Programação por Objectos

Java

Parte 3: Métodos

Métodos (1)

Sintaxe

Qualif* Tipo Ident ([TipoP IdentP [, TipoP IdentP]*]) {
[Variável_local | Instrução]*
}

- Qualif: qualificador (visibilidade, entre outros)
- Tipo: tipo de retorno do método
- Ident: identificador do método
- TipoP: tipo dos parâmetros do método
- IdentP: identificador dos parâmetros do método
- { [Variável | Instrução]* } : corpo do método

Métodos (2)

- **Qualificadores de método**:
  - Visibilidade:
    - **public**: método acessível onde quer que a classe seja acessível.
    - **private**: método acessível apenas na classe.
    - **protected**: método acessível na classe, subclasses e classes no mesmo pacote.
  - **abstract**: método sem corpo.
  - **static**: método de classe.
  - **final**: método que não pode ser redefinido nas subclasses.

Métodos (3)

- No caso de omissão de um qualificador de visibilidade, o método é acessível na classe e classes no mesmo pacote.
- Com excepção dos qualificadores de visibilidade, um método pode ter mais do que um qualificador. Contudo, um método não pode ser ao mesmo tempo **abstract** e **final**.
- Um método estático só pode aceder a atributos estáticos e chamar métodos estáticos da classe.

Métodos (4)

- O tipo de retorno de um método é obrigatório, podendo ser:
  - tipo primitivo (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`)
  - referências (classes e interfaces definidas pelo Java, por exemplo, classe `String`, e classes e interfaces definidas pelo programador)
  - **`void`**
- Valor retornado pela instrução **`return`**.

Métodos (5)

- Um método pode ter zero, um, ou mais parâmetros:
  - Tipos possíveis de parâmetros:
    - tipo primitivo (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`)
    - referências (classes e interfaces definidas pelo Java, por exemplo, classe `String`, e classes e interfaces definidas pelo programador)

Métodos (6)

- O último parâmetro de um método pode ser declarado como uma **sequência de parâmetros**.
  - Permite que o método seja chamado com um número variável de argumentos (incluindo zero).
  - Métodos com sequência de parâmetros são denominados como **métodos de argumento variável**, contrastando com os **métodos de argumento fixo**.
  - A **aridade** de um método é dada pelo número de parâmetros do mesmo. Os métodos de argumento variável são métodos de **aridade variável**, enquanto que, os métodos de argumento fixo são de **aridade fixa**.
  - Definido com reticências (`...`) antes do tipo de parâmetro mas depois do seu identificador.

Métodos (7)

– Declarar um parâmetro como uma sequência de parâmetros é nada mais que pedir ao compilador para construir uma tabela com os elementos que se querem passar ao método.

```
public void imprimeAnimais(String... strings) {
   for (int i=0; i<strings.length; i++)
      System.out.println(strings[i]);
}
```

```
imprimeAnimais("Leão","Tigre","Urso");
```

imprime no terminal

```
Leão
Tigre
Urso
```

Métodos (8)

- Em Java, os parâmetros são sempre **passados por valor**:
  - O valor dos parâmetros são sempre cópias dos valores passados como argumento.
  - O método pode modificar o valor dos seus parâmetros sem afectar o respectivo valor no código que chamou o método.
  - Quando o parâmetro é uma referência para um objecto, é a referência para o objecto que é passada por valor.
    - O método pode alterar o objecto para quem o parâmetro faz referência sem afectar a referência que lhe foi passada.
    - Contudo, alterações a nível dos atributos do objecto recebido como parâmetro, ou chamada a métodos que alterem o estado do objecto recebido como parâmetro, alteram o objecto em todos as partes do programa que tenham uma referência para o mesmo.
  - Um parâmetro pode ser declarado **final**, quando não se quer que o seu valor seja alterado no corpo do método.

Métodos (9)

```
public class Demo {
   static void foo(int arg1) {
      arg1 = 10;
   }
   public static void main(String args[]){
      int arg = 5;
      foo(arg);
      System.out.println("arg = " + arg);
   }
}
```

No terminal é impresso `arg = 5`

Métodos (10)

```
public class Demo {
   public static void foo(Conta c) {
      c.quantia += 10000;
   }
   public static void main(String args[]){
      Conta arg = new Conta(“João Pires”,250);
      foo(arg);
      System.out.println(“arg.quantia = " + arg.quantia);
   }
}
```

No terminal é impresso `arg.quantia = 10250`

Métodos (11)

```
public static void imprimeAnimais(final int i, final String... s) {
   for (; i<s.length; i++)
       System.out.println(s[i]);
}
```

final parameter i may not be assigned
for (; i<strings.length; i++)
1 error

```
public static void imprimeAnimais(final int i, final String... s) {
   s[i]="Foca";
   for (int j=i; j<s.length; j++)
       System.out.println(s[j]);
}
```

```
imprimeAnimais(1,"Leão","Tigre","Urso");
```

imprime no terminal

```
Foca
Urso
```

Métodos (12)

- Um método de uma classe é acedido pelo operador ponto (“`.`”) na forma `referência.metodo(params)`.
- A `referência` é um identificador de:
  - objecto, se o método não tiver qualificador **static**.
  - classe, se o método tiver qualificador **static**.

