



# PROGRAMAÇÃO IV

## AULA 2

## CONVERSA INICIAL

Olá! Seja bem-vindo a esta aula.

Anteriormente, fizemos o download do Android Studio e criamos nosso primeiro projeto para Android.

Hoje, vamos estudar a linguagem Kotlin. Todos os códigos serão executados diretamente pela função `main()`, sem a necessidade de criar um projeto para Android.

Kotlin é a linguagem oficial de desenvolvimento para Android, tendo conquistado esse posto desbancando nada mais nada menos que a famosa e consolidada linguagem Java. Uma linguagem moderna e com mais recursos acelera o desenvolvimento de qualquer software, por isso, o Kotlin ganhou uma força muito grande na comunidade Android.

Nesta aula, vamos aprender a linguagem Kotlin, para que você se acostume com a sintaxe, facilitando seu aprendizado nas próximas aulas específicas sobre Android.

## TEMA 1 – INTRODUÇÃO AO KOTLIN

No evento Google I/O 2017, o Google anunciou oficialmente o suporte ao Kotlin, arrancando aplausos de todos os presentes na plateia. Esse é um grande passo para tornar o desenvolvimento de aplicativos para Android mais produtivo e divertido.

### Saiba mais

É possível assistir ao anúncio no canal oficial do Google. Preste atenção e, 9 min e 20 seg: DEVELOPER KEYNOTE. **Google Developers**. 18 maio 2017. Disponível em: <https://www.youtube.com/watch?v=EtQ8Le8-zyo>. Acesso em: 19 jan. 2021.

Kotlin é uma linguagem de programação desenvolvida pela JetBrains, mesma empresa que criou o Android Studio. Tem uma sintaxe muito simples e agradável, e é compilada para executar na JVM (Java Virtual Machine), portanto, proporciona interoperabilidade total com Java, o que significa que, no código Kotlin, podemos até usar classes escritas em Java.

Uma das grandes vantagens do Kotlin é a sintaxe moderna e expressiva: quando comparado ao Java, podemos notar que é possível escrever o mesmo código com menos linhas.

Veja uma pequena demonstração do código em Java usado no Android para tratar o evento de clique em um botão e chamar um método qualquer chamado **agendar()**:

```
View btAgendar = findViewById(R.id.btAgendar);  
btAgendar.setOnClickListener(new View.OnClickListener() {  
    @Override public void onClick(View v) {  
        agendar();  
    }  
});
```

Em Kotlin, podemos escrever esse mesmo código em apenas uma linha. Vamos estudar essa sintaxe em detalhes nas aulas de tratamento de eventos de interface.

```
findViewById<Button>(R.id.btAgendar).setOnClickListener{  
    agendar()  
}
```

Na época em que estudamos Kotlin, ficamos surpresos com a quantidade de códigos a menos que é digitada em comparação com o Java. Além da própria sintaxe, há bibliotecas e plugins desenvolvidos pela própria JetBrains que simplificam muito o código.

Vamos dar outro exemplo: no Java, podemos fazer uma requisição HTTP do tipo GET utilizando este código:

```
URLConnection urlConnection = null;
BufferedReader reader = null;
try {
    URL url = new URL("http://www....");
    urlConnection = (URLConnection) url.openConnection();
    urlConnection.setRequestMethod("GET");
    urlConnection.connect();
    InputStream inputStream = urlConnection.getInputStream();
    StringBuffer buffer = new StringBuffer();
    if (inputStream == null) {
        return;
    }
    reader = new BufferedReader(new
InputStreamReader(inputStream));
    String line;
    while ((line = reader.readLine()) != null) {
        buffer.append(line + "\n");
    }
    if (buffer.length() == 0) {
        return;
    }
    String result = buffer.toString();
    System.out.println(result);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (urlConnection != null) {
        urlConnection.disconnect();
    }
    if (reader != null) {
        try {
            reader.close();
        } catch (final IOException e) {
            e.printStackTrace();
        }
    }
}
```

É um código de respeito, não é?

Vamos resumir o que esse código faz? Precisamos abrir a URL, abrir `InputStream`, `BufferedReader`, fazer a leitura dos dados, tratar as `exceptions` e, no final, chamar o método `close()` em cada stream. Sabemos que existem bibliotecas que facilitam essa requisição, mas, ao compararmos apenas o suporte-padrão (standard) da linguagem, segue o mesmo código em Kotlin:

```
val result = URL("http://www...").readText()
```

Agora temos apenas uma linha!

Poderíamos citar diversos outros exemplos, principalmente com foco no Android, mas, para isso, precisaríamos ficar comparando trechos de código escritos em Java com aqueles escritos em Kotlin. Como ainda estamos no início dos estudos, isso não faria muito sentido.

Então, para resumir, basta você saber que Kotlin é uma linguagem moderna, compatível com Java, recomendada pelo Google para o desenvolvimento de aplicativos nativos para Android. Nesta aula vamos explorar a sintaxe da linguagem por meio de exemplos simples e práticos.

## 1.1 KOTLIN PLAYGROUND

Para estudar Kotlin, vamos usar uma ferramenta on-line chamada *Kotlin Playground*, criada pelos fundadores da linguagem, justamente para você digitar alguns códigos e se acostumar com a sintaxe.

### Saiba mais

O legal do Kotlin Playground é que ele pode ser acessado diretamente no seu browser, sem a necessidade de instalar nenhum software:

PLAY KOTLINLANG. Disponível em: <<https://play.kotlinlang.org/>>. Acesso em: 19 jan. 2021.


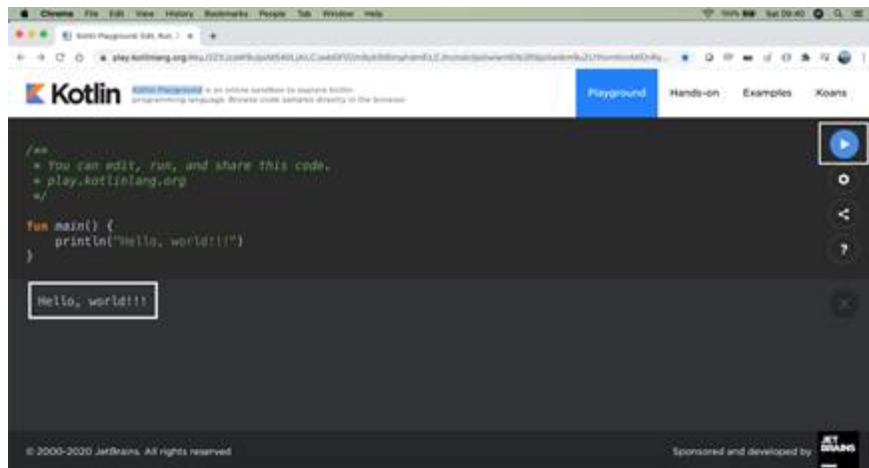
A figura a seguir mostra essa página aberta no browser. No centro da tela, a ferramenta já abriu uma função **main()** com um código bem simples que imprime a famosa mensagem “Hello, world”. Para executar o código, podemos usar o botão **Run** , que fica localizado no canto direito da página. Ao clicar no botão, o código será executado e você verá a mensagem “**Hello, world!!!**” logo abaixo no console.

Figura 1 – Kotlin Playground



Fonte: Kotlin Playground, on-line.

### Saiba mais

**Observação:** nos próximos exemplos, mostraremos o código em Kotlin e o resultado (saída impressa no console). Para testar os exemplos e praticar, copie o código e cole no editor do Kotlin Playground. Execute com o botão **Run**.

