# Java Building

- Capa
- Arquitetura
- Boas Práticas
- Carreira
- Desenvolvimento
- Design
- Planejamento
- Scrum

## Variância

Apr/15
6
3 Commentários | Arquivado em : Desenvolvimento

O conceito de variância aparece quando pensamos em classes genéricas. A variância se relaciona a como a herança de uma classe genérica se coordena com a herança da classe secundária.
Se tivermos uma classe B que herda de A o que podemos dizer sobre a relação de uma classe genérica G<A> e G<B> **?**

Quando falamos de herança neste contexto estamos falando de assignação válida.  O que podemos dizer sobre a validade das seguintes assignações:

```
1  G<A> a = new G<B> ();
2  G<B> b = new G<A> ();
```

A primeira expressão significa que se Todo o B é um A, então Todo o G<B> é um G<A>. A segunda expressão significa que se Todo o B é um A, então Todo o G<A> é um G<B>. Repare que na primeira expressão estamos acompanhando a mesma regra da herança que já existe entre A e  B. No segundo caso estamos seguindo a regra inversa. Apenas uma destas opções pode ser verdadeira para um dado tipo G, ou, nenhuma delas. Em nenhum caso podem ser as duas.

Se a primeira expressão for válida dizemos que G é **covariante**. Se a segunda expressão for válida dizemos que G é **contravariante**.  Se nenhuma das duas é válida, dizemos que G é **invariante**.

Em java a variância não é muito falada porque todas as classes genéricas são , por definição, invariantes. Pode parecer que isto é o mesmo que dizer que em java não existe variância. Bom, isto não é exatamente correto, porque existe um tipo de objeto que é covariante, só que não é genérico: o array.

Em java , arrays são entidades primitivas como int ou double, mas que têm uma estrutura. Esta estrutura os assemelha a objetos em todos os sentidos, excepto na forma de criação. Array em java não são classes genéricas Array<T>, são entidades especiais que o compilador entende e as quais podem ser criadas com o operador *new* e

inclusive têm propriedades , como *length*. Arrays não implementam interfaces e você não pode herdar de um array, Não ha realmente uma classe pública que os representa; são completamente controlados de forma privada pelo compilador e a JVM.

Contudo, tradicionalmente a especificação java sempre primitiu que um array de uma classe fosse assignado a um array da classe mãe como em:

```
1 Object[] objects = new String[2];
```

Isto seguindo a lógica que se uma String é um Objeto, um array de strings é um array de objetos. Esta regra é aceite pela especificação embora não seja formalmente correta. A especificação sabe disto e, portanto, define que se um objeto do tipo errado for colocado no array uma exceção será lançada. Mas como a JVM sabe que o tipo é o errado ? Na realidade não sabe. Quem sabe é a implementação do array que memoriza a classe subjacente que deve ser usada. Os arrays em java, desde a primeira versão. são um tipo reificado, ou seja, o array sabe a classe do objeto dentro dele. O código acima, poderia ser conceptualmente entendido como :

```
1 Array<Object> objects = new Array<String>(2, String.class); // conceptualme
```

De forma que o objeto tem informação sobre qual o tipo de objetos que será colocado dentro dele. É por isso que se executarmos o seguinte código, obteremos uma exceção

```
1 Object[] objects = new String[2];
2
3 objects[0] = "Olá";
4
5 objects[1] = Integer.valueOf(2);
```

Embora um array de objetos possa receber um objeto do tipo Integer , a implementação subjacente só aceita Strings, logo um erro é lançado. Porque existe este mecanismo inerente ao array em java, o array pode ser utilizado na sua forma covariante, embora, formalmente, seja um tipo invariante. Repare que um verdadeiro objeto verdadeiramente covariante não irá lançar exceções. O que acontece é que o array se comporta como um objeto covariante e para os casos em que usamos esta propriedade da especificação, está bom demais.

**Variância e Reificação**

A reificação, como falei, é a capacidade do objeto saber quais os tipos genéricos reais que foram usados no momento da construção do objeto. Por exemplo ou criar um array de string o objeto array contém a informação que o tipo subjacente a ele é String. Um array de int, contém a informação que o tipo subjacente é int, e assim por diante…. Em java, todos os tipos genéricos sofrem *erasure*, que é o processo inverso de reificação: nenhuma informação dos tipos é guardada. Array não é uma classe genérica e por isso não cai nesta regra sendo o único tipo reificado em java (desde a versão original nos anos 90), todos os outros tipos genéricos introduzidos no java 5 e depois, sofrem erasure.

Existe interesse em incluir alguma forma de reificação no futuro, e já existem propostas de como seria possivel. Um exemplo de como poderia ser pode ser visto aqui e num *paper* aqui.

