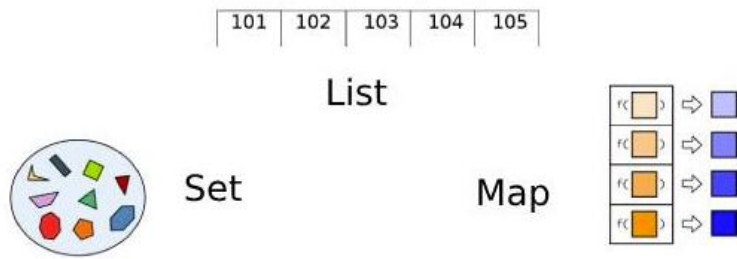


Desenvolvimento de Aplicativos

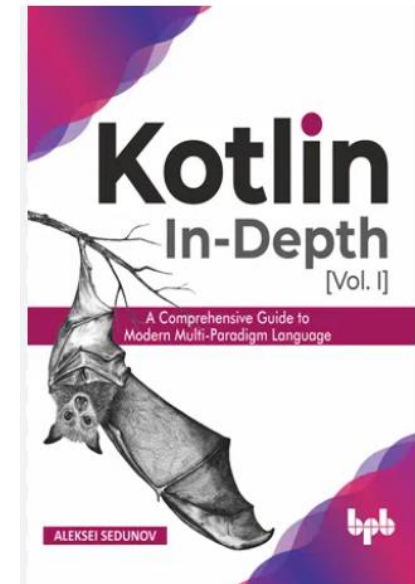
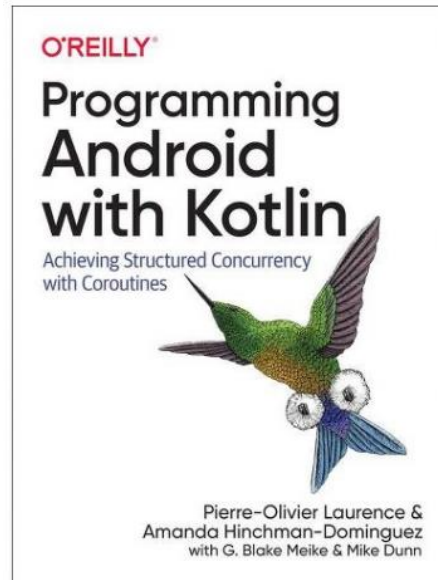
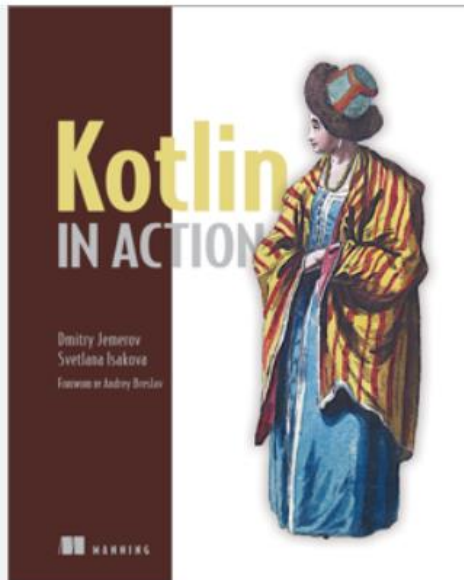
Dispositivos Móveis – Programação de Aplicações para Clientes Móveis

Unidade 4 – Funções e Coleções em Kotlin



Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@online.uscs.edu.br
aparecidovfreitas@gmail.com

Bibliografia



Funções

- ⊕ São parte **essencial** de qualquer linguagem de programação;
- ⊕ Por meio de funções pode-se definir um bloco de código que executa uma determinada tarefa;
- ⊕ Imagine que frequentemente necessita-se imprimir seu nome. Pode-se escrever uma função para realizar esta tarefa.

```
fun printNome() {  
    println("Aparecido Freitas");  
}  
  
fun main() {  
    printNome();  
}
```

Aparecido Freitas

Funções

```
fun printNome() {  
    println("Aparecido Freitas");  
}  
  
fun main() {  
    printNome();  
}
```

- ⊕ O código acima é conhecido como uma declaração de função;
- ⊕ Define-se uma função usando a palavra-chave **fun**;
- ⊕ Com a função definida, você pode usá-la da seguinte forma:

```
printNome();
```

Parâmetros de Funções

- ⊕ Permite que a função tenha comportamento diferente dependendo dos dados passados à ela por meio de parâmetros;

printMultiploDeTres.kt

```
1 fun printSoma10(valor: Int) {  
2     println("$valor + 10 = ${valor + 10}");  
3 }  
4  
5 fun main() {  
6     printSoma10(5);  
7     printSoma10(8);  
8 }
```

5 + 10 = 15

8 + 10 = 18

Named Parameters

- ⊕ Em **Kotlin**, os parâmetros nomeados, ou "**named parameters**", são uma funcionalidade que permite passar valores para os parâmetros de uma função utilizando explicitamente o nome do parâmetro ao invés de apenas sua posição na lista de parâmetros;
- ⊕ Isso torna o código mais **legível** e **claro**, pois pode-se ver imediatamente qual argumento corresponde a qual parâmetro, especialmente em chamadas de função com muitos argumentos ou com valores literais (como números ou strings) que podem ser confusos por si só.