## TEMA 2 – SINTAXE BÁSICA E DECLARAÇÃO DE VARIÁVEIS

Vamos ser bem práticos, certo? A ideia é mostrar diversos exemplos de códigos. Você deve copiá-los, colá-los e testá-los no Kotlin Playground. Vamos começar?

### 2.1 STRING TEMPLATES

Para imprimir variáveis no console ou para criar strings, podemos usar a sintaxe **`${variável}`** ou **`${objeto.propriedade}`** para imprimir os valores. Esse recurso é chamado de *String Templates*; seu

principal objetivo é evitar a concatenação de strings. Por exemplo, se tivermos uma variável *nome* e *sobrenome*, podemos concatená-la com o operador `+` da seguinte forma:

```
fun main() {  
    val nome = "Steve"  
    val sobrenome = "Jobs"  
    val nomeCompleto = nome + " " + sobrenome  
    println("> $nomeCompleto")  
}
```

Esse código vai imprimir "Steve Jobs". Contudo, usando a sintaxe do `$`, o código fica bem mais legível. Além do mais, tecnicamente falando, sempre que concatenamos uma *String* com o operador `+`, uma nova *String* é gerada em memória. Portanto, ao usar o `$` também estamos economizando recursos e memória.

```
fun main() {  
    val nome = "Steve"  
    val sobrenome = "Jobs"  
    val nomeCompleto = "$nome $sobrenome"  
    println("> $nomeCompleto")  
}
```

Vamos dar outro exemplo. Copie e cole o código no Kotlin Playground para brincar:



```
fun main() {  
    val nome = "Ricardo"  
    println("Olá, $nome")  
    println("$nome tem ${nome.length} caracteres")  
    val msg = "Meu nome é $nome"  
    println(msg)  
}
```

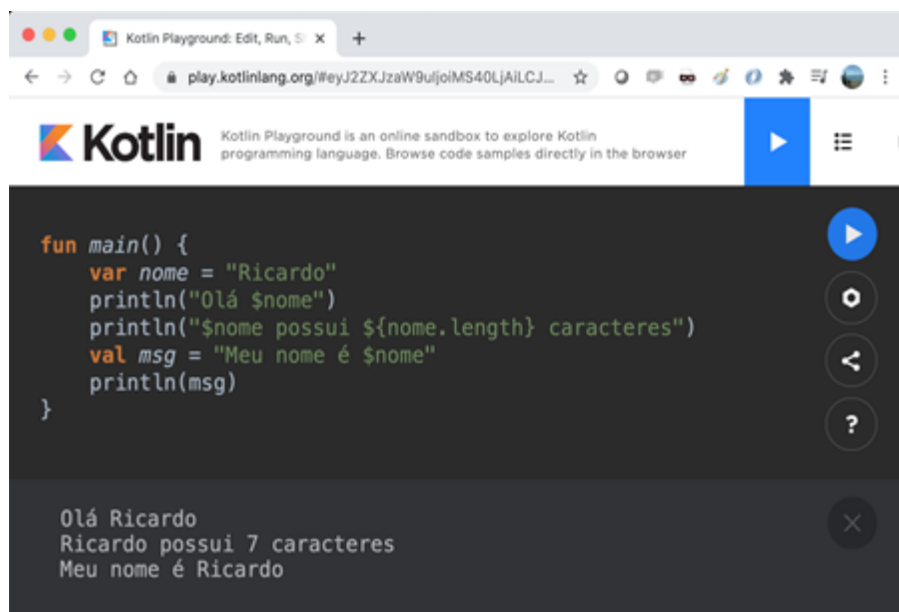
Observe que, quando temos que acessar o `objeto.propriedade`, devemos abrir e fechar chaves, que foi o caso do `nome.length`.

**Resultado:**

Olá, Ricardo  
Ricardo tem 7 caracteres  
Meu nome é Ricardo

A figura a seguir mostra o resultado no Kotlin Playground:

Figura 2 – Resultado no Kotlin Playground



## Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/basic-types.html#string-templates>>. Acesso em: 19 jan. 2021.

## 2.2 – VARIÁVEIS COM VAR E VAL

Uma variável é criada com a sintaxe a seguir. Observe que o tipo da variável é sempre definido com os dois-pontos (:) depois do nome da variável.

```
var nome:String = "Ricardo"
```

O legal é que o tipo da variável pode ser omitido se um valor for atribuído, portanto, nesse caso, não precisamos falar que a variável é uma String, pois o Kotlin é esperto e sabe disso.

```
var nome = "Ricardo"
```

Neste próximo exemplo, vamos declarar uma variável *nome* do tipo *String* e logo depois alterar o seu valor.

```
fun main() {  
    var nome = "Steve"  
    println("Olá, $nome")  
    nome = "Steve Jobs"  
    println("Olá, $nome")  
}
```

### Resultado:

```
Olá, Steve  
Olá, Steve Jobs
```

Só podemos alterar o valor de variáveis declaradas com a palavra reservada **var**. Se a palavra reservada **val** for utilizada, essa variável será tratada como uma constante, e não poderá ser alterada – é o mesmo conceito de **final** no Java. Esse trecho de código tem um erro de compilação, pois a variável *nome* foi declarada como **val**.

Código (não compila):

```
fun main() {  
    val nome = "Steve"  
    println("Olá $nome")  
    nome = "Steve Jobs" // Error: Val cannot be reassigned  
    println("Olá $nome")  
}
```

### Saiba mais

**Dica:** o compilador costuma verificar se a variável está sendo criada e usada apenas uma vez, recomendando que você use **val** na maioria das vezes. Isso vai otimizar questões internas de gerenciamento de memória da linguagem. Sempre que você for alterar o valor da variável, utilize o **var**.

Consulte também: KOTLINLANG. **Documentação oficial**. Disponível em: <https://kotlinlang.org/docs/reference/basic-syntax.html#defining-variables>. Acesso em: 19 jan. 2021.

## 2.3 VARIÁVEIS DO TIPO LISTA

Apenas para dar outro exemplo de como declarar variáveis, imagine este código, que cria uma lista de strings com alguns nomes:

```
val nomes:ArrayList<String> = ArrayList<String>()  
nomes.add("Fulano")  
nomes.add("Beltrano")  
nomes.add("Sicrano")  
print(nomes)
```

Nesse caso, criamos uma variável do tipo *ArrayList<String>*. A parte marcada em amarelo pode ser omitida, pois o Kotlin consegue deduzir o tipo da variável.

Existem outras sintaxes para criar listas – por exemplo, criar tudo na mesma linha. Nesse código, a parte amarela também pode ser omitida.

```
val nomes:List<String> = listOf("Fulano", "Beltrano", "Sicrano")  
print(nomes)
```

## 2.4 CONVERSÃO DE TIPOS: AS, AS? E IS

*Cast* é o processo de converter um tipo para outro. Isso é muito comum ao escrever códigos quando você não sabe qual o tipo de determinada variável. Primeiro fazemos o teste para verificar se a variável é do tipo esperado; depois podemos fazer a conversão, que é chamada de cast.

