

**Programação Orientada a Objetos**

Linguagem de  
**Programação JAVA**

# Programação Orientada a Objetos

---

## Sumário

Estrutura do curso .....	4
Breve Histórico do Java.....	5
Introdução a Linguagem Java.....	6
Java Virtual Machine.....	17
Coletor de Lixo .....	18
Fundamentos da Linguagem Java .....	20
Identificadores, palavras reservadas, tipos, variáveis e Literais.....	22
Aplicativos independentes em Java.....	31
Operadores.....	34
String .....	40
Para que serve.....	41
StringBuffer .....	43
Resultado .....	44
Classe Integer .....	48
Classes Long, Short, Byte, Float e Double.....	51
Classe Character .....	53
Objetos X Tipos primitivos de dados .....	54
Fluxo de Controle .....	55
Laços.....	58
Arrays .....	62
Tratamento de exceções.....	65
Empacotamento de Classes .....	68
Conversão de Tipos .....	69
A referência this.....	71
Coleções .....	72
Pacotes .....	77
Acessibilidade.....	79
Programação Orientada a Objetos .....	85
Abstração .....	85
Classe Indivíduos.....	87
Encapsulamento.....	87
Herança.....	88
Hierarquia de Agregação .....	89
Hierarquia de Generalização / Especialização.....	90
Polimorfismo .....	90
Principais conceitos em orientação a objetos .....	92
Métodos .....	93
Construtores.....	94
O que são construtores?.....	94
<b>Anotações</b> .....	<b>2</b>

# Programação Orientada a Objetos

---

Atributos e variáveis .....	95
A referência super .....	96
Classe Abstrata e Finais .....	97
Interfaces .....	99
Utilitários .....	101
JavaDoc (Documentação) .....	102
Jar (Compactação, Agrupamento e Distribuição) .....	104
Capítulo I - Fundamentos da Linguagem .....	105
Capítulo II - Modificadores e Controle de Acesso .....	128
Capítulo III - Operadores e atribuições .....	164
Capítulo IV - Controle de fluxo, exceções e assertivas .....	188
Capítulo VI - java.lang - a classe Math, Strings e Wrappers .....	251
Capítulo VII - Objetos e conjuntos .....	280
Capítulo VIII - Classes internas .....	304
Capítulo IX - Threads .....	328
Exercícios Teóricos .....	354
Exercícios Práticos .....	358

## Anotações

3

# Programação Orientada a Objetos

---

## Estrutura do curso

- Introdução
- Tipos de dados e variáveis;
- Estruturas de programação no Java;
- Conceitos de Orientação a Objetos;
- Objetos da biblioteca Swing;
- Bancos de dados e SQL.

## Bibliografia

DEITEL & DEITEL. Java – como programar. 4ª Edição, Bookman, 2003.

FURGERI, SÉRGIO – Java 2 Ensino Didático. Editora Érica, 2002.

SIERRA, KATHY & BATES, BERT. JAVA 2 – Certificação SUN – Programador e desenvolvedor. 2ª Edição, AltaBooks.

Anotações

4

## Breve Histórico do Java

- 1991 – início do projeto Green
  - Requisitos do projeto
  - Não ficar dependente de plataforma
  - Poder rodar em pequenos equipamentos
  - Linguagem oak(carvalho)
- Em 1992 – O projeto Green apresenta seu primeiro produto. (Start Seven)
  - Revolucionar a indústria de TV e vídeo oferecendo mais interatividade.
- 1992 – Crise do Projeto Green
- 1993 – explode a WWW (World Wide Web)
  - Duke – Mascote Java
- 1995 – Maio - Nascimento oficial do Java.
- 1996 - Janeiro - Release do JDK 1.0.
- 1996 - Maio - Realizado o primeiro JavaOne, conferencia máxima da tecnologia Java.
  - Apresentados a tecnologia JavaBeans e Servlets.
- 1996 - Dezembro - Release do JDK 1.1 Beta.
- 1997 - Fevereiro - Release do JDK 1.1.
- 1997 - Abril - Anunciada a tecnologia Enterprise JavaBeans (EJB), além de incluir a Java
  - Foundation Classes (JFC) na plataforma Java.
- 1998 - Março - início do projeto JFC/Swing.
- 1998 - Dezembro - Formalizado o Java Community Process (JCP).
- 1999 - Fevereiro - Release do Java 2 Platform.
- 1999 - Junho - Anuncio da "divisão" da tecnologia Java em três edições (J2SE, J2EE, J2ME).
- 2000 -Maio - Release da J2SE v. 1.3.
- 2001 -Abril - Release do J2EE 1.3 beta, contendo as especificações EJB 2.0, JSP 1.2 e Servlet 2.3.
- 2002 - Dezembro - Release do J2EE 1.4 Beta.
- 2004 - Outubro - Release do Java 5.0, chamado de Java Tiger.
- 2005 - Março - 10º aniversário da tecnologia.
- 2005 - Junho - JavaOne de número 10.
- 2006 - JavaOne de número 11.

## Introdução a Linguagem Java

### Características do Java

- Java é sintática e morfolologicamente muito parecido com a linguagem C++, entretanto, existem diferenças:
- Inexistência de aritméticas de ponteiros (ponteiros são apenas referências);
- Independência de plataforma;
- Arrays são objetos;
- Orientação a Objetos;
- Multithreading
- Strings são objetos;
- Gerenciamento automático de alocação e deslocação de memória (Garbage Collection);
- Não existe Herança Múltiplas com classes, apenas com interfaces;
- Não existem funções, mas apenas métodos de classes;
- Bytecode;
- Interpretado;
- Compilado;
- Necessita de ambiente de execução (runtime), ou seja, a JVM (Java Virtual Machine).

