Programação Orientada a Objetos

Alexandre César Muniz de Oliveira

Métodos e Atributos

Parte III

Métodos

- [mod] tipo nome ([tipo arg]) [throws exc]{
  - [mod]: zero ou mais modificadores separados por espaços
  - Tipo: tipo de dados retornado pelo método
  - Identificador: nome do método
  - [arg]: zero ou mais argumentos, com tipo declarado, separados por vírgula
  - [throws exceção]: declaração de exceções

Métodos

- Chamada a métodos pode resultar em um valor de retorno
- Funciona como uma expressão com um tipo particular de dados

```
int fewerSticks = game1.getSticks() - 1;
boolean done = game1.gameOver();
System.out.println(game1.getPlayer());
game1.getSticks();
```

Acesso e modificação

- Recuperar e armazenar dados em objetos
- Métodos públicos em detrimento a atributos públicos
- Métodos *set* e *get* armazenam e recuperam valores em/de variáveis de instâncias privadas.
- Exemplo: Ponto, Racional, etc
- Métodos que
  - modificam são chamados *modificadores*,
  - recuperam são chamados *de acesso*,
  - Ambos públicos.

Acesso e modificação

- Convenção para uma variável:
  - *getVarName()* e
  - *setVarName()*.

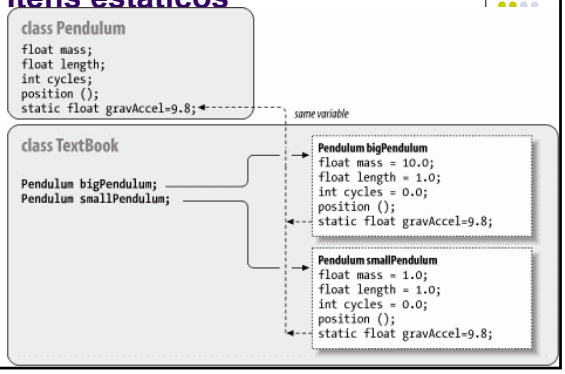
```
public int getSticks()
{   return nSticks;
}

public int getPlayer()
{   return player;
}
```

Itens estáticos

- Cada objeto tem valores específicos para seus membros
- O qualificador *static* em um atributo permite que o valor dele seja único para todas as instâncias de uma classe
- Funciona como se o membro fosse global para todos os objetos criados
- Atributos/métodos estáticos são também chamados de atributos/métodos de classe;

Itens estáticos



Itens estáticos

```
class Pendulum {
   ...
   static float gravAccel = 9.80;
   ...

class Pendulum {
   ...
   static final float EARTH_G = 9.80;
   ...

Pendulum.gravAccel = 8.76;
```

Itens estáticos

- Métodos estáticos, da mesma forma, podem ser acessados diretamente através da classe;
- Um método estático pode ser acessado somente por outro membro estático da classe
- A ordem de definição afeta a visibilidade
- Métodos estáticos não podem acessar diretamente variáveis de instância ou métodos de instância

Itens estáticos

- Exemplos
  - Math.sqrt(),
  - System.out.println ()
  - …

```
class Bird {
   ...
   static String [] getBirdTypes( ) {
     String [] types;
    // Create list...
       return types;    }
   }
…
String [] names = Bird.getBirdTypes( );
```

Itens estáticos

- *Membros de instância só existem se houver um objeto*

main() não faz parte do objeto!

:Circulo
+x: 0
+y: 0
+raio: 0
area():double

Errado!

```
public class Circulo {
   public int raio;
   public int x, y;
   public double area() {
      return Math.PI * raio * raio;
   }
   public static void main(String[] a){
      raio = 3;
      double z = area();
   }
}
```

membros de instância
Pode. Porque area() faz parte do objeto!
qual raio? existe?
qual area? existe?
Não pode. Não existe objeto em main()!

Certo!

```
public class Circulo {
   public int raio;
   public int x, y;
   public double area() {
      return Math.PI * raio * raio;
   }
   public static void main(String[] a){
      Circulo c = new Circulo();
      c.raio = 3;
      double z = c.area();
   }
}
```

tem que criar pelo menos um objeto!
raio de c
area() de c

Escopo

- Referência a contexto (método, bloco, etc) em que uma variável ou método pode ser usado (visualizado)

Escopo

- Parâmetros
  - Escopo local
    - similar ao escopo de variáveis locais
    - limitado ao corpo do método
- Variáveis de instância, de classe e métodos
  - Escopo de classe
    - extensivo a toda a classe.
    - usado e visualizado em todo método da classe
  - Exceções ...

