UTFPR – Cornélio Procópio – Espec. Java – Professor Arthur Pereira Gregório

Aluno: Elvir Moraes

Atividade Web Conferência 01

TAREFA NO CASO DA NÃO PARTICIPAÇÃO DA WEBCONFERÊNCIA:

1) Assistir a gravação da Web Conferência 1 e fazer um relato de forma textual dos conteúdos e atividades ministradas durante a conferência. Entregar um documento PDF.

INTRODUÇÃO GENERICS

O Professor Arthur Gregório, iniciou a aula com uma problemática no desenvolvimento do código, de um sistema bancário, para demonstrar o princípio de Generics, falou do fato de a herança (extends) não ser bom.

Herdar ou Implentar ??? “Sempre dependa de uma composição, muito melhor que usar herança”

“Ter cuidado com getters e setters, não se deve ter setSaldo(), para ter saldo, deve-se depositar( )”, para não quebrar o código, pois em Java existe o Reflection que se pode manipular qualquer coisa, mas este conteúdo ficará pra uma próxima aula.

Obs.: O Eclipse IDE usa instrumentação via

Reflection(um pouco mais lento), diferente do NetBeans que usa instrumentação via JDK.

Problemática:

Caixa eletrônico que consiga manipular uma conta bancária;

Com o código inicialmente apresentado, era necessária uma agência bancária ter caixas eletrônicos para diferentes tipos de conta; Conta Universitária e Conta Premium, por exemplo.

Nesse momento o professor falou da inferência da IDE sabendo-se o tipo, evitar castings desnecessários.

A evolução dos conceitos apresentados pelo professor evoluía com a evolução do código, das classes, dos métodos de sacar() e depositar().

O professor falou dos erros comuns de programadores, que copiam e colam métodos e só alteram o nome e commitam para a Branch Master do projeto, onde se vê solução aplicada pelo programador, utilizada de forma errada, tendo problemas futuros.

O famoso funciona mais não está certo. Em seis meses pode surgir um problema no código, por quaisquer razões e motivos, tal por exemplo, a startup do programador pode ser vendida, o sistema exigia caixas eletrônicos para tipos de contas diferentes, a maioria dos clientes que frequentavam uma determinada agência tinham somente um tipo de conta, e como resultado, somente um dos caixas estava sempre cheio e outro completamente vazio.

Onde surgiu a solução do código do sistema, deveria se tornar mais genérico.

Uma breve história da popularização do Java através de um erro, o Java ME popularizou o Java, com referências na Java Magazine, a história conta que foi um erro de curso que vingou, que criou visibilidade para a plataforma Java, que trazia a computação em cima de uma máquina virtual, a JVM.

Escreva uma vez e execute de qualquer lugar. Até onde era possível o Java ME ser usado, até ser descontinuado, e o legado deixado com a evolução do Java.

As melhorias do Java com o Generics, do Java 1.4 ao 1.5

O problema da herança e a melhoria com o Generics,

O var é uma feature que foi introduzida no Java 11, que permite que meus tipos sejam detectados em Runtime.

Aplicar Generics para um código limpo como solução de flexibilidade na programação do código.

O processo de ERASURE, o processo da compilação do código, e a inferência/substituição de letras convencionadas em operações diamante, por exemplo <T> e <K>, aplicada em tempo de compilação, na substituição do Generics pelo Type real do objeto.

Fazendo-se:

Desnecessário <Type Parameter> na declaração diamante do tipo da classe, pois pode ser usado <T> para um código limpo e genérico. Existe uma convenção de uso de letras reservadas por tipo, todos serão trocados pela inferência no ERASURE:

T = type

K = key

V = value

E = element

I = index

Dentro da mecânica do código de ArrayList, tem um objeto, Array de Object[ ], transiente, elementData, que é feito um casting por debaixo dos panos para o tipo do objeto, flexibilizando o uso Generics para praticamente qualquer objeto, onde também é possível manter uma compatibilidade com as versões anteriores de Java.

O código genérico torna simples ser reutilizado, resolvendo o problema do caixa eletrônico pra cada tipo de conta, mas a declaração <Type TIPADA> de um objeto <ContaBancaria>, <ContaPremium>, ..., ainda não uso o recurso do processo de ERASURE, a leitura do código se torna excessiva e não aproveita todo o recurso do uso do Generecs, não só do ArrayList mais em diversos tipos de projetos. O Generics é o tipo real.

