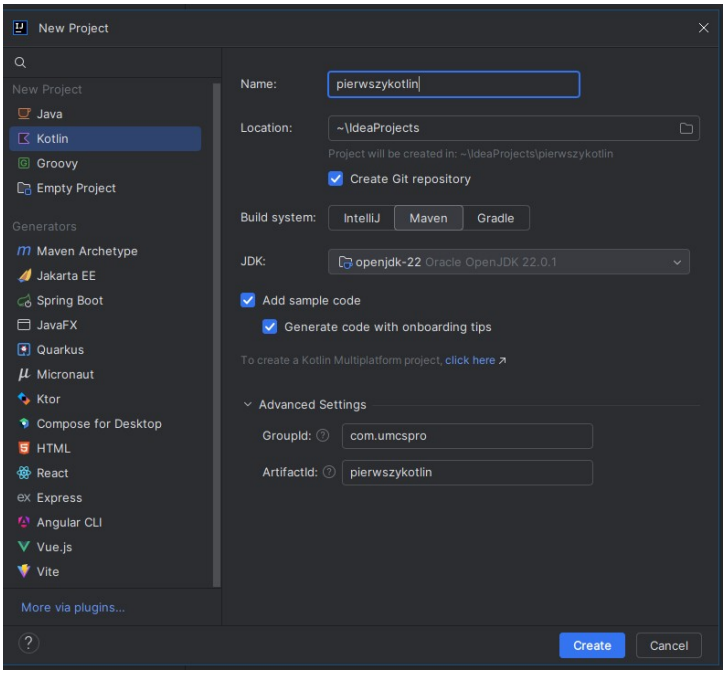
Krótki wstęp do języka Kotlin

Kotlin jest opracowanym przez JetBrains nowoczesnym językiem programowania. Jest wykorzystywany m.in. do tworzenia aplikacji na Androida, ale także do rozwoju aplikacji backendowych, desktopowych i webowych. Jest interoperacyjny z Javą.

https://kotlinlang.org/docs/getting-started.html

W IntelJ Idea projekt możemy utworzyć tak samo jak w przypadku projektu w języku java:



Pliki z kodem źródłowym w Kotlinie mają rozszerzenie .kt (od Kotlin).
W Kotlinie funkcja main może być napisana poza klasą.
W Kotlinie średniki są opcjonalne.

```
package com.umcspro

fun main() {
    val name = "Kotlin"
    println("Hello, $name!")
}
```

val oznacza stałą.
println wyświetla tekst na ekran ($name wstawia zmienną do tekstu).

```
var liczba = 5
var liczba2 : Double = liczba.toDouble() + 0.5
println("Hello, $name! Liczby: $liczba, $liczba2")
println("Typ liczby to: ${liczba2::class.simpleName}")
```

Jeżeli typ zmiennej wynika z wartości, nie trzeba jej deklarować.
Jeżeli nie, deklarujemy typ zmiennej po jej nazwie i po dwukropku:

```
var liczba2 : Double
```

Typy liczbowe: Int, Long, Double, Float, Short, Byte
Typ logiczny: Boolean
Typ znakowy: Char
Typ tekstowy: String

Pobranie danych od użytkownika:

```
var line : String = readLine() ?: ""
var line2 : String? = readlnOrNull()
line = line2.orEmpty()
```

Jeżeli null to readLine zwraca pusty String. ?: ""
Zmienna może być danego typu lub null: String?
line = line2.orEmpty() //zamienia na String

Tworzenie nowej instancji obiektu na przykładzie StringBuilder:

```
var sb: StringBuilder = StringBuilder()
```

Pętle, na przykładzie złączania stringów:

```
for (i in 1..10) {
    sb.append(i)
}
println(sb.toString())
sb.clear()
```

```
for (i in 1 until 10) {
    sb.append(i)
}
println(sb.toString())
sb.clear()
```

```
for (i in 10 downTo  1) {
    sb.append(i)
}
println(sb.toString())
sb.clear()
```

Pętle while, do-while:

```
var i = 1
while (i <= 5) {
    println("i = $i")
    i++
}

i = 6
do {
    println("i = $i")
    i++
} while (i <= 5)
```

break:

```
for (i in 1..10) {
    if (i == 5) {
        println("Przerywam pętlę,  i = $i")
        break
    }
    println("i = $i")
}
```

continue:

```
for (i in 1..10) {
    if (i == 5) {
        println("kontynuacja petli,  i = $i")
        continue
    }
    println("i = $i")
}
```

tablice:

```
val liczby =  arrayOf(1, 2, 3, 4, 5)
var suma = 0
for (liczba in liczby) {
    suma += liczba
}
println("Suma: $suma")

for (index in liczby.indices) {
    println("Element na indeksie $index to ${liczby[index]}")
}
```

```
liczby[0] = -14
suma = 0
for (liczba in liczby) {
    suma += liczba
}
println("Suma: $suma")
```

