Programação Orientada a Objetos

Passagem e Retorno de Objetos, Membros de Classe, Constantes, Enumerados

Prof. Rui César das Neves, Prof. José Cordeiro

Departamento de Sistemas e Informática
Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal
2014/2015

Sumário

- Passagem e Retorno de Objetos em Métodos
- Tipos primitivos vs. Tipos referenciados
  - Tipos Primitivos do Java
- Membros de Classe (“`static`”)
  - Atributos de Classe (versus os nossos conhecidos atributos de instância)
  - Métodos de Classe (versus os nossos conhecidos métodos de instância)
- Constantes (“`final`”)
- Enumerados (“`enum`”)

Passagem e Retorno de Objectos

```
public class Relogio {

    private int horas;
    private int minutos;
    private double segundos;

    public Relogio () { this(0,0,0.0); }
    public Relogio ( int horas, int minutos, double segundos ) {
       this.horas = horas; this.minutos = minutos; this.segundos = segundos;
    }

    // [...] geters & seters

    // retorna um objeto da classe Relogio
    public Relogio copiar () {
           return new Relogio(horas, minutos, segundos);
    }

     // recebe um objeto da classe relogio
    public boolean isIgual ( Relogio relogio ){
       return ((relogio.horas==horas) &&
               (relogio.minutos==minutos) &&
               (relogio.segundos==segundos));
     }
}
```

Passagem e Retorno de Objectos

```
public class Programa {

    public static void main ( String[] args ){
        Relogio timex = new Relogio(10,25,55.0);

        System.out.println(timex.getHoras() + ":" +
                           timex.getMinutos() + ":" +
                           timex.getSegundos());

        timex.avancarMinutos( 55 );

        System.out.println(timex.getHoras() + ":" +
                           timex.getMinutos() + ":" +
                           timex.getSegundos());


        Relogio omega = timex.copiar();

        System.out.println("omega é igual a timex: " + omega.isIgual(timex));
    }
}
```

Passagem e Retorno de Objectos

- Note-se que a passagem de objectos como argumento/parâmetro é especificada da mesma forma que para os outros tipos de dados simples:
  - o nome da classe é o tipo de dados do parâmetro.

- Note-se que o retorno de objetos é da mesma forma que para os outros tipos de dados simples:
  - O nome da classe é o tipo de dados retornado.

Tipos Primitivos do Java

Tipo	Precisão (bits)	Gama de Valores	Observações
byte	8	[-128, 127]	
short	16	[-32 768, 32 767]	
int	32	[-2 147 483 648, 2 147 483 647]	
long	64	[-9 223 372 036 854 775 808, 9 223 372 036 854 775 807]	
float	32	$\approx\pm 2.0 \times 10^{-38}$ $\approx\pm 2.0 \times 10^{38}$	IEEE 754 Precisão Simples
double	64	$\approx\pm 2.0 \times 10^{-308}$ $\approx\pm 2.0 \times 10^{308}$	IEEE 754 Precisão Dupla
boolean	N.D.	{true, false}	
char	16	[0, 65535]	Caracteres UNICODE

Passagem e Retorno de Objectos

- Convém relembrar que em JAVA as variáveis que guardam objetos (sejam elas atributos, parâmetros dos métodos ou variáveis locais) apenas armazenam as suas referências e não os objetos diretamente;

- Este comportamento faz com que a passagem de objetos, através dos parâmetros dos métodos ou do retorno dos métodos, seja feita copiando as suas referências.

- Por isto, não existe uma duplicação do objeto (exceto quando explicitamente indicado) nestas passagens, não se criando dois objetos distintos: um dentro do método e outro fora.

- Por existir apenas um objeto, qualquer alteração a ele, no interior dos métodos, mantêm-se após o método terminar.