Por exemplo, em Java, esse código faz a verificação para checar se um objeto é do tipo *String*, e depois faz a conversão. O cast no Java tem a sintaxe entre parênteses – por exemplo: *(String)*.

```
Object variavelQualquer = "Ricardo";  
if(variavelQualquer instanceof String) { // valida se é String  
    String s = (String) variavelQualquer; // faz o cast  
    System.out.println(s);  
}
```

Em Kotlin, podemos escrever esse código assim:

```
val variavelQualquer:Any = "Ricardo"  
if(variavelQualquer is String) {           // valida se é String  
    val s = variavelQualquer as String    // faz o cast  
    println(s)  
}
```

Na verdade, o Kotlin é um pouco mais esperto. Depois de fazer o teste (if) para verificar se é uma String, não precisamos fazer a conversão (cast) dentro do if, pois isso é automático. Portanto, dentro do if podemos usar métodos da classe String, pois o cast já foi feito de forma implícita.

```
val variavelQualquer:Any = "Ricardo"  
if(variavelQualquer is String) {  
    println(variavelQualquer.length)  
    // O método length só existe na classe String  
}
```

Depois dessa breve introdução, vamos estudar mais alguns detalhes.

Para fazer cast no Kotlin, ou seja, converter de um tipo para outro, é utilizado o operador **as**. Esse operador pode lançar uma exceção do tipo *ClassCastException*, caso a conversão não possa ser feita. Outra opção é utilizar o operador **as?** (cast seguro), que retorna *null* em vez de lançar uma exceção. O

exemplo a seguir demonstra como fazer cast de um objeto e também mostra como utilizar o operador **is**, a fim de verificar se a variável é do tipo informado. O **is** funciona como o **instanceof** do Java, com a vantagem de que dentro do if não é preciso fazer o cast novamente.

```
fun main() {  
    val s:Any = "Steve"  
    println(s as String)  
    println(s as? Int) // Aqui imprime nulo pois s não é um  
    Int  
    if (s is String) {  
        println("$s é uma String") // smart cast  
    }  
}
```

#### Resultado:

```
Steve  
null  
Steve é uma String
```

Veja que a segunda linha imprimiu **null**, pois uma String não pode ser convertida para Int. Dependendo do caso, esse comportamento pode ser melhor do que lançar uma exceção. Essa é a diferença entre **as?** e **as**. Altere o código para usar o operador **as** e veja o que acontece. BOOOM! Rs!

Outro detalhe importante é que, depois de testar o tipo de uma variável em alguma condição if, não é preciso fazer o cast dentro do bloco, pois, se a condição é verdadeira, o Kotlin faz o cast automaticamente. Isso é chamado de **Smart Cast**.

**Saiba mais**

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/typecasts.html>>. Acesso em: 19 jan. 2021.

## 2.5 OPERADOR TERNÁRIO

Imagine o seguinte código:

```
fun main() {  
    println(parOuImpar(1)) // imprime ímpar  
    println(parOuImpar(2)) // imprime par  
}  
  
fun parOuImpar(a: Int): String {  
    if (a % 2 == 0) {  
        return "par"  
    } else {  
        return "ímpar"  
    }  
}
```

O operador ternário existente em várias linguagens e é utilizado para evitar o uso do if/else em várias linhas. No Java, o operador ternário é representado por uma interrogação '?' seguida de dois-pontos ':'. No Kotlin, podemos fazer o if/else todo na mesma linha:

```
fun main() {  
    println(parOuImpar(1)) // imprime ímpar  
    println(parOuImpar(2)) // imprime par  
}  
  
fun parOuImpar(a: Int): String {  
    return if (a % 2 == 0) "par" else "ímpar"  
}
```

Outra sintaxe interessante do Kotlin é que as funções que têm apenas uma linha podem ser escritas como vemos a seguir. Basta colocar o símbolo de igual '=' seguido da expressão que você quer utilizar. Simples, não é?

```
fun main() {  
    println(parOuImpar(1)) // imprime ímpar  
    println(parOuImpar(2)) // imprime par  
}  
  
fun parOuImpar(a: Int) = if (a % 2 == 0) "par" else "ímpar"
```

### Saiba mais

KOTLINLANG. **Documentação oficial sobre o operador ternário e controles de fluxo.** Disponível em: <https://kotlinlang.org/docs/reference/control-flow.html#if-expression>. Acesso em: 19 jan. 2021.

## 2.6 OPERADOR ELVIS

Outro operador famoso é o **Elvis**, identificado por **"?:"**. Esse operador pode ser utilizado para responder à seguinte pergunta: se o valor da variável não for nulo, use seu próprio valor; caso contrário,



use outro. Esse operador é muito usado no dia a dia para evitar o uso de if/else.

Exemplo:

```
fun main() {  
    println(enviarEmail("Steve"))  
    println(enviarEmail("Steve", "Welcome"))  
}  
fun enviarEmail(usuario: String, titulo: String? = null): String {  
    val s = titulo?: "Bem-vindo,"  
    return "$s $usuario"  
}
```

Com o operador Elvis, a variável 's' sempre terá um valor. Ela vai receber o valor de título, caso ele não seja nulo; caso contrário, ficará com o texto "Bem-vindo". Resultado:

```
Bem-vindo, Steve  
Welcome Steve
```

Essa função utiliza o parâmetro **título**, caso não seja nulo, ou utiliza a expressão à direita do operador Elvis. Veja que o parâmetro **título** na função que criamos pode aceitar valores nulos, pois foi criado como **String?**. Ele também tem o valor **null** como padrão, caso nenhum valor seja informado, o que é conhecido como **default arguments**. Esse recurso permite deixar parâmetros de funções como sendo opcionais.

### Saiba mais

**Nota:** vamos estudar sobre valores nulos no Kotlin em outro tópico.

Consulte também: KOTLINLANG. **Documentação oficial do operador Elvis**. Disponível em: <https://kotlinlang.org/docs/reference/null-safety.html#elvis-operator>. Acesso em: 19 jan. 2021.

## TEMA 3 – FUNÇÕES

A declaração de funções segue a seguinte sintaxe:

```
fun test(param1: Tipo, param2: Tipo, ...): TipoRetorno {  
    // Código  
}
```

Para demonstrar, vamos criar duas funções: a função ***imprimir(String)*** recebe uma String como parâmetro e a imprime no console. Ela não tem retorno, portanto, é identificada como **Unit**, semelhante ao **void** do Java. A função **soma(Int,Int)** recebe dois inteiros e retorna outro inteiro com a soma.

```
fun main() {  
    val nome = "Steve"  
    imprimir(nome)           // Imprime: Steve  
    val soma = somar(2,3)  
    imprimir("Soma: $soma")  // Imprime: Soma: 5  
}  
  
fun imprimir(s: String): Unit {  
    println(s)  
}  
  
fun somar(a: Int, b: Int): Int {  
    return a + b  
}
```

**Resultado:**

```
Steve  
Soma: 5
```

Um detalhe importante é que, quando a função não tem retorno, a palavra **Unit** pode ser omitida. Portanto, a função imprimir pode ficar assim:

```
fun imprimir(s: String) {  
    println(s)  
}
```

Outro recurso interessante do Kotlin é a sintaxe resumida ao declarar funções, chamada de **Single-Expression functions**. Sempre que uma função tiver apenas uma linha, não será preciso abrir e fechar chaves { }: basta usar o operador de igual '=' e escrever tudo em uma única linha. Veja que inclusive o tipo do retorno pode ser omitido, pois o Kotlin pode descobrir isso sozinho. O código a seguir tem o mesmo resultado que o anterior, mas repare na diferença ao declarar as funções:

```
fun main() {  
    val nome = "Steve"  
    imprimir(nome)  
    val soma = somar(2,3)  
    imprimir("Soma: $soma")  
}  
  
fun imprimir(s: String) = println(s) // Tudo em apenas uma linha  
  
fun somar(a: Int, b: Int) = a + b // Tudo em apenas uma linha
```

**Saiba mais**

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/functions.html>>. Acesso em: 19 jan. 2021.

