fusberti@ic.unicamp.br



# Métodos e Variáveis

prof. Fábio Luiz Usberti

MC322 - Programação Orientada a Objetos

Instituto de Computação - UNICAMP - 2014


# Sumário

# Chamada de métodos

## Modos de chamar um método

Existem três formas de chamar um método:

1. Utilizando uma variável que faz referência a um objeto, seguido de um ponto (.) e o nome de um método não estático.

```
GradeBook gradeBook1 = new GradeBook( "MC302 Programação Orientada a Objetos" );
myGradeBook.displayMessage();
```

2. Utilizando o nome de uma classe, seguida de um ponto (.) e o nome de um método estático.

```
Math.sqrt(900.0)
```

3. Dentro de um método, chamar outro método da mesma classe diretamente por seu nome.

```
double result = maximum(number1, number2, number3);
```

# Chamada de métodos

## Visibilidade de métodos estáticos

- Um método estático não consegue enxergar métodos e atributos não estáticos da classe.
- Para que um método consiga acessar um método ou atributo não estático da classe, é necessário que o método tenha a referência de um objeto.
- Justificativa: supondo que um método estático chame um método não estático diretamente. Como saber a qual objeto o método estático está se referindo? E se nenhum objeto da classe existir no momento da chamada?

# Chamada de métodos

## Visibilidade de métodos estáticos

Resumo das visibilidades entre métodos e atributos de instância e de classe.

- Métodos de instância *podem* acessar variáveis de instância e métodos de instância diretamente.
- Métodos de instância *podem* acessar variáveis de classe e métodos de classe diretamente.
- Métodos de classe *podem* acessar variáveis de classe e métodos de classe diretamente.
- Métodos de classe não podem acessar variáveis de instância e métodos de instância diretamente. É necessário o uso de uma referência a um objeto. Além disso, métodos de classe não podem utilizar a palavra-chave `this` dado que não há nenhuma instância associada.

# Parâmetros

## Parâmetros e argumentos

- Parâmetros referem-se à lista de variáveis declarada em um método.
- Argumentos referem-se aos valores que são passados em uma chamada de função.
- Quando um método é chamado, os argumentos precisam ser correspondentes aos tipos e à ordem dos parâmetros declarados.
- Os tipos dos parâmetros podem ser primitivos ou referenciados.
- A linguagem Java não permite a passagem de métodos como parâmetro, no entanto, é possível passar um objeto contendo métodos que poderão ser chamados.

# Parâmetros

## Passagem de tipos primitivos

- Em Java, a passagem de argumentos de tipos primitivos é realizada por valor.
- Isso implica que qualquer mudança de valor dos parâmetros persiste somente no escopo do método.
- Quando o método retorna, os parâmetros são liberados da memória e os valores dos argumentos são recuperados para seus valores iniciais.

# Parâmetros

## Passagem de tipos primitivos

```
// PassPrimitiveByValue.java
// Passagem de argumentos primitivos.
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 2;
        int y = 3;

        // passa a variável local x como argumento
        badSwap(x,y);

        System.out.println("Após chamada do método, x = " + x + " e y = " + y);

    } // fim método main

    // mudança de valor do parâmetro p
    public static void badSwap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    } // fim método passMethod(int)
} // fim classe badSwap
```

# Parâmetros

## Passagem de tipos referenciados

- Em Java, a passagem de argumentos de tipos referenciados, como objetos e vetores, também é realizada por valor.
- Isso significa que após o retorno do método o argumento vai continuar referenciando o mesmo objeto de antes da chamada do método.
- Ainda assim, os atributos do objeto referenciado poderão ter seus valores modificados, se seus modificadores de acesso assim o permitirem.

# Parâmetros

## Passagem de tipos referenciados

```
// Point.java
// Passagem de argumentos referenciados.
public class Point {

    // classe que representa um ponto no espaço 2D discreto
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    } // fim construtor

    // tenta realizar (sem sucesso) uma troca entre os objetos
    public static void badSwap(Point pnt1, Point pnt2) {
        pnt1.x = 100;
        pnt1.y = 100;
        Point temp = pnt1;
        pnt1 = pnt2;
        pnt2 = temp;
    } // fim método badSwap

    public static void main(String[] args) {
        Point pnt1 = new Point(0, 0);
        Point pnt2 = new Point(1, 1);
        System.out.println("Ponto 1 - X: " + pnt1.x + " Y: " + pnt1.y);
        System.out.println("Ponto 2 - X: " + pnt2.x + " Y: " + pnt2.y);
        System.out.println("Após o método badSwap: ");
        badSwap(pnt1, pnt2);
        System.out.println("Ponto 1 - X: " + pnt1.x + " Y: " + pnt1.y);
        System.out.println("Ponto 2 - X: " + pnt2.x + " Y: " + pnt2.y);
    } // fim main
} // fim classe Point
```

