DESENVOLVIMENTO AVANÇADO EM JAVA

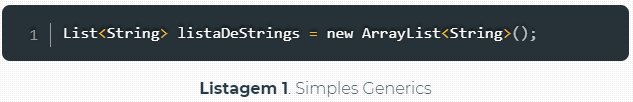
PARADIGMA FUNCIONAL JAVA

Antes de tudo é necessário entender os conceitos de Java Generics e Collections: List e Set.

JAVA GENERICS

O conceito foi introduzido na versão Java SE 5.0 e trouxe consigo mais robustez, segurança e qualidade aos softwares desenvolvidos com este recurso.

O generics é delimitado pelos caracteres “<>”, ou seja, quando houver esse par de caracteres em uma parte qualquer do código, significa que o Generics está sendo utilizado.



Vamos entender o que acontece na **Listagem 1**: o uso do Generics “” na “**[List](http://www.devmedia.com.br/java-collections-set-list-e-iterator/29637" \o "List" \t "_blank)**” determina e obriga que apenas objetos do tipo String podem ser adicionados nesta lista, chamada de “listaDeStrings”. Significa que o código da **Listagem 2** irá apresentar um erro em tempo de design.



O erro mostrado ocorre porque não foi encontrado um método com a seguinte assinatura “add(int)” e o compilador tentou usar o método “add(int,string)”, que obviamente não irá aceitar os argumentos passados.

Com a adição do Generics no Java 5, todas as Collections foram reescritas para uso do Generics e consequentemente evitar o famoso “ClassCastException” (ocorre quando tentamos fazer um cast (coerção - conversão explícita) de uma classe para outra classe diferente desta ou que não seja uma superclasse desta) que ocorria com mais frequência do que queiramos antes de tal recurso existir.

O Generics não é obrigatório, mas é ideal para garantir o type-safety, que em outras palavras irá garantir que determinada collection só tenha o tipo de objetos que você deseja, sem necessidade de casting excessivo.

Se você tentar adicionar qualquer objeto que não seja do tipo definido pelo Generics em tempo de design, automaticamente o compilador já irá apontar o erro e evitar que a aplicação seja iniciada.

Sendo assim, nota-se que o Generics traz robustez, segurança e qualidade:

* Robustez porque podemos tornar nosso software muito mais abstrato e reflexível, adaptável a várias situações;
* Segurança porque garantimos que nenhum tipo de dado que não desejamos será inserido ou usado em nossa Classe;
* Qualidade porque diminuímos consideravelmente a quantidade de erros que nosso código teria (ClassCastException), menos type casting e mais simplicidade no código.

COLLECTIONS: LIST E SET

List é uma sequência ordenada de elementos, ao passo que Seté uma lista distinta de elementos que não é ordenada.

**List<E>:**

Uma coleção ordenada (também conhecida como sequência). O usuário desta interface tem controle preciso sobre onde na lista cada elemento é inserido. O usuário pode acessar elementos por seu índice inteiro (posição na lista) e procurar elementos na lista.

**Set<E>:**

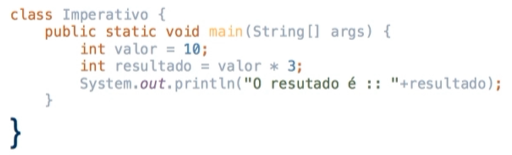
Uma coleção que não contém elementos duplicados. Mas formalmente, os conjuntos não contêm par de elementos e1 e e2, de modo que e1.equals(e2) e, no máximo, um elemento nulo. Como está implícito no nome, essa interface modela a abstração do conjunto matemático.

PARADIGMA FUNCIONAL NO JAVA

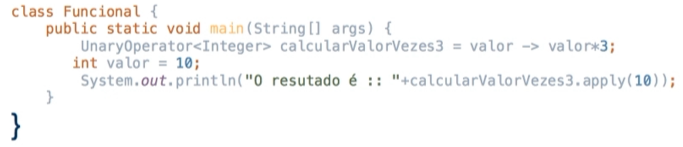
Programação funcional é o processo de construir software através de composição de funções puras, evitando compartilhamento de estados, dados mutáveis e efeitos colaterais. É declarativa ao invés de imperativa, essa é uma definição de Eric Elliott.

A linguagem Java respeita um conceito imperativo, ou seja, ela respeita ordens, portanto todas as vezes que declaramos uma linha estamos dando uma ordem para a JVM processar aquilo.

