Genéricos

Genéricos

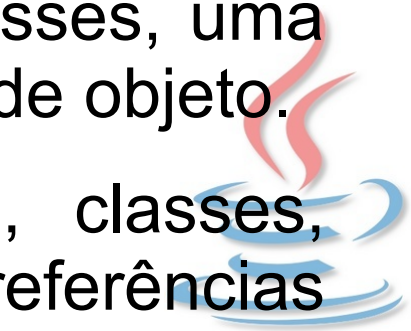
- Desde sua versão original, muitos recursos novos foram adicionados a Java. Todos melhoraram e expandiram seu escopo, mas o que teve impacto mais profundo é o tipo genérico, porque seus efeitos foram sentidos em toda a linguagem.
- Não é exagero dizer que sua inclusão basicamente reformulou a natureza Java.
- À primeira vista, a sintaxe dos genéricos pode parecer um pouco complicada, mas não se preocupe, os genéricos são muito fáceis de usar. Você terá uma noção dos conceitos-chave que estão por trás dos genéricos e terá conhecimento suficiente para usá-los de maneira eficaz em seus próprios programas.

Fundamentos

- Na verdade, com o termo genéricos queremos nos referir aos tipos parametrizados.
- Os tipos parametrizados são importantes porque nos permitem criar classes, interfaces e método sem que o tipo de dado usado seja especificado como parâmetro.
- Uma classe, interface ou método que opera sobre um parâmetro de tipo é chamado de genérico, como em classe genérica ou método genérico.
- Uma vantagem importante do código genérico é que ele funciona automaticamente com o tipo de dado passado para seu parâmetro de tipo.

Fundamentos

- Com os genéricos, você pode definir um algoritmo uma única vez, independentemente do tipo de dado, e então aplicá-lo a uma ampla variedade de tipos de dados sem nenhum esforço adicional.
- É importante entender que Java sempre permitiu a criação de classes, interfaces e métodos generalizados usando referências de tipo Object.
- Já que Object é a superclasse de todas as classes, uma referência Object pode referenciar qualquer tipo de objeto.
- Logo, em códigos anteriores aos genéricos, classes, interfaces e métodos generalizados usavam referências Object para operar com vários tipos de dados.

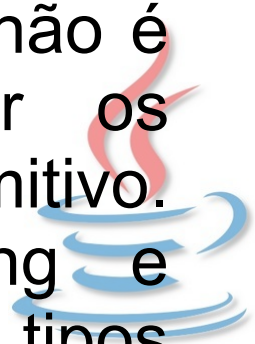


Fundamentos

- O problema é que eles não faziam isso com segurança de tipos, já que coerções eram necessárias para converter explicitamente Object no tipo de dado que estava sendo tratado.
- Portanto, era possível gerar acidentalmente discrepância de tipo. Os genéricos adicionam a segurança de tipos que estava faltando, porque tornam essas coerções automáticas e implícitas.
- Resumindo, eles expandem nossa habilidade de reutilizar código e nos permite fazê-lo de maneira segura e confiável.
- Exemplo da pg. 437

Funcionamento

- Na declaração de um tipo genérico, o argumento de tipo passado para o parâmetro d tipo deve ser um tipo de classe.
- Você não pode usar um tipo primitivo, como int ou char. Por exemplo, com Gen, é possível passar qualquer tipo de classe para T, mas você não pode passar um tipo primitivo para T.
- Certamente, não pode especificar um tipo primitivo não é uma restrição grave, porque você pode usar os encapsuladores de tipos para encapsular um tipo primitivo. Além disso, o mecanismo Java de autoboxing e autounboxing torna o uso do encapsulador de tipos transparente.



Tipos Genéricos

- Um ponto chave que devemos entender sobre os tipos genéricos é que uma referência de uma versão específica de um tipo genérico não tem compatibilidade de tipo com outra versão do mesmo tipo genérico.
- Você pode declarar mais de um parâmetro de tipo em um tipo genérico.
- Para especificar dois ou mais parâmetros de tipo, apenas use uma lista separada por vírgulas.
- Exercício da pg. 442.

Tipos Limitados

- Suponhamos que você quisesse criar uma classe genérica que armazenasse um valor numérico e pudesse executar várias funções matemáticas, como calcular o recíproco ou obter o componente fracionário.
- Você também quer usar a classe para calcular esses valores para qualquer tipo de número, inclusive Integer, Float, Double.
- Logo, quer especificar o tipo dos números genericamente, usando um parâmetro de tipo.
- Exemplo da pg. 444.
- Exercício da pg. 445.

Argumentos Curingas

- Mesmo sendo útil, as vezes a segurança de tipos pode invalidar construções perfeitamente aceitáveis.
- Dada a classe NumericFns mostrada no fim da seção anterior, suponhamos que você quisesse adicionar um método chamado absEqual() que retornasse verdadeiro se dois objetos NumericFns contivessem números cujos valores absolutos fossem iguais.
- Além disso, você quer que esse método funcione apropriadamente, não importando o tipo de número que cada objeto contém.