# Promoção de argumentos

## Promoções válidas

- Uma característica da chamada de métodos corresponde à promoção de tipo dos argumentos.
- Uma promoção de tipo corresponde a tomar o valor da variável de um certo tipo e transformá-lo para o tipo correspondente ao parâmetro do método chamado. Exemplo:

```
Math.sqrt( 4 );
```

- O método `sqrt()` da classe Math está declarado com um parâmetro `double` , ainda assim o compilador realiza corretamente a conversão do inteiro `4` para um `double` e a função retorna `2.0` .

# Promoção de argumentos

## Promoções válidas

- Existe um conjunto de promoções válidas de tipo que são realizadas automaticamente pelo compilador.
- As promoções de tipo também se aplicam a expressões contendo valores de dois ou mais tipos primitivos.
- No caso de expressões, cada valor é sempre promovido para o tipo mais alto.

| Tipo | Promoções válidas |
|---|---|
| `double` | None |
| `float` | `double` |
| `long` | `float` ou `double` |
| `int` | `long`, `float` ou `double` |
| `char` | `int`, `long`, `float` ou `double` |
| `short` | `int`, `long`, `float` ou `double` (mas não `char`) |
| `byte` | `short`, `int`, `long`, `float` ou `double` (mas não `char`) |
| `boolean` | Nenhuma (os valores `boolean` não são considerados como números em Java) |

# Promoção de argumentos

## Promoções inválidas

- Conversões de tipo fora do conjunto válido podem provocar erros de compilação ou perda de informações.
- Rebaixar um tipo `double` para um tipo `int` trunca a parte fracionária e portanto ocorre perda de informação.
- Nas conversões fora do conjunto válido, o compilador exige um operador denominado cast para explicitamente indicar que uma conversão está sendo feita.

```
double valDouble = (1.0+Math.sqrt(5))/2.0;
int  valInt  = (int) valDouble; // parte fracionária é truncada
```

- O operador cast informa ao compilador que o programador está ciente de que a conversão pode causar perda de informação.

# Escopos

## Escopo de entidades

- O escopo de declaração de uma classe, método ou variável refere-se à porção do código onde a entidade declarada pode ser chamada pelo seu nome.
- O Java possui as seguintes regras básicas de escopo:
  - O escopo de um parâmetro se encontra no corpo do método onde ele está declarado.
  - O escopo de uma variável local se inicia no ponto de sua declaração até o final do bloco (método, laço, desvio condicional) que a contém.
  - O escopo de um método ou atributo se define pelo corpo da classe onde eles estão declarados.
- Se uma variável local ou parâmetro tem o mesmo nome de um atributo, então o atributo é ocultado em um processo denominado sombreamento.

# Escopos

## Escopo de entidades

```
// Scope.java
// Demonstra o escopo de atributos e variáveis locais.
public class Scope {

        private static int x = 1;          // variável de classe privada

        public static void main(String[] args) {

                useLocalVariable();
                System.out.println();
                useLocalVariable();

        } // fim main

        // a cada chamada do método useLocalVariable() uma nova variável local x é criada
        public static void useLocalVariable() {

                int x = 25; // variável local que causa o sombreamento da variável de classe

                System.out.printf("ANTES: variável local x = %d\n", x);
                ++x;
                System.out.printf("DEPOIS: variável local x = %d\n", x);

                System.out.printf("ANTES: variável da classe x = %d\n", Scope.x);
                Scope.x *= 10;
                System.out.printf("DEPOIS: variável da classe x = %d\n", Scope.x);
        } // fim método useLocalVariable

} // fim classe Scope
```

# Sobrecarga de métodos

## Polimorfismo ad hoc

- Métodos com o mesmo nome podem ser declarados em uma mesma classe, processo denominado sobrecarga de métodos, contanto que eles tenham listas de parâmetros distintas.
- As listas de parâmetros devem ser distintas pelo número, tipos e/ou ordem dos parâmetros.
- Quando um método sobrecarregado é chamado, o compilador seleciona o método adequado examinando o número, tipos e ordem dos argumentos na chamada.
- Em geral, sobrecarga de métodos é comum para criar múltiplos métodos que realizam tarefas similares mas sobre parâmetros de diferentes tipos e em diferentes quantidades.