Métodos (13)

- Chamada sucessiva:
  - Um método pode retornar um objecto, ao qual se pode chamar outro método. Existem duas formas de chamar sucessivamente os métodos:
    1. Guardar objecto numa variável

```
Classe var = obj.metodo1();
var.metodo2();
```

    2. Chamar directamente

```
obj.metodo1().metodo2();
```

Métodos (14)

- Dentro de um método de instâmcia, o objecto sobre o qual está a ser chamado o método pode ser referênciado pelo **`this`**.
- Não existe a referência `this` em métodos estáticos.
- Normalmente, o `this` é usado para passar o objecto sobre o qual está a ser chamado o método como argumento para outros métodos.

Métodos (15)

```
public void deposito(float valor){
   quantia += valor;
}
```

```
public void deposito(float valor){
   this.quantia += valor;
}
```

- Também pode ser necessário usar o `this` na presença de um **atributo escondido** por uma variável local ou parâmetro.

```
public void deposito(float quantia){
   this.quantia += quantia;
}
```

Métodos (16)

- É possível definir dois métodos, na mesma classe, com o mesmo nome mas com diferente número ou tipo de parâmetros.
- Neste caso, diz-se que um método é uma **sobreposição** do outro.
- O compilador decide qual o método a chamar consoante o número e tipo dos parâmetros passados ao método em causa.
- Normalmente, a sobreposição é usada quando um método pode receber a mesma informação de diferentes formas, ou quando alguns dos parâmetros podem tomar valores por omissão (e nesse caso não precisam de ser passados ao método).

Métodos (17)

- No caso de métodos de argumento variável, uma sequência de parâmetros `T...` é tratada como sendo um parâmetro de tipo tabela `T[]`.
  - Se dois métodos sobrepostos diferem apenas porque um declara uma sequência de parâmetros `T...` e outro declara um parâmetro de tipo tabela `T[]`, tal dará origem a um erro de compilação.
- Em caso de ambiguidade, um método de argumento fixo será sempre escolhido antes de um método de argumento variável.

Métodos (18)

```
public static void print(String title){...}
public static void print(String title, String... messages){...}
public static void print(String... messages){...}
```

```
print(“Hello”);
```

Ok, chamaria o 1º método (argumento fixo)!

```
print(“Hello”,”World”);
```

Daria origem a um erro de compilação!

A solução seria:

```
print(“Hello”, new String[] {”World”});
print(new String[] {“Hello”,”World”});
```

Método `main` (1)

- O interpretador JVM executa sempre o **método `main`** da classe indicada na linha do comando:
  - Qualificadores: **public static**
  - Retorno: **void**
  - Parâmetros: **String[] args**
- Todas as classe numa aplicação podem ter um método `main`. O método `main` a executar é especificado de cada vez que se corre o programa.

Método `main` (2)

```
Class Echo {
   public static void main(String[] args) {
      for(int i=0; i<args.length; i++)
         System.out.print(args[i]+ “ ”);
      System.out.println();
   }
}
```

```
> javac Echo.java
> java Echo estou aqui
> estou aqui
```

Método `main` (3)

- O programa termina quando é executada a última instrução do `main`.
- Se for necessário terminar antecipadamente o programa, chamar o método:

  System.exit(int status);

  onde o parâmetro `status` identifica o código de terminação (sucesso - 0 em Linux, 1 em Windows).

Variáveis locais

- A declaração de uma variável local é semelhante à declaração de um atributo, sendo que o único qualificador que pode ser aplicado a esta é **final**.
  Qualif* Tipo Ident [= Expr] [, Ident = Expr]* ;
- As variáveis locais têm de ser inicializadas antes de ser usadas, ou quando são declaradas, ou por atribuição antes de ser usada.
  - Ao contrário dos atributos, não há inicialização por omissão de variáveis locais.
- Tal como no caso dos atributos, quando a incialização de uma variável local com o qualificador **final** é feita fora da sua declaração, essa variável local é dita **final em branco**.

