Niepoprawnie zapisany kod

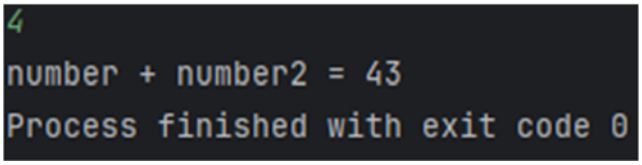
```
3
number + number2 = 33
Process finished with exit code 0
```

Grafika 3.5: Wynik działania dla liczby 3

Jak można zauważyć liczby nie są dodawane w sposób matematyczny tylko wyświetlane jako kolejne znaki. Jest tak dlatego, że typem danych w tym przypadku jest łańcuch znaków(string), gdy dodajemy do siebie łańcuchy znaków to otrzymujemy taki efekt końcowy, gdy znak + zamienimy na funkcję dodawania nic to nie zmieni:

```
print("number + number2 = "+number?.plus(number2))
```

Listing 3.15: Kod po zmianie



Grafika 3.6: Wynik działania po wprowadzonych zmianach

3.2.2.2 Zmienne Val

Zmienne Val cechują się tym, że możemy przypisać je tylko raz. Możemy to zrobić przed uruchomieniem lub w trakcie działania programu. Tak więc można użyć **val** do wczytania danych z klawiatury natomiast jeżeli, w trakcie dalszego działania programu konieczne będzie podanie nowej wartości do tej samej zmiennej, to przy wartości **val** nie będziemy mogli zrobić. Podsumowując **val** to zmienna niezmienna, można zmienić ją tylko raz i na tym koniec. Teraz jest odpowiedni moment na napisanie krótkiego kodu.

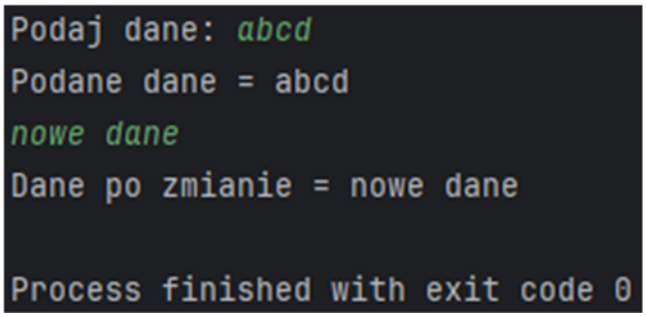
```
print("Podaj dane: ")
val dane = readln()
println("Podane dane = $dane")
dane = readln()
println("Dane po zmianie = "+dane)
```

Listing 3.17: Kod programu z użyciem val

Ten program się nie skompiluje zamiast tego kompilator zaproponuje nam zmianę z val na var, z powodu drugiego przypisania wartości do zmiennej dane. W programie tym także można zauważyć dwie metody na wypisanie danych w wyjściu. Symbol $ daje znać kompilatorowi że nazwę do niego przyklejoną ma czytać jako nazwę zmiennej. Gdy podmienimy val na var program zacznie działać.

```
print("Podaj dane: ")
var dane = readln()
println("Podane dane = $dane")
dane = readln()
println("Dane po zmianie = "+dane)
```

Listing 3.17: Kod pod podmianie na var



Grafika 3.7: Zachowanie programu

3.2.2.3 Stałe

Stałe są podobne do zmiennych, różnią się od siebie czasem inicjalizacji i jej zasadami. Stałym możemy przypisać wartość tylko raz przed uruchomieniem programu. A więc jakie jest ich zastosowanie? Możemy ich użyć np. w kalkulatorze jako liczba pi lub jako wartość do przeliczania stopni Celsjusza na Fahrenheita. Co bardzo ważne stałe możemy używać tylko na poziomie obiektów lub klasy.

Dlatego to zagadnienie rozwiniemy w działach o klasach i obiektach. Warto też wiedzieć że zmienną **val** możemy zainicjalizować przed uruchomieniem programu, co tak naprawdę oznacza, że jest to stała. Aby utworzyć stałą należy przed słowem kluczowym **val** dopisać **const**:

```
const val pi = 3.14159
```

Listing 3.18: Stała pi w klasach i obiektach

Konkluzja jest taka, że jeżeli potrzebujemy stałej np. pi to w klasach oraz obiektach używamy **const val** lub **va**l, a poza tymi konstrukcjami np. w funkcjach lub jako zmienna lokalna używamy **val**.

```
val pi = 3.14159
```

Listing 3.19: Stała pi poza klasami i obiektami

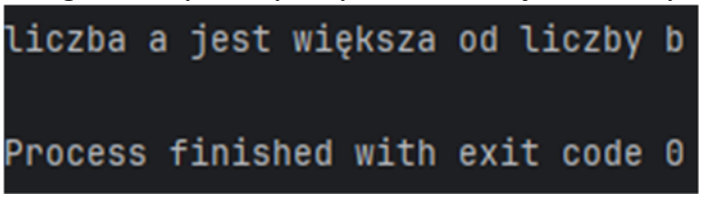
3.3 Instrukcje warunkowe

Instrukcja warunkowa – to blok kodu, który wykonuje jedną z dwóch lub więcej operacji, w zależności od spełnienia określonego warunku logicznego.