### 3.1 FUNÇÕES – DEFAULT ARGUMENTS

Um dos recursos mais interessantes no Kotlin refere-se aos parâmetros das funções que podem ter valores-padrão, o que evita ter de criar vários métodos com a mesma assinatura (method overloading). Para exemplificar, veja o seguinte código, que tenta converter uma **String** para **Int**, permitindo que, caso a conversão não seja possível, um valor inteiro padrão seja utilizado.

```
fun main() {  
    println(enviarEmail("Steve"))           // Imprime: Bem vindo Steve  
    println(enviarEmail("Steve", "Welcome")) // Imprime: Welcome Steve  
}  
fun enviarEmail(usuario: String, titulo:String = "Bem-vindo,"): String  
{  
    return "$titulo $usuario"  
}
```

**Resultado:**

```
Bem-vindo, Steve  
Welcome Steve
```

#### Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/functions.html#default-arguments>>. Acesso em: 19 jan. 2021.

## 3.2 FUNÇÕES: NAMED ARGUMENTS

O Kotlin permite que o nome dos parâmetros seja utilizado no momento de chamar uma função, possibilitando inclusive que a passagem de parâmetros seja feita fora de ordem. Isso muitas vezes facilita a leitura do código, principalmente se tivermos muitos parâmetros envolvidos.

```
fun main() {  
    teste("Ricardo", "Lecheta", "Android")  
  
    teste("Ricardo")  
  
    teste("Ricardo", disciplina = "Android")  
}  
  
fun teste(  
    nome: String?,  
    sobrenome: String? = null,  
    disciplina: String? = null) {  
    println("Nome: $nome $sobrenome, disciplina: $disciplina")  
}
```

### Resultado:

```
Nome: Ricardo Lecheta, disciplina: Android  
Nome: Ricardo null, disciplina: null  
Nome: Ricardo null, disciplina: Android
```

**Saiba mais**

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/functions.html#named-arguments>>. Acesso em: 19 jan. 2021.

### 3.3 FUNÇÕES – VARARGS

Algo muito comum em programação é um tipo especial de parâmetro (normalmente o último) chamado **varargs**, que pode receber um ou mais parâmetros separados por vírgula. Em Kotlin, isso é feito com a palavra reservada **vararg**:

```
fun toList(vararg args: String): List<String> {  
    val list = ArrayList<String>()  
    for (s in args)  
        list.add(s)  
    return list  
}  
  
fun main() {  
    val frutas = toList("Banana", "Laranja", "Maçã")  
    println(frutas) // Imprime: [Banana, Laranja, Maçã]  
}
```

A função **toList** recebe vários argumentos por **vararg** e, nesse caso, está criando e retornando uma lista com os valores passados. O resultado é um *ArrayList* com o nome das frutas.

```
[Banana, Laranja, Maçã]
```

**Saiba mais**

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/functions.html#variable-number-of-arguments-varargs>>. Acesso em: 19 jan. 2021.

### 3.4 TIPOS GENÉRICOS

Tipos genéricos (Generics) é um assunto que pode ser um tanto complexo, principalmente se você for novo em programação. Vamos dizer que queremos criar uma função que crie uma lista de Strings, como fizemos no tópico anterior. Mas como fazemos para criar outra função que crie uma lista de inteiros?

Temos que criar duas funções e duplicar o código? E se o algoritmo for exatamente o mesmo? Como fazemos?

Como as funções são exatamente iguais, podemos utilizar tipos genéricos para isso, pois, na prática, não importa se o tipo é String ou Int, o objetivo é criar uma lista.

Para criar tipos genéricos (Generics) em Kotlin, utilizamos a seguinte sintaxe:

```
fun <T> toList(vararg args: T): List<T> {  
    val list = ArrayList<T>()  
    for (s in args)  
        list.add(s)  
    return list  
}  
  
fun main() {  
    val strings = toList<String>("Banana", "Laranja", "Maçã")  
    println(strings) // Imprime: [Banana, Laranja, Maçã]  
  
    val ints = toList<Int>(1,2,3,4,5)  
    println(ints) // Imprime: [1, 2, 3, 4, 5]  
}
```

**Resultado:**

```
[Banana, Laranja, Maçã]  
[1, 2, 3, 4, 5]
```

Observe que, na declaração da função, definiu-se o tipo genérico `<T>`, que será substituído no momento de chamar a função por `<String>`, `<Int>` etc., tornando o algoritmo dessa função genérica. Para que você entenda bem, veja esta linha de código:

```
val strings = toList<String>("Banana", "Laranja", "Maçã")
```

O compilador é tão esperto, que o tipo genérico até pode ser omitido, pois ele vai deduzir que você está passando Strings ou Ints e vai criar e retornar o tipo correto de lista:

```
val strings = toList("Banana", "Laranja", "Maçã")  
val ints = toList(1, 2, 3, 4, 5)
```

Note que, geralmente, declaramos as variáveis com **var** ou **val** e também omitimos o tipo na declaração, pois o Kotlin é esperto e sabe deduzir isso também. No entanto, às vezes, para deixar o código mais claro ou facilitar a sua leitura, dependendo do gosto do programador, pode-se fazer assim:

```
val strings: List<String> = toList("Banana", "Laranja", "Maçã")  
val ints: List<Int> = toList(1, 2, 3, 4, 5)
```



Nesse exemplo, as variáveis estão sendo declaradas especificando o seu tipo. Isso é opcional e pode ajudar enquanto você está aprendendo. Com o tempo, provavelmente você vai começar a preferir a sintaxe mais resumida.

### Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/generics.html>>. Acesso em: 19 jan. 2021.

## TEMA 4 – CLASSES E ORIENTAÇÃO A OBJETOS

Vamos falar um pouco sobre orientação a objetos. Para prosseguir, é muito importante que você conheça bem os conceitos de classes, interfaces, métodos, atributos, construtor, herança, encapsulamento etc. Portanto, caso seja necessário, por favor revise seus estudos sobre orientação a objetos.

Criar classes em Kotlin é simples. Basta utilizar a palavra reservada **class**. A grande diferença é que o construtor-padrão é declarado na mesma linha da classe.

O código a seguir mostra como criar uma classe **Carro** com os atributos *nome* e *ano* e um construtor que recebe esses parâmetros. O construtor primário em Kotlin não pode ter nenhum código, portanto o código foi inicializado com a palavra `init` (bloco de inicialização). O código também mostra como sobrescrever o método **toString()**, que é chamado sempre que a classe é convertida para String.

```
class Carro(nome:String, ano:Int) {  
    val nome:String  
    val ano:Int  
    init {  
        // bloco de inicialização  
        this.nome = nome  
        this.ano = ano  
    }  
    override fun toString():String {  
        return "Carro $nome, ano: $ano"  
    }  
}  
  
fun main() {  
    val c1 = Carro("Fusca",1950)  
    // Imprime: Carro Fusca, ano: 1950  
    println(c1)  
    println("Carro ${c1.nome}, ano: ${c1.ano}")  
}
```

### Resultado:

```
Carro Fusca, ano: 1950  
Carro Fusca, ano: 1950
```

### Saiba mais

**Nota:** para sobrescrever um método, é preciso utilizar a palavra reservada **override**.

Veja que, para criar uma instância do objeto *Carro*, não é preciso utilizar o famoso operador **new** como no Java; basta informar o nome da classe e os parâmetros:

```
val c1 = Carro("Fusca",1950)
```

No Java, seria assim:

```
Carro c1 = new Carro("Fusca",1950)
```

### Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/classes.html>>. Acesso em: 19 jan. 2021.

