03

# Desenvolver código que usa classes wrappers como Boolean, Double e Integer

## Java 8 - Trabalhando com tipos de dados em Java

### Desenvolver código que usa classes wrappers como Boolean, Double e Integer

::Wrappers:: são classes de objetos que representam tipos primitivos. Existe um ::wrapper:: para cada primitivo, conforme a lista a seguir:

- `boolean` : `Boolean`
- `byte` : `Byte`
- `short` : `Short`
- `char` : `Character`
- `int` : `Integer`
- `long` : `Long`
- `float` : `Float`
- `double` : `Double`

### Criando ::wrappers::

Todos os ::wrappers:: numéricos possuem dois construtores, um que recebe o tipo primitivo, e um que recebe `String` . O construtor que recebe `String` pode lançar `NumberFormatException` , se a `String` fornecida não puder ser convertida ao tipo em questão:

```
Double d1 = new Double(22.5);
Double d2 = new Double("22.5");
Double d2 = new Double("abc"); //throws NumberFormatException
```

COPIAR CÓDIGO

A classe `Character` possui apenas um construtor, que recebe um `char` como argumento:

```
Character c = new Character('d');
```

COPIAR CÓDIGO

A classe `Boolean` também possui dois construtores, um que recebe `boolean` e outro que recebe `String`. Caso a `String` passada como argumento tenha o valor `"true"`, com maiúsculas ou minúsculas, o resultado será `true`; qualquer outro valor resultará em `false`:

```
Boolean b1 = new Boolean(true);   // true
Boolean b2 = new Boolean("true"); // true
Boolean b3 = new Boolean("TrUe"); // true
Boolean b4 = new Boolean("T");    // false
```

COPIAR CÓDIGO

## Convertendo de ::wrappers:: para primitivos

Para converter um ::wrapper:: em um primitivo, existem vários métodos no formato `xxxValue()`, onde `xxx` é o tipo para o qual gostaríamos de realizar a

conversão:

```
Long l = new Long("123");

byte b = l.byteValue();
double d = l.doubleValue();
int i = l.intValue();
short s = l.shortValue();
```

COPIAR CÓDIGO

Todos os tipos numéricos podem ser convertidos entre si. Os tipos `Boolean` e `Character` só possuem método para converter para o próprio tipo primitivo:

```
boolean b = new Boolean("true").booleanValue();
char c = new Character('z').charValue();
```

COPIAR CÓDIGO

## Convertendo de `String` para ::wrappers:: ou primitivos

Além dos construtores dos ::wrappers:: que recebem `String` como parâmetro, também existem métodos para realizar transformações entre ::strings::, ::wrappers:: e primitivos.

Vamos começar convertendo de ::strings:: para primitivos. Cada ::wrapper:: possui um método no formato `parseXXX` onde `XXX` é o tipo do ::wrapper::. Este método também lança `NumberFormatException` caso não consiga fazer a conversão:

```
double d = Double.parseDouble("23.4");
long l = Long.parseLong("23");
int i = Integer.parseInt("444");
```

COPIAR CÓDIGO

Os ::wrappers:: de números inteiros possuem uma variação do `parseXXX` que recebe como segundo argumento a base a ser usada na conversão:

```
short i1 = Short.parseShort("11",10); // 11  Decimal
int i2 = Integer.parseInt("11",16);   // 17  HexaDecimal
byte i3 = Byte.parseByte("11",8);     // 9   Octal
int i4 = Integer.parseInt("11",2);    // 3   Binary
int i5 = Integer.parseInt("A",16);    // 10  HexaDecimal
int i6 = Integer.parseInt("FF",16);   // 255 HexaDecimal
```

COPIAR CÓDIGO

Já para converter uma `String` diretamente para um ::wrapper:: podemos ou usar o construtor como vimos anteriormente, ou usar o método `valueOf`, disponível em todos os ::wrappers::. A assinatura destes métodos é idêntica à do `parseXXX`, inclusive com versões que recebem a base de conversão para tipo inteiros:

```
Double d = Double.valueOf("23.4");
Long l = Long.valueOf("23");
Integer i1 = Integer.valueOf("444");
Integer i2 = Integer.valueOf("5AF", 16);
```

