**Zestaw 1**

**Zad.1.**

W podanym kodzie stała *lastName* nie została zdefiniowana i zainicjalizowana - kompilator nie wie do czego odnieść się w przypadku implementacji stałej *fullName*.

**Zad.2.**

answer1 = true

answer2 = false

answer3 = true

answer4 = true

answer5 = false

**Zad.3**.

val Number: Double = 5.0  
val PowerOf2: Boolean = *log2*(Number) == *round*(*log2*(Number))  
*println*(PowerOf2)

**Zad.4.**

var odd2: Double = 0.0  
var odd3: Double = 0.0  
var odd4: Double = 0.0  
var odd5: Double = 0.0  
var odd6: Double = 0.0  
var odd7: Double = 0.0  
var odd8: Double = 0.0  
var odd9: Double = 0.0  
var odd10: Double = 0.0  
var odd11: Double = 0.0  
var odd12: Double = 0.0  
var pierwszaKostka: Int = 1  
while (pierwszaKostka < 7) {  
 var drugaKostka: Int = 1  
 while (drugaKostka < 7) {  
 when(pierwszaKostka+drugaKostka) {  
 2 -> odd2 += 1.0/36  
 3 -> odd3 += 1.0/36  
 4 -> odd4 += 1.0/36  
 5 -> odd5 += 1.0/36  
 6 -> odd6 += 1.0/36  
 7 -> odd7 += 1.0/36  
 8 -> odd8 += 1.0/36  
 9 -> odd9 += 1.0/36  
 10 -> odd10 += 1.0/36  
 11 -> odd11 += 1.0/36  
 12 -> odd12 += 1.0/36  
 }  
 drugaKostka += 1  
 }  
 pierwszaKostka += 1  
}

**Zestaw 2**

**Zad.1.**

Wartość parametru *sum* będzie równa 55, natomiast pętla for wykona się 11 razy.

**Zad.2.**

Ranges reprezentują przedziały liczb całkowitych i mogą być zamknięte lub otwarte z jednej strony.

Przedział zamknięty:

val closedRange = 0..10 //przedział zawiera liczby od 0 do 10

Przedział otwarty prawostronnie:

val halfOpenRange = 0 *until* 10 //przedział zawiera liczby od 0 do 9

Możliwe jest również tworzenie przedziałów, w których liczby maleją:

val downRange = 10 *downTo* 0 //przedział zawiera liczby od 10 do 0

Dodatkowo można określać krok zwiększania lub zmniejszania liczb w przedziale:

val downRange2 = 10 *downTo* 0 *step* 2 //przedział zawiera kolejne liczby: 10,8,6…

val closedRange2 = 0..10 *step* 2 //przedział zawiera kolejne liczby: 0,2,4…

**Zad.3.**

Przedział zamknięty nigdy nie może być pusty, ponieważ zawiera zarówno dolną, jak i górną granicę, zatem nawet w przypadku zapisu:

val halfOpenRange = 0..0

podany przedział będzie zawierał jeden element - 0.

**Zad.4.**

When wykorzystywane jest, kiedy ma być wykonywany różny kod w zależności od wartości danej zmiennej. W przypadku, gdy dla dwóch wartości zmiennej ma być wykonana ta sama instrukcja stosuje się przecinek.

when (x) {  
 0 -> *println*("x jest równy 0")  
 1,2 -> *println*("x jest równy 1 lub 2")  
 else -> *println*("coś innego")  
}

When może być dodatkowo użyte jako zastąpienie bloku ifów:

when {  
 bokA <= 0 && bokB <= 0 -> *println*("to NIE jest prostokąt")  
 bokA > 0 && bokB <= 0 -> *println*("to tylko bok a")  
 bokA <= 0 && bokB > 0 -> *println*("to tylko bok b")  
 else -> *print*("to jest prostokąt")  
}

Switch występujący w C/C++ może być wykorzystywany jedynie w przypadku wykonywania różnych kodów w zależności od wartości zmiennej, nie posłuży nigdy do zastąpienia bardziej skomplikowanej sekwencji ifów. When można stosować zarówno z argumentem (when (x) {…}), jak i bez (when {…}), natomiast stwitcha tylko z argumentem. Dodatkowo konstrukcja when jest znacznie prostsza, bardziej intuicyjna i przejrzysta w stosunku do switcha.

