ФУНКЦІЇ. ФУНКЦІОНАЛЬНІ ІНТЕРФЕЙСИ. ЛЯМБДА-ВИРАЗИ.

KOTLIN. ФУНКЦІЇ

```
fun main() {
    println("2+17 = " + sum(2,17))
}
// створення підпрограми-функції
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

```
// функція без параметрів
// не повертає результату
fun greeting() {
    println("Hello, Kotlin!")
}
```

fun — ключове слово для визначення функції.

sum — ім'я функції.

(a: Int, b: Int) — параметри (з іменами та типами).

: Int — тип, що повертається (Int).

return a + b — повернення значення.

ПАРАМЕТРИ ЗА ЗАМОВЧУВАННЯМ

```
fun greetUser(name: String = "Guest")
{
    println("Hello, $name!")
}
```

ВКЛАДЕННЯ ФУНКЦІЙ

```
fun outerFunction() {
    fun innerFunction() {
        println("Це вкладена функція")
    }
    innerFunction()
}
```

ПЕРЕДАЧА ІМЕНОВАНИХ АРГУМЕНТІВ

```
fun main() {
    fullName(firstName = "Gorge", lastName = "Orwell")
}

fun fullName(firstName: String, lastName: String) {
    println("Full name: $firstName $lastName")
}
```

ОДНОРЯДКОВА ФУНКЦІЯ

```
fun main() {
    println("5 * 9 = " + multiply(5, 9))
}
// рядкова функція
fun multiply(a: Int, b: Int) = a * b
```

ЗМІННА КІЛЬКІСТЬ ПАРАМЕТРІВ

```
fun main() {
    printNumbers(1, 2, 3, 4, 5)
}
fun printNumbers(vararg numbers: Int) {
    for (num in numbers) {
        println(num)
    }
}
```

РОЗШИРЮВАЛЬНІ ФУНКЦІЇ (МЕТОДИ)

```
fun main() {
    "John".greet()
    // Hello, John!
}
fun String.greet() {
    println("Hello, $this!")
}
```

РЕКУРСИВНА ФУНКЦІЯ

```
fun factorial(n: Int): Int {
    return if (n == 1) 1 else n * factorial(n - 1)
}
```

KOTLIN. ФУНКЦІЇ ВИЩОГО ПОРЯДКУ

```
import kotlin.math.abs
fun main() {
    println(String.format("I = %10.5f",
      integrate(0.0, 2.0, ::cube, 100)))
    }
fun integrate(a: Double, b: Double,
    f: (Double) -> Double, N: Int ): Double {
    var s = 0.0; var x = a
    val h = abs(b - a) / N
    while (x < b) { s += f(x) * h; x += h }
    return s
}
fun cube(t:Double):Double = t*t*t
```

Аналог на Java

```
public class Integrator {
    public static void main(String[] args) {
        System.out.printf("I = %10.5f",
            integrate(0,Math.PI, Math::sin, 100));
    }
    private static double integrate(double a, double b,  func f, int N) {
        double s = 0;
        double h = Math.abs(b-a) / N;
        for (double x = a; x < b; x+=h) s += f.calc(x)*h;
        return s;
    }
}
interface func{double calc(double x);}
```

Функціональні інтерфейси

Особливе місце займають інтерфейси, що мають оголошений рівно один метод. Їх називають функціональними інтерфейсами і їх можна використовувати для передачі методу в якості параметра іншого методу, на кшталт вказівників на функцію у C++ чи функцій як об’єктів в Python.

INTERFACE **COMPARABLE**

- Interface **`Comparable`** allows ***custom sorting*** of objects when implemented.
  - When a class implements this interface, we must add the public method **`compareTo(Object o)`**.

```
public class Person implements Comparable<Person> {
    private String name;
    private int age;
    // constructors, getters, setters, toString

    @Override
    public int compareTo(Person p) {
        if (this.name.compareTo(p.name) != 0 )
            return this.name.compareTo(p.name);
        else
            return Integer.compare(this.age, p.age);
    }
}
```

INTERFACE `COMPARATOR`

Interface Comparator allows ***custom sorting*** of objects when implemented.

When a class implements this interface, we must add the public method `compare(Object o1, Object o2)`.

Methods compare can throw an exception *ClassCastException*, if the object types are not compatible in the comparison.

Вбудовані функціональні інтерфейси

Вбудовані функціональні інтерфейси Java:

Predicate<T> {boolean test(T t);}

Consumer<T>{void accept(T t);}

Function<T,R>{R apply(T t);}

Supplier<T> {T get();}

UnaryOperator<T> {T apply(T t);}

BinaryOperator<T> {T apply(T t1, T t2);}

Лямбда-вирази в Java

Анонімна реалізація функціональних інтерфейсів починаючи з Java 8 отримала новий синтаксис у формі популярних в інших мовах лямбда-виразів. Він значно простіший ніж у C++, бо відсутній оператор захоплення змінних оточення (всі вони захоплюються автоматично за принципом вбудованого класу), а тип результату виводиться з типу об'єкта, в який перетворюється лямбда-вираз.

Тобто лямбда складається з круглих дужок, знака лямбда-виразу «->» та блоку коду.

```
interface MathOperation {
    int operate(int a, int b);
}
public class LambdaExample {
    public static void main(String[] args) {
        MathOperation addition = (a, b) -> {
            int res = a + b;
            return res;
        };
        MathOperation multiplication = (a, b) -> a * b;
        System.out.println(addition.operate(5, 3));
        System.out.println(multiplication.operate(5, 3));
    }
}
```

Лямбда-вирази в Java

```
public class LambdaComparatorExample {

    public static void main(String[] args) {
      String [] names = {"Charlie", "Alice", "Bob"};

      Arrays.sort(names,
                        (a, b) -> a.length() - b.length() );

      System.out.println(Arrays.toString(names));
    }
}
```

Лямбда-вирази в Kotlin

Загальний синтаксис:

{ список параметрів -> тіло лямбда-виразу }

Наприклад:

```
val sum = { a: Int, b: Int -> a + b }
println(sum(3, 5))
```

Лямбду можна типізувати:

```
val multiply: (Int, Int) -> Int = { a, b -> a * b }
```

Тип параметрів можна опустити,

якщо компілятор може їх визначити.

Якщо лямбда має

один параметр,

його можна замінити на it:

```
val square: (Int) -> Int = { it * it }
println(square(4))
```

Лямбда-вирази в Kotlin

Лямбду можна передавати у функцію вищого порядку:

```
fun operate(a: Int, b: Int,
            operation: (Int, Int) -> Int): Int {
    return operation(a, b)
}
val result = operate(6, 2) { x, y -> x - y }
println(result)
```

Якщо останній аргумент функції — лямбда, її можна винести за дужки

Для виходу з лямбди використовується return@ :

```
val divide = { a: Int, b: Int ->
    if (b == 0) return@divide 0
    a / b
}
println(divide(10, 2))
```