Tipos primitivos vs. Tipos referenciados

```
int numero1;
numero1=23;
```

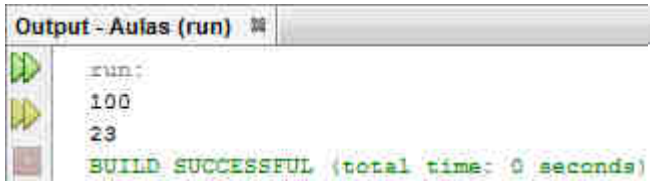
numero1	~~100~~ 23

```
int numero2;
numero2=numero1;
```

numero2	23

```
numero1=100;

System.out.println( numero1 );
System.out.println( numero2 );
```



Este slide deve ser visto em modo de animação!

```
Relogio timex;
timex = new Relogio(11,12,13.0);
```

timex	

horas	11
minutos	12
Segundos	13.0

```
Relogio omega;
omega = new Relogio (21,22,23.0 );
timex = omega;
omega.setHoras(15);
omega.setMinutos(0);
omega.setSegundos(0.0);
System.out.println( timex.getHoras() + ":" +
     timex.getMinutos() + ":" +
     timex.getSegundos());
System.out.println(omega.getHoras() + ":" +
     omega.getMinutos() + ":" +
     omega.getSegundos());
```

omega	

horas	15
Minutos	0
Segundos	0.0

Tipos primitivos vs. Tipos referenciados

Qual o resultado?

```
int a;
int b;
a = 32;
b = a;
a = a + 1;
System.out.println(b);
```

```
Pessoa a;
Pessoa b;

a = new Pessoa("Miguel");
b = a;

a.setNome("Bruno");

System.out.println(b.getNome());
```

Membros de Classe – "static"

Motivação

- Como temos visto até agora, quando criamos instâncias de uma classe, cada objeto possui os seus membros (atributos e métodos) de instância:
  - O estado (o valor) dos atributos é próprio de cada objeto (atributos de instância)
  - Só pode ser alterado pelos métodos do próprio objeto (métodos de instância)
  - Acedemos a esses atributos através dos métodos do objeto
  - EX: identificadorDoObjeto.getAtributo() bruno.getNome()

- Por vezes é necessário ter atributos que pertencem à Classe e não a um objeto em particular:
  - Imagine-se a querer saber quantos Relógios foram fabricados?
  - **Será que atributos como**
    - **quantidadeDeRelogiosFabricados**
  - **Deverão definir o estado:**
    - **do objeto?**
    - **ou da classe?**

Membros de Classe – "static"

- **Membros de Classe:**
  - **Atributos de Classe**: definem o estado da Classe como um todo e não o estado de um objeto em particular.
    - São partilhados por todos os objetos da classe
    - O seu valor é igual para todos os objetos da classe
  - **Métodos de Classe:** definem aspetos do comportamento da Classe e não de um objeto em particular .
    - São partilhados por todos os objetos da classe
    - Servem, fundamentalmente, para manipular os valores dos atributos de classe
    - Podem servir para fornecer funcionalidades próprias da classe quando estas não estão relacionadas com uma instância em particular
    - São sempre acessíveis pelas instâncias da classe, mas …
    - … não têm acesso a qualquer dos métodos ou atributos de instância.
  - Não é necessário criar um objecto para utilizar um método de classe

Membros de Classe – "static"

```
public class Relogio {

    // 1 - Atributos de Classe
        private static int quantidadeDeRelogiosFabricados = 0;

    // 2- Métodos de Classe
        public static int getQuantidadeDeRelogiosFabricados (){
            return quantidadeDeRelogiosFabricados; }
        private static void incrementaQuantidadeDeRelogiosFabricados(){
            quantidadeDeRelogiosFabricados++;}

    // 3 – atributos de instância
        private int horas;
        private int minutos;
        private double segundos;
    // 4 - Construtores
        Relogio () {
            this(0,0,0.0);
        }
        Relogio ( int horas, int minutos, double segundos ) {
        // Note-se que os métodos de instância têm acesso aos métodos de classe
            incrementaQuantidadeDeRelogiosFabricados();
            this.horas = horas;
            this.minutos = minutos;
            this.segundos = segundos;}
    //[…]restantes métodos
}
```

