Java

Regras de casting

segunda-feira, 03/12/2001 às 03h12, por Cristiano Trindade

Utilizamos o casting ou moldagem para converter
um objeto ou primitiva de um tipo para outro. Podemos converter
um double em um int, ou uma subclasse em uma superclasse.

A regra fundamental para moldagem é: o casting
não modifica o objeto ou valor que está sendo moldado; porém,
o receptor do cast constitui um novo objeto ou um novo tipo.

O casting de dados primitivos ocorre com bastante
frequência. A maneira pelo qual o casting age depende da precisão
dos dados envolvidos. Por sua vez, a precisão depende da quantidade
de informações que um tipo pode conter; assim, um tipo de
ponto flutuante double tem maior precisão que um float, pois
o primeiro é um tipo de 64 bits e o segundo, de 32 bits.

Sempre que houver uma transferência de dados
de um tipo menos preciso para um mais preciso, o casting explícito
não é obrigatório.

```
int i = 3;
double pi = i + .14159;
```

Já a transferência de dados de um tipo mais
preciso para um menos preciso requer um casting explícito.
Isso ocorre porque pode haver perda de dados durante a moldagem.
Java pretende alertar o programador do possível perigo de
uma conversão desse tipo:

```
double pi = 3.14159;
int i = (int)pi; // variável definida como 3; perda de .14159
```

O casting entre objetos requer um pouco mais
de cuidado. Para ilustrar uma aplicação de casting para objetos,
vamos analisar a classe Hashtable do JDK da Sun. Ela utiliza
instâncias da classe Object e as insere em uma tabela de hash,

por meio de um método chamado put(), no qual um String é usado como chave.

Vamos supor que você tenha uma classe chamada MinhaClasse e um String usado como chave em uma variável chave. Incluímos essa chave em um objeto Hashtable da seguinte maneira:

```
MinhaClasse MeuObjeto;
hash.put(chave.MeuObjeto);
```

Vamos supor que seja necessária a armazenagem de MeuObjeto como Object próprio, ou seja, um cast de uma subclasse como uma superclasse. Isso seria feito da seguinte forma:

```
hsh.put(chave,(Object)MeuObjeto);
```

Ao fazer este cast, perde-se a funcionalidade de MeuObjeto.

É possível também converter o Object retronado por get() no MeuObjeto original, através de:

```
MinhaClasse MeuObjeto;
hash.put(chave,MeuObjeto);
MinhaClasse MeuObjeto2 = (MinhaCLasse)hash.get(chave);
// obtém de volta o MeuObjeto original
```

Não é possível fazer o cast de tipos de dados primitivos como objetos ou vice-versa. Entretanto, pode-se realizar essas conversões utilizando as classes wrappers de tipos discutidas no início deste capítulo.

Regras de Escopo

Quando uma variável é referenciada no interior de um método, Java procura primeiro uma declaração em um bloco mais externo. Se não encontra enhuma declaração, Java percorre os blocos alinhados, até alcançar a definição do método. Se a variável se encontrar em qualquer lugar no interior de um método, ela tem prioridade sobre uma variável de instância ou de classe com nome semelhante. Por isso, encontramos frequentemente códigos como este:

```
public RegistroSimples(String
nome, String email) {
this.nome = nome;
this.email = email;
}
```

A palavra this
é usada nesse caso para diferenciar a variável de instância
da variável local.

Tratamento de Exceções

Um grande desafio atualmente em programação
é saber como lidar com erros durante a execução de maneira
eficiente. Tradicionalmente, os programadores gerenciam os
problemas pela passagem de códigos de sucesso ou falha em
instruções return. Então, as instruções de chamada verificam
o código de retorno em uma instrução if … else. Se a função
é bem-sucedida, é chamada uma sequência de ações: caso contrário,
é executada outra série de funções.

Existem alguns problemas nesse tipo de abordagem.
O excesso de instruções if … else resulta em um aumento
considerável do tamanho do código. Além disso, essa solução
não implementa uma forte verificação de erros.

