Java Developer

Java Core

Sumário

Objetivos deste Módulo

Ao final deste módulo, objetiva-se que o aluno adquira:

Conhecimentos:

- Wrapper Classes, unboxing, autoboxing
- Enum

1. Wrapper Classes

Em Java, as Wrapper Classes são classes que fornecem um invólucro (wrapper) para tipos de dados primitivos, permitindo que esses tipos primitivos sejam tratados como objetos. Isso é útil em situações em que você precisa trabalhar com tipos primitivos em contextos que exigem objetos, como quando se trabalha com coleções (por exemplo, ArrayList) ou ao usar funcionalidades avançadas de programação orientada a objetos.

As Wrapper Classes em Java são as seguintes:

Integer -> int

Double -> double

Float -> float

Character -> char

Byte -> byte

Short - > short

Long -> long

Boolean -> boolean

Essas classes fornecem métodos e funcionalidades adicionais para manipular e converter os tipos primitivos correspondentes. Além disso, elas também são usadas para representar nulos, uma vez que objetos podem ser nulos (null), ao contrário dos tipos primitivos.

Aqui está um exemplo de como você pode usar uma Wrapper Class, como Integer, para converter um valor inteiro primitivo em um objeto:

```
// “Embalar” -> colocar um primitivo dentro de um invólucro
int valorPrimitivo = 42;
Integer objetoInteiro = Integer.valueOf(valorPrimitivo); // Wrapping

// “Desembalar” -> remover um primitivo do invólucro
// Você pode agora usar objetoInteiro como um objeto Integer
int desembrulhado = objetoInteiro.intValue(); // Unwrapping
```

Wrapper Classes são especialmente úteis ao trabalhar com coleções genéricas, como Listas, onde apenas objetos podem ser armazenados. Elas permitem que tipos primitivos sejam usados em tais estruturas de dados sem problemas. Por exemplo:

```
ArrayList<Integer> lista = new ArrayList<>();
lista.add(5); // O valor 5 é automaticamente convertido em um objeto Integer
int valor = lista.get(0); // Você pode obtê-lo de volta como um valor primitivo
```

Wrapper Classes em Java são usadas para fornecer uma representação orientada a objetos dos tipos de dados primitivos, permitindo que você os utilize em contextos onde objetos são necessários, como coleções e manipulações de dados mais avançadas.

Autoboxing e **Unboxing**: Java oferece recursos de autoboxing e unboxing para simplificar a conversão entre tipos primitivos e suas Wrapper Classes.

O **autoboxing** permite que você atribua um valor primitivo a uma variável de Wrapper Class diretamente, enquanto o **unboxing** permite que você extraia o valor primitivo de um objeto Wrapper Class de forma transparente. Aqui está um exemplo:

Integer **objetoInteiro** = 42; // Autoboxing

int valorPrimitivo = **objetoInteiro**; // Unboxing

Comparação: Ao comparar valores armazenados em objetos Wrapper, é importante usar o método equals() em vez do operador ==. Isso ocorre porque == compara as referências dos objetos, enquanto equals() compara os valores contidos neles.

```
Integer a = 10;
Integer b = 10;
System.out.println(a == b); // true (devido a autoboxing e à cache de valores inteiros)
System.out.println(a.equals(b)); // true

Integer c = new Integer(10);
Integer d = 10;
System.out.println(c == d); // false
System.out.println(c.equals(d)); // true
```

Nulos (null): Uma característica importante das Wrapper Classes é que elas podem ser nulas (null). Isso é útil quando você precisa representar a ausência de um valor. Por exemplo:

```
Integer valor = null;
if (valor == null) {
    System.out.println("O valor é nulo.");
} else {
    System.out.println("O valor não é nulo.");
}
```

Conversão: Você pode converter entre tipos de Wrapper Classes usando métodos de conversão ou construtores apropriados. Por exemplo:

Integer numero = Integer.valueOf("42"); // Convertendo uma String em **Integer**
Double valorDouble = Double.valueOf(3.14); // Convertendo um double em **Double**

Desempenho: Embora as Wrapper Classes forneçam flexibilidade, é importante observar que usar tipos primitivos é geralmente mais eficiente em termos de desempenho, uma vez que não há sobrecarga

associada com a criação e manipulação de objetos. Portanto, ao trabalhar com grandes volumes de dados ou em cenários onde o desempenho é crítico, é recomendável usar tipos primitivos sempre que possível.

Wrapper Classes para Tipos Numéricos: As Wrapper Classes para tipos numéricos (Integer, Double, Float, Byte, Short, Long) também possuem **métodos** e **constantes** úteis. Por exemplo, você pode usar Integer.MAX_VALUE para obter o valor máximo que um int pode representar.

```
int maxInt = Integer.MAX_VALUE;
System.out.println("Máximo valor de int: " + maxInt);
```

Autoboxing e Unboxing em Expressões: Além de atribuições, autoboxing e unboxing também podem ocorrer em expressões, tornando a conversão entre tipos primitivos e Wrapper Classes ainda mais transparente.

```
Integer a = 5;
Integer b = 3;
Integer soma = a + b; // Autoboxing e soma de valores como objetos
int resultado = soma; // Unboxing para obter o valor primitivo
```

Uso de Wrapper Classes em Estruturas de Dados Complexas: Além de coleções, as Wrapper Classes são úteis em muitos outros cenários, como ao trabalhar com bibliotecas que exigem objetos em vez de tipos primitivos, ou ao modelar classes de domínio que precisam representar valores nulos.

