A Natureza Evolutiva das Interfaces Java

ENTENDENDO A HERANÇA MÚLTIPLA EM JAVA

Na série ocasional “New to Java”, eu tento escolher temas que convidam uma compreensão mais profunda do fundo conceitual de um construtor de linguagem. Freqüentemente, os programadores iniciantes têm um conhecimento prático de um conceito - isto é, podem usá-lo em muitas situações, mas carecem de uma compreensão mais profunda dos princípios subjacentes que levariam a escrever códigos melhores, criar melhores estruturas e tomar melhores decisões sobre eles. quando usar uma determinada construção. Interfaces Java são geralmente apenas um tópico desse tipo.

Neste artigo, suponho que você tenha um entendimento básico de herança. As interfaces Java estão intimamente relacionadas à herança, assim como as palavras-chave `extends` e `implements`. Então, vou discutir por que o Java tem dois mecanismos de herança diferentes (indicados por essas palavras-chave), como classes abstratas são usadas e para quais tarefas as interfaces podem ser usadas.

Como tantas vezes acontece, a história desses recursos começa com algumas idéias bastante simples e elegantes que levam à definição de conceitos nas primeiras versões do Java, e a história se torna mais complicada à medida que o Java avança para lidar com problemas mais complicados do mundo real. Este desafio levou à introdução de métodos padrão no Java 8, o que turvou um pouco as águas.

Um pouco de antecedentes sobre herança

A herança é simples de entender em princípio: uma classe pode ser especificada como uma extensão de outra classe. Nesse caso, a classe atual é chamada de subclasse e a classe que está sendo estendida é chamada de superclasse. Objetos da subclasse possuem todas as propriedades da superclasse e da subclasse. Eles têm todos os campos definidos em subclasse ou superclasse e também todos os métodos de ambos. Por enquanto, tudo bem.

A herança é, no entanto, o equivalente ao canivete suíço na programação: ele pode ser usado para atingir alguns objetivos muito diversos. Eu posso usar a herança para reutilizar algum código que escrevi antes, posso usá-lo para subdividir e despachar dinamicamente, posso usá-lo para separar especificação de implementação, posso usá-lo para especificar um contrato entre diferentes partes de um sistema e pode usá-lo para uma variedade de outras tarefas. Todas são ideias importantes, mas muito diferentes. É necessário entender essas diferenças para ter uma boa noção de herança e interfaces.

Herança de Tipo Versus Herança de Código

Dois recursos principais que a herança fornece são a capacidade de herdar o código e a capacidade de herdar um tipo. É útil separar essas duas ideias conceitualmente, especialmente porque a herança padrão de Java as mescla. Em Java, toda classe que eu defino também define um tipo: assim que eu tenho uma classe, eu posso criar variáveis desse tipo, por exemplo.

Quando eu crio uma subclasse (usando a palavra-chave `extends`), a subclasse herda tanto o código quanto o tipo da superclasse. Os métodos herdados estão disponíveis para serem chamados (vou me referir a isso como "o código"), e os objetos da subclasse podem ser usados em locais onde os objetos da superclasse são esperados (assim, a subclasse cria um subtipo).

Vamos ver um exemplo. Se `Estudante` for uma subclasse de `Pessoa`, então os objetos da classe `Estudante` terão o tipo `Estudante`, mas eles também terão o tipo `Pessoa`. Um estudante é uma pessoa. O código e o tipo são herdados.

A decisão de vincular herança de tipo e herança de código em Java é uma escolha de design de linguagem: isso foi feito porque geralmente é útil, mas não é a única maneira pela qual uma linguagem pode ser projetada. Outras linguagens de programação permitem herdar o código sem herdar o tipo (como herança privada C++) ou herdar o tipo sem herdar o código (que o Java também suporta, como explicarei em breve).

Herança Múltipla

A próxima ideia que entra no mix é herança múltipla: uma classe pode ter mais de uma superclasse. Deixe-me dar um exemplo: os estudantes de doutorado na minha universidade também trabalham como instrutores. Nesse sentido, eles são como professores (eles são instrutores de uma classe, têm um número de sala, um número de folha de pagamento e assim por diante). Mas também são estudantes: estão matriculados em um curso, têm um número de identificação de estudante e assim por diante. Eu posso modelar isso como herança múltipla (veja a Figura 1).