## 4.1 CLASSES – HERANÇA

Para herdar de uma classe, basta utilizar a sintaxe dos dois-pontos, seguida do nome da classe-mãe. O código a seguir cria uma classe-mãe *Automóvel* e uma classe-filha *Carro*. Veja que a classe *Carro* precisa invocar o construtor da classe-mãe para que o código funcione corretamente.

```
open class Automovel(nome:String, ano:Int) {
    val nome:String
    val ano:Int
    init {
        this.nome = nome
        this.ano = ano
    }
    open fun acelerar(velocidade:Int) {
        print("Acelerando este automóvel $velocidade");
    }
    override fun toString():String {
        return "Automóvel $nome, ano: $ano"
    }
}

class Carro(nome:String, ano:Int): Automovel(nome,ano) {
    override fun acelerar(velocidade:Int) {
        print("Acelerando este carro $velocidade km/h");
    }
}

fun main() {
    val c1 = Carro("Fusca",1950)
    // Imprime: Automóvel Fusca, ano: 1950
    println(c1)

    // Imprime: Carro Fusca, ano: 1950
    println("Carro ${c1.nome}, ano: ${c1.ano}")

    // Imprime: Acelerando este carro 100 km/h
    c1.acelerar(100)
}
```

**Resultado:**

```
Automóvel Fusca, ano: 1950  
Carro Fusca, ano: 1950  
Acelerando este carro 100 km/h
```

Para que a herança possa ser realizada, a classe *Automóvel* foi anotada como **open**, pois, no Kotlin, todas as classes são **final** por padrão, ou seja, não é possível utilizar herança. O mesmo vale para métodos. Observe que o método **acelerar(Int)** foi anotado como **open** – somente assim ele pode ser sobrescrito.

### Saiba mais

**Nota:** é recomendado ler a documentação oficial do Kotlin para estudar bem a sintaxe, pois não vamos conseguir estudar tudo aqui. De qualquer forma, bons conceitos sobre programação e orientação a objetos são necessários para entender questões sobre herança, construtores, interfaces etc. Explicar esses conceitos não faz parte do objetivo deste estudo. Aqui estamos apenas estudando a sintaxe da linguagem para facilitar seu aprendizado ao desenvolver aplicativos para Android.

## 4.2 DATA CLASSES

Frequentemente, em programação, utilizamos classes que apenas contêm informações.

Em Java, esses objetos são chamados de *Java Beans* e têm os clássicos métodos getters/setters, com a sintaxe mostrada a seguir.

Veja que também implementamos os métodos **equals()**, **hashCode()** e **toString()**:

- **Carro.java**

```
public class Carro {
    private String nome;
    public Carro(String nome) {
        super();
        this.nome = nome;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((nome == null) ? 0 :
nome.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Carro other = (Carro) obj;
        if (nome == null) {
            if (other.nome != null)
                return false;
        } else if (!nome.equals(other.nome))
            return false;
        return true;
    }
}
```

```
@Override  
public String toString() {  
    return "Carro [nome=" + nome + "];"  
}  
}
```

Em Kotlin, essa mesma classe pode ser escrita utilizando **Data Classes** em apenas uma linha. Não precisa falar mais nada, não é?

- **Carro.kt**

```
data class Carro(val nome:String)
```

Data Classes têm uma sintaxe resumida, mas automaticamente implementam as seguintes funções:

- equals() e hashCode()
- toString()
- copy()

Permitem acessar os atributos facilmente, sem a necessidade dos get/set.

Para testar a data class, crie o arquivo e execute o código. Fácil, não é?

```
data class Carro(val nome:String)  
  
fun main() {  
    val c = Carro("Fusca")  
    print("Carro: ${c.nome}")  
}
```

## Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/data-classes.html>>. Acesso em: 19 jan. 2021.

## 4.3 O OBJETO COMPANION

Métodos estáticos – ou métodos de classe, como são chamados – permitem criar métodos que podem ser chamados diretamente com a sintaxe "**Classe.metodo()**", sem a necessidade de criar uma instância da classe. Geralmente esses métodos podem ser utilizados para funções que retornam os dados e não precisam armazená-los nos atributos na classe, ou seja, não precisam salvar o estado do objeto.

A maneira de criar métodos e atributos estáticos no Kotlin é declará-los dentro do objeto companion (companion object).



```
data class Carro(val nome:String)

class CarroService {
    companion object {
        fun getCarros(): List<Carro> {
            val carros = mutableListOf<Carro>()
            for (i in 1..3) {
                val c = Carro("Carro $i")
                carros.add(c)
            }
            return carros
        }
    }
}

fun main() {
    val carros = CarroService.getCarros() // Chamada estática
    for(c in carros) {
        println(c)
    }
}
```

### Resultado:

```
Carro(nome=Carro 1)
Carro(nome=Carro 2)
Carro(nome=Carro 3)
```

Observe que, nesse exemplo, utilizamos uma **Data Class** chamada *Carro*.

Ao chamar a função **getCarros()**, não foi necessário criar uma instância da classe **CarroService**, pois utilizamos a sintaxe estática **Classe.funcao()**.

## Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/object-declarations.html#companion-objects>>. Acesso em: 19 jan. 2021.

**Nota avançada:** recomendamos a leitura da documentação oficial na seção de interoperabilidade com a linguagem Java. Como podemos escrever código em Kotlin e em Java ao mesmo tempo, às vezes temos de nos preocupar com as chamadas entre as duas linguagens. O objeto companion do Kotlin, na prática, é um singleton. Embora possamos usar a sintaxe de métodos estáticos (sem criar instância do objeto) para chamar as funções, ele não é estático na verdade. Isso é apenas magia do compilador. Se for necessário exportar a classe para o Java de forma que o método seja estático no nível de bytecode, é necessário anotá-lo com **@JvmStatic**.

## 4.4 SINGLETONS

Um singleton é um padrão que permite que uma classe tenha apenas uma instância (objeto) em memória. Para criar um singleton em Java, são necessárias várias linhas de código, mas, em Kotlin, basta utilizar a palavra reservada **object** em lugar de **class**.

O próximo exemplo é muito parecido com o anterior, portanto, como já foi explicado, é importante que sua base de programação seja boa para entender a diferença entre uma classe com métodos estáticos e um singleton. No Kotlin, ambos têm chamadas com a mesma sintaxe, mas o comportamento, dependendo do caso, será diferente.

```
data class Carro(val nome:String)

object CarroService {
    fun getCarros(): List<Carro> {
        val carros = mutableListOf<Carro>()
        for (i in 1..3) {
            val c = Carro("Carro $i")
            carros.add(c)
        }
        return carros
    }
}

fun main() {
    val carros = CarroService.getCarros()
    for(c in carros) {
        println(c)
    }
}
```

### Resultado:

```
Carro(nome=Carro 1)
Carro(nome=Carro 2)
Carro(nome=Carro 3)
```

### Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/object-declarations.html#object-declarations>>. Acesso em: 19 jan. 2021.

