Autoboxing e unboxing em Java

Veja nesse artigo uma discussão sobre as funções Java autoboxing / unboxing. Essas funções são basicamente um recurso para converter tipos de dados primitivos para objetos e reverter.

Todo o processo é feito automaticamente em tempo de execução Java. Mas cuidados devem ser tomados durante a implementação desse recurso caso contrário, terá efeito negativo sobre o desempenho do aplicativo.

Antes do JDK 1.5, não era fácil para converter tipos de dados primitivos, por exemplo int, char, float, double em seu objeto equivalente (classes Wrapper) - Integer, Character, Float, Double. O JDK 5 veio com o recurso de conversão automática de tipos de dados primitivos em seu objeto equivalente. Esse recurso é conhecido como autoboxing. O contrário disto é conhecido como unboxing, ou seja, o processo de conversão de objetos em tipos primitivos de dados correspondentes é chamado de unboxing. Exemplo de código para ambos autoboxing e unboxing é mostrado como abaixo:

autoboxing

```
Integer integer = 9;
```

unboxing

```
int in = 0;
```

```
in = new Integer(9);
```

Quando Autoboxing e Unboxing são usados?

Autoboxing é aplicada pelo compilador do Java nas seguintes condições:

- Quando um valor primitivo é passado como um parâmetro para um método que espera um objeto da classe Wrapper correspondente.
- Quando um valor primitivo é atribuído a uma variável da classe Wrapper correspondente.

Considere o seguinte exemplo:

Listagem 1: Exemplo de código utilizando Autoboxing

```
public int sumEvenNumbers(List<Integer> intList ) {
```

```
    int sum = 0;
```

```
    for (Integer i: intList )

        if ( i % 2 == 0 )

            sum += i;

        return sum;

}
```

Antes do JDK 1.5, o trecho de código acima resultaria em erro de compilação desde o operador resto - '%' e a expressão - '+ =' não poderia ser aplicada em números inteiros. Mas desde o JDK 1.5 este pedaço de código compila e roda sem nenhum erro, pois converte um Integer para int em tempo de execução.

Unboxing é aplicada pelo compilador do Java nas seguintes condições:

- Quando um objeto é passado como um parâmetro para um método que espera um valor primitivo correspondente.
- Quando um objeto é atribuído a uma variável do tipo primitivo correspondente.

Considere o seguinte exemplo:

Listagem 2: Exemplo de código mostrando Unboxing

```
import java.util.ArrayList;

import java.util.List;



public class UnboxingCheck {



    public static void main(String[] args) {

        Integer in = new Integer(-8);



        // 1. Unboxing through method invocation

        int absVal = absoluteValue( in );
```

```
        System.out.println( "absolute value of " + in + " = " + absVal );



        List<Double> doubleList = new ArrayList<Double>();

    // It is autoboxed through method invocation.

        doubleList.add(3.1416);



        // 2. Unboxing through assignment

        double phi = doubleList.get(0);

        System.out.println( "phi = " + phi );

    }



    public static int absoluteValue( int i ) {

        return (i < 0) ? -i : i;

    }

}
```

Autoboxing e Unboxing permitem que o desenvolvedor escreva seu código forma fácil de ler e entender. A tabela a seguir mostra os tipos de dados primitivos e seus objetos delimitadores correspondentes:

Tipo Primitivo	**Classe Wrapper**
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short

Tabela 1: tipo de dados primitivos e suas classes Wrapper equivalentes

Com operadores de comparação: Autoboxing e Unboxing podem ser feitos com os operadores de comparação. O seguinte trecho de código ilustra como isso pode ser feito:

Listagem 3: Exemplo de código mostrando Autoboxing e unboxing usando operadores de comparação

```
public class BoxedComparator {

    public static void main(String[] args) {

        Integer in = new Integer(25);

        if (in < 35)

            System.out.println("Value of int = " + in);

    }

}
```

Autoboxing e Unboxing com sobrecarga de método: Autoboxing ou Unboxing podem ser feitos em caso de sobrecarga de método. Isto acontece com base nas seguintes regras:

Ampliação bate boxe - Quando há uma situação de escolher entre alargamento e boxe, ampliação leva a preferência:

Listagem 4: Exemplo de código mostrando preferência de sobrecarga

```
public class WideBoxed {

    public class WideBoxed {

    static void methodWide(int i) {

        System.out.println("int");

    }



    static void methodWide( Integer i ) {

        System.out.println("Integer");

    }
```

```
    public static void main(String[] args) {

        short shVal = 25;

        methodWide(shVal);

    }

   }

}
```