**Zad.5.**

var countdown: Int = 10  
while (countdown >= 0) {  
 *println*(countdown)  
 countdown -= 1  
}

**Zestaw 3**

**Zad.1.**

Funkcja *doSomething* wykona się – zwróci *i*-razy wypisane słowo wprowadzone do funkcji jako argument *msg*. Jedyne co możnaby poprawić to to, żeby funkcja, po przyjęciu wartości *i* mniejszej od 0 zwracała informację o tym:

fun doSomething(msg: String, i: Int): Unit {  
 if (i > 0) {  
 var x = 0  
 while (x < i) {  
 *println*(msg)  
 x = x + 1  
 }  
 } else {  
 *println*("Nieprawidłowy argument")  
 }  
}

Funkcja *threeMultiplication* nie wykona się – argument przekazany do funkcji jest zawsze stałą (val) i nie można go modyfikować wewnątrz funkcji. Należy dodać dodatkową zmienną wewnątrz funkcji i to ją zwrócić:

fun threeMultiplication(x: Int): Int {  
 var temp = x \* 3  
 return temp  
}

Funkcja *maxValue* nie wykonuje się – typ parametru zwracanego nie został określony, zatem przyjmuje się, że jest on Unit (funkcja nie powinna nic zwracać returnem), chociaż w rzeczywistości funkcja zwraca Int. Aby poprawić tę funkcję wystarczy określić typ zwracanego parametru jako Int:

fun maxValue(args: Array<Int>): Int {  
 var max = args[0]  
 var x = 1  
 while (x < args.size) {  
 var item = args[x]  
 max = if (max >= item) max else item  
 x = x + 1  
 }  
 return max  
}

**Zad.2.**

fun printResult(userChoice: String, gameChoice: String) {  
 val result: String  
 //Figure out the result  
 if (userChoice == gameChoice) result = "Tie!"  
 else if ((userChoice == "Rock" && gameChoice == "Scissors") ||  
 (userChoice == "Paper" && gameChoice == "Rock") ||  
 (userChoice == "Scissors" && gameChoice == "Paper")) result = "You win!"  
 else result = "You lose!"  
 *println*("You chose $userChoice. I chose $gameChoice. $result")  
}

**Zestaw 4**

**Zad.1**.

Nullable Types – typ zmiennej, która może zawierać zarówno wartość, jak i null (brak wartości zmiennej).

Null Safety – ochrona przed niespodziewanym wystąpieniem NullPointerException w trakcie kompilacji. Występowanie Nullable i Non-Nullable Types pozwala stwierdzić, która ze zmiennych w ogóle może przechowywać null. W przypadku Nullable Types należy sprawdzać w kodzie, czy dana zmienna nie zawiera null – można to robić na kilka sposobów:

* Dodanie sprawdzenia czy zmienna jest typu null za pomocą ifa
* Za pomocą Safe Call Operator (?.) wraz z funkcją let() lub bez niej
* Za pomocą Elvis Operator (?:)
* Not null assertion Operator (!!)

**Zad.2.**

Elvis Operator (?:) umożliwia sprawdzenie, czy Nullable Type przechowuje null. Operator ten przyjmuje dwa parametry: Nullable Type i wartość domyślną – jeżeli Nullable Type jest null zwróci wartość domyślną, w przeciwnym wypadku zwróci wartość Nullable Type.

**Zad.3.**

Safe Call Operator (?.) umożliwia zastąpienie bloku if…else, sprawdzającego, czy dana zmienna zawiera null, jedną linijką kodu:

var nameLength = nullableName?.length

Jeżeli zmienna *nullableName* będzie zawierała null funkcja length się nie wykona.

Not null assertion Operator (!!) służy do konwertowania Nullable Type do Non-Nullable Type. W przypadku, gdy Nullable Type przechowuje null wywoła wyjątek NullPointerException, dlatego zazwyczaj nie jest używany.