### Tecnologia Java

A tecnologia java oferece um conjunto de soluções para desenvolvimento de aplicações para diversos ambientes.

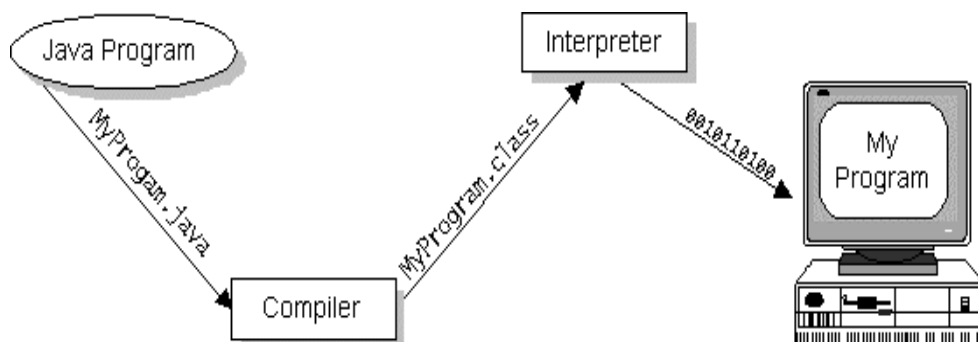
- J2SE – Java 2 Standard Edition (Core/Desktop)
- J2EE – Java 2 Enterprise Edition (Enterprise/Server)
- J2ME – Java 2 Micro Edition (Mobile/Wireless)

# Programação Orientada a Objetos

# O QUE É JAVA ?

É uma Linguagem de programação Orientada a objetos, portátil entre diferentes plataformas e sistemas operacionais.

1. Todos os programas Java são compilados e interpretados;
2. O compilador transforma o programa em bytecodes independentes de plataforma;
3. O interpretador testa e executa os bytecodes
4. Cada interpretador é uma implementação da JVM - Java Virtual Machine;



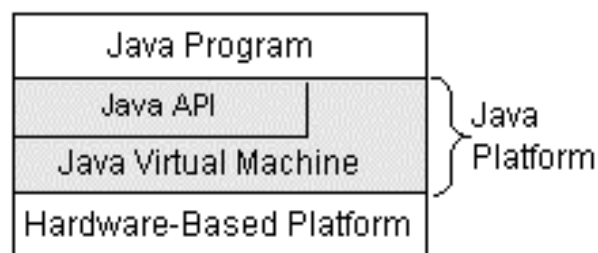
# Plataforma Java

Uma plataforma é o ambiente de hardware e software onde um programa é executado. A plataforma Java é um ambiente somente de software.

Componentes:

## Java Virtual Machine (Java VM)

## Java Application Programming Interface (Java API)



## Anotações

7

# Programação Orientada a Objetos

---

## Mitos da Linguagem

O Java é da Sun?

Java é uma linguagem direcionada para a Internet?

Java é igual a JavaScript? (LiveScript)

Java é lento?

Anotações

8

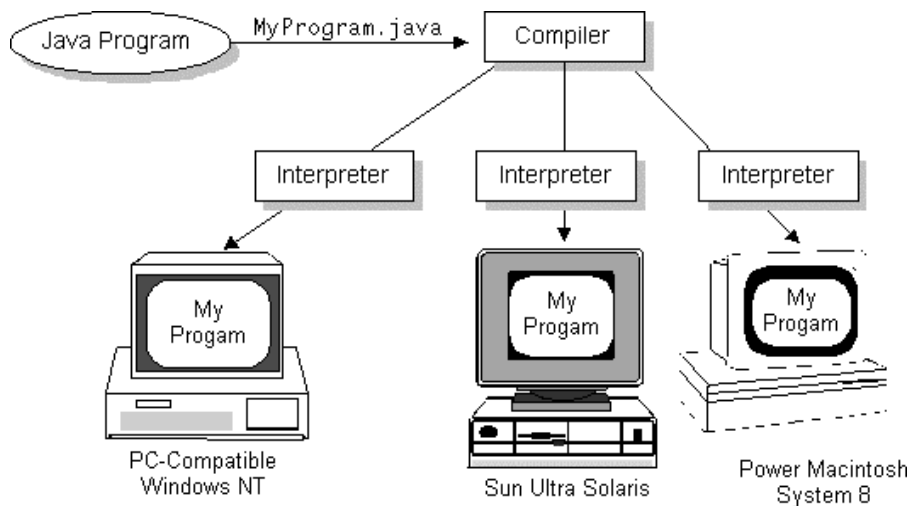


# Programação Orientada a Objetos

---

## Portabilidade: “A independência de plataforma”

A linguagem Java é independente de plataforma. Isto significa que o desenvolvedor não terá que se preocupar com particularidades do sistema operacional ou de hardware, focando o seu esforço no código em si. Mas o que isto realmente significa?



A maioria das linguagens é preciso gerar uma versão para cada plataforma que se deseja utilizar, exigindo em muitos casos, alterações também no código fonte. Em Java o mesmo programa pode ser executado em diferentes plataformas. Veja o exemplo abaixo:

```
public class HelloWorldApp{
    public static void main (String arg []){
        System.out.println("Hello World!");
    }
}
```

### Compilação:

```
> javac HelloWorldApp.java
```

### Execução:

```
> java HelloWorldApp
```

### Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## Gerando Aplicações

Para criar aplicações ou programas na linguagem Java temos que seguir os alguns passos como: Edição, Compilação e Interpretação.

