07

Conversões e outros tipos

Transcrição

Haverá momentos em que queremos misturar os tipos de variáveis, como o `double` e o `int`. Vimos que um `int` cabe no `double`, mas o caminho inverso não funciona. Vamos, então, criar uma classe denominada "TestaConversao".

Incluiremos uma variável do tipo `salario` com os `1270.50`, que por algum motivo queremos que esteja em uma variável do tipo inteiro. E então guardaremos `salario` em `valor`:

```
public class TestaConversao {

    public static void main(String[] args) {
        double salario = 1270.50;
        int valor = salario;
    }
}
```

Já vimos que isto não funciona, pois o compilador do Java é rígido e não deixa que isto ocorra sem que afirmemos com total segurança de estarmos cientes de que perderemos o `.50`. Por conta disso, deixaremos as duas linhas comentadas, e mostraremos que o caminho inverso é possível:

```
public class TestaConversao {

    public static void main(String[] args) {
        // double salario = 1270.50;
        // int valor = salario;

        double valor = 3;
    }
}
```

Ou seja, a conversão de um valor inteiro para um tipo `double` é possível, academicamente chamada de **promoção**, ou "ser promovido a um `double`", e acontece de maneira automática.

Para tentarmos fazer com que a parte do código comentada acima funcione, poderemos forçar a conversão, moldando um `double` para que ele se encaixe em um `int`.

É claro que não haverá encaixe perfeito, resultando em arestas que provavelmente serão perdidas. Faremos isso utilizando uma sintaxe comum a outras linguagens, o *casting*, para que o `double` seja transformado em um `int`.

```
public class TestaConversao {

    public static void main(String[] args) {
        double salario = 1270.50;
        int valor = (int) salario;
        System.out.println(valor);
    }
}
```

Se printarmos `valor`, será mostrada apenas a parte inteira daquele número: `1270`. É isso que chamamos de *casting* que, nestas variáveis que guardam números, não é algo muito complexo.

Mais adiante, veremos o *casting* de variáveis que são referência, e têm a ver com orientação a objetos, se são compilados ou não, se darão *exceptions*; é um mundo à parte.

Basicamente, para os tipos chamados **primitivos**, as variáveis básicas que estamos vendo aqui e são `double` com "d" minúsculo, e na cor roxa, possuem funcionamento mais simples. O *casting* faz a conversão quando ela não é possível de forma automática.

Neste caso, sem o `(int)`, assim, entre parênteses, a compilação não ocorre, e a aplicação não rodará.

Como saberemos quais valores se encaixam em quê, e outros tipos numéricos?

No Java, o `int` e o `double` são os tipos mais usados, os outros aparecem de maneira muito esporádica. A nível de curiosidade, em `int` cabem `32bits` com sinais, isto é, números positivos e negativos. Mais especificamente, cabem de `2` elevado a `31` negativos, a `2` elevado a `31` positivos menos `1`, por conta do `0` (zero), o que dá uma quantidade de cerca de `2` bilhões.

O `int` pode guardar até `2` bilhões e, passando dessa quantidade, ocorrerá um *overflow*. Caso se queira guardar um número maior ou menor que este, será preciso um número com `64bits`, que no Java é o `long`, e guarda um número de até `2` elevado a `63` menos `1`. É um número absurdo, que inclusive precisa de um `L` no fim, em caixa alta ou baixa, para indicar que estouramos os `2` bilhões!

```
long numeroGrande = 32432423523L;
```

Por padrão, quando não é um `double`, um número no Java é considerado um `int`. O `L` indica "literal", um valor específico, como um `long`. Em contrapartida, há números menores: o `short`, que guarda um número de `16bits` menos `1`, e o `byte`, que é menor ainda, de até `2` elevado a `8`, que dá `256` com `128` negativos, a `127` com `1` a menos:

```
short valorPequeno = 2131;
byte b = 127;
```

E se o número for maior do que `64bits`, um número gigantesco? Daí, não serão usados tipos primitivos, ou estas variáveis. Podem ser objetos, e então usaremos bibliotecas.

Nesse caso, usaremos este exemplo:

```
double valor1 = 0.2;
double valor2 = 0.1;
double total = valor1+valor2;
```

Esta operação deveria resultar em `0.3`, certo? Ao acrescentarmos `System.out.println(total);` e rodarmos o código, porém, obteremos `0.30000000000000004`. Que número maluco é esse?

Há várias questões matemáticas por trás dele. Se pesquisarmos o valor no Google, encontramos diversos resultados de pessoas buscando uma explicação. Existe até o site 0.30000000000000004.com (http://0.30000000000000004.com/), com a explicação matemática para esse *floating point*, do porquê, em muitas linguagens, essa soma dar exatamente esse valor.

Não é à toa - como uma representação de decimal do inteiro é utilizada para se obter um ponto flutuante, fica complicado fazer uma operação aritmética deste tipo e guardar o resultado internamente. Por isto, o Java, como muitas outras linguagens, segue a especificação **IEEE 754**, de leitura complexa, que remete à Engenharia. De qualquer forma, é normal que este resultado apareça quando utilizamos o `double`.

Para lidarmos com dinheiro sem que apareçam centavos, por exemplo, usaríamos o `BigDecimal`, de que falaremos mais para a frente. Por ora continuaremos com o `double` pois ainda estamos iniciando na linguagem, e queremos usar variáveis que são palavras chave do Java.

Os quatro tipos de tipo primitivo são: `int`, `long`, `byte` e `short`. Quanto aos tipos flutuantes, além do `double`, há o `float` e, se tentarmos definir a variável como recebendo `3.14`, ocorre o mesmo problema do `long`, mesmo se tratando de ponto flutuante.

Para o Java, `3.14` é um `double` com `64bits`. É um valor que cabe em um tipo flutuante com `32bits`? Não, e informações podem ser perdidas. Neste caso, usa-se o *casting*, o que seria estranho, ou se indica que este literal, o valor `3.14`, é um `float`, colocando-se "**f**" no fim:

```
float pontoFlutuante = 3.14f;
```

Mais uma vez, o mais importante é o enfoque no `double` e no `int`, que aparecem com muito mais frequência. E no `long` em alguns casos, o qual será visto em alguns exercícios.