O paradigma imperativo é aquele que expressa o código através de comandos ao computador, nele é possível ter controle de estado dos objetos. Linguagens como C, Java, .NET, entre outras, em suas primeiras versões eles obedeciam a esse paradigma imperativo, em todas as instruções passadas por essas linguagens eram interpretadas como ordens dadas ao computados, por isso é chamado de imperativo, porque em toda linha era dada uma ordem. Ele funciona como um script de ordens onde o computador vai obedecendo linha por linha daquele script. No momento que estamos passando o comando ele está sendo executado.

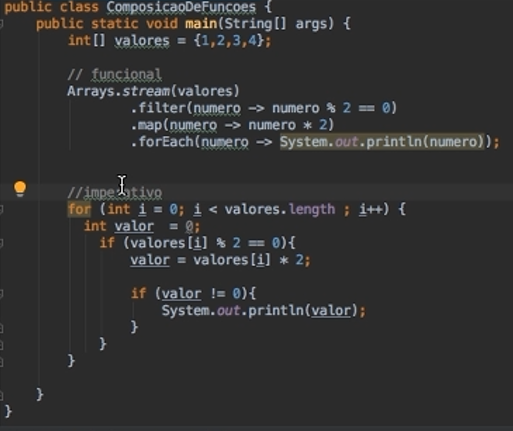


No paradigma funcional damos uma regra, uma declaração de como queremos que o programa se comporte. Criamos então uma função que ao receber um parâmetro faça alguma coisa.



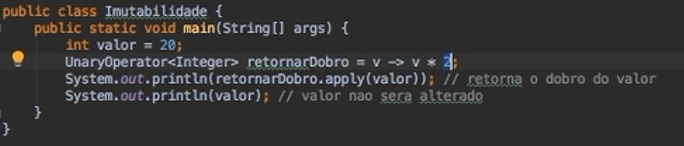
CONCEITOS FUNDAMENTAIS DA PROGRAMAÇÃO FUNCIONAL

Composição de funções é criar uma nova função através da composição de outras. Por exemplo, vamos criar uma função que vai filtrar um array, filtrando somente os números pares e multiplicando por dois:



Funções Puras: Uma função **pura** é aquela que não provoca efeitos colaterais, ou seja, ela não muda qualquer estado na aplicação. Mas não é só isso, ela precisa sempre gerar o mesmo resultado com os mesmos argumentos, ou seja, ela precisa ser completamente determinística. Tendo F(x) = x + 2, temos então que se x for 2 o resultado de f(x) sempre será 4 e assim por diante. Então uma função pura deve retornar sempre os mesmos resultado aos receber os mesmos parâmetros.

Imutabilidade: significa que uma vez que uma variável recebeu um valor, vai possuir esse valor para sempre, ou quando criamos um objeto ele não pode ser modificado.



IMPERATIVO x DECLARATIVO

É comum aprender programação de forma imperativa, onde mandamos alguém fazer algo. Busque o usuário 15 no BD, valide essas informações do usuário, etc.

Na programação funcional tentamos programar de forma declarativa, onde declaramos o que desejamos, sem explicitar como será feito. Qual o usuário 15? Quais os erros dessas informações?

LAMBDA

Os lambdas obedecem ao conceito do paradigma funcional, com eles podemos facilitar a legibilidade do nosso código além disso com a nova API Funcional do Java podemos ter uma alta produtividade para lidar com objetos. Primeiramente, devemos entender o que são interfaces funcionais.

Interfaces funcionais: são interfaces que possuem apenas um método abstrato. Exemplo:

Geralmente possuem uma anotação em nível de classe (@FunctionalInterface), para forçar o compilador a apontar um erro se a interface não estiver de acordo com as regras de uma interface funcional. Esta anotação não é obrigatória, pois o compilador consegue reconhecer uma interface em tempo de compilação.

Antes do Java 8, se quiséssemos implementar um comportamento especifico para uma única classe deveríamos utilizar uma classe anônima para implementar este comportamento.

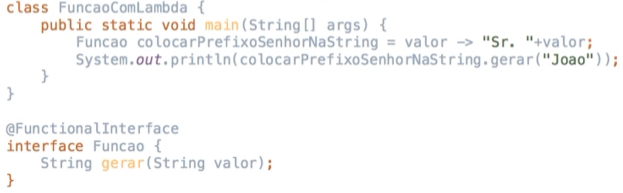


Código muito verboso.

Agora que sabemos como se define uma interface funcional, podemos aprender como se define uma lambda.

Estrutura de um lambda:

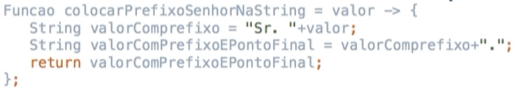
InterfaceFuncional nomeVariavel = parâmetro -> logica



Tornamos o mesmo código, antes verboso, mais simples.