A Reificação , quando existe, pode ajudar a controlar a variância através do lançamento de exceções. Este é um tipo de variância por comportamento em vez de por design, mas que é boa o suficiente em muitos casos. O objeto não têm uma variância definida, ele apenas se comporta como se tivesse. É por isto que linguagens mais modernas se esforçam em ser reificadas e não perder a informação de tipos genéricos subjacentes em tempo de execução como faz o java. Sendo a linguagem reificada e permitindo definir a variância desejada o compilador pode fazer o resto do trabalho e das inferências. Se algo der errado o próprio objeto está protegido internamente porque ele conhece os tipos em causa.

**Declarando a Variância**

Como falei, em java todos os tipos genéricos são invariantes e não ha nada que possamos fazer por enquanto. No c# o mecanismo ainda é limitado e apenas é possível ditar a variância de interfaces genéricas ( não em classes). Em scala, kotlin e ceylon qualquer tipo genérico pode ser anotado com instruções de variância. A forma para isto depende de cada linguagem. Vou adotar aqui a sintaxe do C# e do Kotlin que acho mais intuitiva.

Duas palavras reservadas, **in** e **out**, podem ser usadas para anotar o tipo genérico como no exemplo a seguir

```
1 public interface Association< in K, out V>
```

O **in** representa um tipo contravariante  e o **out** um tipo covariante. Se o tipo é invariante, nenhuma anotação é colocada. Esta nomenclatura é bem simples porque ,como veremos a seguir, o tipo de variância está relacionado como a posição em que o tipo genérico é usado : se em pontos de retorno (out) então é covariante, e se em pontos de entrada (in), como argumentos de métodos,  é contravariante.

**Variância e Mutabilidade**

Ao dizermos que um determinado objeto do tipo G<T> é covariante ou contravariante estamos implicitamente estipulando que tipo de operações podem ser feitas em G que tenham como parâmetros o tipo genérico, T. O tipo do objeto é covariante apenas se T aparece em posições de retorno e nenhum operação tem T como argumento. O tipo é contravariante se tem apenas operações em que  T está nos argumentos dos métodos e nunca no retorno.  Se T aparece em posições de retorno e também em posições de argumento para o mesmo tipo, o tipo tem que ser invariante. Construtores não são métodos e portanto podem receber argumentos de T mesmo quando o tipo é covariante.

Arrays em java sempre foram entendidos como objetos covariantes. Em rigor, ele teria que ser invariante pois a operação de escrita array[i] é também a operação de leitura e ambas têm T como parâmetro. Portanto, como T aparece simultaneamente como retorno de um método e parâmetro de um método, o tipo tem que ser invariante. É exatamente por isto que todas as coleções em java  são invariantes e o código seguinte não compila, qualquer que seja a classe de coleção usada, já que todas as classes contém métodos que retornam T (como get) e aceitam T (como add)

```
1 List<String> strings = ... // inicializado de alguma forma válida.
2 List<Object> objetcs = strings; // esta linha não compila.
```

Vimos que a covariância está relacionada a interfaces e a operações só de retorno.Peguemos agora como exemplo  a interface Comparator<T>. Esta interface tem a seguinte assinatura:

```
1 public interface Comparator<T> {
2
3 public int compare(T a, T b);
4
5 }
```

O único método de Comparator apenas recebe T e nunca o retorna.  Isto tonaria possível que Comparator seja contravariante (apenas posições *in*) e podemos escrever

```
1 Comparator<Object> objectsCompator = ... // inicializado de alguma forma vá
2
3 Comparator<String> stringComparator = objectsCompator; // repare que a assi
```

A ideia por detrás é: se o comparador compara quaisquer objetos com um certa regra, ele poderá comparar também Strings da mesma forma pois toda a String é um Object. Este código é conceptualmente válido, mas impossível de escrever em java porque em java todas as classes genéricas são invariantes. Mas seria perfeitamente possível em C#(com a interface homologa : IComparer) , ou Kotlin ( com aquela mesma sintaxe que usei), por exemplo.

Vemos aqui claramente que a linguagem java não permite informar todas as propriedades de um tipo. Isto é devido, em parte, à API de coleções ter que continuar válida, e em parte à escolha de variância em local de invocação em vez de variância como propriedade dos tipos genéricos da classe. Apenas recentemente com o java 8 se tornou  possível colocar anotações em todos os pontos onde existem definições de tipos genéricos, e esta funcionalidade era necessária para incluir o conceito em Java. Em C# com a modificação da própria linguagem isto não foi um problema. Também não é problema para as linguagens mais modernas que incluem o conceito de raiz como Kotlin ( que inclusive usa a mesma API que Java).