Exemplo sem Named Parameters

Considerando uma função que configura um perfil de usuário:

```
fun configurarPerfil(nome: String, idade: Int, email: String) {  
    println("Nome: $nome, Idade: $idade, Email: $email")  
}
```

```
// Chamada da função sem named parameters  
configurarPerfil("João", 30, "joao@email.com")
```

Neste caso, você precisa lembrar a ordem dos parâmetros quando chama a função.



Exemplo com Named Parameters

Agora, vamos usar a mesma função, mas chamá-la usando named parameters:

kotlin

```
// Chamada da função com named parameters  
configurarPerfil(nome = "João", idade = 30, email = "joao@email.com")
```

Com named parameters, a ordem dos argumentos pode ser alterada, já que cada um é especificado pelo nome:

kotlin

```
// Chamada da função com named parameters em ordem diferente  
configurarPerfil(email = "joao@email.com", nome = "João", idade = 30)
```


Named Parameters – Vantagens

- ⊕ **Legibilidade:** É mais fácil entender o propósito de cada argumento, especialmente em chamadas de função com muitos parâmetros ou quando alguns parâmetros são opcionais;
- ⊕ **Flexibilidade** na Ordem dos Argumentos: Você pode alterar a ordem dos argumentos sem afetar a execução da função, o que é particularmente útil quando alguns parâmetros têm valores padrão definidos na função;
- ⊕ **Evita Erros:** Reduz a chance de passar argumentos na ordem errada, especialmente em funções com muitos parâmetros.

Exemplo com Named Parameters e Default

configurarPerfil.kt

```
1 fun configurarPerfil(nome: String, idade: Int = 18, email: String = "email@default.com") {  
2     println("Nome: $nome, Idade: $idade, Email: $email")  
3 }  
4  
5 fun main() {  
6     configurarPerfil(nome = "Maria")  
7 }
```

Nome: Maria, Idade: 18, Email: email@default.com

Argumentos default

- ⊕ Os parâmetros de função podem ter valores padrão em **Kotlin**, o que permite que o chamador (da função) omita alguns argumentos no local da chamada;
- ⊕ Pode-se adicionar um padrão à assinatura de uma função atribuindo um valor ao seu parâmetro.



Argumentos default

Kotlin_03_5.kt

```
1 fun conectDB (  
2     hostname: String = "localhost",  
3     username: String = "mysql",  
4     password: String = "secret") {  
5     println("hostname = $hostname")  
6     println("username = $username")  
7     println("password = $password")  
8 }  
9  
10 fun main() {  
11     conectDB("myComputer", "root");  
12 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_  
Kotlin_03_5.jar && java -jar Kotli  
hostname = myComputer  
username = root  
password = secret
```

Declarando Funções

- ⊕ Pode-se escrever funções em três locais:
 1. Dentro de uma classe. Estas são chamadas de **funções-membro**.
 2. Fora de uma classe. Estas são chamadas de **funções de nível superior**.
 3. Dentro de outras funções. Estas são chamadas de **funções locais**.

- ⊕ Independentemente de onde se coloca suas funções, a forma de declará-las não muda muito:

```
fun nomeDaFuncao(parametro1: Tipo1, parametro2: Tipo2, ...): TipoRetorno {  
    // Corpo da função  
    return valorDoTipoRetorno  
}
```

Declarando Funções

```
fun nomeDaFuncao(parametro1: Tipo1, parametro2: Tipo2, ...): TipoRetorno {  
    // Corpo da função  
    return valorDoTipoRetorno  
}
```

1. **Palavra-chave `fun`:** Toda declaração de função começa com a palavra-chave **`fun`**, que é abreviação de "function" (função).
2. **Nome da Função:** Segue o padrão de nomenclatura usual em programação, usando letras minúsculas e camelCase para nomes compostos.
3. **Parâmetros:** Entre parênteses, você lista os parâmetros da função, cada um seguido pelo seu tipo. Por exemplo, **`parametro1: Tipo1`**. Se a função não tiver parâmetros, os parênteses ficam vazios.

Declarando Funções

```
fun nomeDaFuncao(parametro1: Tipo1, parametro2: Tipo2, ...): TipoRetorno {  
    // Corpo da função  
    return valorDoTipoRetorno  
}
```

4. **Tipo de Retorno:** Após os parâmetros, você coloca dois pontos `:` e o tipo de dado que a função retorna. Se a função não retorna nenhum valor, você pode usar `Unit` ou omitir o tipo de retorno (o `Unit` é inferido automaticamente).
5. **Corpo da Função:** Entre chaves `{}`, você escreve o código que define o que a função faz.
6. **Instrução `return`:** Se a função tem um tipo de retorno diferente de `Unit`, você deve retornar um valor desse tipo com a instrução `return`.