List usa Generics também, e a não declaração diamante <> numa lista de genéricos, deixa aberto a inserção de elementos nessa lista de tipos diferentes como add(1), add("tres"), está usando rawtype ao invés de Generics:

final List números = new ArrayList();

é traduzido TIPADO pelo ERASURE como:

final List<Object> números = new ArrayList<>();

É possível passar <Object> para qualquer tipo de objeto, e por <TYPE Parametrizado> para um tipo definido de objeto como o <Integer> por exemplo definindo o tipo da lista.

Passado o conceito de auto boxing inserido no Java, para as formas de inserção de elementos em uma lista, add(1), add(new Integer(2))

// PECS

// producer extends / consumer super

Outro conceito para utilização de Generics é o WildCards,

Final List<? Extends Number> producer = new ArrayList<>();

Final List<? Super Number> consumer = new ArrayList<>();

Onde numa lista <? Extends Number> producer não consigo adicionar e numa lista <? Super Number> consumer consigo add(1)

Posso fazer em um mas no outro não porque extends e super?

Se eu não consigo por nada aí dentro, como vou usar isso? Baixar o nível de herança, fazendo cast.

```
Consumer.forEach(numero -> {
        If (numero instanceof BigDecimal decimal) {
                System.out.println(decimal);
        } else {
                System.out.println("é outra coisa");
        }
})
```

Tem um projeto chamado Project coiin que contribui para pequenas melhorias na linguagem e uma delas é o pattern math. Aproveitando a instanciação do instanceof vinda na feature de alguma versão nova Java, talvez Java 19.

O Question mark < ? > é uma palavra reservada para fazer um WildCard / raw type.

A INFERÊNCIA DO VAR EM SUA ATRIBUIÇÃO

Por inferência: var inteiro = 1;

é do tipo inteiro por ter recebido um valor numérico inteiro, o compilador vai inferir o tipo através do processo de ERASURE, onde não torna possível reatribuição de outro tipo para o mesmo objeto.

A verbosidade foi removida pela inferência um lado inferi o tipo pelo outro lado, o tipo definido da variável inteiros.

Final List<Integer> inteiros = new ArrayList<Integer>();

Por inferência o tipo de uma lista é definido pelo valor que estou adicionando. Add(1) podendo ser usado o var.

Se eu não adicionei nada o tipo é object.

Final var inteiros = new ArrayList<>();

Se adicionei o tipo é do valor inserido:

Final var inteiros2 = List.of(1,2,3);

A inferência de tipos também acontece pelos valores.

DEMONSTRANDO UMA FACTORY COM GENERICS

Neste exemplo foi criado um sistema para fábrica de bolos, com métodos para fazer um bolo.

-Apresentado o Type arps que são os três pontinhos <>... que é usado para passar por parâmetro, se tornando um Array

Falado sobre Optional: O Optional pode ser usado bastante no dia a dia da programação, o professor por exemplo, usa muito Optional.

Um exemplo de uso do Optional é poder retornar um objeto opcional após ser tratado com try catch.

-Sintaxe sugar, onde se tem estruturas de decisão prontas para uso, para tornar o código do programador mais limpo e expressivo em uma leitura semântica, evitando verbosidades.

```
Public final class fabricadeDeBolos {
        Private FabricaDeBolos() { }
        Public static <T extends Bolo> Optional<T> fazerBolo(Class<T> tipoDeBolo){
                Try{
                        Return
        Optional.of(tipoDeBolo.getDeclaredConstructor().newInstance());
                } catch(Exception ex){
                        ...
                        return Optional.empty();
                }
        }
}
```

-Apresentado record: Que é a forma mais sintaxe sugar para se criar uma classe imutável, sem precisar declarar os métodos de equals, hashCode e toString como até antes do Java 17.

Dado o exemplo de Strings serem imutáveis. Pelo de terem os métodos para tornar um objeto imutável.

Ter atenção ao usar, ou tirar o @Data para persistência, sem implementar o hashCode, equals, ... da maneira correta.

Pois os objetos podem ser tratados todos como iguais, sendo diferentes, mutáveis.

Ou usar o lambock.

Recomendado pelo professor não usar record com JPA, pois JPA não foi concebido para ser utilizada com objetos imutáveis. Talvez funcione, de alguma maneira, mas não deveria. Funciona mais não está certo. Refletir sobre isso.

Todo material citado na vídeo aula, é material de base.