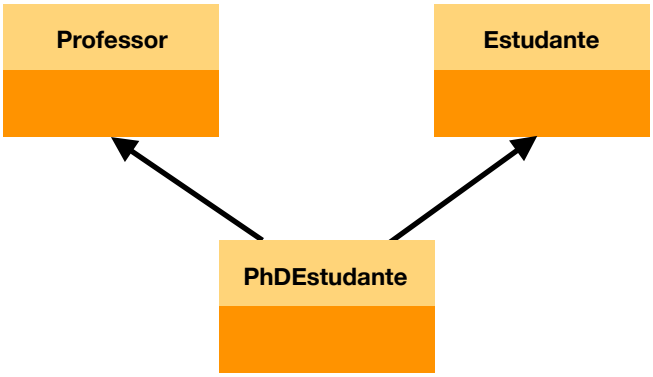

Figura 1. Um exemplo de herança múltipla

`PhDEstudante` é uma subclasse de `Professor` e `Estudante`. Desta forma, um estudante de doutorado terá os atributos de estudantes e professores. Conceitualmente isso é simples. Na prática, no entanto, a linguagem se torna mais complicada se for permitido declarar herança múltipla, porque isso introduz novos problemas: E se ambas as superclasses tiverem campos com o mesmo nome? E se eles tiverem métodos com a mesma assinatura, mas diferentes implementações? Para esses casos, preciso de construções de linguagem que especifiquem alguma

solução para o problema de ambigüidade e sobrecarga de nomes. No entanto, fica pior.

Herança de Diamante

Um cenário mais complicado é conhecido como herança de diamantes (veja a Figura 2). É aqui que uma classe (`PhDStudent`) tem duas superclasses (`Professor` e `Estudante`), que por sua vez possuem uma superclasse comum (`Pessoa`). O gráfico de herança forma uma forma de diamante.

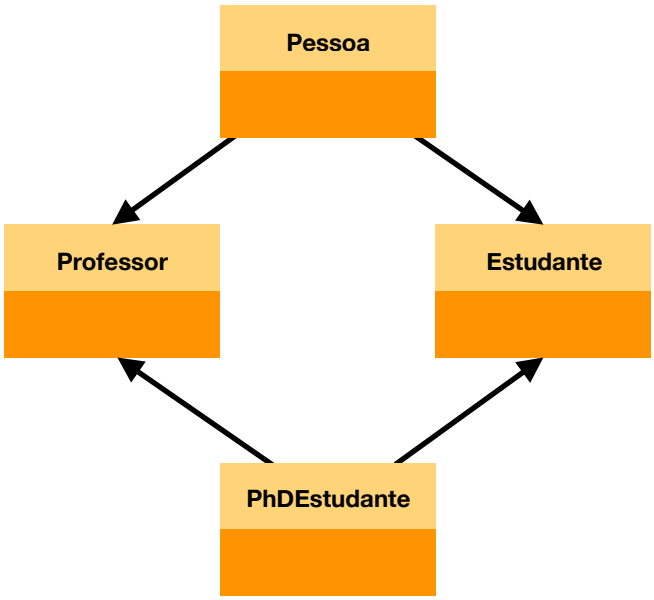

Figura 2. Um exemplo de herança de diamante

Agora, considere esta pergunta: Se há um campo na superclasse de nível superior (`Pessoa`, neste caso), a classe na parte inferior (`PhDStudent`) deve ter uma cópia desse campo ou dois? Ele herda esse campo duas vezes, afinal, uma vez via cada um dos seus ramos de herança.

A resposta é: depende. Se o campo em questão for, digamos, um número de ID, talvez um estudante de doutorado deva ter dois: um ID de estudante e um ID de professor / folha de pagamento que pode ser um número diferente. Se o campo for, no entanto, o nome da família da pessoa, você só deseja um (o estudante de doutorado tem apenas um nome de família, embora seja herdado de ambas as superclasses).

Em suma, as coisas podem se tornar muito confusas. Idiomas que permitem herança múltipla completa precisam ter regras e construções para lidar com todas essas situações, e essas regras são complicadas.