Funções em Kotlin – Exemplo

Kotlin_03_1.kt

```
1 fun soma(a: Int, b: Int): Int {  
2     return a + b  
3 }  
4  
5 fun main() {  
6     var a = 10  
7     var b = 20  
8     println (soma(a,b))  
9 }  
10
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS  
Kotlin_03_1.jar && java -jar Kotlin_03_1.jar  
30
```


Funções em Kotlin – Exemplo

Kotlin_03_2.kt

```
1 fun exibeMensagem(msg: String, contador: Int) {  
2     var i = 1  
3     while(i++ <= contador ) {  
4         println(msg)  
5     }  
6 }  
7  
8 fun main() {  
9     var a = 5  
10    exibeMensagem("USCS",a)  
11 }
```

```
[Running] cd "d:\USCS\DISCIPL  
Kotlin_03_2.jar && java -jar  
USCS  
USCS  
USCS  
USCS  
USCS
```

Funções em Kotlin - Exemplo

Equivale a
void em Java

Kotlin_03_3.kt

```
1 fun exibeMensagem(msg: String, contador: Int) : Unit {
2     var i = 1
3     while( i <= contador ) {
4         println(msg)
5         i++
6     }
7 }
8
9 fun main() {
10     var a = 5
11     exibeMensagem("USCS",a)
12 }
13
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_Ap
Kotlin_03_3.jar && java -jar Kotlin_03_3.jar
USCS
USCS
USCS
USCS
USCS
```

Funções Single Expression

- ⊕ Vimos que funções em **Kotlin** são declaradas por:

```
fun nomeDaFuncao(parametro1: Tipo1, parametro2: Tipo2, ...): TipoRetorno {  
    // Corpo da função  
    return valorDoTipoRetorno  
}
```

- ⊕ Há uma outra forma de se declarar funções onde pode-se omitir tanto o statement **return**, quanto as **chaves** e o **tipo** de **retorno**;
- ⊕ Esta outra forma de se escrever funções em Kotlin é denominada: **Funções Single Expression**.

Funções Single Expression

❏ Kotlin_03_4.kt

```
1 fun somaInteiros(a: Int, b: Int) = a + b
2
3 fun main() {
4     var a = 5
5     var b = 3
6     println(somaInteiros(a,b))
7 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS"
Kotlin_03_4.jar && java -jar Kotlin_03_4.jar
8
```

Número variável de argumentos

- ⊕ Em **Kotlin**, o conceito de **número variável de argumentos** é implementado através do modificador **vararg**;
- ⊕ Ele permite que se passe um **número variável de argumentos** do mesmo tipo para uma função;
- ⊕ Quando uma função é declarada com um parâmetro **vararg**, pode-se passar qualquer número de argumentos para esse parâmetro (incluindo nenhum) quando a função for chamada.

Número variável de argumentos

```
fun soma(vararg numeros: Int): Int {  
    return numeros.sum()  
}  
  
fun main() {  
    println(soma(1, 2, 3))           // Saída: 6  
    println(soma(10, 20, 30, 40))   // Saída: 100  
    println(soma())                 // Saída: 0  
}
```

Neste exemplo, a função `soma` aceita um número variável de argumentos inteiros. Ela usa a função `sum()` para calcular a soma de todos os inteiros passados. Na função `main`, `soma` é chamada com diferentes números de argumentos para demonstrar sua flexibilidade.

Extension Functions

- ⊕ Em **Kotlin**, as "**extension functions**" (funções de extensão) permitem que se adicione novas funções à classes existentes sem ter que modificar suas definições;
- ⊕ Com as **funções de extensão**, pode-se estender uma classe com novas funcionalidades, como se estivesse adicionando métodos a essa classe;
- ⊕ Isso é particularmente útil para adicionar **funcionalidades** a **classes** sobre as quais não se tem controle, como as classes da biblioteca padrão ou de bibliotecas de terceiros.

Extension Functions

Kotlin_03_6.kt

```
1  // Extensão da classe String com uma função chamada "contarLetraA"
2  fun String.contarLetraA(): Int {
3      var contador = 0
4      for (letra in this) {
5          if (letra == 'A' || letra == 'a') {
6              contador++
7          }
8      }
9      return contador
10 }
11
12 fun main() {
13     val exemplo = "Kotlin - linguagem mAravilhosA"
14     println("Total de letras 'A' em \"$exemplo\": ${exemplo.contarLetraA()}")
15 }
```


```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_Aplicativos\Foo"
Kotlin_03_6.jar && java -jar Kotlin_03_6.jar
Total de letras 'A' em "Kotlin - linguagem mAravilhosA": 4
```


Estendendo a Classe String

1. Estendendo a Classe String

Suponha que você queira adicionar um método `inverte()` para a classe `String` que retorna a string invertida. Você pode fazer isso da seguinte maneira:

kotlin

 Copy code

```
fun String.inverte(): String {  
    return this.reversed()  
}  
  
val minhaString = "Kotlin"  
println(minhaString.inverte()) // ntiloK
```

Neste exemplo, a função `inverte()` é uma extension function da classe `String`. Ela usa o método `reversed()` da própria classe `String` para retornar a string invertida.