Argumentos Curingas

- Por exemplo, se um objeto tiver o valor Double 1,25 e o outro tiver o valor Float -1,25, absEqual() retornará verdadeiro.
- Uma maneira de implementar absEqual() é passar para ele um argumento NumericFns e então comparar o valor absoluto desse argumento com o valor absoluto do objeto chamador, só retornando verdadeiro se os valorem forem iguais.
- A primeira vista, criar absEqual() parece uma tarefa fácil. Infelizmente, os problemas começam a surgir assim que tentamos declarar um parâmetro de tipo NumericFns.

Argumentos Curingas

- Que tipo devemos especificar como parâmetro de NumericFns?
- Inicialmente, poderíamos pensar em uma solução como a dada a seguir, em que T é usado como parâmetro de tipo:

```
//Este código não funcionará!
//Determina se os valores absolutos de dois objetos são iguais.
boolean absEqual(NumericFns<T> ob) {
    if(Math.abs(num.doubleValue()) ==
        Math.abs(ob.doubleValue())){
        return true;
    }
    return false;
}
```

- Exercício da pg. 448.

Curingas Limitados

- Os argumentos curingas podem ser limitados de maneira semelhante a como fizemos com o parâmetro de tipo.

- Um curinga limitado é particularmente importante quando estamos criando um método projetado para operar somente com objetos que sejam subclasses de uma superclasse específica.

Métodos Genéricos

- Os métodos de uma classe genérica podem fazer uso do parâmetro de tipo da classe e, portanto, são automaticamente genéricos de acordo com o parâmetro de tipo.
- Entretanto, podemos declarar um método genérico que use um ou mais parâmetros de tipo exclusivamente seus.
- Além disso, podemos criar um método genérico embutido em uma classe não genérica.
- Exercício da pg. 453 e 455.

Interfaces Genéricas

- Além das classes e métodos genéricos, você também pode ter interfaces genéricas.
- As interfaces genéricas são especificadas como as classes genéricas.
- Temos um exercício que cria uma interface chamada Containment, que pode ser implementada por classes que armazenem um ou mais valores.
- Também declara um método chamado contains() que determina se um valor especificado está contido no objeto chamador.
- Exercício da pg. 456 e 458.

Tipos brutos

- Já que o suporte aos genéricos não existia antes do JDK5, era necessário que o Java fornecesse algum meio dos códigos antigos anteriores aos genéricos fazerem a transação.
- Os códigos legados anteriores aos genéricos tinham que ser ao mesmo tempo funcionais e compatíveis com os genéricos. Ou seja, os códigos pré-genéricos devem funcionar com os genéricos e os códigos têm de funcionar com os códigos pré-genéricos.

Tipos brutos

- Para realizar a transição para os genéricos, Java permite que uma classe genérica seja usada sem nenhum argumento de tipo.
- Isso cria um tipo bruto para a classe. Esse tipo bruto é compatível com códigos legados, que não têm conhecimento dos genéricos.
- A principal desvantagem do uso do tipo bruto é a segurança de tipos dos genéricos ser perdida.
- Exercício da pg. 463.

Inferência de tipos com o operador losango

- A partir de JDK 7, podemos encurtar a sintaxe usada na criação de uma instância de um tipo genérico. Em versões de Java anteriores usávamos uma instrução parecida com essa:

Classe<Integer, String> classe = new Classe<Integer,String> (42, “teste”);

- A versão JDK 7 permite que a expressão anterior seja escrita dessa forma:

Classe<Integer, String> classe = new Classe<>(42,”teste”);

Inferência de tipos com o operador losango

- Observe que a parte que cria a instância usa simplesmente<>, que é uma lista de argumentos de tipo. Isso se chama operador *losango.*
- Ele solicita ao compilador que infira os argumentos de tipo requeridos pelo construtor na expressão **new**.
- A principal vantagem dessa sintaxe de inferência de tipos é que ela encurta o que às vezes gera instruções de declaração muito longas. É praticamente útil para os tipos genéricos que especificam limites.

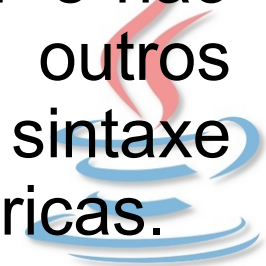
Inferência de tipos com o operador losango

- Quando a inferência de tipos é utilizada, a sintaxe para a declaração de uma referência e de uma instância genéricas tem a forma geral a seguir:

classe<lista-arg-tipo> var = new classe<>(lista-args-const);

- Embora seja mais usada em instruções de declaração, a inferência de tipos também pode ser aplicada à passagem de parâmetros.