# Sobrecarga de métodos

## Polimorfismo ad hoc

- O tipo de retorno dos métodos não entra como um critério de diferenciação entre métodos, portanto declarar mais de um método com a mesma lista de parâmetros, ainda que com tipos de retorno distintos, resulta em erro de compilação.
- O método `Math.max()` foi sobrecarregado para tratar quatro tipos de parâmetros:

```
static double max(double a, double b)
static float  max(float a, float b)
static int    max(int a, int b)
static long   max(long a, long b)
```

# Sobrecarga de métodos

## Polimorfismo ad hoc

```
// MethodOverload.java
// Declaração de métodos sobrecarregados.
public class MethodOverload {

        public static void main(String[] args) {
                System.out.printf("O quadrado de 7 é %d\n", square(7));
                System.out.printf("O quadrado de 7.5 é %f\n", square(7.5));
        } // end main

        // eleva ao quadrado com parâmetro int
        public static int square(int intValue) {
                System.out.printf("\nChamando método com parâmetro int: %d\n", intValue);
                return intValue * intValue;
        } // fim método square(int)

        // eleva ao quadrado com parâmetro double
        public static double square(double doubleValue) {
                System.out.printf("\nChamando método com parâmetro double: %f\n", doubleValue);
                return doubleValue * doubleValue;
        } // fim método square(double)
} // fim classe MethodOverload
```

## Impressão no terminal:

```
Chamando método com parâmetro int: 7
O quadrado de 7 é 49
Chamando método com parâmetro double: 7.500000
O quadrado de 7.5 é 56.250000
```

# Classes Empacotadoras

## Objetos para tipos primitivos

- Todo tipo primitivo possui uma classe empacotadora associada no pacote `java.lang`.
- O nome classe empacotadora é devido aos tipos primitivos serem empacotados dentro de um objeto.
- Existem oito classes empacotadoras: `Boolean, Byte, Character, Double, Float, Integer, Long and Short`.
- Os objetos instanciados são imutáveis, ou seja, o primitivo empacotado é declarado como `final`.

# Classes Empacotadoras

## Objetos para tipos primitivos

```
// WhyImmutable.java
// Demonstrando porque classes wrappers e String são imutáveis
class WhyImmutable {
    public static void main(String[] args) {
        String name = "Pedro";
        Double sal = 1000.00;
        displayTax(name, sal);
        System.out.println("Nome: " + name + ". Salário: R$" + sal);
    }

    static void displayTax(String name, Double num) {
        name = "Olá " + name.concat("!");
        num = num * 30 / 100;
        System.out.println(name + " Seus impostos somaram $" + num);
    }
}
```

## Impressão no terminal:

```
Olá Pedro! Seus impostos somaram $300.0
Nome: Pedro. Salário: R$1000.0
```

# Classes Empacotadoras

## Objetos para tipos primitivos

- Há pelo menos três razões para utilizar as classes empacotadoras:
  1. Métodos cujos argumentos são objetos.
  2. Utilizar as constantes definidas pela classe, como `MIN_VALUE`, `MAX_VALUE` que fornecem os limites inferior e superior de cada tipo.
  3. Conversão de valores entre diferentes tipos primitivos, conversão de e para `String`, conversão entre bases numéricas (binária, decimal, octal, hexadecimal).

# Classes Empacotadoras

## Embalamento e desembalamento automáticos

- O embalamento automático (autoboxing) consiste em uma conversão realizada pelo compilador de um tipo primitivo para sua classe empacotadora correspondente. Por exemplo, a conversão de um valor `int` para um objeto `Integer`, ou de um valor `double` para um objeto `Double`, e assim sucessivamente.
- Um autoboxing é efetuado quando um tipo primitivo é passado como argumento de um método cujo parâmetro é o objeto empacotador correspondente ou quando um tipo primitivo é atribuído para uma variável que referencia uma classe empacotadora.

# Classes Empacotadoras

## Embalamento e desembalamento automáticos

```
Character ch = 'a'; // autoboxing
```

- O literal `'a'` do tipo primitivo `char` está sendo atribuído a um objeto. A princípio essa instrução deveria dar um erro de compilação por incompatibilidade de tipos, no entanto isso não ocorre pois o compilador transforma a instrução acima em:

```
Character ch = Character.valueof('a');
// outra instrução equivalente seria:
// Character ch = new Character('a');
```

# Classes Empacotadoras

## Autoboxing e unboxing

- A conversão no sentido contrário, de Objeto empacotador para o tipo primitivo, é denominado desembalamento (unboxing).
- O compilador Java realiza unboxing quando um objeto empacotador é passado como parâmetro de um método que requer um tipo primitivo (correspondente) ou quando é atribuído a uma variável primitiva (correspondente).

```
Double piObject = new Double(3.14);
double piValue = piObject;   //  unboxing
```