Desestruturação de Funções

- ⊕ A desestruturação (ou **destructuring**) em **Kotlin** é um recurso que permite "desmontar" um objeto em várias partes, de modo que você possa extrair os valores armazenados dentro dele e atribuí-los a variáveis separadas, de uma forma simples e direta.
- ⊕ Kotlin tem uma classe chamada **Pair**, que guarda dois valores.

pair.kt

```
1 fun MulDiv(n: Int, fator: Int): Pair<Int, Int> {  
2     val p = n * fator;  
3     val q = n / fator;  
4  
5     return Pair(p, q);  
6 }  
7 fun main() {  
8     val (x,y) = MulDiv(4, 2);  
9     println("Produto: $x, Quociente: $y")  
10 }
```

Produto: 8, Quociente: 2

Overloading de Funções

- ⊕ Overloading (ou sobrecarga) em **Kotlin** é o conceito que permite definir várias funções ou métodos com o mesmo nome, mas com diferentes listas de parâmetros;
- ⊕ Isso significa que você pode criar várias versões de uma função ou método, e cada versão pode aceitar um conjunto diferente de argumentos.

Overloading de Funções

overloading.kt

```
1 fun ola() {
2     println("Hello!");
3 }
4
5 fun ola(nome: String) {
6     println("Hello, $nome!");
7 }
8
9 fun ola(nome: String, idade: Int) {
10    println("Hello, $nome! Você tem $idade anos de idade!");
11 }
12
13 fun main() {
14     ola();
15     ola("Aparecido Freitas");
16     ola("Aparecido Freitas", 18);
17 }
```

```
Hello!
Hello, Aparecido Freitas!
Hello, Aparecido Freitas! Você tem 18 anos de idade!
```

Funções Recursivas

fibonacci.kt

```
1 fun fibonacci(n: Int): Int {
2     return if (n <= 1) {
3         n
4     } else {
5         fibonacci(n - 1) + fibonacci(n - 2)
6     }
7 }
8
9 fun main() {
10     print("Digite um número: ")
11     val n = readLine()?.toIntOrNull();
12     if (n == null || n < 0 ) {
13         println ("Entrada inválida!");
14         return;
15     }
16     println("O Fibonacci de $n é: ${fibonacci(n)} ");
17 }
```

Quiz 4_01

O que será exibido na console?

Exercicio_4_1.kt

```
1  //-----
2  fun msg(c: Char, contador: Int): Unit {
3      var i = 1;
4
5      while (i <= contador) {
6          print(c + " ");
7          i++;
8      }
9  }
10 //-----
11 fun main() {
12     val lista = charArrayOf('a', 'b', 'c', 'd', 'e')
13     for (i in lista.indices) {
14         if (i % 2 == 0)
15             msg(lista[i], i)
16     }
17 }
```

Resposta: _____

Listas

- ⊕ Uma lista é uma estrutura de dados que é conceitualmente muito semelhante a um array;
- ⊕ Uma lista tem implementações concretas em tipos como ArrayList, LinkedList e outros;
- ⊕ Arrays geralmente são mais eficientes do que listas em termos de desempenho bruto, mas as listas têm a característica adicional de serem de tamanho dinâmico;
- ⊕ Ou seja, arrays têm tamanho fixo, mas as listas podem ser configuradas para crescer e encolher conforme necessário, como você verá mais adiante ao aprender sobre listas mutáveis.

Criando listas

- ⊕ **Kotlin** tem a função **listOf()** na biblioteca padrão para criar listas;
- ⊕ Listas criadas com **listOf()** em **Kotlin** são imutáveis por padrão;
- ⊕ Isso significa que você não pode adicionar, remover ou modificar elementos dessa lista depois que ela foi criada;
- ⊕ Se você precisa de uma lista mutável, que permite alterações, deve-se usar **mutableListOf()** para criar a lista.

list_01.kt

```
1  //-----
2  fun main() {
3      val frutas = listOf("Maçã", "Banana", "Laranja");
4
5      println("Lista imutável original: $frutas");
6
7      val mutableFrutas = frutas.toMutableList();
8
9      mutableFrutas.add("Uva");
10     println("Lista mutável após adicionar 'Uva': $mutableFrutas");
11
12     mutableFrutas[0] = "Melancia";
13     println("Lista mutável após alterar a lista: $mutableFrutas");
14
15     mutableFrutas.remove("Banana");
16     println("Lista mutável após remover 'Banana': $mutableFrutas");
17 }
```

```
Lista imutável original: [Maçã, Banana, Laranja]
Lista mutável após adicionar 'Uva': [Maçã, Banana, Laranja, Uva]
Lista mutável após alterar a lista: [Melancia, Banana, Laranja, Uva]
Lista mutável após remover 'Banana': [Melancia, Laranja, Uva]
```