Tipo de Herança para o Resgate

Quando você pensa sobre esses problemas com cuidado, percebe que todos os problemas com herança múltipla estão relacionados ao código herdado: implementações de métodos e campos. A herança de vários códigos é confusa, mas a herança de vários tipos não causa problemas. Este fato é acoplado a outra observação: a herança de múltiplos códigos não é muito importante, porque você pode usar delegação (usando uma referência a outro objeto), mas muitas vezes a subtipagem é muito útil e não é facilmente substituída de maneira elegante.

É por isso que os projetistas de Java chegaram a uma solução pragmática: permitir herança única para código, mas permitir herança múltipla para tipos.

Interfaces

Para tornar possível ter regras diferentes para tipos e códigos, o Java precisa ser capaz de especificar tipos sem especificar o código. É isso que uma interface Java faz.

Interfaces especificam um tipo de Java (o nome do tipo e as assinaturas de seus métodos) sem especificar qualquer implementação. Nenhum campo e nenhum corpo de método é especificado. Interfaces podem conter constantes. Você pode deixar de fora os modificadores (`public static final` para constantes e `public` for methods) - eles são assumidos implicitamente.

Esse arranjo me fornece dois tipos de herança em Java: posso herdar uma classe (usando `extends`), na qual herdo o tipo e o código, ou posso herdar apenas um tipo (usando `implements`) herdando de uma interface. E agora posso ter regras diferentes sobre herança múltipla: Java permite herança múltipla para tipos (interfaces) mas apenas herança única para classes (que contêm código).

Benefícios da herança múltipla para tipos

Os benefícios de permitir a herança de vários tipos - essencialmente de poder declarar que um objeto pode ser visto como tendo um tipo diferente em momentos diferentes - são bem fáceis de ver. Suponha que você esteja escrevendo uma simulação de tráfego e nela você tenha objetos da classe `Carro`. Além dos carros, existem outros tipos de objetos ativos em sua simulação, como pedestres, caminhões, semáforos e assim por diante. Você pode então ter uma coleção central em seu programa - digamos, um `List` - que contém todos os atores:

```
private List atores;
```

`Ator`, neste caso, poderia ser uma interface com um método de `agir`:

```
public interface Ator {
      void agir();
}
```

Sua classe `Carro` pode, então, implementar essa interface:

```
class Carro implements Ator {
      public void agir() {
            …
      }
}
```

Observe que, como o `Carro` herda apenas o tipo, incluindo a assinatura do método `agir`, mas nenhum código, ele próprio deve fornecer o código para implementar o tipo (a implementação do método `agir`) antes de poder criar objetos a partir dele.

Até agora, isso é apenas uma herança única e poderia ter sido obtida herdando uma classe. Mas imagine agora que há também uma lista de todos os objetos a serem desenhados na tela (o que não é o mesmo que a lista de atores, porque alguns atores não são desenhados, e alguns objetos desenhados não são atores):

```
private List desenhos;
```

Você também pode querer salvar uma simulação para armazenamento permanente em algum momento, e os objetos a serem salvos podem, novamente, ser uma lista diferente. Para ser salvo, eles precisam ser do tipo `Serializable`:

```
private List objetosParaSalvar;
```

Nesse caso, se os objetos `Carro` fizerem parte de todas as três listas (eles agem, são desenhados e devem ser salvos), a classe `Carro` pode ser definida para implementar todas as três interfaces:

```
class Carro implements Ator, Drawable, Serializable …
```

Situações como esta são comuns, e permitir múltiplos supertipos permite que você visualize um único objeto (o carro, neste caso) de diferentes perspectivas, focando em diferentes aspectos para agrupá-los com outros objetos similares ou para tratá-los de acordo com um certo subconjunto de seus possíveis comportamentos.

O modelo de processamento de eventos da GUI do Java é construído com base na mesma ideia: o tratamento de eventos é obtido através de ouvintes de eventos - interfaces (como `ActionListener`) que geralmente implementam apenas um método - para que objetos que o implementem possam ser vistos como ouvintes quando necessário.