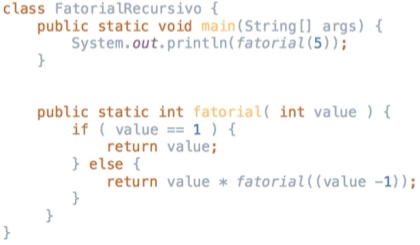
Quando um lambda possui apenas uma instrução no corpo de sua lógica não é necessário o uso de chaves, como pode-se ver no exemplo acima. Ele entende que deve retornar aquele valor.

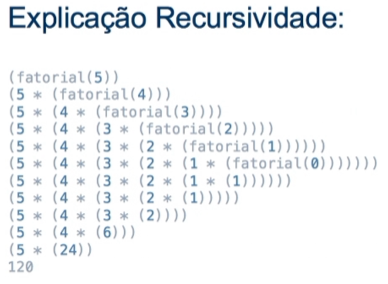
Se a função possui mais de uma instrução devemos utilizar chaves e além disso deve explicitar o retorno se o retorno for diferente de void. Exemplo:



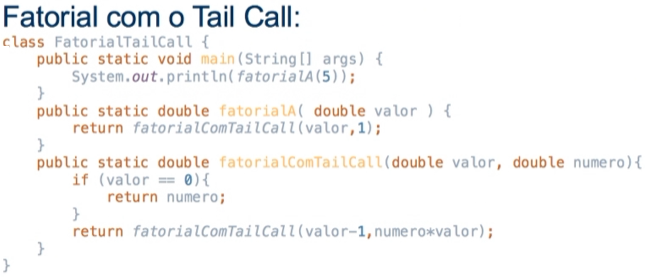
RECURSIVIDADE

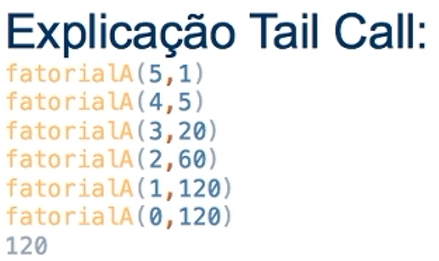
Uma função chama a si mesma repetidamente, até atingir uma condição de parada. No caso de Java, um método chama a si mesmo, passando para si objetos primitivos. Cada chamada gera uma nova entrada na pilha de execução, e alguns dados podem ser disponibilizados em um escopo global ou local, através de parâmetros em um escopo global ou local.

 Recursividade tem um papel importante em programação funcional, facilitando que evitemos estados mutáveis e mantenhamos nosso programa mais declarativo, e menos imperativo.



Tail Call (Recursividade em cauda): recursão em cauda é uma recursão onde não há nenhuma linha de código após a chamada do próprio método e, sendo assim, não há nenhum tipo de processamento a ser feito após a chamada recursiva.

Obs: a JVM não suporta recursão em cauda, ele lança um estouro de pilha (StackOverFlow).

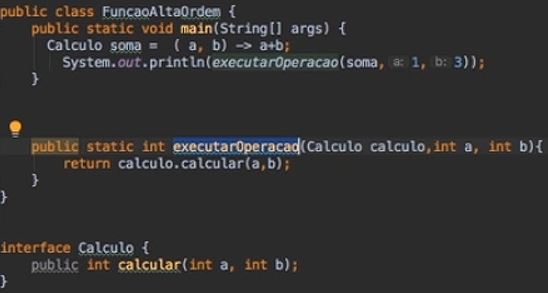


Memoization: é uma técnica de otimização que consiste no cache do resultado de uma função, baseado nos parâmetros de entrada. Com isso conseguimos ter uma resposta mais rápida.

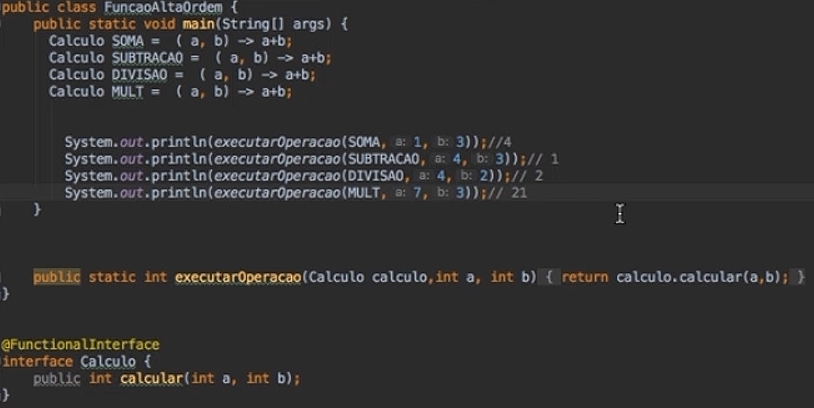
INTERFACES FUNCIONAIS

São interfaces que possuem apenas um método abstrato para estar respeitando as regras para estar implementando um lambda.