Propriedades das listas

- `size`: Retorna o número de elementos na lista.

```
val size = list.size
```

- `first`: Retorna o primeiro elemento da lista. Lança uma exceção se a lista estiver vazia.

```
val firstElement = list.first()
```

- `last`: Retorna o último elemento da lista. Lança uma exceção se a lista estiver vazia.

```
val lastElement = list.last()
```

- `isEmpty`: Retorna `true` se a lista estiver vazia, caso contrário, retorna `false`.

```
val isEmpty = list.isEmpty()
```

- `isNotEmpty`: Retorna `true` se a lista não estiver vazia, caso contrário, retorna `false`.

```
val isNotEmpty = list.isNotEmpty()
```

Funções de Acesso em listas

- `get(index)` ou `[]`: Retorna o elemento na posição especificada.

```
val element = list.get(2)
val element = list[2]
```

- `indexOf(element)`: Retorna o índice da primeira ocorrência de um elemento na lista ou -1 se o elemento não estiver presente.

```
val index = list.indexOf("Banana")
```

Funções de Acesso em listas

- `lastIndexOf(element)`: Retorna o índice da última ocorrência de um elemento na lista ou -1 se o elemento não estiver presente.

```
val lastIndex = list.lastIndexOf("Banana")
```

- `subList(fromIndex, toIndex)`: Retorna uma sublista da lista entre os índices especificados.

```
val subList = list.subList(1, 3)
```

Funções de Verificação em listas

- `contains(element)`: Retorna `true` se a lista contém o elemento especificado, caso contrário, retorna `false`.

```
val containsElement = list.contains("Banana")
```

- `containsAll(elements)`: Retorna `true` se a lista contém todos os elementos da coleção especificada.

```
val containsAllElements = list.containsAll(listOf("Banana", "Maçã"))
```

Funções de Transformação em listas

- `sorted()`: Retorna uma nova lista ordenada em ordem crescente.

```
val sortedList = list.sorted()
```

- `reversed()`: Retorna uma nova lista com os elementos na ordem inversa.

```
val reversedList = list.reversed()
```

Funções de Mutabilidade p/listas mutáveis

- `add(element)`: Adiciona um elemento à lista.

```
mutableList.add("Uva")
```

- `addAll(elements)`: Adiciona todos os elementos da coleção especificada à lista.

```
mutableList.addAll(listOf("Abacaxi", "Melancia"))
```

Funções de Mutabilidade p/listas mutáveis

- `remove(element)`: Remove a primeira ocorrência do elemento especificado da lista.

```
mutableList.remove("Banana")
```

- `removeAt(index)`: Remove o elemento na posição especificada.

```
mutableList.removeAt(2)
```

- `clear()`: Remove todos os elementos da lista.

```
mutableList.clear()
```


Quiz 4_02

O que será exibido na console?

list_02.kt

```
1 fun main() {  
2     val list1 = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9);  
3     val list2 = list1.toMutableList();  
4     val r = mutableListOf<Int>();  
5  
6     for (k in list2.indices) {  
7         if (list2.get(k) % 2 == 0) {  
8             r.add(list2.get(k));  
9         }  
10    }  
11    println(r);  
12 }
```

Resposta: _____

Quiz 4_03

O que será exibido na console?

list_03.kt

```
1 fun main() {
2     val list1 = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9);
3     val list2 = list1.toMutableList();
4     val r = mutableListOf<Int>();
5     if (!list2.isEmpty())
6         print("${list2.first() + list2.last()}" + " ");
7     for (k in list2.indices)
8         if (list2[k] % 2 == 0)
9             r.add(list2[k])
10    if (!r.isEmpty())
11        print("${list2.last() - list2.first()}" + " ");
12    val v = r.indexOf(6)
13    if (v != -1)
14        print("USCS")
15    else
16        print("CC")
17 }
```

Resposta: _____

Quiz 4_04

O que será exibido na console?