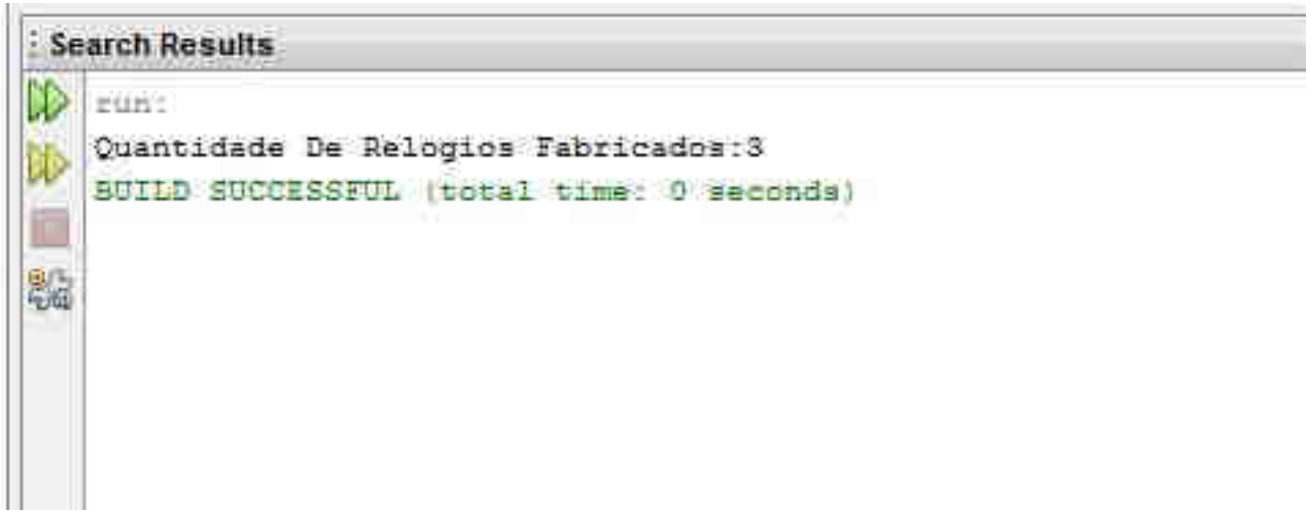
Membros de Classe – “static”

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

    Relogio timex1 = new Relogio(0, 11, 22);
    Relogio timex2 = new Relogio(1, 33, 44);
    Relogio timex3 = new Relogio(2, 55, 59);

    System.out.println("Quantidade De Relogios Fabricados: " +
        Relogio.getQuantidadeDeRelogiosFabricados());

}
```



Constantes – "final"

- **Motivação:**
  - Quando pretendemos definir um valor que deverá permanecer inalterado e inalterável permanentemente definimo-lo como uma constante.
- `final` – modificador que define o atributo como uma constante
- **Declaração:**
  - Normalmente as contantes são definidas como atributos de classe:
    - EX: `public static final double PI = 3.1415926;`
  - Mas podem também ser definidas como atributos de instância:
    - EX: `public final double PI = 3.1415926;`
  - Mas então qual a diferença entre defini-las como atributos de classe e de instância?
- **Identificadores:**
  - Por regra as constantes são identificadas usando apenas LETRAS_MAIUSCULAS, com o underscore ´_´ para separar as palavras.

Enumeração de Valores

- Supor que se queria acrescentar informação sobre a cor do relógio
- A primeira abordagem seria criar um atributo, do tipo `String`, que registasse a cor:

```
public class Relogio {
    //...
    private String cor;
    public Relogio() { this(0, 0, 0, "preto"); }
    public Relogio(int horas, int minutos, int segundos, String cor) {
        //...
        this.cor = cor; }
    //...
    public String getCor(){
        return cor;
    }
    //...
    public void setCor(String cor) {
        this.cor = cor;
    }
    //...
}
```

Problemas com a Abordagem

- Não havia garantia da correta indicação da cor:
  - Erros de escrita: "vermlho"
  - Diferentes formas de escrita: "vermelho", "vermelha", "encarnado"
  - Possibilidade de ausência de cor: ""
- Impossibilidade de limitar apenas às cores disponíveis (fazer uma validação com `switch` e usar cor por omissão)
- Espaço ocupado é "grande" pois em cada objeto teremos que armazenar todos os caracteres do nome da cor

Abordagem Alternativa

- Criar constantes para os diferentes tipos de cores (uso de byte poupa espaço):

```
public class Relogio {
    public static final byte PRETO = 1;
    public static final byte VERMELHO = 2;
    public static final byte DOURADO = 3;

    //...
    private byte cor;

    //...
```

Abordagem Alternativa

```
public Relogio() {
    this(0, 0, 0, PRETO);
}

public Relogio(int horas, int minutos) {
    this(horas, minutos, 0, PRETO);
}

public Relogio(int horas, int minutos, int segundos, byte cor) {
    //...
    if (cor >= PRETO && cor <= DOURADO) {
        this.cor = cor;
    } else {
        this.cor = PRETO;
    }
}

//...
```

Abordagem Alternativa

```
    public String getCor() {
        switch (cor) {
            case PRETO: return "preto";
            case VERMELHO: return "vermelho";
            case DOURADO: return "dourado";
            default: return "cor impossível";
        }
    }
    //...
    public void setCor(byte cor) {
        if (cor >= PRETO && cor <= DOURADO) {
            this.cor = cor;
        }
    }
    //...
}
```

- Uso no programa principal:

```
public static void main(String[] args) {
    Relogio timex = new Relogio(11, 12, 13, Relogio.PRETO);
    Relogio omega = new Relogio(21, 22, 23, Relogio.DOURADO);
    //...
}
```

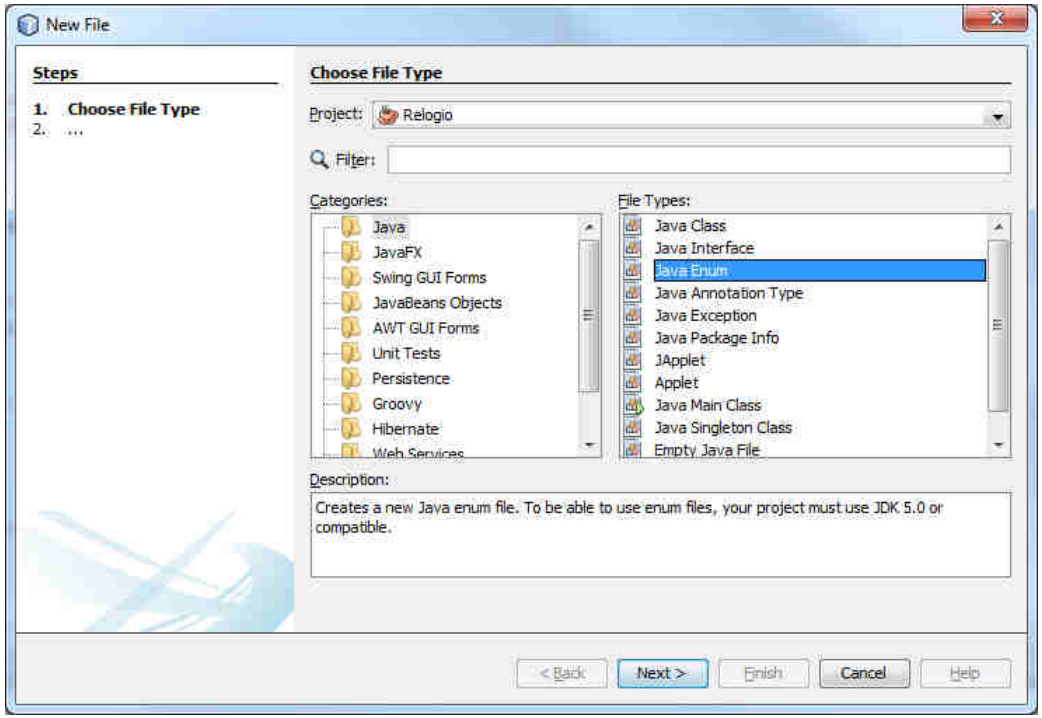
Abordagem Alternativa

- Conseguiu-se indicar quais as alternativas de cores disponíveis
- Evitam-se erros de escrita
- Conseguiu-se poupar espaço (byte ocupam menos que Strings)
- Mas ainda não há garantia de que apenas se usem aqueles 3 valores inteiros (é preciso validar nos sets e no construtor)

Tipos Enumerados

- O que se pretende é ter atributos de um *tipo especial* `Cor`, em que apenas se possam ter os três valores possíveis, e não do tipo `byte` ou `String` com "infinitas" possibilidades.
- Tipo enumerado:

```
public enum Cor {
    PRETO, VERMELHO, DOURADO
}
```



Uso de Enums

- O atributo passa a ser do tipo Cor:

```
public class Relogio {
    //Desaparecem as constantes
    //...
    private Cor cor;

    public Relogio() { this(0, 0, 0, Cor.PRETO); }
    public Relogio(int horas, int minutos, int segundos, Cor cor) {
        //...
        this.cor = cor;
    }
    //...
    public Cor getCor() { //System.out.println(Cor.PRETO); escreve PRETO
        return cor;
    }
    //...
    public void setCor(Cor cor) {
        this.cor = cor;
    }
    //...
}
```

Tipos Enumerados

- Criação de uma "classe especial" onde são indicados (*enumerados*) os diferentes valores possíveis (são constantes e por isso devem ser escritos em MAIÚSCULAS)
- Podem ser utilizados nos `switch`
- São criados, automaticamente, os métodos de classe:
  - `Cor.valueOf(String name)` que devolve o valor associado à `String` que o representa (o nome tem de ser escrito exatamente igual à constante)
  - `Cor.values()` que devolve um array (`Cor[]`) com todas as alternativas (interessante para usar em ciclos)

Tipos Enumerados

- É "classe especial" pois podem ser definidos métodos:

```
public enum Cor {
    PRETO, VERMELHO, DOURADO;

    @Override
    public String toString() {
        switch (this) {
            case PRETO: return "Escuro";
            case VERMELHO: return "Chamativo";
            case DOURADO: return "Brilhante";
            default: return "";
        }
    }
}
```

- É ainda possível definir valores associados a cada constante, criando-se atributos, construtores, seletores, etc.

Resumindo

- Passagem e Retorno de Objetos em Métodos
- Tipos primitivos vs. Tipos referenciados
  - Tipos Primitivos do Java
- Membros de Classe ("`static`")
  - Atributos de Classe
    - Declaração: static int quantidadeDeInstanciasCriadas;
    - Utilização: NomeDaClasse. quantidadeDeInstanciasCriadas;
  - Métodos de Classe
    - Declaração: static int getQuantidadeDeInstanciasCriadas() { […] }
    - Utilização: NomeDaClasse. getQuantidadeDeInstanciasCriadas();
- Constantes ("`final`")
  - De Classe: static final int MAXIMO_DE_ELEMENTOS = 1500;
  - De instância (raramente usados): final int MAXIMO_DE_ELEMENTOS = 1500;
- Tipos enumerados ("`enum`")

Leitura Complementar

- Capítulo 3
  - Páginas 65 a 112
  - Páginas 347 a 356