Classes Abstratas

Devo dizer algumas palavras sobre classes abstratas, porque é comum imaginar como elas se relacionam com as interfaces. Classes abstratas ficam a meio caminho entre classes e interfaces: elas definem um tipo e podem conter código (como as classes fazem), mas também podem ter métodos abstratos - métodos que são especificados apenas, mas não implementados. Você pode pensar neles como classes parcialmente implementadas com algumas lacunas nelas (código que está faltando e precisa ser preenchido por subclasses).

No meu exemplo acima, a interface `Ator` poderia ser uma classe abstrata. O método de `agir` em si pode ser abstrato (porque é diferente em cada ator específico e não há um padrão razoável), mas talvez contenha algum outro código comum a todos os atores.

Nesse caso, posso escrever `Ator` como uma classe abstrata, e a declaração de herança da minha classe `Carro` seria assim:

```
class Carro extends Ator implements Drawable, Serializable …
```

Se eu quiser que várias das minhas interfaces contenham código, transformá-las em classes abstratas não funcionará. Como afirmei anteriormente, Java permite apenas herança única para classes (isso significa que apenas uma classe pode ser listada após a palavra-chave extends). Herança múltipla é apenas para interfaces.

Existe uma saída, no entanto: os métodos padrão, que foram introduzidos no Java 8. Eu os alcançarei em breve.

Interfaces vazias

Às vezes, você se depara com interfaces vazias - elas definem apenas o nome da interface e nenhum método. `Serializable`, mencionado anteriormente, é essa interface. `Cloneable` é outro. Essas interfaces são conhecidas como interfaces de marcadores. Eles marcam certas classes como possuindo uma propriedade específica, e sua finalidade está mais relacionada ao fornecimento de metadados do que à implementação de um tipo ou à definição de um contrato entre partes de um programa. Java, desde a versão 5, teve anotações, que são uma maneira melhor de fornecer metadados. Há pouca razão hoje para usar interfaces de marcadores em Java. Se você for tentado, procure usar anotações.

Um novo amanhecer com o Java 8

Até agora, ignorei intencionalmente alguns novos recursos que foram introduzidos com o Java 8. Isso ocorre porque o Java 8 adiciona funcionalidades que contradizem algumas das decisões de design anteriores da linguagem (como “única herança única para código”), o que explica a relação de algumas construções é bastante difícil. Argumentar a diferença entre a justificativa e a existência de interfaces e classes abstratas, por exemplo, torna-se bastante complicado. Como mostrarei em instantes, as interfaces no Java 8 foram estendidas para se tornarem mais semelhantes às classes abstratas, mas com algumas diferenças sutis.

Na minha explicação dos problemas, eu levei você para o caminho histórico - explicando primeiro a situação pré-Java 8 e agora adicionando os recursos mais novos do Java 8. Fiz isso de propósito, porque entender a justificativa para a combinação de recursos como são hoje é possível apenas à luz dessa história.

Se a equipe Java fosse projetar o Java do zero agora, e se quebrar a compatibilidade com versões anteriores não fosse um problema, eles não o projetariam da mesma maneira. A linguagem Java, no entanto, não é antes de tudo um exercício teórico, mas um sistema para uso prático. E no mundo real, você deve identificar maneiras de evoluir e estender sua linguagem sem quebrar

tudo o que foi feito antes. Métodos padrão e métodos estáticos em interfaces são dois mecanismos que tornaram possível o progresso no Java 8.

Interfaces em Evolução

Um problema no desenvolvimento do Java 8 foi como evoluir interfaces. O Java 8 incluiu lambdas e vários outros recursos na linguagem Java que tornaram desejável adaptar algumas das interfaces existentes na biblioteca Java. Mas como você evolui uma interface sem quebrar todo o código existente que usa essa interface?

Imagine que você tenha uma interface VarinhaMagica na sua biblioteca existente:

```
public interface VarinhaMagica {
      void executeMagica();
}
```

Essa interface já foi usada e implementada por muitas classes em muitos projetos. Mas agora você tem novas funcionalidades realmente boas e gostaria de adicionar um novo método realmente útil:

```
public interface VarinhaMagica {
      void executeMagica();
      void executeMagicaAvancada();
}
```

Se você fizer isso, todas as classes que implementaram anteriormente essa interface quebram, porque elas precisam fornecer uma implementação para esse novo método. Assim, à primeira vista, parece que você está preso: ou você quebra o código de usuário existente (o que você não quer fazer) ou está fadado a ficar com suas bibliotecas antigas sem a chance de melhorá-las facilmente. (Na verdade, existem algumas outras abordagens que você poderia tentar, como estender interfaces em subinterfaces, mas elas têm seus próprios problemas, que eu não discuto aqui.) O Java 8 criou um truque inteligente para obter o melhor de ambos mundos: a capacidade de adicionar interfaces existentes sem quebrar o código existente. Isso é feito usando métodos padrão e métodos estáticos, que discuto agora.