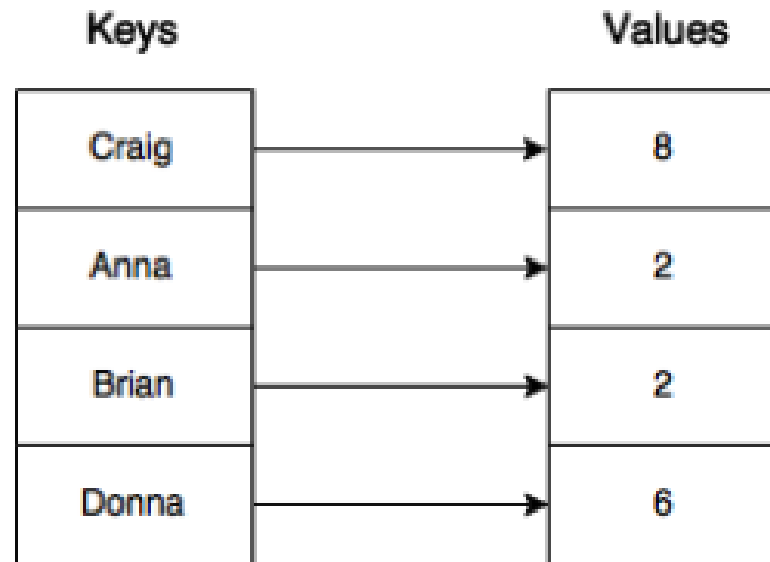
list_04.kt

```
1 fun main() {
2     val list1 = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9);
3     val list2 = list1.toMutableList();
4     val r = mutableListOf<Int>();
5     if (!list2.isEmpty())
6         print("${list2.indexOf(list2.first()) + list2.indexOf(list2.last())}" + " ");
7     for (k in list2.indices)
8         if (list2[k] % 2 == 1)
9             r.add(list2[k]);
10    if (!r.isEmpty())
11        print("${list2.indexOf(list2.last()) + list2.indexOf(list2.first())}" + " ");
12    val s1 = r.subList(1, r.size - 1);
13    print (s1);
14 }
```

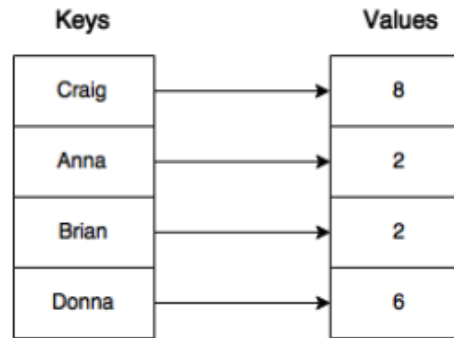
Resposta: _____

maps

- Um **map** é uma coleção não ordenada de pares, onde cada par é formado por uma chave e um valor;



maps



- ⊕ As chaves são **exclusivas**;
- ⊕ A mesma chave não pode aparecer duas vezes em um mapa, mas diferentes chaves podem apontar para o mesmo valor;
- ⊕ Todas as chaves devem ser do mesmo tipo, e todos os valores devem ser do mesmo tipo;
- ⊕ Pode-se pensar em um **map** como um dicionário onde se procura um valor (definição) usando-se uma chave (palavra).

Criação de maps

```
fun main() {  
    // Criando um Map imutável com a função mapOf  
    val capitais = mapOf(  
        "Brasil" to "Brasília",  
        "Estados Unidos" to "Washington D.C.",  
        "França" to "Paris"  
    )  
  
    // Exibindo o Map  
    println(capitais)  
}
```

- Usamos a função `mapOf` para criar um `Map` imutável.
- Cada entrada no `Map` é criada usando a sintaxe `chave to valor`.
- O `Map` resultante associa os nomes dos países às suas capitais.

Criação de maps

```
fun main() {  
    // Criando um Map imutável com a função mapOf  
    val capitais = mapOf(  
        "Brasil" to "Brasília",  
        "Estados Unidos" to "Washington D.C.",  
        "França" to "Paris"  
    )  
  
    // Exibindo o Map  
    println(capitais)  
}
```

```
{Brasil=Brasília, Estados Unidos=Washington D.C., França=Paris}
```

Criação de maps

```
fun main() {  
    // Criando um Map imutável com a função mapOf  
    val capitais = mapOf(  
        "Brasil" to "Brasília",  
        "Estados Unidos" to "Washington D.C.",  
        "França" to "Paris"  
    )  
  
    // Exibindo o Map  
    println(capitais)  
}
```

- `mapOf`: Esta função cria um `Map` imutável. A imutabilidade significa que, depois que o `Map` é criado, você não pode adicionar, remover ou alterar as chaves e valores no `Map`.
- Pares `chave to valor`: Cada entrada no `Map` é definida usando a sintaxe `chave to valor`. Neste exemplo, a chave é o nome do país, e o valor é a capital correspondente.

maps imutáveis

```
fun main() {  
    val capitais = mapOf(  
        "Brasil" to "Brasília",  
        "Estados Unidos" to "Washington D.C.",  
        "França" to "Paris"  
    )  
  
    // Tentativa de modificar o Map (Isso causará um erro)  
    // capitais["Brasil"] = "Rio de Janeiro" // Erro de compilação  
    // capitais.put("Itália", "Roma") // Erro de compilação  
  
    println(capitais)  
}
```

- ⊕ Ao se tentar modificar um map imutável, o **Kotlin** gerará um **erro de compilação**.