A **Edição** é a criação do programa, que também é chamado de código fonte.

Com a **compilação** é gerado um código intermediário chamado Bytecode, que é um código independente de plataforma.

Na **Interpretação**, a máquina virtual Java ou JVM, analisa e executa cada instrução do código Bytecode.

Na linguagem Java a compilação ocorre apenas uma vez e a interpretação ocorre a cada vez que o programa é executado.

## Plataforma de Desenvolvimento

A popularidade da linguagem Java fez com que muitas empresas desenvolvessem ferramentas para facilitar desenvolvimento de aplicações. Estas ferramentas também são conhecidas como IDE (Ambiente de Desenvolvimento Integrado), que embutem uma série de recursos para dar produtividade. Todavia, cada uma delas tem suas próprias particularidades e algumas características semelhantes.

As principais ferramentas do mercado são:

Jbuilder ([www.borland.com](http://www.borland.com))

NetBeans(<http://www.netbeans.org>)

Java Studio Creator ([www.sun.com](http://www.sun.com))

Jedit([www.jedit.org](http://www.jedit.org))

IBM Websphere Studio Application Developer(WSAD) ([www.ibm.com](http://www.ibm.com))

Eclipse([www.eclipse.org](http://www.eclipse.org))

Jdeveloper([www.oracle.com](http://www.oracle.com))

## Instalação do ambiente de desenvolvimento (JDK)

O J2SDK é ambiente de desenvolvimento da linguagem Java. Na verdade é conjunto de ferramentas para compilar, depurar, executar e documentar um programa escrito em java. As versões para Solaris, Linux e Windows podem ser obtidas no endereço: [www.java.sun.com/j2se](http://www.java.sun.com/j2se) (Página oficial da linguagem).

---

## Anotações

10

# Programação Orientada a Objetos

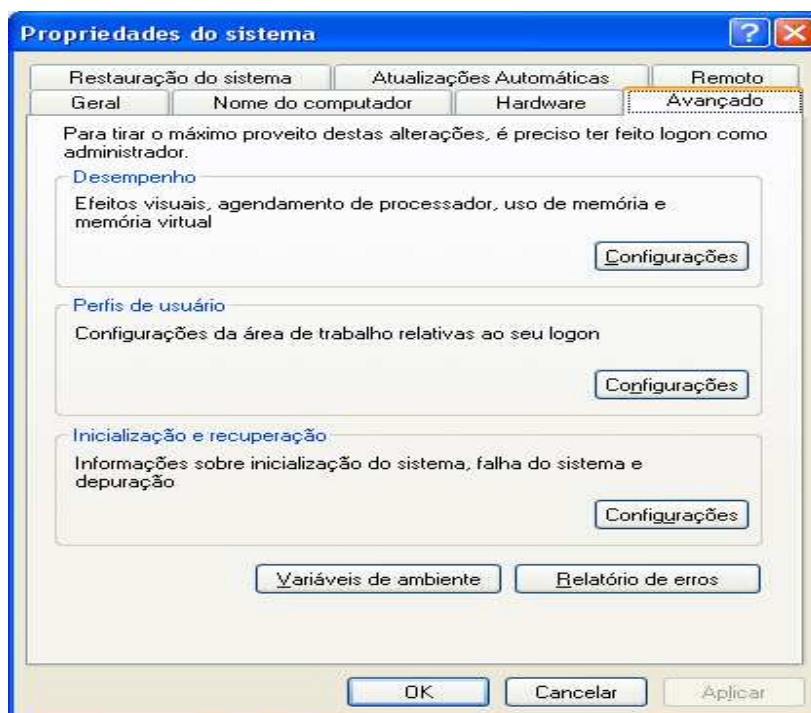
---

## Procedimentos de Instalação:

Para instalar o J2SDK basta executar o programa de instalação e seguir as instruções. Caso o J2SDK esteja compactado será necessário descompactá-lo primeiro, os formatos mais populares de arquivos compactados são zip (para Windows) e tar (para Unix).

## Instalação do ambiente de desenvolvimento (JDK)

A partir do Menu iniciar, selecione Configurações, depois selecione Painel de Controle e enfim Sistema. Estando em Sistema seleciona o guia “Avançado” e depois variáveis de ambiente.



