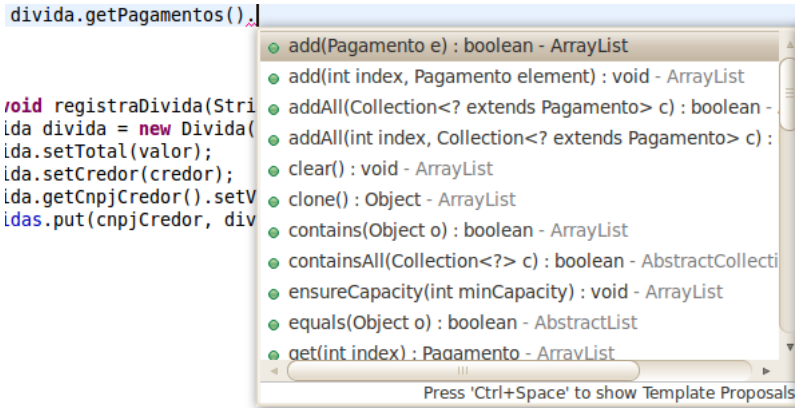
01

Herança: quando não usar

Transcrição

No capítulo anterior, criamos a classe `Pagamentos` para representar uma lista de pagamentos. Fizemos ela herdar de `ArrayList`, afinal ela é uma lista de `Pagamento`. Mas veja só os métodos que nossa classe ganhou com a herança:



Fizemos o método `registra`, mas temos um método `add`. Um deles calcula os impostos incidentes no pagamento e atualiza o valor total pago, outro não. Qual deles devemos chamar? Podemos resolver isso sobrescrevendo o método `add`, para que ele contenha a lógica que está no método `registra`.

```
public class Pagamentos extends ArrayList<Pagamento> {
  private double valorPago;

  @Override
  public boolean add(Pagamento pagamento) {
    double valor = pagamento.getValor();
    if (valor < 0) {
      throw new IllegalArgumentException("Valor invalido para pagamento");
    }
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
    return super.add(pagamento);
  }

  // outros métodos
}
```

Daí o método `registra` passa a chamar diretamente o método `add`:

```
public class Pagamentos extends ArrayList<Pagamento> {
  private double valorPago;

  @Override
  public boolean add(Pagamento pagamento) {...}

  public void registra(Pagamento pagamento) {
    this.add(pagamento);
```

```
    }

    // outros métodos
}
```

Podemos até apagar o método `registra`, para evitar confusões. Mas, veja só, temos ainda os métodos `add(int index, Pagamento element)` e os métodos `addAll`. Será que precisamos reimplementá-los também? O método `addAll` deve usar o `add`, certo? Veja o que acontece se só implementarmos o `add`:

```
Pagamento p1 = new Pagamento();
Pagamento p2 = new Pagamento();

p1.setValor(105);
p2.setValor(25);

Pagamentos pagamentos1 = new Pagamentos();
pagamentos1.add(p1);
pagamentos1.add(p2);

System.out.println("Valor já pago: " + pagamentos1.getValorPago()); // 122.0

Pagamentos pagamentos2 = new Pagamentos();
ArrayList<Pagamento> adicionar = new ArrayList<Pagamento>();
adicionar.add(p1);
adicionar.add(p2);
pagamentos2.addAll(adicionar);

System.out.println("Valor já pago: " + pagamentos2.getValorPago()); // 0.0
```

Então podemos sobrescrevê-lo também. Fazemos ele calcular o valor pago, já com descontos, e, em seguida, delegamos a adição dos elementos para a implementação da classe mãe.

```
@Override
public boolean addAll(Collection<? extends Pagamento> adicionar) {
  for (Pagamento pagamento : adicionar) {
    double valor = pagamento.getValor();
    if (valor < 0) {
      throw new IllegalArgumentException("Valor invalido para pagamento");
    }
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
  }
  return super.addAll(adicionar);
}
```

Podemos fazer isso sem medo porque agora sabemos que a implementação do `addAll` da classe mãe não chama o `add`. Não corremos o risco de calcular duas vezes os impostos e o valor da nota. Mas será que isso vale para todas as coleções do Java?