Criação de maps mutáveis

- ✦ Para criar maps mutáveis em Kotlin, utiliza-se a função **mutableMapOf**;
- ✦ Um map mutável permite que se adicione, remova e modifique as entradas (pares de chave-valor) após a criação.

maps mutáveis

```
fun main() {  
    val capitais = mutableMapOf(  
        "Brasil" to "Brasília",  
        "Estados Unidos" to "Washington D.C."  
    )  
    println("Map inicial: $capitais")  
  
    capitais["França"] = "Paris"  
    println("Após adicionar França: $capitais")  
  
    capitais["Brasil"] = "Rio de Janeiro"  
    println("Após modificar Brasil: $capitais")  
  
    capitais.remove("Estados Unidos")  
    println("Após remover Estados Unidos: $capitais")  
}
```

maps mutáveis

```
fun main() {  
    val capitais = mutableMapOf(  
        "Brasil" to "Brasília",  
        "Estados Unidos" to "Washington D.C."  
    )  
    println("Map inicial: $capitais")  
  
    capitais["França"] = "Paris"  
    println("Após adicionar França: $capitais")  
  
    capitais["Brasil"] = "Rio de Janeiro"  
    println("Após modificar Brasil: $capitais")  
  
    capitais.remove("Estados Unidos")  
    println("Após remover Estados Unidos: $capitais")  
}
```

```
Map inicial: {Brasil=Brasília, Estados Unidos=Washington D.C.}  
Após adicionar França: {Brasil=Brasília, Estados Unidos=Washington D.C., França=Paris}  
Após modificar Brasil: {Brasil=Rio de Janeiro, Estados Unidos=Washington D.C., França=Paris}  
Após remover Estados Unidos: {Brasil=Rio de Janeiro, França=Paris}
```

Acessando-se valores em maps

⊕ A forma mais comum de acessar um valor em um **map** é com o uso da chave associada a ele. Isso pode ser feito por:

⊕ Usando-se **colchetes** []

```
val capitais = mapOf("Brasil" to "Brasília", "França" to "Paris")  
val capitalBrasil = capitais["Brasil"] // Retorna "Brasília"  
println(capitalBrasil) // Imprime: Brasília
```

⊕ Usando-se a função **get**:

```
val capitalFranca = capitais.get("França") // Retorna "Paris"  
println(capitalFranca) // Imprime: Paris
```

Tratando-se valores nulos em maps

- ⊕ Se a chave não existir no **map**, o valor retornado será **null**. Pode-se lidar com isso de várias formas:

```
val capitalAlemanha = capitais["Alemanha"]
if (capitalAlemanha != null) {
    println("A capital da Alemanha é $capitalAlemanha")
} else {
    println("Chave não encontrada no Map")
}
```

- ⊕ Com **arrays**, o acesso a um índice fora dos limites causa um erro de tempo de execução (**exception**), mas isso não acontece com **maps**.

Propriedades e métodos de maps

`get(key)`:

- Retorna o valor associado à chave fornecida ou `null` se a chave não existir.
- Exemplo:

```
val map = mapOf("A" to 1)
println(map.get("A")) // Imprime: 1
println(map.get("B")) // Imprime: null
```

Propriedades e métodos de maps

`put(key, value)` (disponível apenas para `MutableMap`):

- Adiciona uma entrada ao `Map` ou atualiza o valor se a chave já existir.
- Exemplo:

```
val map = mutableMapOf("A" to 1)
map.put("B", 2)
println(map) // Imprime: {A=1, B=2}
```


Propriedades e métodos de maps

`remove(key)` (*disponível apenas para `MutableMap`*):

- Remove a entrada associada à chave fornecida.
- Exemplo:

```
val map = mutableMapOf("A" to 1, "B" to 2)
map.remove("A")
println(map) // Imprime: {B=2}
```

Propriedades e métodos de maps

`clear()` (disponível apenas para `MutableMap`):

- Remove todas as entradas do `Map`.
- Exemplo:

```
val map = mutableMapOf("A" to 1, "B" to 2)
map.clear()
println(map) // Imprime: {}
```

Propriedades e métodos de maps

`putAll(from: Map)` *(disponível apenas para `MutableMap`):*

- Adiciona todas as entradas de outro `Map` ao `Map` atual.
- Exemplo:

```
val map1 = mutableMapOf("A" to 1)
val map2 = mapOf("B" to 2, "C" to 3)
map1.putAll(map2)
println(map1) // Imprime: {A=1, B=2, C=3}
```

sets

- ⊕ Um conjunto (**set**) é uma coleção não ordenada de valores únicos do mesmo tipo;
- ⊕ Isso pode ser extremamente útil quando se deseja garantir que um **item não** apareça mais de uma vez em uma coleção e quando a ordem dos seus itens não é importante;
- ⊕ Um **set** imutável é uma coleção em que você não pode adicionar ou remover elementos após a criação;
- ⊕ Para se criar um set imutável em Kotlin, usa-se a função **setOf**.

sets imutáveis

```
fun main() {  
    // Criando um Set imutável  
    val frutas = setOf("Maçã", "Banana", "Laranja")  
  
    // Exibindo o Set  
    println(frutas)  
}
```

- `setOf`: Esta função cria um `Set` imutável. No exemplo, o `Set` chamado `frutas` contém os elementos "Maçã", "Banana" e "Laranja".
- **Elementos Únicos**: O `Set` não permite elementos duplicados. Se você tentar criar um `Set` com elementos duplicados, os duplicados serão automaticamente ignorados.
- **Ordem**: A ordem dos elementos em um `Set` não é garantida e pode não refletir a ordem de inserção.