---

## Anotações

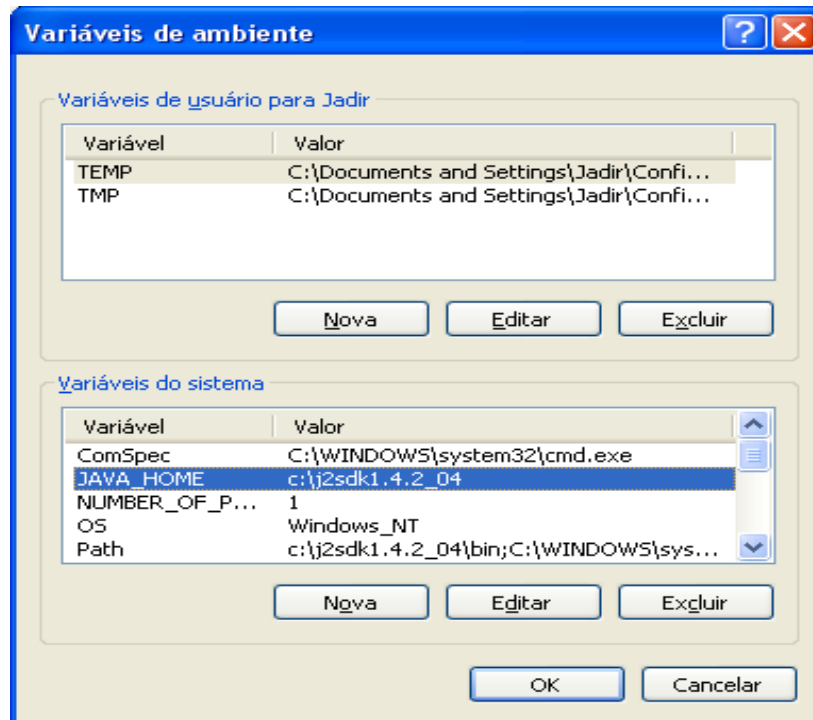
11

# Programação Orientada a Objetos

---

## Passos 1:

1 - Crie uma nova variável de ambiente chamada "JAVA\_HOME" com o seguinte valor C:\j2sdk1.4.2\_04



Exemplo: JAVA\_HOME = C:\j2sdk1.4.2\_04. A variável JAVA\_HOME deve ter o mesmo nome do diretório onde foi instalado o J2SDK.

Anotações

12

# Programação Orientada a Objetos

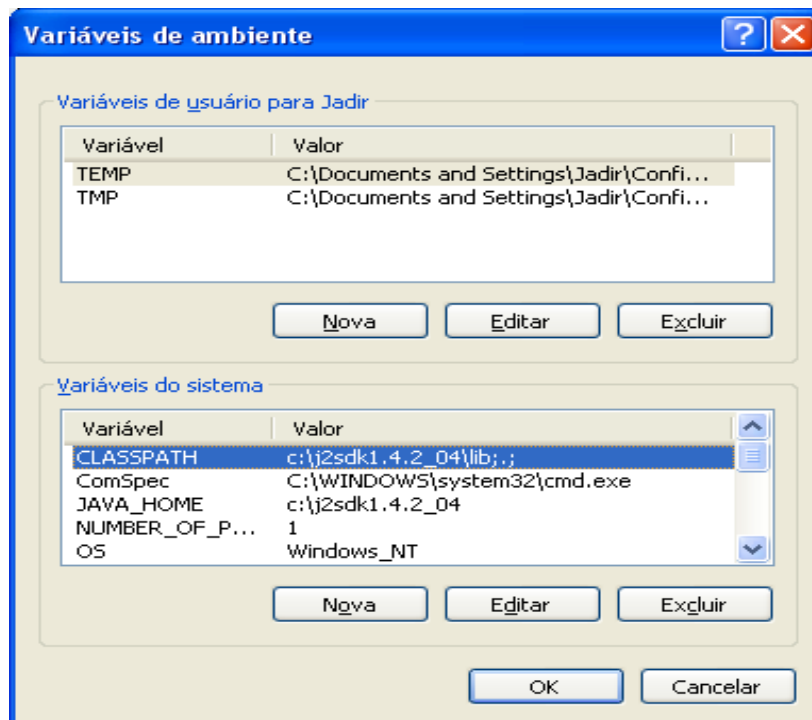
---

## Passo 2:

### Classpath

O Java defini uma variável de ambiente chamada ClassPath. O Java procura pelas classes e pacotes através desta variável.

1 - Crie ou edite a variável de ambiente chamada CLASSPATH, informa o seguinte valor para ela. classpath=C:\j2sdk1.4.2\_04\lib;.;

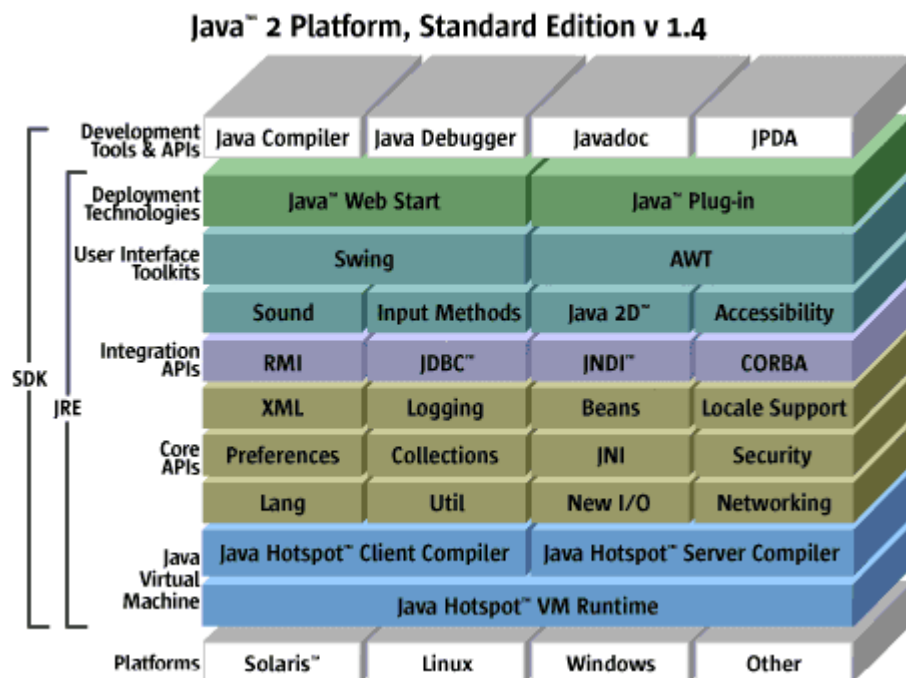


Anotações

13

# Programação Orientada a Objetos

---



A figura acima demonstra uma visão do pacote de desenvolvimento J2SDK e também do ambiente de execução (JRE). Ambos são necessários para desenvolver uma aplicação.