- Na instrução acima um objeto `Double` é atribuído para um tipo primitivo `double`, o que também deveria causar um erro de compilação.
- O compilador, no entanto, converte a instrução acima da seguinte forma:

```
double piValue = piObject.doubleValue();
```

# Classe Random

## Gerador de números pseudo-aleatórios

- Um gerador de números pseudo-aleatórios é um programa computacional que gera uma sequência de números ou símbolos que não apresentam padrão, ou seja, aparenta ser aleatória.
- Muitas aplicações que utilizam de aleatoriedade motivaram o desenvolvimento de diferentes métodos para a geração de números aleatórios.
- Em computação, aplicações que requerem aleatoriedade envolvem simulações de sistemas reais, criptografia, jogos e amostragem estatística.
- Um elemento aleatório pode ser introduzido em um aplicativo java por duas formas:
  - O método `Math.random()` produz valores `double` no intervalo [0, 1).
  - Um objeto da classe `Random` (no pacote `java.util`) pode produzir aleatoriedade para os tipos `boolean`, `byte`, `float`, `double`, `int`, `long` e distribuição Gaussiana para o tipo `double`[1].

[1] Documentação: http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

# Classe Random

## Gerador de números pseudo-aleatórios

- A seguinte instrução instancia um gerador de números pseudo-aleatórios:

```
Random randomNumbers = new Random();
```

- Quando um programa está sendo depurado, em geral é útil que a sequência de números pseudo-aleatórios gerada seja repetível.
- A repetibilidade da sequência permite testar sua aplicação para uma única sequência de números aleatórios antes de testar com diversas outras sequências.
- Para atingir repetibilidade, o objeto `Random` deve ser criado com uma semente inicial `seedValue` de tipo `long`.

```
Random randomNumbers = new Random(seedValue);
```

- Se uma semente inicial não for passada como argumento, então o construtor adota uma semente inicial gerada com base na hora atual.

# Classe Random

## Gerador de números pseudo-aleatórios

- Uma vez instanciado o objeto `Random` números aleatórios do tipo `int` são obtidos com o seguinte método:

```
int randomValue = randomNumbers.nextInt();
```

- Esse método retorna um valor `int` no intervalo $[-2.147.483.648, +2.147.483.647]$.
- Para obter valores em algum intervalo definido por $[minValue, maxValue] \subset [-2.147.483.648, +2.147.483.647]$, basta adaptar a instrução da seguinte forma:

```
int randomValue = minValue + randomNumbers.nextInt(maxValue-minValue+1);
```

# Classe Random

## Gerador de números pseudo-aleatórios

```
// RollDie.java
// Lança um dado de seis faces 6,000,000 de vezes.
import java.util.Random;

public class RollDie {
        public static void main(String[] args) {
                Random randomNumbers = new Random(); // gerador de números pseudo-aleatórios

                // conta a frequência de cada face
                int frequency1 = 0, frequency2 = 0, frequency3 = 0, frequency4 = 0, frequency5 = 0, frequency6 = 0;
                int face; // face atual do dado

                for (int roll = 1; roll <= 6000000; roll++) {
                        face = 1 + randomNumbers.nextInt(6); // lança o dado
                        // determina a face em que o dado caiu
                        switch (face) {
                                case 1: ++frequency1; break;
                                case 2: ++frequency2; break;
                                case 3: ++frequency3; break;
                                case 4: ++frequency4; break;
                                case 5: ++frequency5; break;
                                case 6: ++frequency6; break;
                        } // fim switch
                } // fim for
                System.out.println("Frequência das faces do dado:"); // output headers
                System.out.printf("1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",frequency1, frequency2,
                                frequency3, frequency4, frequency5, frequency6);
        } // fim main
} // fim classe RollDie
```

# Referências

1. Java: Como Programar, Paul Deitel & Heivey Deitel; Pearson; 7a. Ed. (no. chamada IMECC – 05.133 D368j)
2. Data Structures and Algorithms with Object Oriented Design Patterns in Java, Bruno Preiss; (http://www.brpreiss.com/books/opus6/)
3. The Java Tutorials (Oracle) (http://docs.oracle.com/javase/tutorial/)
4. Guia do Usuário UML, Grady Booch et. al.; Campus(1999)
5. Java Pocket Guide - Robert Liguori & Patricia Liguori; O'Reilley, 2008.