Wrapper Classes para Booleanos: A classe Boolean é usada para envolver o tipo primitivo boolean. Ela também fornece métodos úteis para lidar com valores booleanos, como a conversão de strings em valores booleanos.

```
Boolean verdadeiro = Boolean.TRUE; // Representa o valor booleano true
Boolean falso = Boolean.FALSE; // Representa o valor booleano false

// Conversão de uma String em um valor booleano
Boolean valorBooleano = Boolean.valueOf("true");
```

Wrapper Classes Imutáveis: As Wrapper Classes são imutáveis, o que significa que, após criadas, seus valores não podem ser alterados. Qualquer operação que pareça modificar um objeto Wrapper na verdade cria um novo objeto.

```
Integer a = 10;
Integer b = a + 5; // Isso cria um novo objeto Integer, não modifica 'a'
System.out.println(a); // Ainda será 10
System.out.println(b); // Será 15
```

Atentar a alguns castings:

```
// primitivos
double d = 10.3;
float a = 10.3f;
```

```
    // wrapper
    Integer numero = 102;
    Long outroNumero = 102l;
    Long outroNumero2 = 102L;
    Long valor = Long.valueOf(102);

    Double numero2 = 10.3;
    Float numero3 = 10.3f;
    Float numero4 = 10.3F;
    Float numero5 = Float.valueOf(10.3f);
```

2. Enum

Em Java, "**enum**" é uma palavra-chave usada para criar um tipo de dado enumerado, que é uma lista fixa de valores constantes. Os enums são úteis quando você deseja representar um conjunto predefinido de valores que não devem ser alterados durante a execução do programa. Por exemplo, você pode usar enums para representar os dias da semana, os meses do ano, os tipos de cartão de crédito, etc.

Aqui está um exemplo simples de como definir e usar um enum em Java:

```
public enum DiaDaSemana {
    DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA, SABADO
}


public class ExemploEnum {

    public static void main(String[] args) {

        DiaDaSemana dia = DiaDaSemana.SEGUNDA;

        // Você pode usar um enum em estruturas de controle, como um switch
        switch (dia) {
            case DOMINGO:
                System.out.println("Hoje é domingo.");
                break;
            case SEGUNDA:
                System.out.println("Hoje é segunda-feira.");
                break;
            // Outros casos para os outros dias da semana...
        }

    }
}
```

Alguns conceitos importantes relacionados aos enums em Java:

Valores Enumerados:

Os valores em um enum são chamados de "valores enumerados" e são definidos como constantes em maiúsculas. No exemplo anterior, DOMINGO, SEGUNDA, TERCA, etc., são os valores enumerados.
Iteração em Enums:

Você pode iterar por todos os valores de um enum usando o método values(). Por exemplo:

```
for (DiaDaSemana dia : DiaDaSemana.values()) {
    System.out.println(dia);
}
```

Campos e Métodos em Enums:

Os enums podem conter campos e métodos como qualquer outra classe em Java. Isso permite adicionar funcionalidades específicas aos valores do enum. Aqui está um exemplo com um campo adicional:

```
public enum DiaSemanaEnum {

    DOMINGO("Domingo"), SEGUNDA("Segunda"), TERCA("Terça"), QUARTA("Quarta"), QUINTA("Quinta"), SEXTA("Sexta"), SABADO("Sábado");

    private final String nome;

    DiaSemanaEnum(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}
```

Comparação de Enums:

Você pode comparar enums usando o operador de igualdade (==) porque eles são constantes. Por exemplo:

```
DiaDaSemana dia1 = DiaDaSemana.SEGUNDA;
DiaDaSemana dia2 = DiaDaSemana.TERCA;

if (dia1 == dia2) {
    System.out.println("Os dias são iguais.");
} else {
    System.out.println("Os dias são diferentes.");
}
```

Usos Comuns:

Enums são comumente usados para representar estados, tipos, opções e valores fixos em um programa. Eles ajudam a tornar o código mais legível e a evitar o uso de valores mágicos (valores literais sem contexto) em seu código.
Em resumo, enums em Java são uma ferramenta poderosa para representar conjuntos fixos de valores

constantes, tornando o código mais seguro e legível. Eles podem conter campos e métodos personalizados, além de suportar iteração e comparação direta entre os valores enumerados. É uma prática recomendada usá-los sempre que você tiver um conjunto fixo de valores relacionados em seu programa.

Enums Ordinais:

Cada valor enum tem um número ordinal associado a ele, que é a posição do valor no conjunto enum (começando de 0). Por exemplo, DiaDaSemanaEnum.SEGUNDA.ordinal() retornaria 1, já que SEGUNDA é o segundo valor na enumeração.

3. Exercícios

1) Crie um programa que peça ao usuário para inserir um número inteiro e, em seguida, verifique se esse número é par ou ímpar usando Integer.

2) Desenvolva um programa que receba um número em ponto flutuante (float) do usuário e, em seguida, arredonde-o para o inteiro mais próximo usando Float e Math.round().

3) Crie um programa que peça ao usuário para inserir um caractere e, em seguida, verifique se é um espaço em branco usando Character. Atenção, um char é declarado com aspas simples e não aspas duplas como o String. Exemplo: char a = ‘a’

4) Estações do Ano: Crie um enum chamado EstacaoAnoEnum que represente as quatro estações do ano: PRIMAVERA, VERAO, OUTONO e INVERNO. Escreva um programa que, com base em um mês informado pelo usuário (1 a 12), determine a estação do ano correspondente.

5) Opções de Menu: Crie um enum chamado OpcaoMenuEnum que represente as opções de um menu de restaurante, como BEBIDAS, PRATO_PRINCIPAL, SOBREMESA e SAIR. Em seguida, escreva um programa que permita ao usuário selecionar uma opção do menu e exiba uma mensagem correspondente.