Anotações

14

# Programação Orientada a Objetos

---

## O Compilador javac

Sintaxe: **javac [opções] NomeDoArquivo.java**

Argumento	Descrição
classpath path	Localização das classes. Sobrepoë a variável de ambiente Classpath;
-d dir	Determina o caminho onde as classes compiladas são armazenadas;
-deprecation	Faz a compilação de código em desuso, geralmente de versões anteriores e faz aviso de advertência;
-g	Gera tabelas de "debugging" que serão usadas pelo depurador JDB;
-nowarn	Desabilita as mensagens de advertência;
-verbose	Exibe informações adicionais sobre a compilação;
-O	Faz otimização do código;
-depend	Faz a compilação de todos os arquivos que dependem do arquivo que está sendo compilado. Normalmente somente é compilado o arquivo fonte mais as classes que este invoca.

### Exemplos

```
> javac Hello.java  
> javac -d Hello.java  
> javac -deprecation Hello.java  
> javac -O -deprecation -verbose Hello.java  
> javac -O Hello.java
```

Anotações

15

# Programação Orientada a Objetos

---

## O Interpretador java

Sintaxe: **java [opções] NomeDoArquivo [lista de Argumentos]**

Argumento	Descrição
classpath path	Localização das classes. Sobrepõe a variável de ambiente Classpath;
-help	Exibe a lista de opções disponíveis;
-version	Exibe a versão do interpretador;
-debug	Inicia o interpretador no modo de "debug", geralmente em conjunto com JDB;
-D	propriedade=valor Possibilita redefinição de valores de propriedades. Pode ser usado várias vezes;
-jar	Indica o nome do arquivo (com extensão .jar) que contém a classe a ser executada;
-X	Exibe a lista de opções não padronizadas do interpretador;
-v ou -verbose	Exibe informações extras sobre a execução, tais como, mensagens indicando que uma classe está sendo carregada e etc;
Lista de Argumentos	Define a lista de argumentos que será enviada a aplicação.

### Exemplos

```
> java Hello  
> javac -version  
> java -D nome="Meu Nome" Hello  
> java -verbose Hello  
> javac Hello MeuNome
```

Anotações

---

---

---

---

16



# Programação Orientada a Objetos

---

## Java Virtual Machine

A JVM é parte do ambiente de "runtime" Java e é a responsável pela interpretação dos bytecodes (programa compilado em java), ou seja, a execução do código. A JVM consiste em um conjunto de instruções, conjunto de registradores, a pilha (stack) , garbage-collected heap e a área de memória (armazenamento de métodos).

### Funcões da JVM Java Virtual Machine :

- Segurança de código – Responsável por garantir a não execução de códigos maliciosos (ex. applets).
- Verificar se os bytecodes aderem às especificações da JVM e se não violam a integridade e segurança da plataforma;
- Interpretar o código;
- Class loader – carrega arquivos .class para a memória.

OBS: Em tempo de execução estes bytecodes são carregados, são verificados através do Bytecode Verifier (uma espécie de vigilante) e somente depois de verificados serão executados.

# Programação Orientada a Objetos

---

## Coletor de Lixo

A linguagem Java tem alocação dinâmica de memória em tempo de execução. No C e C++ (e em outras linguagens) o programa desenvolvido é responsável pela alocação e deslocamento da memória. Isto geralmente provoca alguns problemas. Durante o ciclo de execução do programa, o Java verifica se as variáveis de memória estão sendo utilizadas, caso não estejam o Java libera automaticamente esta área para o uso. Veja exemplo abaixo:

```
import java.util.*;
class GarbageExample {
    private static Vector vetor;
    public static void main(String args[]) {
        vetor = new Vector();
        for (int a=0; a < 500; a++){
            vetor.addElement(new StringBuffer("teste"));
            Runtime rt = Runtime.getRuntime();
            System.out.println("Memória Livre: " + rt.freeMemory());
            vetor = null;
            System.gc();
            System.out.println("Memória Livre: " + rt.freeMemory());
        }
    }
}
```

# Programação Orientada a Objetos

---

## Escrevendo um pequeno programa

1 - Abra o editor de programas e crie o seguinte programa.

```
public class Hello{  
    public static void main (String arg []){  
        String s = "world";  
        System.out.println("Hello " + s);  
    }  
}
```

2 - Salvar como: **Hello.java**

3 - Compile o programa com o seguinte comando:  
javac **Hello.java**

4 - Para executar, digite o comando:  
java **Hello**

---

Anotações

19

# Programação Orientada a Objetos

---

## Fundamentos da Linguagem Java

### Estrutura da Linguagem

#### Comentários

Temos três tipos permitidos de comentários nos programas feitos em Java:

- // comentário de uma única linha
- /\* comentário de uma ou mais linhas \*/
- /\*\* comentário de documentação \*/ (este tipo de comentário é usado pelo utilitário
- Javadoc, que é responsável por gerar documentação do código Java)

#### Exemplo

```
int x=10; // valor de x Comentário de linha
```

```
/*  
A variável x é integer  
*/  
int x;
```

Exemplo onde o comentário usa mais que uma linha. Todo o texto entre "/\*" e "\*/", inclusive, são ignorados pelo compilador.

```
/**  
x -- um valor inteiro representa  
a coordenada x  
*/  
int x;
```

Todo o texto entre o "/\*" e "\*/", inclusive, são ignorados pelo compilador mas serão usados pelo utilitário javadoc.