- Já que o operador losango foi adicionado por JDK 7 e não funcionará com compiladores mais antigos, os outros exemplos de genéricos continuarão usando a sintaxe completa na declaração de instâncias de classe genéricas.

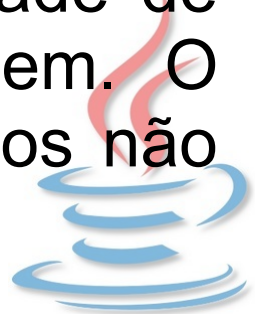


Erasure

- Assim, funcionarão com qualquer compilador Java que dê suporte aos genéricos.
- O uso da sintaxe completa também deixa muito claro o que está sendo criado, o que é útil quando o exemplo de código é mostrado. É claro que, em um código seu, o uso da sintaxe de inferência de tipos otimizará as declarações.
- Geralmente, não é necessário o programador saber os detalhes de como o compilador Java transforma o código-fonte em código-objeto.

Erasure

- No entanto, no caso dos genéricos, algum conhecimento geral do processo é importante, já que ele explica por que os recursos genéricos funcionam como funcionam – e por que às vezes seu comportamento surpreende.
- Logo, é útil falarmos brevemente como os genéricos são implementados em Java.
- Uma restrição importante que conduziu a maneira de os genéricos serem adicionados à Java foi a necessidade de compatibilidade com versões anteriores à linguagem. O código genérico tinha que ser compatível com códigos não genéricos preexistentes.



Erasure

- A maneira de Java implementar os genéricos respeitando essa restrição é com a técnica *erasure.*
- Em geral, é assim que o erasure funciona. Quando o código Java é compilado, todas as informações de tipos genéricos são removidas(em inglês, erased). Ou seja, é feita a substituição dos parâmetros de tipo por seu tipo limitado, que é **Object** quando nenhum limite explícito é especificado, e a aplicação das coerções apropriadas(como determinado pelos argumentos de tipo) para que seja mantida a compatibilidade de tipos.
- Essa abordagem dos genéricos não permite que existam parâmetros de tipo no tempo de execução. São simplesmente um mecanismo do código-fonte.

Erros de ambiguidade

- A inclusão dos genéricos fez surgir um novo tipo de erro contra o qual você deve se proteger: a *ambiguidade.* Erros de ambiguidade ocorrem quando o erasure faz duas declarações genéricas aparentemente distintas produzirem o mesmo tipo, causando um conflito.
- Uma classe declara dois tipos genéricos **T e V**. Dentro da classe é feita uma tentativa de sobrecarregar o meto que seta o valor com base em parâmetros do tipo **T e V**. Isso é considerado correto porque **T e V** parecem ser tipos diferentes.
- No entanto há dois problemas de ambiguidade aqui.

Erros de ambiguidade

- Em primeiro lugar, dependendo o modo de criação da classe não é necessário que **T e V** seja tipos diferentes.
- O segundo e mais grave problema é que a remoção de tipos reduz as duas versões ao seguinte:

```
void set(Object o) {//...}
```

- Logo a sobrecarga de **set()** como tentada anteriormente é internamente ambígua. A solução nesse caso é usar dois nomes de métodos distintos em vez de tentar sobrecarregar o método **set().**

Restrições dos genéricos

- Há algumas restrições das quais você deve lembrar ao usar genéricos. Elas envolvem a criação de objetos de um parâmetro de tipo, membros estáticos, exceções e arrays. Todas elas veremos aqui.

Parâmetros de tipos não podem ser instanciados

- Não é possível criar uma instância de um parâmetro de tipo. Por exemplo:

```
//Não é possível criar uma instância de T
class Gen<T> {
    T ob;
    Gen() {
        Ob = new T(); //inválido!!!
    }
}
```

Restrições dos genéricos

- Não é válido tentar criar uma instância de **T.** A razão deve ser fácil de entender: o compilador não tem como saber que tipo de objeto criar. **T** é simplesmente um espaço reservado que é apagado.

Restrições aos membros estáticos

- Nenhum membro **static** pode usar um parâmetro de tipo declarado pela classe externa. Por exemplo:

```
Class Wrong<T> {
    //Errado, não há variáveis estáticas de tipo T.
    static T ob;
    //Errado, nenhum método estático pode usar T.
    static T getob() {
        return ob;
    }
}
```

Restrições dos genéricos

- Embora você não possa declarar membros **static** que usem um parâmetro de tipo declarado pela classe que os contêm, *pode* declarar métodos genéricos **static**, que definam seus próprios parâmetros de tipo, como foi feito anteriormente neste capítulo.

Restrições aos arrays estáticos

- Há duas restrições importantes dos genéricos aplicáveis aos arrays. Em primeiro lugar, você não pode instanciar um array cujo tipo do elemento seja um parâmetro de tipo. Em segundo lugar, não pode criar um array de referências genéricas específicas de um tipo.