Escopo

- Exceção
  - **Variáveis e métodos de instância** não podem ser usados dentro de **métodos de classe** (métodos estáticos)
  - Necessidade de instanciação de um objeto e acesso ao método e/ou atributo em questão
- Qualificação de nomes
  - Na mesma classe, referência apenas ao nome, sem qualificação

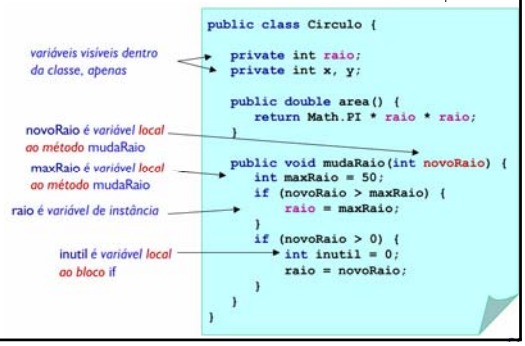
Escopo

- Visibilidade de classes:
  - **public** – indica que o conteúdo público da classe pode ser usado livremente por outras classes do **pacote** ou de outro **pacote**
  - **private** – só é permitido seu uso para aninhar classes, fazendo com que uma classe seja interna a outra

Escopo

- Visibilidade de atributos e métodos
  - O acesso a atributos e métodos é controlado através de modificadores **public**, **private** e **protected** (posteriormente)
  - Membros **public** permitem o acesso ao membro a partir de outras classes
  - Membros **private** não permite à classes clientes ter acesso ao membro; somente a própria classe que contêm este membro pode acessá-lo.
  - Restringir o acesso a membros através da cláusula **private** é chamado encapsulamento

Escopo



Parâmetros

- Também chamado de *Parâmetro Formal*
- Variável usada para passar informação para um método quando este é invocado.
- Tipo e nome devem aparecer na lista de parâmetros
- Armazena temporariamente o valor a ser passado para o método.

Argumento

- Parâmetro se refere ao parâmetro formal
- Argumento se refere ao atual valor passado para o método quando de sua invocação
- O tipo do argumento deve ser compatível com o parâmetro formal definido para um dado método
- Linguagens fortemente tipadas fazem verificação de tipos em tempo de compilação e em tempo de execução

Construtor

- Métodos modificadores usados para iniciar objetos, atribuindo valores iniciais às variáveis de instância
- Definidos com o mesmo nome da classe e não podem declarar nenhum tipo de retorno
- Não são considerados membros da classe

Construtor

- Exemplos

```
public OneRowNim()
{   nSticks = 7;
    player = 1;
}
OneRowNim game1 = new OneRowNim();
OneRowNim game2 = new OneRowNim();
```

Construtor

- Exemplos

```
public OneRowNim(int sticks)
{   nSticks = sticks;
}
OneRowNim game3 = new OneRowNim(21);
OneRowNim game4 = new OneRowNim(13);
```

Construtor

- *Default*
  - Automaticamente provido
  - Não recebe parâmetros
  - Se a classe for pública
    - Construtor default é público
    - Acessível por outros objetos

```
public OneRowNim() {}
…
OneRowNim game5 = new OneRowNim();
```

Construtor

- Redundância e flexibilidade

OneRowNim
− nSticks : int − player : int
+ OneRowNim() + OneRowNim(in sticks : int) + OneRowNim(in sticks : int, in starter : int) + takeSticks(in num : int) + getSticks() : int + getPlayer() : int + gameOver() : boolean + report()

Sobrecarga

- Tipo de polimorfismo
  - O termo polimorfismo é originário do grego e significa "muitas formas" (poli = muitas, morphos = formas).
  - Permite que a semântica de uma interface seja efetivamente separada da implementação que a representa.
  - Formas abstratas e formas concretas
- Operador associado a mais de uma operação

Sobrecarga

- Na prática, permite a existência de vários métodos de mesmo nome
- **Métodos** diferem apenas por suas "**assinaturas**"
- Assinaturas variando no **número** e **tipo de argumentos** e no **valor de retorno**.
- O compilador escolhe de acordo com as listas de argumentos os procedimentos ou métodos a serem executados.