FUNÇÕES DE ALTA ORDEM

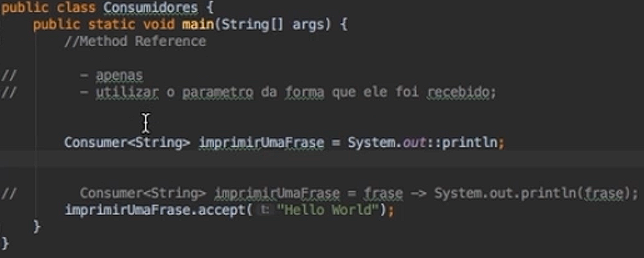


A função executarOperacao funciona como uma função de alta ordem por ser uma função/método que recebe outra função como parâmetro.

Portanto a definição de função de alta ordem é uma função que retorna uma função ou que recebe uma função como parâmetro. Em Java é mais comum receber uma função do que retornar uma função.

Com funções de alta ordem passamos apenas a implementação dele e os valores, pois ele já sabe como se comportar. Isso facilita a manutenção de código. Chamamos o executarOperacao, dizemos qual será o lambda a ser implementado (no caso SOMA, SUBTRACAO, DIVISAO e MULT), passamos os valores, e o método se encarrega de executar a operação de acordo com o que foi chamada.

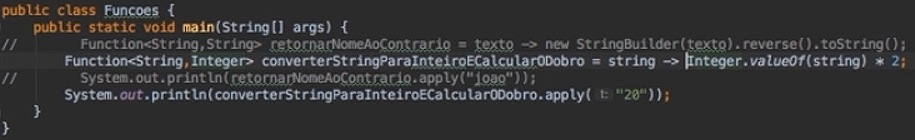
CONSUMER INTERFACE



Com o método CONSUMER podemos tanto passar a função que queremos que o método execute quanto um parâmetro e uma função o implementando como mostrado acima.

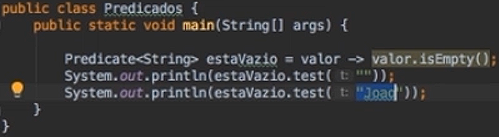
FUNCTIONS

Elas recebem um parâmetro e devem retornar um parâmetro também. Em seu generics deve ser passado <dadoRecebido, dadoResposta>. Por exemplo na imagem abaixo recebemos um String em uma das funções, mas o retorno é um Integer, portando deve ser passado Function<String,Integer>.



PREDICATE

Eles recebem um parâmetro qualquer e retornar um boolean.



Observações quanto a formas de implementação do lambda: quando tratamos diretamente com o parâmetro, lidando diretamente com o objeto recebido, ou seja, quando não precisamos fazer nada com o que foi recebido, apenas passa-lo, podemos utilizar o modelo apresentado no CONSUMER e no PREDICATE de passarmos diretamente a função. Ex.: System.out::println, String::isEmpty.

SUPPLIERS

São supridores, eles não recebem parâmetro e retornam o que foi especificado no generics.

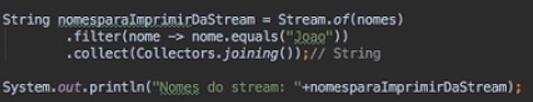


ITERAÇÕES

ANTES

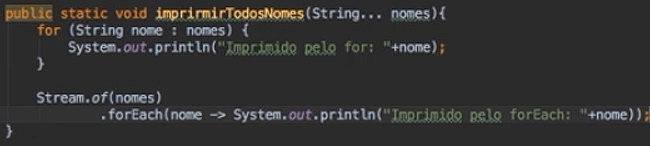


DEPOIS

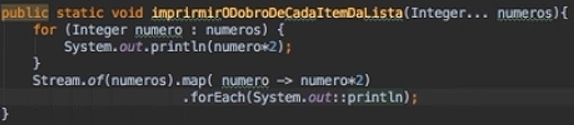


Ele irá concatenar as Strings toda vez que o nome “Joao” aparecer, caso aparecessem duas vezes no array retornaria JoaoJoao.

Outras formas de utilizar o Stream.of



O mesmo pode ser utilizado quando queremos dobrar os valores de um vetor, por exemplo. Antes utilizávamos o for para isso, após a chegada da API podemos usar o Stream.of.



Em questões de performance o Stream.of pode apresentar um leve atraso em comparação ao for, portanto ao focar em performance, para obter um sistema mais performático, a utilização do for é mais indicada.