Exemplo com Tentativa de Duplicação

```
fun main() {  
    // Tentando criar um Set com elementos duplicados  
    val numeros = setOf(1, 2, 3, 3, 4, 5)  
  
    // Exibindo o Set  
    println(numeros)  
}
```

[1, 2, 3, 4, 5]

Criação de sets a partir de array

`toSet()`: Converte um `Array` em um `Set` imutável.

`toMutableSet()`: Converte um `Array` em um `MutableSet`, que pode ser modificado.

```
fun main() {  
    // Criando um Array  
    val arrayDeFrutas = arrayOf("Maçã", "Banana", "Laranja", "Banana")  
  
    // Convertendo o Array em um Set  
    val frutasSet = arrayDeFrutas.toSet()  
  
    // Exibindo o Set  
    println(frutasSet)  
}
```

Criação de sets a partir de array

```
fun main() {  
    // Criando um Array  
    val arrayDeFrutas = arrayOf("Maçã", "Banana", "Laranja", "Banana")  
  
    // Convertendo o Array em um Set  
    val frutasSet = arrayDeFrutas.toSet()  
  
    // Exibindo o Set  
    println(frutasSet)  
}
```

- `arrayOf`: Cria um `Array` em Kotlin. No exemplo, o `Array` chamado `arrayDeFrutas` contém os elementos "Maçã", "Banana", "Laranja" e uma duplicata de "Banana".
- `toSet`: Converte o `Array` em um `Set`. Isso remove automaticamente qualquer duplicata e mantém apenas elementos únicos.
- **Resultado:** O `Set` resultante (`frutasSet`) conterá apenas os elementos únicos do `Array`.

[Maçã, Banana, Laranja]

Criação de sets a partir de array

`toSet()`: Converte um `Array` em um `Set` imutável.

`toMutableSet()`: Converte um `Array` em um `MutableSet`, que pode ser modificado.

```
fun main() {  
    // Criando um Array  
    val arrayDeFrutas = arrayOf("Maçã", "Banana", "Laranja", "Banana")  
  
    // Convertendo o Array em um MutableSet  
    val frutasMutableSet = arrayDeFrutas.toMutableSet()  
  
    // Adicionando um novo elemento ao MutableSet  
    frutasMutableSet.add("Uva")  
  
    // Exibindo o MutableSet  
    println(frutasMutableSet)  
}
```

Criação de sets a partir de array

```
fun main() {  
    // Criando um Array  
    val arrayDeFrutas = arrayOf("Maçã", "Banana", "Laranja", "Banana")  
  
    // Convertendo o Array em um MutableSet  
    val frutasMutableSet = arrayDeFrutas.toMutableSet()  
  
    // Adicionando um novo elemento ao MutableSet  
    frutasMutableSet.add("Uva")  
  
    // Exibindo o MutableSet  
    println(frutasMutableSet)  
}
```

Saída Esperada:

```
[Maçã, Banana, Laranja, Uva]
```

Acessando-se elementos de um set

A maneira mais comum de acessar elementos em um `Set` é usando um loop `for`.

```
fun main() {  
    val frutas = setOf("Maçã", "Banana", "Laranja")  
  
    // Iterando sobre os elementos do Set  
    for (fruta in frutas) {  
        println(fruta)  
    }  
}
```

Saída esperada:

```
Maçã  
Banana  
Laranja
```

Verificando-se a presença de um elemento em um set

- ⊕ Pode-se verificar se um elemento específico está presente em um set usando-se a palavra-chave **in** ou o método **contains**.

```
fun main() {  
    val frutas = setOf("Maçã", "Banana", "Laranja")  
  
    // Verificando se um elemento está no Set  
    if ("Banana" in frutas) {  
        println("Banana está no conjunto.")  
    }  
  
    // Usando o método contains  
    if (frutas.contains("Laranja")) {  
        println("Laranja está no conjunto.")  
    }  
}
```

Verificando-se a presença de um elemento em um set

```
fun main() {  
    val frutas = setOf("Maçã", "Banana", "Laranja")  
  
    // Verificando se um elemento está no Set  
    if ("Banana" in frutas) {  
        println("Banana está no conjunto.")  
    }  
  
    // Usando o método contains  
    if (frutas.contains("Laranja")) {  
        println("Laranja está no conjunto.")  
    }  
}
```

Saída esperada:

```
Banana está no conjunto.  
Laranja está no conjunto.
```

Convertendo-se para uma lista ou array

- ⊕ Caso se necessite acessar os elementos de um set por índice, pode-se converter o **set** em uma lista ou array;

```
fun main() {  
    val frutas = setOf("Maçã", "Banana", "Laranja")  
  
    // Convertendo o Set em uma Lista para acessar por índice  
    val frutasLista = frutas.toList()  
    println(frutasLista[0]) // Acessando o primeiro elemento  
}
```

Saída esperada:

Maçã