Java possui um considerável esquema de verificação
de erros. Muitos deles são interceptados pelo próprio compilador,
ao invés de serem detectados somente na execução. Porém, se
alguma falha "passar" pelo compilador, o exame em
uma pilha (veremos mais tarde que a virtual machine de Java
é toda baseada na pilha) irá facilitar a busca do problema.

Os tratadores de exceção e
as classes de exceções

As exceções podem ser de diferentes tipos ou,
mais especificamente, de classes distintas. Por exemplo, NumberFormatException
se refere a uma classe específica. Quando uma exceção é gerada,
o sistema runtime do Java procura um tratador apropriado para
interceptar a falha. O fato de um tratador ser ou não apropriado
para uma determinada exceção está relacionado com a classe
da exceção.

Vejamos um exemplo: a classe ArithmeticException
representa erros causados por operações aritméticas inválidas
(geralmente uma divisão por zero). Os objetos da classe ArrayIndexOutOfBoundsException
são emitidos quando é acessado um índice de array inválido.
A classe Exception éutilizada para representar a classe geral
de exceções; na verdade, as outras classes de exceções que
foram discutidas são subclasses de Exception. Vejamos como
essas exceções são usadas:

```
try {

int x = Integer.parse(s);
int y = Integer.parse(s2);
int index = Integer.parseInt(s3);
int a[] = new Int[5];

a[index] = x/y;
System.out.println("Tudo está funcionando");
}
catch (ArithmeticException e) {
System.out.println("Exceção aritmética");
}
catch (ArrayIndexOutOfBoundsException e) {
System.out.println("Exceção de índice de array fora dos
limites");
}
catch (NumberFormatException e) {
System.out.println("Impossível converter String em número");
}
catch (Eception e) {
System.out.println("Exceção genérica");
}
```

A primeira parte do código utiliza três Strings
de entrada e tenta convertê-los em inteiros; se a conversão
falha, é emitido um objeto NumberFormatException. No entanto,
a terceira cláusula catch irá interceptar o objeto emitido.
Se a cláusla não existisse, a última cláusula catch iria detectar
o problema, porque NumberFormatException é uma subclasse de
Exception.

Depois que os Strinfs são convertidos em números,
o programa tenta dividir os números. Se houver uma tentativa

de divisão por zero, é emitido um objeto ArithmeticException.
O objeto é então interceptado pela cláusula catch de ArthmeticException.

Também poderemos ter exceções aninhadas. Vejamos
uma variação do código anterior:

```
int y = 0;
int z = 0;
int x = 0;
int index = 0;
int a[] = new int[5];

try {

x = Integer.parseInt(s);
y = Integer.parseInt(s2);

index = Integer.parseInt(s3);
a[index] = x/y;
System.out.println('Tudo está funcionando");
}
catch (ArithmeticException e) {
System.out.println("Exceção aritmética. Experimente atribuir
zero");

try {
a[index] = x/z;
}
catch (ArithmeticException e2) {
System.out.println("Outra exceção aritmética");
}
catch (ArrayIndexOutOfBoundsException e)
System.out.println("Exceção de índice de arrays fora
dos limites");
}
catch (NumberFormatException e) {
System.out.println("Não é possível converter String em
número");
}
catch (Exception e) {
System.out.println("Exceção genérica");
}
```

Se a variável y do tipo int tiver valor 0,
é emitida uma ArithmeticException. Quando esta exceção é interceptada,
o código tenta executar outra divisão, que também é inválida.
Este novo erro é interceptado por um bloco catch de ArithmeticException.
Se o erro fosse, ao invés de uma divisão por zero (emissão
da exceção ArithmeticException), um índice inválido, seria
emita a exceção ArrayIndexOutOfBoundsException.

Por hoje, ficamos por aqui. Até a semana que
vem e bons estudos!

Cristiano Trindade
é desenvolvedor de sistemas e atualmente trabalha no mercado de aplicações para dispositivos móveis. Possui sólidos conhecimentos em Java, ASP, ASP.Net, ColdFusion e Javascript, C/C++, C# e Pascal.