Imagine que queremos evitar pagamentos duplicados na dívida. Fazemos nossa classe `Pagamentos` estender `HashSet`, mantendo nossa implementação do `addAll`. Veja só o que acontece agora com os valores dos impostos e da nota:

```
Pagamento p1 = new Pagamento();
Pagamento p2 = new Pagamento();

p1.setValor(105);
p2.setValor(25);

Pagamentos pagamentos1 = new Pagamentos();
pagamentos1.add(p1);
pagamentos1.add(p2);

System.out.println("Valor já pago: " + pagamentos1.getValorPago()); // 122.0

Pagamentos pagamentos2 = new Pagamentos();
ArrayList<Pagamento> adicionar = new ArrayList<Pagamento>();
adicionar.add(p1);
adicionar.add(p2);
pagamentos2.addAll(adicionar);

System.out.println("Valor já pago: " + pagamentos2.getValorPago()); // 244.0
```

A implementação de `addAll` da classe `HashSet`, ao contrário da classe `ArrayList`, chama o método `add`. Acabamos somando o valor dos pagamentos duas vezes! Note como precisamos conhecer a classe que estendemos para implementar nossa lógica. E note que, se a implementação da classe mãe mudar por algum motivo, nossa classe pode parar de funcionar. Isso significa que o acoplamento com a classe mãe é muito alto!

Ainda temos um outro problema. Faz sentido termos um método para remover pagamentos já realizados de uma dívida? Mesmo se a resposta for não, temos os métodos `remove`, `removeAll` e `clear`.

E trocar um pagamento registrado por outro? Talvez não faça sentido, mas temos o método `set`, que troca o elemento em uma posição. Veja quanto comportamento herdamos sem querer! E o quanto podemos fazer, sendo que não queremos liberar!

Perceba que estamos estendendo a classe `ArrayList` mas precisamos mudar muito o comportamento dela para adequá-la às nossas necessidades. Ainda por cima, temos métodos indesejados na nossa classe. Isso é um sinal de que não deveríamos estar usando herança nesse caso. Mas como reaproveitar o código que já existe na classe `ArrayList` sem usar herança?

Em vez de fazer nossa classe `Pagamentos` estender `ArrayList`, podemos fazê-la ter uma instância de `ArrayList` como atributo e aproveitar o código da classe `ArrayList` invocando os métodos que desejarmos. Veja, por exemplo, o método `registra`, como fica.

```
public class Pagamentos {
  private double valorPago;
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();

  public void registra(Pagamento pagamento) {
    double valor = pagamento.getValor();
    if (valor < 0) {
      throw new IllegalArgumentException("Valor invalido para pagamento");
```

```
    }
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
    this.pagamentos.add(pagamento);
  }

  // outros métodos
}
```

Note que agora, para reaproveitar um comportamento da classe `ArrayList`, precisamos invocar explicitamente um método dessa classe. Não ganhamos automaticamente todos os métodos que a classe `ArrayList` tem. Para ganharmos um comportamento, precisamos deixar explícito que queremos reaproveitá-lo. Perceba como continuamos reaproveitando o comportamento mas, desse modo, conseguimos controlar o que queremos reaproveitar ou não.

Uma outra coisa muito importante a perceber é que agora, se a implementação da classe `ArrayList` mudar, nossa implementação não quebra. Nossa implementação só quebrará se os métodos de `ArrayList` mudarem, ou seja, se a interface de `ArrayList` mudar. Não precisamos mais conhecer intimamente a classe que queremos reaproveitar. Diminuímos o acoplamento entre a classe `ArrayList` e a classe `Pagamentos`.

Isso que acabamos de fazer foi trocar uma herança por uma composição. Agora, em vez de nossa classe ser um `ArrayList`, ela tem um `ArrayList`. Sempre que temos uma classe herdando de outra, é possível trocar essa herança por uma composição (http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-heranca/), ou seja, substituir o `extends` por uma instância da classe que queremos herdar e métodos na nossa classe que chamam métodos dessa instância.

Veja que, ao substituirmos herança por composição, perdemos, por exemplo, o método `contains`. Mas se precisarmos saber se um pagamento foi efetuado, podemos recuperar esse comportamento. Mais ainda: podemos dar um nome mais significativo para ele.

```
public class Pagamentos {
  private double valorPago;
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();

  public boolean foiRealizado(Pagamento pagamento) {
    return pagamentos.contains(pagamento);
  }

  // outros métodos
}
```

Herança é um recurso que deve ser usado com muito cuidado! Existem casos mesmo dentro da linguagem Java em que a herança foi mal utilizada, como na implementação da classe `Stack`, que estende `Vector`, ou da classe `Properties`, que estende `Hashtable`. Essas classes acabaram herdando métodos que não fazem sentido para elas e, para o desenvolvedor saber quais métodos ele pode ou não usar, ele precisa olhar na documentação da classe.

Dando preferência a composição sobre herança, evitamos quebra de encapsulamento de nossas classes. Diminuímos o acoplamento entre nossas classes, evitando, assim, que uma mudança em uma única classe quebre várias partes de nosso sistema.