Métodos Padrão

Os métodos padrão são métodos em interfaces que possuem um corpo de método - a implementação padrão. Eles são definidos usando o modificador padrão no início da assinatura do método e eles têm um corpo de método completo:

```
public interface VarinhaMagica {
      void executeMagica();

      default void executeMagicaAvancada() {
            ... // algum código aqui
```

```
    }
}
```

As classes que implementam essa interface agora têm a chance de fornecer sua própria implementação para esse método (substituindo-a) ou podem ignorar completamente esse método, caso em que recebem a implementação padrão da interface. O código antigo continua a funcionar, enquanto o novo código pode usar essa nova funcionalidade.

Métodos estáticos

As interfaces agora também podem conter métodos estáticos com implementações. Estes são definidos usando o modificador estático usual no início da assinatura do método. Como sempre, ao escrever interfaces, o modificador público pode ficar de fora, porque todos os métodos e todas as constantes nas interfaces são sempre públicos.

Então, o que acontece com o problema do diamante?

Como você pode ver, as classes e interfaces abstratas tornaram-se bastante semelhantes agora. Ambos podem conter métodos e métodos abstratos com implementações, embora a sintaxe seja diferente. Ainda existem algumas diferenças (por exemplo, classes abstratas podem ter campos de instância, enquanto interfaces não podem), mas essas diferenças suportam um ponto central: desde o lançamento do Java 8, você tem herança múltipla (via interfaces) que pode conter código!

No início deste artigo, apontei como os designers de Java trabalharam com muito cuidado para evitar a herança de vários códigos por causa de possíveis problemas, principalmente relacionados a herdar várias vezes e nomear conflitos. Então, qual é a situação agora?

Como sempre, os projetistas de Java criaram as seguintes regras práticas e sensatas para lidar com esses problemas:

- Herdar vários métodos abstratos com o mesmo nome não é um problema - eles são vistos como o mesmo método.
- A herança de campos de diamante - um dos problemas difíceis - é evitada, porque as interfaces não podem conter campos que não sejam constantes.
- Herdar métodos e constantes estáticos (que também são estáticos por definição) não é um problema, porque eles são prefixados pelo nome da interface quando são usados, portanto seus nomes não entram em conflito.
- Herdar de diferentes interfaces vários métodos padrão com a mesma assinatura e implementações diferentes é um problema. Mas aqui o Java escolhe uma solução muito mais pragmática do que algumas outras linguagens: em vez de definir uma nova construção de linguagem para lidar com isso, o compilador apenas relata um erro. Em outras palavras, é problema seu. Java apenas diz: "Não faça isso".

Conclusão

Interfaces são um recurso poderoso em Java. Eles são úteis em muitas situações, inclusive para definir contratos entre diferentes partes do programa, definindo tipos para o despacho dinâmico, separando a definição de um tipo de sua implementação e permitindo a herança múltipla em Java. Eles são muito úteis em seu código; você deve se certificar de que entende bem o comportamento deles.

Os novos recursos de interface no Java 8, como os métodos padrão, são mais úteis quando você escreve bibliotecas; eles são menos propensos a serem usados no código do aplicativo. No entanto, as bibliotecas Java agora fazem uso extensivo delas, portanto, saiba o que elas fazem. O uso cuidadoso das interfaces pode melhorar significativamente a qualidade do seu código.

Michael Kölling é um campeão de Java e professor da Universidade de Kent, Inglaterra. Ele publicou dois livros-texto Java e vários artigos sobre orientação a objetos e tópicos de educação em computação, e é o desenvolvedor líder do BlueJ e do Greenfoot, dois ambientes de programação educacional. Kölling também é um Educador Distinto do ACM.