# Programação Orientada a Objetos

---

## Estilo e organização

- No Java, blocos de código são colocados entre chaves { };
- No final de cada instrução usa-se o ; (ponto e vírgula);
- A classe tem o mesmo nome do arquivo .java;
- Todo programa Java é representado por uma ou mais classes;
- Normalmente trabalhamos com apenas uma classe por arquivo.
- Case Sensitive;

## Convenções de Códigos

### Nome da Classe:

O primeiro caracter de todas as palavras que compõem devem iniciar com maiúsculo e os demais caracteres devem ser minúsculos.

Ex. HelloWorld, MeuPrimeiroPrograma, BancoDeDados.

### Método, atributos e variáveis:

Primeiro caracter minúsculo;

Demais palavras seguem a regra de nomes da classes.

Ex. minhaFunção, minhaVariavelInt.

### Constantes:

Todos os caracteres maiúsculos e divisão de palavras utilizando underscore “\_”.

Ex. MAIUSCULO, DATA\_NASCIMENTO.

### **Exemplo:**

```
public class Exercicio1 {  
    public static void main (String args []) {  
        valor=10;  
        System.out.println(valor);  
    }  
}
```

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## Identificadores, palavras reservadas, tipos, variáveis e Literais

### Identificadores

Que são identificadores ?

Identificadores são nomes que damos as classes, aos métodos e as variáveis.

**Regra:** Um identificador deverá ser inicializado com uma letra, sublinhado ( \_ ), ou sinal de cifrão (\$). Em Java existe uma diferença entre letras maiúsculas e minúsculas.

Veja alguns exemplos:

**Teste** é diferente de **TESTE**

**teste** é diferente de **Teste**

#### Exemplos de identificadores:

Alguns identificadores válidos:

valor - userName - nome\_usuario - \_sis\_var1 - \$troca

Exemplo: *public class **PessoaFisica***

Veja alguns inválidos:

- 1nome - \TestClass - /metodoValidar

# Programação Orientada a Objetos

---

## Palavras Reservadas

As Palavras Reservadas, quer dizer que nenhuma das palavras da lista abaixo podem ser usadas como identificadores, pois, todas elas foram reservadas para a Linguagem Java.

<b>abstract</b>	<b>boolean</b>	<b>break</b>	<b>byte</b>	<b>case</b>	<b>catch</b>
<b>char</b>	<b>class</b>	<b>const</b>	<b>int</b>	<b>double</b>	<b>float</b>
<b>for</b>	<b>long</b>	<b>short</b>	<b>if</b>	<b>while</b>	<b>do</b>
<b>transient</b>	<b>volatile</b>	<b>strictpf</b>	<b>assert</b>	<b>try</b>	<b>finally</b>
<b>continue</b>	<b>instanceof</b>	<b>package</b>	<b>static</b>	<b>private</b>	<b>public</b>
<b>protected</b>	<b>throw</b>	<b>void</b>	<b>switch</b>	<b>throws</b>	<b>native</b>
<b>new</b>	<b>import</b>	<b>final</b>	<b>implements</b>	<b>extends</b>	<b>interface</b>
<b>goto</b>	<b>else</b>	<b>default</b>	<b>return</b>	<b>super</b>	<b>this</b>
<b>synchronized</b>					

Veja o exemplo:

```
public class TestPalavraReservada{
    private int return =1;
    public void default(String hello){
        System.out.println("Hello ");
    }
}
```

Este programa provocará erros ao ser compilado:

```
----- Compiler Output -----
TestEstrutura.java:3: <identifier> expected
private int return =1;
^
TestEstrutura.java:6: <identifier> expected
public void default(String hello)
```

---

## Anotações

23





# Programação Orientada a Objetos

---

## Exemplos

```
int i = 10, int i = 11;  
byte b = 1;
```

Todo número Inteiro escrito em Java é tratado como int, desde que seu valor esteja na faixa de valores do int.

Quando declaramos uma variável do long é necessário acrescentar um literal L, caso contrário esta poderá ser tratada como int, que poderia provocar problemas.  
Long L = 10L;

## Ponto Flutuante

São os tipos que têm suporte às casas decimais e maior precisão numérica. Existem dois tipos em Java: o float e o double. O valor default é double, ou seja, todo vez que for acrescentado a literal F, no variável tipo float, ela poderá ser interpretada como variável do tipo double.

## Exemplos

```
float f = 3.1f;  
float div = 2.95F;  
double d1 = 6.35, d2 = 6.36, d3 = 6.37;  
double pi = 3.14D;
```

## Regra:

Os tipos float e double quando aparecem no mesmo programa é necessário identificá-los, para que não comprometa a precisão numérica:

```
float f = 3.1F;  
double d = 6.35;
```

Uma vez não identificado, ao tentar compilar o programa, será emitida uma mensagem de erro.