Używając instrukcji warunkowych możemy tworzyć programy, które zachowują się inaczej w zależności od podanych danych lub operacji użytkownika. Podstawową i najprostszą instrukcją warunkową w Kotlinie jest **if**. Spójrzmy na przykład wykorzystania.

```
var a = 3
var b = 1
if (a>b){
    println("liczba a jest większa od liczby b")
}
```

Listing 3.20: Przykład wykorzystania instrukcji warunkowych

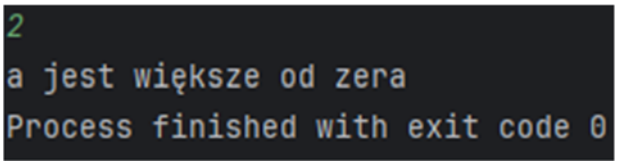


Grafika 3.8: Komunikat wypisany w konsoli

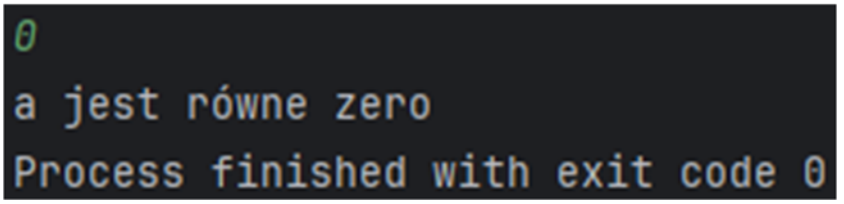
W tym programie sprawdzamy czy a jest większe od b i jeżeli jest to prawdą, wyświetla się komunikat. Możemy ten program rozbudować o więcej warunków niż tylko jeden.

```
var a = readlnOrNull()
var number = a?.toInt()
if (number != null) {
    if(number > 0)
        print("a jest większe od zera")
    else if (number == 0)
        print("a jest równe zero")
    else if (number > 5)
        print("a jest większe od 5")
    else
        print("a jest mniejsze od zera")
}
```

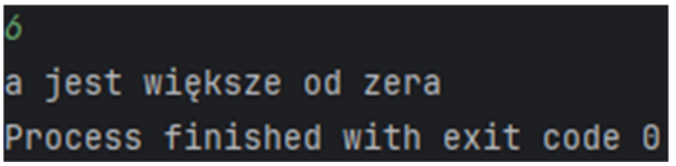
Listing 3.21: Program z wykorzystaniem instrukcji if, else if, else



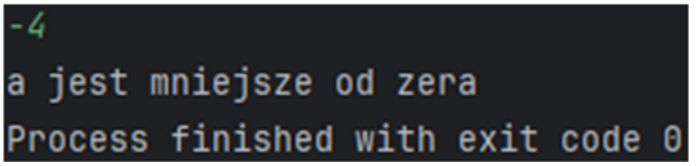
Grafika 3.9: działanie przy wpisanych danych = 2



Grafika 3.10: działanie przy wpisanych danych = 0



Grafika 3.11: działanie przy wpisanych danych = 6



Grafika 3.12: działanie przy wpisanych danych = -4

W zaktualizowanym programie dane podaje użytkownik co wiąże się z niebezpieczeństwem wpisania złych danych, tak więc dodatkowo zabezpieczamy się instrukcją warunkową **if**, która sprawdza czy podane dane nie są wartością **null**, następnie program sprawdza pozostałe warunki logiczne, jak można zauważyć każdy kolejny **if** jest zapisany jako **else if**. Jest tak dlatego że instrukcje są wtedy w jednym bloku, co zmienia zachowanie programu.

Blok instrukcji warunkowej zamykamy instrukcją **else**, która jest wywoływana w momencie gdy żaden wcześniejszy warunek nie zostanie spełniony. Instrukcje warunkowe są sprawdzane po kolei, jeżeli warunek zostanie spełniony cały blok jest pomijany oznacza to w praktyce, że w omawianym programiku nigdy nie zobaczymy komunikatu “a jest większe od 5”, ponieważ każda liczba większa od 5 jest również większa od zera co oznacza, że program wykona pierwszą