Instruções (1)

- Um **bloco** agrupa zero ou mais instruções.
- Um bloco é delimitado pelos parêntesis { e }.
- Uma variável local existe apenas enquanto o bloco que a contém está a ser executado.
- Uma variável local pode ser declarada em qualquer ponto do bloco a que pertence, e não apenas no início, mas sempre antes de ser usada.

Instruções (2)

Atribuição

Var = Expr [, Var = Expr]*;

- Var: variável local
- Expr: expressão

- Relembrar operador de atribuição:
  - A atribuição Var = Var op Expr é equivalente a Var op= Expr.

Instruções (3)

- Atribuição de objectos produz duas referências distintas para o mesmo objecto!

```
Conta c1 = new Conta(), c2;
c1.depositar(1000);
c2 = c1; // c2 referencia mesmo objecto que c1
System.out.println(“Saldo c1 = " + c1.saldo());
System.out.println(“Saldo c2 = " + c2.saldo());
c1.depositar(100);
c2.depositar(200);
System.out.println(“Saldo c1 = " + c1.saldo());
System.out.println(“Saldo c2 = " + c2.saldo());
```

No terminal é impresso

```
Saldo c1 = 1000
Saldo c2 = 1000
Saldo c1 = 1300
Saldo c2 = 1300
```

Instruções (4)

Execução condicional

`if` (Expr-Bool) instrução1 [`else` instrução2]

- Expr_Bool: expressão Booleana

Instruções (5)

```
char c;
/* identifica espécie de caractere */
if (c>=‘0’ && c<=‘9’)
   System.out.println("Digito!");
else if ((c>=‘a’ && c<=‘z’) || (c>=‘A’ && c<=‘Z’))
   System.out.println(“Letra!”);
else
   System.out.println("Outra coisa!");
```

Instruções (6)

Selecção valores

```
switch (Expr) {
        case literal: instrução1
        case literal: instrução2
        ...
        default: instruçãoN
    }
```

- Valor de expressão Expr (`char`, `byte`, `short` ou `int`, ou correspondentes classes de embrulho, ou enum) é comparado com literais.
- Execução interrompida pela instrução **`break`**.

Instruções (7)

```
int i = 3;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, ");
    case 0: System.out.println("Boom!");
    default: System.out.println("Um numero,por favor!");
}
```

No terminal é impresso
```
3, 2, 1, Boom!
Um numero, por favor!
```

Instruções (8)

```
int i = 3;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, "); break;
    case 0: System.out.println("Boom!");
    default: System.out.println("Um numero,por favor!");
}
```

No terminal é impresso `3, 2, 1,`

Instruções (9)

```
int i = 4;
switch (i) {
    case 3: System.out.print("3, ");
    case 2: System.out.print("2, ");
    case 1: System.out.print("1, ");
    case 0: System.out.println("Boom!");
    default: System.out.println("Um numero,por favor!");
}
```

No terminal é impresso `Um numero, por favor!`

Instruções (10)

Ciclo condicional

`while` (Expr-Bool) instrução-corpo

- A instrução-corpo é executada enquanto a expressão booleana Expr-Bool for avaliada a `true`.

`do` instrução-corpo `while` (Expr-Bool)

- O teste pode ser feito depois de executar a instrução-corpo.
- Em qualquer dos ciclos condicionais:
  - Dentro da instrução-corpo o programa pode regressar à avaliação da expressão boolean Expr-Bool mediante a instrução **`continue`**.
  - Saída do ciclo pode ser antecipada pela instrução **`break`**.

Instruções (11)

Ciclo iterativo

`for` (inicialização; Expr-Bool; actualização)

instrução-corpo

- Variável de controlo que percorre gama de valores.
- A inicialização é feita antes de entrar no ciclo (tipicamente, variável de controlo colocada no início da gama de valores).
- Expressão booleana Expr-Bool é avaliada no início do ciclo, que é executado enquanto for `true` (tipicamente, testa se variável de controlo atingiu fim da gama de valores).
- A actualização é executada depois de executado o corpo do ciclo instrução-corpo (tipicamente, altera variável de controlo em direcção ao valor final da gama).

Instruções (12)

- Variável de controlo pode ser declarada na inicialização, e é acessível no corpo.
- Podem existir várias variáveis de controlo, podendo todas elas ser inicializadas na inicialização (separadas por vírgulas). Nesse caso, a actualização pode conter tantas instruções (separadas por vírgulas) quantas as variáveis de controlo a alterar em direcção ao valor final da gama.
- Os três campos podem ser nulos.
  - Por omissão, o segundo campo é avaliado a `true`.
  - As instruções **`for(;;)`** e **`while(true)`** são instruções equivalentes.
- Dentro da instrução-corpo o programa pode regressar à avaliação da expressão Booleana Expr-Bool mediante a instrução **`continue`**.
- Saída do ciclo pode ser antecipada pela instrução **`break`**.