---

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

Tipos primitivos de Dados			
Tipo	Nome	Tamanho em bytes	Intervalo de valores
int	inteiro	4	- 2147483648 até 2147483647
short	inteiro curto	2	- 32768 até 32767
byte	byte	1	-128 até 127
long	inteiro longo	8	- 922372036854775808 até 922372036854775807
float	ponto flutuante	4	dependente da precisão
double	ponto flutuante	8	dependente da precisão
boolean	booleano	1	true ou false
char	caracter	2	todos os caracteres unicode

## Exemplo

```
public class TiposPrimitivos {
    public static void main ( String args [] ){
        Int x=10, y = 20;    // int
        double dolar = 2.62; // double
        float f = 23.5f;    // float
        String nome = "JAVA"; // string
        char asc = 'c';
        boolean ok = true;
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("dolar = " + dolar);
        System.out.println("f= " + f);
        System.out.println("nome = " + nome);
        System.out.println("asc = " + asc);
        System.out.println("ok = " + ok);
    }
}
```

## Anotações

26

# Programação Orientada a Objetos

---

## Inicialização de variáveis

As variáveis dentro de um método devem ser inicializadas explicitamente para serem utilizadas. Não é permitido o uso de variáveis indefinidas ou não inicializadas.

### Exemplo

```
int i;  
int a = 2;  
int c = i + a;
```

Neste caso ocorre erro, pois, o valor de **i** está indefinido.

```
public class TestVarInic {  
    public static void main(String[] args) {  
        int valor = 10;  
        valor = valor + 1;  
        System.out.println("valor = " + valor);  
    }  
}
```

Resultado: **valor = 11**

As variáveis ou atributos definidos dentro de uma de classe, são inicializadas através do construtor, que usa valores default. Valores default para boolean é **false**, para os tipos numéricos é **0** e tipo referencia é **null**;

```
public class TestVarInicClasse {  
    private static int valor;  
    public static void main(String[] args) {  
        System.out.println("valor " + valor);  
    }  
}
```

Resultado: **valor = 0**

---

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## Tipos Reference

Variáveis do tipo reference armazenam o endereço de memória estendida para um determinado objeto e a quantidade de memória varia de acordo com o objeto em questão.

Criação e Inicialização

<tipo\_de\_dado><nome\_da\_variável> = new <tipo\_da\_variável>

Somente o valor null, representando que a variável não armazena nenhuma referência.

### Exemplos

```
String s = new String();  
String s2 = "teste";
```

A classe String é a única que pode ser criada da seguinte forma:

```
String s2 = "teste";
```

## Variáveis Locais

Variáveis declaradas dentro de métodos ou blocos de código são definidas como locais. Devem ser inicializadas, senão ocorrerá um erro de compilação.

```
public class VariaveisLocais{  
    public static void main (String args []) {  
        int x=10;  
        int y;  
        System.out.println(x);  
        System.out.println(y);  
    }  
}
```

Anotações

28

# Programação Orientada a Objetos

---

O Escopo define em qual parte do programa a variável estará acessível.

```
public class Escopo {  
    public static void main (String args []){  
        int x=10;  
    }  
    {  
        System.out.println(x);  
    }  
}
```

**Anotações**

---

---

---

---

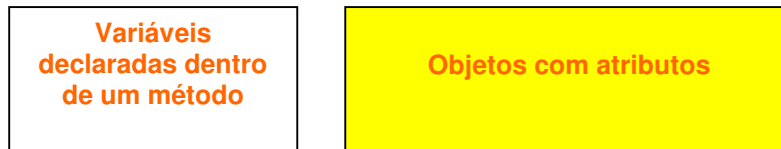
29

# Programação Orientada a Objetos

---

## Armazenamento de variáveis

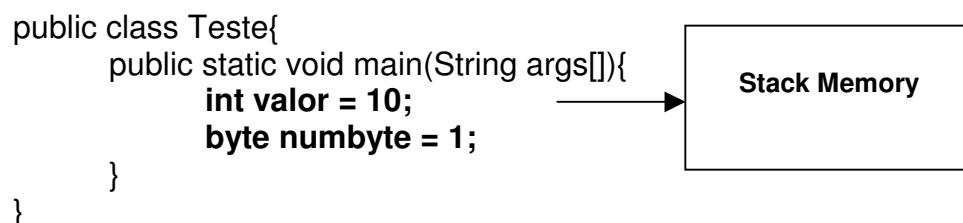
Geralmente as variáveis são armazenadas em memória. A linguagem Java possui dois locais para armazenamento de variáveis de acordo com as características.



Stack Memory

Heap Memory

Objetos e seus atributos e métodos são armazenados no Heap Memory. A Heap é dinamicamente alocada para os objetos esperarem por valor. Já Stack memory armazena as informações que serão utilizadas por um breve intervalo de tempo,



Anotações

30

## Aplicativos independentes em Java

Os aplicativos independentes, assim como qualquer tipo de programa em Java, deve conter pelo menos uma classe, que é a principal, e que dá nome ao arquivo fonte. A classe principal de um aplicativo independente deve sempre conter um método **main**, que é o ponto de início do aplicativo, e este método pode receber parâmetros de linha de comando, através de um **array** de objetos tipo **String**. Se algum dos parâmetros de linha de comando recebidos por um aplicativo precisar ser tratado internamente como uma variável numérica, deve ser feita a conversão do tipo **String** para o tipo numérico desejado, por meio de um dos métodos das classes numéricas do pacote **java.lang**.

### Método main

O método **main** é o método principal de um aplicativo em Java, constituindo o bloco de código que é chamado quando a aplicação inicia seu processamento. Deve sempre ser codificado dentro da classe que dá nome para a aplicação (sua classe principal).