**Nota:** uma classe singleton tem uma única instância durante toda a execução do programa, **portanto tenha cuidado** ao armazenar atributos na classe, pois eles serão compartilhados por todo o sistema. Gostamos de utilizar singletons para criar métodos que retornam listas e objetos, como mostrado aqui. Isso é útil ao fazer consultas no banco de dados ou em web services, pois a sua sintaxe é simples e facilita o código.

## 4.5 LISTAS – LISTOF E MUTABLELISTOF

Se você leu atentamente, percebeu que criamos listas de duas formas diferentes nos exemplos anteriores.

Podemos utilizar um **ArrayList**, que é uma lista ordenada:

```
val list = ArrayList<String>()
```

Contudo, se quisermos que o Kotlin crie essa lista para nós, é possível utilizar a função **mutableListOf()**, que retorna uma lista mutável, ou seja, que pode ser alterada.

```
val carros = mutableListOf<Carro>()
```

Dessa forma, a real implementação da lista fica abstraída. Segundo a documentação do Kotlin, essa é a sintaxe recomendada. Na prática, ambas as sintaxes funcionam de forma idêntica. Vale lembrar também que o Kotlin difere listas mutáveis de imutáveis. Dessa forma, podemos utilizar as funções **mutableListOf()** (mutável) ou **listOf()** (imutável), conforme a necessidade. Para você praticar um pouco mais, segue um exemplo simples que cria uma lista e adiciona alguns objetos do tipo *Carro* a ela:

```
data class Carro(val nome:String)

fun main() {
    val carros = mutableListOf<Carro>()
    val c1 = Carro("Fusca")
    val c2 = Carro("Brasília")
    carros.add(c1)
    carros.add(c2)
    for(c in carros) {
        println(c.nome)
    }
}
```

### Saiba mais

Vale a pena ler a documentação sobre coleções, assim como sobre os outros tipos de listas, como sets e maps: KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/collections.html>>. Acesso em: 19 jan. 2021.

## 4.6 ENUM

Enum é um tipo de objeto que contém constantes predefinidas. A sintaxe é simples, conforme estes exemplos da documentação oficial:

```
enum class Direction { NORTH, SOUTH, WEST, EAST }
```

Uma enum pode ter um construtor:

```
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF)  
}
```

Nessa enum, podemos acessar o rgb da cor desta forma:

```
val azul = Color.BLUE.rgb
```

### Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <<https://kotlinlang.org/docs/reference/enum-classes.html>>. Acesso em: 19 jan. 2021.

## TEMA 5 – KOTLIN AVANÇADO

Os próximos tópicos são avançados. Fique tranquilo se não conseguir entender tudo. O importante é ter este estudo como um guia contínuo para a sua formação, pois sua caminhada como desenvolvedor está apenas começando, e alguns conceitos não se aprendem da noite para o dia.

Um bom desenvolvedor estuda continuamente para melhorar a cada dia, portanto, leia novamente esses tópicos sempre que precisar.

### 5.1 HIGHER-ORDER FUNCTIONS E LAMBDA

Esse é um dos melhores recursos da linguagem e traz muita produtividade no momento de escrever o código. Kotlin é uma linguagem de paradigma funcional que permite que funções recebam outras funções como parâmetro, além de permitir que uma função retorne outra função. Isso é chamado na documentação oficial de **Higher-Order Functions**. Para demonstrar a sintaxe, veja o seguinte exemplo:

```
fun filtrar(list: List<Int>, filtro: (Int) -> (Boolean)):
List<Int> {
    val newList = arrayListOf<Int>()
    for (item in list) {
        if(filtro(item)) {
            newList.add(item)
        }
    }
    return newList
}

fun numerosPares(numero: Int) = numero % 2 == 0
fun numerosMaiorQue3(numero: Int) = numero > 3
fun main() {
    val ints = listOf(1,2,3,4,5)
    println(ints)
    val pares = filtrar(ints, ::numerosPares)
    println(pares)
    val list = filtrar(ints, ::numerosMaiorQue3)
    println(list)
}
```

**Resultado:**

```
[1, 2, 3, 4, 5]
[2, 4]
[4, 5]
```

Esse é um exemplo muito interessante. Leia o código com atenção e leve o tempo que precisar para absorver o conceito, nem que tenha que ler várias vezes. Aliás, faça isso em todos os exemplos que formos estudar.

Nesse código, as funções ***numerosPares(Int)*** e ***numerosMaiorQue3(Int)*** são utilizadas para filtrar a lista de inteiros. Ao declarar a função ***filtrar(list,filtro)***, veja que a sintaxe **(Int) -> (Boolean)** foi utilizada para informar que a função que será passada como parâmetro deve receber um inteiro e retornar um booleano.

A sintaxe **::funcao** é utilizada para passar uma função como parâmetro. Se a função tiver os mesmos argumentos que a função desejada, a chamada será feita.

### Saiba mais

**Dica:** dedique-se a entender esse exemplo e só prossiga se realmente entendê-lo. Caso não tenha entendido, será complicado compreender o que são **lambdas**. Uma vez entendido, vamos dar o próximo passo e conferir o que são **lambdas**.

Podemos dizer que uma lambda é uma forma simplificada de passar o código de uma função como parâmetro para outra função. Uma função lambda tem uma sintaxe simplificada e facilita passar funções como parâmetros. Vamos utilizar o exemplo anterior. Digamos que o objetivo seja filtrar apenas os números pares. Estamos utilizando este código:

```
val pares = filtrar(ints, ::numerosPares)
```

Com lambdas, podemos fazer a chamada entre chaves {}, assim:

```
val pares = filtrar(ints, { /* código aqui */ })
```

Para que o código compile, precisamos passar uma função que receba um inteiro e retorne um booleano. Portanto, a sintaxe ficará assim:



```
val pares = filtrar(ints, {  
    numero:Int -> numerosPares(numero)  
})
```

Parece que o código ficou mais complicado, não é? Mas calma. Continue lendo.

Veja que declaramos um parâmetro inteiro e fizemos a chamada da função passando o número como parâmetro. No entanto, essa sintaxe parece ser mais complicada do que a anterior. A vantagem das lambdas é que, sempre que a função recebe apenas um parâmetro, podemos usar o parâmetro **it** genérico da linguagem. Utilizando o **it**, a chamada fica simplificada desta forma:

```
val pares = filtrar(ints, {numerosPares(it)})
```

Como qualquer parâmetro, a função lambda foi passada separada por vírgulas; a diferença é que ela é feita dentro das chaves {}. Contudo, o Kotlin ainda tem outra sintaxe que facilita o uso das funções lambdas. Caso a função seja o último parâmetro que precisa ser informado, podemos fechar os parênteses do método e abrir uma expressão entre chaves, assim:

```
val pares = filtrar(ints) {  
    numerosPares(it)  
}
```

A sintaxe dessa vez ficou bem enxuta, graças às lambdas. Quebramos a linha para facilitar sua leitura, mas, se preferir, pode deixar tudo em uma única linha.