Instruções (13)

```
/* Imprime pares até 20 */
for(int i=0, j=0; i+j<=20; i++, j++)
   System.out.print(i+j + " ");
System.out.println();
```

No terminal é impresso `0 2 4 6 8 10 12 14 16 18 20`

Instruções (14)

Cliclo iterativo (variante): ***for-each loop***

`for` (Tipo var-ciclo: conjunto) instrução-corpo

- O conjunto é um objecto que define um conjunto de valores sobre os quais se pretende iterar (tabela, …)
- Em cada passo do ciclo a variável local var-ciclo toma um valor diferente do conjunto e é executado a instrução-corpo (tipicamente, usando a variável var-ciclo para algo).
- O ciclo termina quando não existem mais valores no conjunto.

Instruções (15)

```
static double media(int[] valores) {
   double soma = 0.0;
   for (int val : valores)
      soma += val;
   return soma / valores.length;
}
```

```
int[] valores={20,19,18,17,16,15,14,13,12};
System.out.println("A média dos valores é " + media(valores));
```

Imprime no terminal `A média dos valores é 16.0`

Instruções (16)

- A instrução `break` pode ser usada para sair de qualquer bloco.
- Há duas formas possíveis para a instrução `break`:
  - **Não etiquetado**: **`break;`**
  - **Etiquetado**: **`break etiqueta;`**
- Um `break` não etiquetado termina a correspondente instruções `switch`, `for`, `while` ou `do`, e só pode ser usado neste contexto.
- Um `break` etiquetado pode terminar qualquer instrução etiquetada.

Instruções (17)

```
public boolean modificaValor(float[][] matriz, float val) {
   int i, j;
   boolean encontrado = false;
procura:
   for (i=0; i<matriz.length; i++) {
      for (j=0; j<matriz[i].length; j++) {
         if (matriz[i][j]==val) {
            encontrado = true;
            break procura;
         }
      }
   }
   if (!encontrado)
      return false;
   //modificar o valor matriz[i][j] da forma pretendida
   return true;
}
```

Instruções (18)

```
public boolean modificaValor(float[][] matriz, float val) {
   int i, j;
procura:
   {
      for (i=0; i<matriz.length; i++) {
         for (j=0; j<matriz[i].length; j++) {
            if (matriz[i][j]==val)
               break procura;
         }
      }
      //se chegamos aqui não encontramos o valor
      return false;
   }
   //modificar o valor matriz[i][j] da forma pretendida
   return true;
}
```

Instruções (19)

- A instrução **`continue`** pode ser usada apenas em ciclos (`for`, `while` ou `do`) e transfere o controlo para o fim do correspondente corpo do ciclo.
- Esta transferência de controlo faz com que a próxima instrução a ser executada seja:
  - No caso dos ciclos `while` e `do`, a avaliação da expressão booleana Expr-Bool.
  - No caso do ciclo `for` (básico), a instrução seguida da avaliação da expressão boolena Expr-Bool.
  - No caso do ciclo `for` (*for-each*), próximo elemento do conjunto, se existir.

Instruções (20)

- A instrução continue também tem duas formas:
  - **Não etiquetada: `continue;`**
  - **Etiquetada: `continue etiqueta;`**
- Na forma não etiquetada, a instrução `continue` transfere o controlo para o fim do correspondente ciclo.
- Na forma etiquetada, a instrução `continue` transfere o controlo para o fim do ciclo com essa etiqueta.

Instruções (21)

```
static void duplicaMatrizSimetrica(int[][] matriz) {
   int dim = matriz.length;
coluna:
   for (int i=0; i<dim; i++) {
      for (int j=0; j<dim; j++) {
         matriz[i][j]=matriz[j][i]=matriz[i][j]*2;
         if (i==j)
            continue coluna;
      }
   }
}
```

Instruções (22)

- A instrução **`return`** termina a execução dum método e regressa ao ponto que o invocou.
  - Se o método não retorna nenhum valor (`void`), usa-se apenas `return;`
  - Se o método retorna um valor, a instrução `return` deve incluir uma expressão com tipo compatível com o tipo de retorno do método.
- O `return` também pode ser usado no contexto de um construtor (como os contrutores não têm tipo de retorno é sempre usado na forma `return;`).