Listy:

```
    val listaNiemutowalna = listOf("y", "h", "a")
    val mutowalna = mutableListOf("a", "b", "c")

    mutowalna.add("d")
    mutowalna.remove("b")
    for (e in mutowalna) {
        println(e)
    }
    for (i in mutowalna.indices) {
        println("Element $i: ${mutowalna[i]}")
    }

    if ("b" in listaNiemutowalna) {
        println("Lista zawiera b")
    }

    val literyPosortowane = listaNiemutowalna.sorted()
    println(literyPosortowane)
}
```

Klasy, obiekty, wyjątki:

```
class Gamer(val name: String) {
    var points: Int = 0

    fun addPoint(){
        points++
    }
}
```

Klasa posiada konstruktor główny, który ustawia name.
Klasa posiada cechę z points ustawioną na zero.
Klasa posiada funkcję addPoint() zwieszająca ilość punktów o 1.

Przykład:
Klasa Game będzie przeprowadzać rozgrywkę, gracz ma odgarnąć liczbę od 0 do 9.
W pliku będzie przechowywany najlepszy wynik:

```
class Game(val gamer: Gamer) {
    var bestPlayer: String
    var bestScore: Int

    private val filename = "bs.txt"

    init {
        bestPlayer = readBestScore(filename)?.first?:"empty"
        bestScore =  readBestScore(filename)?.second?:0
    }
```

Konstruktor podstawowy ustawia Gracza.
Init wykonuje się po konstruktorze podstawowym. Wczytuje dane o najlepszym graczu z pliku.

Wczytanie danych najlepszego gracza:
Do wczytania pliku używamy obiektu klasy File, oraz metody readText.
Wyjątek przechwytujemy za pomocą bloku try-catch.

```
private fun readBestScore(nazwaPliku: String): Pair<String, Int>? {  //zwracamy parę lub null.
    try {
        val file = File(nazwaPliku)
        if (!file.exists()) {
            file.createNewFile()
            return null
        }
        val linia = file.readText().trim()
        if (linia.isNotEmpty()) {
            val data = linia.split(",")
            if (data.size >= 2) {
                val name = data[0]
                val score = data[1].toIntOrNull() ?: 0          //gdy nie wczyta inta zwraca null,  null zamieniony na zero

                return Pair(name, score)
            } else {
                println("not enough data")
                return null
            }
        } else {
            println("empty file")
            return null
        }
    } catch (e: IOException) {
        println("error: ${e.message}")
        return null
    }
}
```

Klasa Pair składa się z dwóch elementów, jest niemutowalna:

first – pierwszy element pary.
second – drugi element pary.

Zapis:

```
    private fun saveScore(nazwaPliku: String, name: String, score: Int) {
        try {
            val file = File(nazwaPliku)
            file.writeText("$name,$score")
        } catch (e: IOException) {
            println("Error: ${e.message}")
        }
}
```

Rozgrywka:

```
fun play(nRounds: Int){

    println("Hi ${gamer.name}, let's play! $nRounds")
    println("Best Player is $bestPlayer, with $bestScore")
    var number = 0;
    var guess: Int = 0;
    for (i in 1..nRounds){
        number = Random.nextInt(0,10);
        println("what number did I think of")
        guess = readlnOrNull()?.toIntOrNull() ?: 0
        if (guess == number){
            gamer.addPoint()
            println("Correct! I did think about $number")
        }else{
            println("Wrong! I thought about $number")
        }
    }
    println("Total score of ${gamer.name}: ${gamer.points}")
    if (bestScore<gamer.points){
         saveScore(filename,gamer.name,gamer.points)
         print("new record")
    }
}
```

Funkcja main:

```
println("Your name:")
val gname: String = readlnOrNull()?:""
val gamer = Gamer(gname)
println("How many rounds:")
val nrounds = readlnOrNull()?.toIntOrNull() ?: 1
val game = Game(gamer)
game.play(nrounds)
```