A variância ainda é uma propriedade relativamente recente e apenas quem tem contato com linguagens mais moderna a vê em operação na sua forma mais pura. Contudo sempre que você tem que usar uma expressão de tipos genéricos em Java , você está fazendo uso de variância, só que declarada no ponto da invocação (*user-site variance*), em vez de a declarar na definição da classe, por exemplo:

```
1 public <T> List<T> someMethod  (List<? extends T> original)
```

Isto significa "*aceite um List de qualquer tipo que seja T ou filho de T*". O código a seguir só aceita exatamente T e não os filhos.

```
1 public <T> List<T> someMethod  (List<T> original)
```

É preciso dizer que, a variância apenas definida na classe não resolve todos os casos de uso, e as linguagens mais tarde ou mais cedo têm que permitir também variância em ponto de invocação., sobretudo em caso, como no exemplo, em que a classe é invariante mas sabemos pela lógica dos tipos que a transformação é válida. Desta feita, a escolha do java parece mais complicada  mas é porque é a opção mais genérica que resolve todos os casos de uso sem ter que recorrer a reescrever todas as classes que existem ( que foi a opção seguida pelo C#).

A declaração da variância ajuda o compilador a realizar menos operações de cast e ao mesmo tempo o compilador ajuda o programador a verificar as regras de declaração, por exemplo, se declarou um tipo genérico como out e depois o usou como argumento em algum método. Compiladores e linguagens que se utilizam do conceito de variância são mais seguros porque representam um passo além da tipagem forte para uma tipagem ainda mais forte. É por isto que todas as linguagens modernas a incorporam, e a razão de porque elas parecem mais elegantes.

Finalmente vale referir que quando se usa o conceito de variância o design acaba favorecendo o Principio de Segregação de Interfaces e por consequência os métodos e tipos se tornam mais imutáveis o que é bom para promover otimizações, especialmente em ambiente multi-thread. Este é um elemento de design comum a todas as API das novas linguagens que deriva deste único conceito de variância.

« Java 8 – Prólogo
Streams no Java 8 e em outras Linguagens »

## 3 comentários para "Variância"

1. **Douglas Arantes**, em April 7th, 2015 às 12:57 disse:

   Muito bom o post Sérgio.

   Existem outros projetos interessantes em investigação, como:

   State of the Specialization: http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html

   Value Types: http://cr.openjdk.java.net/~jrose/values/values-0.html

2. **sergiotaborda**, em April 7th, 2015 às 13:31 disse:

Realmente são tempos interessantes para a JVM. Ao fazer o MiddleHeaven senti bastante falta destes pontos que referes ( Value Types e Genéricos para Primitivos). Isso influencia a performance o ao seguir um design coerente, ela acaba sendo deixada para trás pois há que escolher e não ha nada que o programador possa fazer para otimizar as coisas. Só espero que não façam as mesmas asneiras que o .NET fez com structs para os Value Types. Pelo que já li das propostas um Value Type irá ter construtores e todo o modelo que uma classe e interface têm (embora por detrás o compilador e a jvm façam magia). No .NET structs não têm como controlar os valores iniciais dentro delas o que leva a erros que o programador não pode evitar. Por exemplo, um struct Fraction pode ter numerador e denominador zero (0/0) o que é um erro. A adição de Value Types pode abrir a porta para Tuplas e melhor performance para cálculos numéricos e tratamento de arrays pode fazer o java realmente tomar conta do campo cientifico onde hoje o C e o C++ ainda são usados com relevância ( e o python oferece uma porta para a chamada a lib em C e C++ que são base para o fortran). Com melhor performance e mais tipos numéricos, java pode substituir o uso de fortran, python e todas essas linguagens e/ou fornecer uma VM onde essas linguagens funcionem. Tempo realmente interesantes, mal posso esperar…

3. **Douglas Arantes**, em April 8th, 2015 às 13:20 disse:

   Parece mesmo que o objetivo da Oracle é fornecer algo bem superior aos Structs do .NET, e abrir portas para tuplas, e tipos numéricos. O legal é que outras linguagem como Scala podem implementar tuplas eficientes.

   Pena que isso vai demorar para chegar, mas se chegar no Java 10, já está ótimo.

   JEP 169: http://openjdk.java.net/jeps/169

   A JEP 128(Generics over Primitive Types), está marcada para o Java 10.

   Existe uma série de 3 posts, mostrando os resultados atuais do project valhalla.

   http://nosoftskills.com/2015/02/primitives-in-generics-part-1/

## Comente

O seu nome - obrigatório

O seu email (não será publicado) - obrigatório

O seu site

Enviar

## JavaBuilding

- Academia

- Arquitetura
- Design Patterns
- Livros
- Oficina
- Principios

## Tags

agil arquitetura boas práticas Camadas carreira conceitos contrato decorator design design pattern Design Patterns diretivas escolhas gerencia ideia java liderança linguagens mercado mitos monad MVC más práticas opinião pacotes plano plataforma portabilidade principios processo produto programação qualidade risco scrum tecnologia tendência valores

## Artigos

Select Month

```
1
```