**Zad.4.**

var name: String? = "Bożena" //prawidłowe - zmienna name może być zarówno String i null  
var age: Int = null //nieprawidłowe - zmienna age może być tylko Int

val distance: Float = 26.7 //nieprawidłowe - przekazywana wartość jest traktowana jako Double (po liczbie należy wstawić 'f' by był to Float)  
var middleName: String? = null //prawidłowe - middleName może być zarówno String, jak i null

**Zad.5.**

fun divideIfWhole(liczba: Int, dzielnik: Int): Int? {  
 var koniec: Boolean = false  
 var parametr: Int = liczba  
 var ileRazy: Int = 0  
 while (!koniec) {  
 if (parametr%dzielnik == 0) {  
 ileRazy += 1  
 parametr /= dzielnik  
 } else {  
 koniec = true  
 }  
 }  
 if (ileRazy==0) {  
 return null  
 } else {  
 return ileRazy  
 }  
}  
  
var answer: Int? = divideIfWhole(64,2)  
if (answer == null) {  
 *println*("Not divisible :[")  
} else {  
 *println*("Yep, it divides $answer times")  
}

**Zestaw 5**

**Zad.1.**

fun removeOne(item: Int, list: List<Int>): List<Int> {  
 val removeList: MutableList<Int> = list.*toMutableList*()  
 removeList.remove(item)  
 removeList.*toList*()  
 return removeList  
}

**Zad.2.**

fun remove(item: Int, list: List<Int>): List<Int> {  
 val removeList: MutableList<Int> = list.*toMutableList*()  
 while (removeList.contains(item)) {  
 removeList.remove(item)  
 }  
 removeList.*toList*()  
 return removeList  
}

**Zad.3.**

fun reverse(array: Array<Int>): Array<Int> {  
 val reverseArray: Array<Int> = Array(array.size,**{**0**}**)  
 for (i in array.*indices*) {  
 reverseArray[i] = array[array.size-1-i]  
 }  
 return reverseArray  
}

**Zad.4.**

fun minMax(numbers: Array<Int>): Pair<Int,Int>? {  
 if (numbers.*isEmpty*()) {  
 return null  
 } else {  
 var min: Int = numbers[0]  
 var max: Int = numbers[0]  
 numbers.*forEach* **{** arg **->** if (arg < min) {  
 min = arg  
 } else if (arg > max) {  
 max = arg  
 }  
 **}** return Pair(min, max)  
 }  
}

**Zestaw 6**

**Zad.1.**

Labda jest funkcją nieposiadającą nazwy (funkcją anonimową). Może być przypisywana do zmiennej lub stałej i przekazywana jak zwykła wartość (również jako argument funkcji). Lambda zwraca wartość ostatniego wyrażenia w swoim ciele, może również zwrócić typ Unit lub nie zwrócić żadnej wartości (używa się wtedy typu Nothing). Lamba nie pozwala na używanie nazw argumentów. Oprócz dostępu do zmiennych z swoim ciele, Lambda może uzyskać również do zmiennych zewnętrznych (przechwytywanie zmiennych).

**Zad.2.**

Słowo kluczowe *it* występuje wówczas, gdy w lambdzie występuje tylko jeden argument. Używamy go wówczas jako określenie tego jedynego argumentu.

**Zad.3.**

fun repeatTask(times: Int, task: () -> Unit) {  
 for (i in 0 *until* times) {  
 task()  
 }  
}  
  
val lambdaWypisz: () -> Unit = **{** *println*("Działa!")  
**}**repeatTask(5,lambdaWypisz)

**Zad.4.**

val appRatings = *mapOf*(  
 "Kalendarz Pro" *to arrayOf*(1,5,5,4,2,1,5,4),  
 "Kurier Messenger" *to arrayOf*(5,4,2,5,4,1,1,2),  
 "Myszojeleniopedia" *to arrayOf*(2,1,2,2,1,2,4,2)  
)  
  
var averageRatings = *mutableMapOf*<String,Double>()  
appRatings.*forEach* **{** (k, v) **->** val ile = (v.size).toDouble()  
 val sum = v.*reduce* **{** a, b **->** a + b  
 **}** val average: Double = sum/ile  
 averageRatings[k] = average  
**}**val moreThan3 = averageRatings.*filter* **{** (\_,v) **->** v > 3 **}**val moreThan3Names = moreThan3.keys  
*println*(moreThan3Names)