Como podemos ver, as funções lambdas têm convenções que, se bem-utilizadas, reduzem muito a quantidade de código escrito, facilitando sua leitura e a manutenção. O melhor das lambdas é que podemos passar funções como parâmetros sem criar essas funções, o que é chamado de **funções anônimas**. Veja que nos exemplos anteriores criamos as funções **numerosPares(Int)** e

**numerosMaiorQue3(Int)**. Agora é a hora do show. Usando lambdas, podemos escrever o mesmo código assim:

```
fun filtrar(list: List<Int>, filtro: (Int) -> (Boolean)):
List<Int> {
    val newlist = arrayListOf<Int>()
    for (item in list) {
        if(filtro(item)) {
            newlist.add(item)
        }
    }
    return newlist
}

fun main(args: Array<String>) {
    val ints = listOf(1,2,3,4,5)
    println(ints)

    // Apenas números pares
    val pares = filtrar(ints) { it % 2 == 0 }
    println(pares)

    // Apenas números maiores que 3
    val list = filtrar(ints) { it > 3 }
    println(list)
}
```

### Resultado:

```
[1, 2, 3, 4, 5]
[2, 4]
[4, 5]
```

Conseguiu entender? É como se tivéssemos passado um código diretamente para a função filtrar. Para isso, não criamos nenhuma função.

### Saiba mais

**Nota:** vamos usar muito esse recurso no Android, para adicionar eventos nos botões e outros componentes.

Consulte também: KOTLINLANG. **Documentação oficial**. Disponível em: <https://kotlinlang.org/docs/reference/lambdas.html>. Acesso em: 19 jan. 2021.

## 5.2 EXTENSÕES

Extensões são muito utilizadas em Kotlin e permitem adicionar métodos em classes sem utilizar herança. Esse recurso é muito útil para criar métodos em classes que são **final** e que, a princípio, não podem ter classes-filhas (subclasses).

Outra utilidade das extensões refere-se ao fato de terem uma alternativa para evitar os clássicos métodos utilitários para fazer algumas tarefas comuns em um sistema. Para você entender o que é um método utilitário, vamos dar um exemplo. Imagine que temos uma classe **Produto** com nome e valor, e o objetivo é mostrar o preço do produto formatado em reais (R\$). Uma maneira de escrever esse código seria criar uma classe utilitária (Utils) com o método que faz essa formatação para R\$. Uma classe utilitária geralmente tem métodos pequenos e objetivos, e na maioria das vezes também são “estáticos”, ou seja, podem ser chamados sem criar uma instância do objeto. Conforme já estudamos no Kotlin, isso é feito com o **companion object**.

```
import java.text.DecimalFormat
import java.util.*

data class Produto(val nome: String, val valor: Double)

class Utils {
    companion object {
        fun toReais(valor: Double): String {
            val format = DecimalFormat
                .getCurrencyInstance(Locale("pt", "br"))
            return format.format(valor)
        }
    }
}

fun main() {
    val p = Produto("Camiseta", 10.50)
    val preco = Utils.toReais(p.valor)
    print("Produto: ${p.nome}, preço: $preco")
}
```

### Resultado:

```
Produto: Camiseta, preço: R$ 10,50
```

Conforme vimos nesse exemplo, tivemos que criar a classe **Utils**, pois a linguagem não oferece nenhuma forma-padrão de formatar um **Double** para reais R\$.

E se a classe Double tivesse um método **toReais()** que já fizesse isso?

Bom, o problema é que não temos como criar uma classe-filha de Double, pois a linguagem não permite.

É nesse momento que as extensões dão o ar da sua graça.

Usando extensões, podemos colocar o método **toReais()** na classe **Double** sem precisar criar uma classe-filha. É mágico. O próximo exemplo demonstra como fazer isso. Veja como o código ficou menor.

```
import java.text.DecimalFormat
import java.util.Locale

data class Produto(val nome: String, val valor: Double)

fun Double.toReais(): String {
    val format = DecimalFormat
        .getCurrencyInstance(Locale("pt", "br"))
    return format.format(this)
}

fun main() {
    val p = Produto("Camiseta", 10.50)
    val preco = p.valor.toReais()
    print("Produto: ${p.nome}, preço: $preco")
}
```

Observe que declarar uma extensão para um tipo ou classe conhecida, como a `Double`, é bem simples; a sintaxe é a mesma de qualquer função. Basta colocar o “tipo” antes do nome da função:

```
fun Double.toReais(): String { ... }
```

Para praticar o conceito de extensões, vamos criar outro exemplo e melhorar o código que mostramos anteriormente com lambdas. Naquele exemplo, o método **filtrar(list,filtro)** recebia a lista e a função de filtro, mas uma maneira mais moderna de criar esse tipo de função é criar uma extensão na própria lista, assim:

```
fun List<Int>.filtrar(filtro: (Int) -> (Boolean)): List<Int> {  
    val newList = arrayListOf<Int>()  
    for (item in this) {  
        if(filtro(item)) {  
            newList.add(item)  
        }  
    }  
    return newList  
}  
  
fun main() {  
    val ints = listOf(1,2,3,4,5)  
    println(ints)  
  
    // Apenas números pares  
    val pares = ints.filtrar { it % 2 == 0 }  
    println(pares)  
}
```

**Resultado:**

```
[1, 2, 3, 4, 5]  
[2, 4]  
[4, 5]
```

O resultado é o mesmo de antes, mas essa alteração tornou o código mais legível e expressivo, pois escrevemos de uma forma semelhante à linguagem natural que falamos. O Kotlin é conhecido por ser uma linguagem bastante expressiva, porque o código, muitas vezes, lembra a forma como falamos. No próximo exemplo, o código está dizendo "se a variável `s` é uma `String`".

```
if (s is String)
```

Esse código diz: "crie a variável s como uma String".

```
var s = obj as String
```

Esses são exemplos simples, mas demonstram o significado da expressividade em linguagens de programação.

### Saiba mais

KOTLINLANG. **Documentação oficial**. Disponível em: <https://kotlinlang.org/docs/reference/extensions.html>. Acesso em: 19 jan. 2021.

## 5.3 COLEÇÕES E LAMBDA: MAP E FILTER

Há várias funções nativas da linguagem que são criadas para processar e filtrar listas, sets, mapas, dentre outras coisas. Ao utilizar lambdas, a sintaxe fica muito simples. O código a seguir mostra como transformar todas as strings de uma lista em caixa alta (upper case) e também como filtrar outra lista para apenas objetos pares.

```
fun main() {  
    val nomes = listOf("Steve", "Jobs")  
    // Lista multiplicada por 2  
    val uppercaseList = nomes.map { it.toUpperCase() }  
    println(uppercaseList)  
  
    // Filtrar apenas os pares  
    val ints = listOf(1,2,3,4,5)  
    val pares = ints.filter { it % 2 == 0 }  
    println(pares)  
}
```

### Resultado:

```
[STEVE, JOBS]  
[2, 4]
```

As funções **map** e **filter** são autoexplicativas, verdadeiros clássicos das linguagens modernas. Na sua vida como programador, com certeza vai se deparar muito com elas. Confira mais detalhes na documentação oficial:

- KOTLINLANG. **Documentação oficial**. Disponível em: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/map.html>. Acesso em: 19 jan. 2021.
- KOTLINLANG. **Documentação oficial**. Disponível em: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html>. Acesso em: 19 jan. 2021.
- Recomendamos ler a documentação oficial e explorar as outras funções que podem percorrer e alterar valores de uma coleção: KOTLINLANG. **Documentação oficial**. Disponível em: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/>. Acesso em: 19 jan. 2021.

## 5.4 OBJETOS NULOS (NULL SAFETY)



Este tópico pode ser um pouco complexo de entender, por isso deixamos para o fim. Novamente, orientamos: leia quantas vezes for necessário para assimilar os conceitos; sempre revise quando achar necessário. Essa sintaxe será muito utilizada no dia a dia na programação Android e, é claro, nas próximas aulas. Ainda vamos praticá-la muito.