Sobrecarga

- Construtores sobrecarregados

```
public OneRowNim() {} // Constructor #1
public OneRowNim(int sticks)   // Constructor #2
{  nSticks = sticks;
}
```

- *Assinatura:*
  OneRowNim()
  OneRowNim(int)

Sobrecarga

- *Em Java, considera como assinatura:*
  - *nome do método,*
  - *Número de parâmetros,*
  - *tipos de parâmetros,*
  - *ordem de declaração*
- *Não pode existir dois métodos com mesma assinatura na mesma classe*

Sobrecarga

- Exemplos

```
public OneRowNim(int sticks, int starter)
{  nSticks = sticks;   // Set the number of sticks
   player = starter;   // Set who starts
}

OneRowNim game5 = new OneRowNim(14, 2);
OneRowNim game6 = new OneRowNim(31, 1);
```

Sobrecarga

- Exemplos

```
OneRowNim game1 = new OneRowNim();
OneRowNim game2 = new OneRowNim(21);
OneRowNim game3 = new OneRowNim(19, 2);

 OneRowNim game4 = new OneRowNim("21");
OneRowNim game5 = new OneRowNim(12, 2, 5);
                                Error     ^
```

Sobrecarga

- Exemplos

```
...
public void paint (Graphics g){
  int x = 10;
  g.drawString(x, x*2, "Booo!");
}
...
```

```
class T1 {
  private int a; private int b;
  public int soma () {
    return a + b;
  }
}
```

```
class T2 {
  int x, y;
  public int soma () {
    return x + y;
  }
  public static int soma(int a, int b){
    return a + b;
  }
  public static int soma(int a,
                         int b, int c){
    return soma(soma(a, b), c);
  }
}
```

Sobrecarga

```
public class Sobrecarga {
    public static void main (String args[]) {
        System.out.println (“área de um quadrado... ” + area(3));
        System.out.println (“área de um retângulo... ” + area(3,2));
        System.out.println (“área de um cubo... ” + area(3,2,5));
    }
    public static double area (int x) {
        return x * x;
    }
    public static double area (int x, int y) {
        return x * y;
    }
    public static double area (int x, int y, int z) {
        return x * y * z;
    }}
```

Destrutores

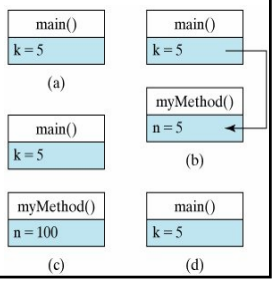
- Construtores alocam recursos durante a inicialização dos objetos
- Para efetuar a liberação destes recursos podem ser utilizados destrutores de classe
- Destrutores são invocados quando um objeto é removido da memória
  - Problema: em Java não é possível saber quando isto ocorrerá, por conta da ***coleta de lixo*** (*garbage collection)*
- Um destrutor deve ser implementado através do método:
  - void finalize ();
- A coleta de lixo pode ser “forçada” através da chamada ao método System.gc()

Valor e referência

- Passagem de parâmetros por valor
  - Valor primitivo (int, boolean, double)

```
public class PrimitiveCall
{
  public static void myMethod(int n)
  {  System.out.println("myMethod: n= " + n);
     n = 100;
     System.out.println("myMethod: n= " + n);
  }

  public static void main(String argv[])
  {  int k = 5;
     System.out.println("main: k= " + k);
     myMethod(k);
     System.out.println("main: k= " + k);
  }
}
```



Valor e referência

- Passagem de parâmetros por referência
  - Funciona como se o próprio objeto e não sua cópia fosse passado como parâmetro
  - A referência é uma abstração que equivale a um ponteiro
  - O argumento é uma referência ao objeto original
  - O objeto propriamente não é passado, pois seria bastante ineficiente
  - O objeto original poderá ser alterado de dentro do método em que ele for passado por referência

Valor e referência

```
public class ReferenceCall
{
   public static void myMethod(OneRowNim g)
   {  System.out.print("myMethod: Number of sticks: ");
      System.out.println(g.getSticks());
      g.takeSticks(3);
      System.out.print("myMethod: Number of sticks: ");
      System.out.println(g.getSticks());
   } // myMethod()
   public static void main(String argv[])
   {  OneRowNim game = new OneRowNim(10);
       System.out.print("main: Number of sticks: ");
       System.out.println(game.getSticks());
       myMethod(game);
       System.out.print("main: Number of sticks: ");
       System.out.println(game.getSticks());
}
```

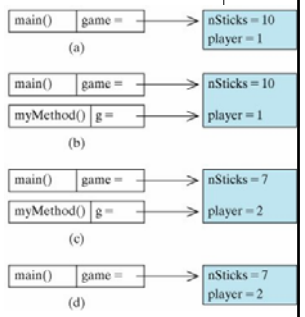
Valor e referência

main: Number of sticks: 10

myMethod: Number of sticks: 10

myMethod: Number of sticks: 7

main: Number of sticks: 7



Exemplos

- Inicialização de atributos de classe:

```
public class Produto {
   public static int total = 0;
   public int serie = 0;
   public Produto() {
      serie = total + 1;
      total = serie;
   }
}
```

```
public class Livro {
   private String titulo;
   public Livro() {
      titulo = "Sem titulo";
   }
   public Livro(String umTitulo) {
      titulo = umTitulo;
   }
}
```