COPIAR CÓDIGO

## Convertendo de primitivos ou ::wrappers:: para String

Assim como todo objeto Java, os ::wrappers:: também possuem um método `toString`:

```
Integer i = Integer.valueOf(256);
String number = i.toString();
```

COPIAR CÓDIGO

Além do `toString` padrão, há uma versão estática sobrecarregada, que recebe o tipo primitivo como argumento. Ademais, os tipos `Long` e `Integer` possuem uma versão que, além do primitivo, também recebem a base:

```
String d = Double.toString(23.5);
String s = Short.toString((short)23);
String i = Integer.toString(23);
String l = Long.toString(20, 16);
```

COPIAR CÓDIGO

Além destes, as classes `Long` e `Integer` ainda possuem outros métodos para fazer a conversão direta para a base escolhida:

```
String binaryString = Integer.toBinaryString(8); //1000,
binary
String hexString = Long.toHexString(11);       // B,
Hexadecimal
String octalString = Integer.toOctalString(22);  // 26 Octal
```

COPIAR CÓDIGO

## Autoboxing

Até o Java 1.4 não era possível executar operações em cima de ::wrappers::. Por exemplo, para somar 1 em um `Integer` era necessário o seguinte código:

```
Integer intWrapper = Integer.valueOf(1);
int intPrimitive = intWrapper.intValue();
intPrimitive++;
intWrapper = Integer.valueOf(intPrimitive);
```

COPIAR CÓDIGO

A partir do Java 5, foi incluído um recurso chamado **autoboxing**. O próprio compilador é responsável por transformar os ::wrappers:: em primitivos (::unboxing::) e primitivos em ::wrappers:: (::boxing::). A mesma operação de somar 1 em um `Integer` agora é:

```
Integer intWrapper = Integer.valueOf(1);
intWrapper++; //will unbox, increment, then box again.
```

COPIAR CÓDIGO

Repara que não há magia. A única diferença é que, em vez de você mesmo escrever o código que faz o ::boxing:: e ::unboxing::, agora esse código é gerado pelo compilador.

## Comparando ::wrappers::

Veja o seguinte código:

```
Integer i1 = 1234;
Integer i2 = 1234;
System.out.println(i1 == i2);       //false
System.out.println(i1.equals(i2)); //true
```

COPIAR CÓDIGO

Apesar de parecer que estamos trabalhando com primitivos, estamos usando objetos aqui, logo, quando esses objetos são comparados usando `==` o resultado é `false`, já que são duas instâncias diferentes de `Integer`.

Agora veja o seguinte código:

```
Integer i1 = 123;
Integer i2 = 123;
System.out.println(i1 == i2);       //true
System.out.println(i1.equals(i2)); //true
```

COPIAR CÓDIGO

Repare que o resultado do `==` foi `true`, mas o que aconteceu? É que o Java, para economizar memória, mantém um **cache** de alguns objetos, e toda vez que é feito um ::boxing:: ele os reutiliza. Os seguintes objetos são mantidos no cache:

- Todos `Boolean` e `Byte`;
- `Short` e `Integer` de -128 até 127;
- `Caracter` ASCII, como letras, números etc.

Sempre que você encontrar comparações usando `==` envolvendo ::wrappers::, preste muita atenção aos valores: se forem baixos, é possível que o resultado seja `true` mesmo sendo objetos diferentes!

## NullPointerException em operações envolvendo ::wrappers::

Fique atento, pois, como ::wrappers:: são objetos, eles podem assumir o valor de `null`. Qualquer operação executada envolvendo um objeto `null` resultará em um `NullPointerException`:

```
Integer a = null;
int b = 44;
System.out.println(a + b); //throws NPE
```

COPIAR CÓDIGO