Vamos lá! Kotlin não permite que variáveis e objetos tenham valores nulos, a não ser que isso seja explicitamente indicado no código. Exemplificando, o código a seguir não compila, pois a variável "nome" não pode ser nula. Isso resolve vários problemas de execução no código com objetos nulos.

```
fun main() {  
    var nome = "Steve"  
    println("Olá $nome")  
    nome = null // Error: Null can not be a value of a non-null  
                type String  
    println("Olá $nome")  
}
```

Para que uma variável possa ter um valor nulo, é obrigatório declarar o seu tipo e utilizar o operador da interrogação '?':

```
fun main() {  
    var nome:String? = "Steve"  
    println("Olá, $nome")  
    nome = null // Agora podemos atribuir nome para null  
    println("Olá, $nome") // Imprime: Olá, null  
}
```

## Resultado:

Olá, Steve

Olá, null

Isso evita muitos erros de programação, pois o fato de saber que uma variável nunca é nula traz segurança no momento de escrever o código. Caso uma variável nula seja acessada de forma indevida, o compilador vai alertá-lo do problema.

Veja outro exemplo: o código a seguir também não compila, pois uma variável nula foi chamada sem a verificação.

```
fun main() {  
    var nome:String? = "Steve"  
    println("Olá $nome")  
    nome = null  
    println("Olá $nome")  
    // Error: Only safe (?.) or non-null asserted (!!.) calls  
    // are allowed on a nullable receiver of type String?  
    println("$nome tem ${nome.length} caracteres")  
}
```

Está entendendo? O compilador verifica o código e já informa em tempo de compilação que isso vai dar problema. Ele ajuda a fazer códigos que vão funcionar.

O erro foi causado porque tentamos imprimir o valor de **`${nome.length}`**, mas o compilador sabe que a variável **`nome`** pode ser nula, e isso pode se tornar um problema. Para que o código compile, é preciso validar se a variável não é nula:

```
fun main() {  
    var nome:String? = "Steve"  
    println("Olá, $nome")  
    nome = null  
    println("Olá, $nome")  
    if(nome != null) {  
        println("$nome tem ${nome.length} caracteres")  
    }  
}
```

### Resultado:

```
Olá, Steve  
Olá, null
```

Veja que não foi impresso o texto dentro do *if()*, pois a variável era nula.

Outra opção que temos para que o código compile é utilizar o operador **?**, chamado de **safe call** (chamada segura). Esse operador vai ignorar a chamada se o objeto for nulo.

Para exemplificar, veja este código:

```
fun main() {  
    var nome:String? = "Steve"  
    println("Olá, $nome")  
    nome = null  
    println("Olá, $nome")  
    println("$nome tem ${nome?.length} caracteres")  
}
```

**Resultado:**

```
Olá, Steve  
Olá, null  
null tem null caracteres
```

Veja que, ao imprimir `${nome?.length}`, o código não travou, pois o Kotlin sabe que o objeto é nulo e não tentou acessar essa referência. Isso é útil ao chamar métodos de objetos, pois, se o objeto for nulo, a chamada será ignorada se o operador `?` (safe call – chamada segura) for utilizado.

Ainda existe outra forma de escrever esse código – e, segundo a documentação do Kotlin, essa é para apaixonados por *NullPointerException*. Claro que isso foi uma brincadeira, portanto, você deve evitar sempre que possível usar a técnica mostrada a seguir. Caso queiramos, por algum motivo, forçar que a referência de um objeto seja acessada, podemos utilizar o operador `!!`. Cuidado! Apenas para ilustrar, o código a seguir lança a exception *KotlinNullPointerException*, pois o operador `!!` foi utilizado de forma irresponsável, resultando num erro em tempo de execução.

```
fun main() {  
    var nome:String? = "Ricardo"  
    println("Olá, $nome")  
    nome = null  
    println("Olá, $nome")  
    println("$nome tem ${nome!!} caracteres")  
    // kotlin.KotlinNullPointerException!!!  
}
```

**Resultado:**

```
Olá, Ricardo  
Olá, null  
Exception in thread "main" kotlin.KotlinNullPointerException  
    at MainKt.main(Main.kt:6)  
    at MainKt.main(Main.kt)
```

Para concluir, devemos evitar ao máximo utilizar a sintaxe **!!**, pois isso pode nos levar aos famosos **NullPointerException**, dos quais tanto queremos fugir. O recomendado é sempre validar se as variáveis não são nulas ou utilizar o operador **?** (safe call – chamada segura) no que for necessário.

Achou o tema complicado? Vamos resumir: se você programar da forma adequada em Kotlin e evitar o uso do operador **!!**, nunca verá a exceção **NullPointerException** e evitará muitos bugs.

Esse é um assunto muito importante, e recomendo que você o estude mais a fundo quando puder. Vale ressaltar que o próprio criador das referências nulas para objetos, Tony Hoare, fez uma palestra chamada “The Billion Dollar Mistake”, na qual ele mesmo pede desculpas por ter criado o conceito de referências nulas durante o desenvolvimento da linguagem Algol em 1965. Diga-se de passagem, Tony Hare é um gênio, grande cientista da computação e inventor do famoso algoritmo de ordenação Quick Sort. Sua maior grandeza foi palestrar em público sobre o seu maior erro e não medir forças para explicar o quanto o acesso a objetos nulos pode ser catastrófico. Segue frase do próprio Tony Hoare, ao falar sobre referências nulas em 2009:

Eu chamo isso de meu erro de bilhões de dólares. Foi a invenção da referência nula em 1965. Naquela época, eu estava projetando o primeiro sistema de tipo abrangente para referências em uma linguagem orientada a objetos (Algol W). Meu objetivo era garantir que todo uso de referências fosse absolutamente seguro, com a verificação realizada automaticamente pelo compilador. Mas não pude resistir à tentação de colocar uma referência nula, simplesmente porque era tão fácil de implementar. Isso levou a inúmeros erros, vulnerabilidades e falhas do sistema, que provavelmente causaram bilhões de dólares de dor e danos nos últimos quarenta anos. (QCon London, 2009, tradução nossa)

### Saiba mais

Para ler o texto original em inglês e mais detalhes sobre a palestra, acesse: QCon London.

**Tony Hoare.** Disponível em: <https://qconlondon.com/london-2009/qconlondon.com/london-2009>

[9/speaker/Tony+Hoare.html](https://speaker/Tony+Hoare.html)>. Acesso em: 21 jan. 2021.

INFO Q. **Null References:** The billion dollar mistake. Disponível em: <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>>. Acesso em: 21 jan. 2021.

Para maiores detalhes sobre o tratamento de tipos nulos no Kotlin, leia a documentação oficial: KOTLINLANG. **Documentação oficial.** Disponível em: <https://kotlinlang.org/docs/reference/null-safety.html>>. Acesso em: 21 jan. 2021.

## FINALIZANDO

Nesta aula, aprendemos a sintaxe da linguagem Kotlin. Por meio de exemplos práticos, vimos como declarar variáveis, funções, classes e vários recursos.

Até a próxima!

## REFERÊNCIAS

KOTLINLANG. Disponível em: <https://kotlinlang.org/>. Acesso em: 19 jan. 2021.

KOTLIN PLAYGROUND. Disponível em: <https://play.kotlinlang.org/>. Acesso em: 19 jan. 2021.

QCON LONDON. **Tony Hoare.** 2009. Disponível em: <https://qconlondon.com/london-2009/qconlondon.com/london-2009/speaker/Tony+Hoare.html>>. Acesso em: 21 jan. 2021