Dodatkowy konstruktor, w przykładzie, zmiana pliku z punktacją:

```
    private var bestPlayer: String = "empty"
    private var bestScore: Int = 0

    private var filename = "bs.txt"

//    init {
//        bestPlayer = readBestScore(filename)?.first?:"empty"
//        bestScore =  readBestScore(filename)?.second?:0
//    }


    constructor(gamer: Gamer, filename: String) : this(gamer) {
        this.filename = filename
        val bestData = readBestScore(filename)
        bestPlayer = bestData?.first ?: "empty"
        bestScore = bestData?.second ?: 0
    }
```

Porównywanie obiektów:
Porównujemy za pomocą operatora: ==, który wywołuje metodę equals.
Własna implementacja metody:

```
override fun equals(other: Any?): Boolean {
    if (this === other) return true
    if (other !is Gamer) return false
    return name == other.name && points == other.points
}
//hashCode:
override fun hashCode(): Int {
    return name.hashCode() * 31 + points
}
```

```
this === other //sprawdza czy porównujemy tą samą referencję
```

toString():

```
override fun toString(): String {
    return "Gamer(name=$name, points=$points)"
}
```

Dziedziczenie, interfejsy:
Jeżeli chcielibyśmy dziedziczyć po klasie Gamer, np. dopisując funkcję odejmującą punkty w Kotlinie musimy oznaczyć klasę Gamer jako open, ponieważ domyślnie klasy są finalne.

```
class MinusGamer(name: String, points: Int = 0) : Gamer(name, points)
{
    fun substractPoint() {
        points--;
    }
}
```

podobnie w klasie abstrakcyjnej, już bez open:

```
abstract class AGamer(val name: String, var points: Int = 0) {
    fun addPoint() {
        points++
    }
    fun subtractPoint() {
            points--
        }
```

```
    abstract fun showPoints()
}
```

```
class AbsImplGamer(name: String, points:Int=0) : AGamer(name,points){
    override fun showPoints() {
        println(points)
    }
}
```

Jeżeli chcielibyśmy, aby klasa zamiast dziedziczyć implementowała intrefejs Igamer np:

```
interface IGamer {
    fun addPoint()
    fun substractPoint()
    fun showPoints()
    fun getUserPoints(): Int
    fun getUserName(): String
}
```

```
class MinusGamer(val name: String, var points: Int = 0) :  IGamer

{
    override fun addPoint() {
        points++
    }

    override fun showPoints() {
        println("Player ${this.name} has ${this.points} points" )
    }

    override fun getUserPoints(): Int {
        return this.points
    }

    override fun getUserName(): String {
        return name
    }

    override fun substractPoint() {
        points--;
    }
}
```

Metody statyczne:
Odpowiednikiem metod statycznych w kotlinie mogą być:
funkcje najwyższego poziomu,
obiekt signleton, lub companion object.
Na przykładzie zapisu punktów do pliku:
w dowolnym miejscu:

```
fun saveScore(nazwaPliku: String, name: String, score: Int) {
    try {
        val file = File(nazwaPliku)
        file.writeText("$name,$score")
    } catch (e: IOException) {
        println("Error: ${e.message}")
    }
}
```

lub:

```
object ScoreManager {
    fun saveScore(nazwaPliku: String, name: String, score: Int) {
        try {
            val file = File(nazwaPliku)
            file.writeText("$name,$score")
        } catch (e: IOException) {
            println("Error: ${e.message}")
        }
    }
}
```

lub w klasie:

```
companion object {
    fun saveScore(nazwaPliku: String, name: String, score: Int) {
        try {
            val file = File(nazwaPliku)
            file.writeText("$name,$score")
        } catch (e: IOException) {
            println("Error: ${e.message}")
        }
    }
}
```

```
ScoreManager.saveScore("test","imie",10)
saveScore("test","imie",10)
Gamer.saveScore("test","imie",10)
```

Zadanie0:
Napisz grę papier, nożyce kamień.
Na początku rozgrywki podajemy login, określamy liczbę rund.
Gra wczytuje dane na temat punktów gracza z pliku i wyświetla mu je na konsoli.
Następnie przeprowadzamy grę z komputerem.
Za wygraną dostajemy +1 pkt. Za remis 0 i za przegraną -1.
Po każdej rozgrywce wyświetlamy wynik z wszystkich rozgrywek oraz aktualny wynik.
Po wszystkich rundach zapisujemy wynik do pliku oraz wyświetlamy tabelę z wszystkimi wynikami.