A saída deste programa é – int

Ampliação bate VarArgs - Quando há uma situação de escolher entre ampliação e varargs, ampliação tem a preferência

Listagem 5: Exemplo de código mostrando preferência sobrecarga

```
public class WideVarArgs {



    static void methodWideVar(int i1, int i2) {

        System.out.println("int int");

    }



    static void methodWideVar(Integer... i) {

        System.out.println("Integers");

    }



    public static void main( String[] args) {
```

```
        short shVal1 = 25;

        short shVal2 = 35;

        methodWideVar( shVal1, shVal2);

    }

}
```

Boxe bate VarArgs - Quando houver uma situação que escolher entre boxe e varargs, boxe leva a preferência

Listagem 6: Exemplo de código mostrando preferência sobrecarga

```
public class BoxVarargs {

    static void methodBoxVar(Integer in) {

        System.out.println("Integer");

    }



    static void methodBoxVar(Integer... i) {

        System.out.println("Integers");

    }



    public static void main(String[] args) {

        int intVal1 = 25;

        methodBoxVar(intVal1);

    }

}
```

Guardar na mente para usar Autoboxing

Como sabemos que todo bom recurso vem com alguns inconvenientes, Autoboxing não é uma exceção a este respeito. Algumas dicas importantes que o desenvolvedor deve manter em mente ao usar esse recurso são como abaixo:

Comparando objetos com operador de igualdade - o operador de igualdade - "==" conduz à confusão, pois ele pode ser aplicado a ambos os tipos de dados primitivos e objetos. Quando o operador é aplicada nos objectos, ele realmente compara a referência de um dos objectos, e não os valores.

Listagem 7: Exemplo de código mostrando comparação

```
public class Comparator {

    public static void main(String[] args) {



        Integer istInt = new Integer(1);

        Integer secondInt = new Integer(1);



        if (istInt == secondInt) {

            System.out.println("both one are equal");



        } else {

            System.out.println("Both one are not equal");

        }

    }

}
```

Misturando objeto e primitivo na igualdade e operador relacional - Se compararmos um tipo de dados primitivo com um objeto, o unboxing ocorre que pode lançar um NullPointerException se o objeto for nulo.

Cached Object - Desde o método valueOf () é usada para criar objetos primitivos encaixotados, os objetos usados são armazenados em cache. Desde java armazena números inteiros a partir de: 128-128, estes objetos em cache podem se comportar de forma diferente.

Degradação do desempenho do - Autoboxing ou unboxing diminui a performance de um aplicativo, pois cria um objeto indesejado que leva o GC para executar mais freqüência.

Desvantagem de Autoboxing

Embora autoboxing tenha várias vantagens, ele possui a seguinte desvantagem:

Caso o Autoboxing aconteça dentro de um loop de objetos desnecessários, pode diminuir o desempenho da aplicação. Considere o seguinte código:

Listing 8: Exemplo de código mostrando problema de desempenho

```
public int sumEvenNumbers(List<Integer> intList) {

    int sum = 0;

    for (Integer i: intList)

        if ( i % 2 == 0 )

            sum += i;

        return sum;

}
```

Neste pedaço de código, soma + = i; irá expandir como soma = soma + i;. Desde o '+' operação não pode ser feita em objeto Integer, a JVM dispara o unboxing da soma Integer Objeto e, em seguida, o resultado é executa autoboxing volta.

Antes do JDK 1.5, os tipos de dados int e Integer eram distintos e em caso de sobrecarga de método estes dois tipos foram usados sem qualquer aborrecimento. Agora, com autoboxing e unboxing, este tornou-se mais complicado. Um exemplo disto é o método sobrecarregado em remover ArrayList. Classe ArrayList tem dois métodos de remover - remover (index) e remover (objeto). Neste caso, a sobrecarga de métodos não acontecerá e respectivo método será chamado com parâmetros apropriados.

Conclusão

Autoboxing é o mecanismo para converter um tipo de dados primitivo na respectiva embalagem ou objeto. O compilador usa valueOf () método para converter primitivo para Object e usa intValue (), doubleValue (), etc., para obter o valor primitivo do objeto. Em autoboxing, um boolean é convertido para booleano, byte a byte, char de caráter, as mudanças flutuador para Float, int vai para Integer, long vai convertidos curto eo longo para curto, enquanto que no unboxing a conversão acontece no sentido inverso.

Artigo traduzido e originalmente publicado em: http://mrbool.com/autoboxing-and-unboxing-in-java/28238