Exemplos

```
public class Casa {
   private Porta porta;
   private int numero;
   public java.awt.Color cor;
   public Casa() {
      porta = new Porta();
      numero = ++contagem * 10;
   }
   public void abrePorta() {
      porta.abre();
   }
   public static String arquiteto = "Zé";
   private static int contagem = 0;
   static {
      if ( condição ) {
         arquiteto = "Og";
      }
   }
}
```

Atributos de instância: cada objeto poderá armazenar valores diferentes nessas variáveis.

Procedimento de inicialização de ***objetos*** *(Construtor):* código é executado após a criação de cada novo objeto. Cada objeto terá um número diferente.

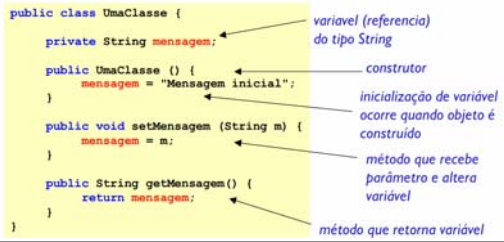
Método de instância: só é possível chamá-lo se for através de um objeto.

Atributos estáticos: não é preciso criar objetos para usá-los. Todos os objetos os compartilham.

Procedimento de inicialização ***estático****:* código é executado uma única vez, quando a classe é carregada. O arquiteto será um só para todas as casas: ou Zé ou Og.

Exemplos

- *Exemplo de classe com um atributo de dados (variável), um construtor e dois métodos*



Exemplos

- **super() e this():**
  - Todo construtor chama algum construtor de sua superclasse
  - Por default, chama-se o construtor sem argumentos, através do comando super() (implícito)
  - Pode-se chamar outro construtor, identificando-o através dos seus argumentos (número e tipo) na instrução super()
  - super(), se presente, deve sempre ser a primeira instrução do construtor (substitui o super() implícito)

Exemplos

```
public class Livro {
   private String titulo;
   public Livro() {
      titulo = "Sem titulo";
   }
   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

```
public class Livro {
   private String titulo;
   public Livro() {
      this("Sem titulo");
   }
   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

Exemplos

Escreva uma classe Ponto

- *contém x e y que podem ser definidos em construtor*
- *métodos getX() e getY() que retornam x e y*
- *métodos setX(int) e setY(int) que mudam x e y*

Escreva uma classe Circulo, que contenha

- *raio inteiro e origem Ponto*
- *construtor que define origem e raio*
- *método que retorna a área*
- *método que retorna a circunferência*
- *use java.lang.Math.PI (Math.PI)*

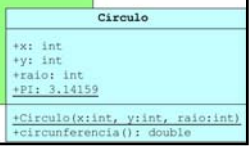
Crie um segundo construtor para Circulo que aceite um raio do tipo int e coordenadas x e y

Exemplos

```
public class Circulo {
   public int x;
   public int y;
   public int raio;
   public static final double PI = 3.14159;

   public Circulo (int x1, int y1, int r) {
      x = x1;
      y = y1;
      raio = r;
   }

   public double circunferencia() {
      return 2 * PI * raio;
   }
}
```

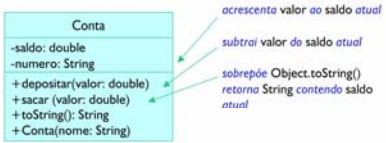


Exemplos

```
Circulo c1, c2, c3;
c1 = new Circulo(3, 3, 1);
c2 = new Circulo(2, 1, 4);
c3 = c1; // mesmo objeto!
System.out.println("c1: (" + c1.x + ", "
                           + c1.y + ")");
int circ = (int) c1.circunferencia();
System.out.print("Raio de c1: " + c1.raio);
System.out.println("; Circunferência de c1: "
                   + circ);
```

Exercícios

- *1. Classe Conta e TestaConta*
  - *a) Crie a classe **Conta**, de acordo com o diagrama UML abaixo*



  - *b) Crie uma classe **TestaConta**, contendo um método main(), e simule a criação de objetos Conta, o uso dos métodos depositar() e sacar() e imprima, após cada operação, os valores disponíveis através do método toString()*
  - *c) Gere a documentação javadoc das duas classes*