No caso das Applets, a estrutura do programa é um pouco diferente, e o método **main** é substituído por métodos específicos das Applets como: **init**, **start**, etc, cada um deles tendo um momento certo para ser chamado.

```
public static void main(String[ ] parm)
```

- ↳ O método **main** pode receber argumentos de linha de comando.
- ↳ Estes argumentos são recebidos em um array do tipo String
- ↳ Um argumento de linha de comando deve ser digitado após o nome do programa quando este é chamado:

**Exemplo** → java Nomes "Ana Maria"

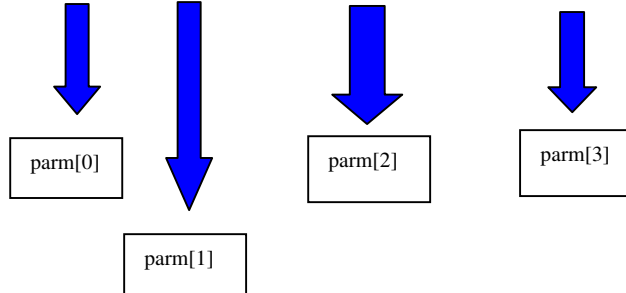
# Programação Orientada a Objetos

## Como o Java recebe os argumentos de linha de comando

Array do tipo  
String

parm[0]	parm[1]	parm[2]	parm[3]	etc...
---------	---------	---------	---------	--------

C:...\> java Nomes Renata Pedro "José Geraldo" Tatiana



- Espaços em branco são tratados como separadores de parâmetros, portanto, quando duas ou mais palavras tiverem que ser tratadas como um único parâmetro devem ser colocadas entre aspas.
- Para saber qual é o número de argumentos passados para um programa, este deve testar o atributo **`length`** do array.

```
for (i=0; i< parm.length; i++)  
    → processamento envolvendo parm[i];
```

`parm.length = 0` → indica que nenhum parâmetro foi digitado na linha de comando  
`parm.length = 5` → indica que o aplicativo recebeu 5 parâmetros pela linha de comando

Anotações

32



# Programação Orientada a Objetos

## Exemplo de programa que trabalha

Este programa mostra na tela os argumentos digitados pelo usuário na linha de comando de chamada do programa

```
public class Repete
{
    public static void main (String args[ ] )
    {
        int i;
        for (i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Anotações

33

# Programação Orientada a Objetos

---

## Operadores

Os operadores na linguagem Java, são muito similares ao estilo e funcionalidade de outras linguagens, como por exemplo o C e o C++.

### Operadores Básicos

.	referência a método, função ou atributo de um objeto
,	separador de identificadores
;	finalizador de declarações e comandos
[]	declarador de matrizes e delimitador de índices
{ }	separador de blocos e escopos locais
( )	listas de parâmetros

### Operadores Lógicos

>	Maior
>=	Maior ou igual
<	Menor
<=	Menor ou igual
= =	Igual
!=	Diferente
&&	And (e)
	Or (ou)

#### Exemplos:

i > 10;

x == y;

"Test" != "teste"

!y

x || y

Anotações

34

# Programação Orientada a Objetos

---

## Operadores Matemáticos

Operadores Binários	
+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)
Operadores Unários	
++	Incremento
--	Decremento
+=	Combinação de soma e atribuição
-=	Combinação de subtração e atribuição
*=	Combinação de multiplicação e atribuição
/=	Combinação de divisão e atribuição
%=	Combinação de módulo e atribuição
Operadores Terciários	
? :	Verificação de condição (alternativa para o if...else)

### Exemplos

```
int a = 1;  
int b = a + 1;  
int c = b - 1;  
int d = a * b;  
short s1 = -1;  
short s2 = 10;  
short s1++;  
int c = 4 % 3;
```

Anotações

35

# Programação Orientada a Objetos

---

## Outros Operadores

**Instanceof** Faz a comparação do objeto que está “instanciado” no objeto.  
**new** Este operador é usado para criar novas “instance” de classes.

### Exemplos

#### instanceof

```
Objeto obj = new String("Teste");  
if (obj instanceof String)  
    System.out.println("verdadeiro");
```

#### new

```
Hello hello = new Hello();
```

## Precedências de Operadores

. [ ] ( )  
\* / %  
+ -

### Exemplo:

Com a precedência definida pelo Java

```
int c = 4 / 2 + 4;
```

Neste caso, primeiro ocorrerá a divisão e após a soma.

Com a precedência definida pelo desenvolvedor

```
int a = 4 / (2 + 4);
```

Já neste caso, primeiro ocorrerá a soma e depois a divisão.

# Programação Orientada a Objetos

## Operadores Binários

Java fornece extensa capacidade de manipulação de bits. Todos os dados são representados internamente como uma sequência de bits. Cada bit pode assumir o valor de 0 ou 1. No geral uma sequência de bits formam um byte, que é a unidade de armazenamento padrão, para uma variável tipo byte. Os tipos de dados são armazenados em quantidade maiores que byte. Os operadores sobre os bits são utilizados para manipular os bits de operandos integrais (isto é aqueles do tipo byte, char, short, int e long).

Operador	Nome	Descrição		
&	AND	Mask Bit	Input Bit	Output Bit
		1	1	1
		1	0	0
		0	1	0
		0	0	0
		AND=1 and 1 produz 1. Qualquer outra combinação produz 0		
		Exemplo: & 00110011 11110000 00110000		
	OR	Mask Bit	Input Bit	Output Bit
		1	1	1
		0	1	1
		1	0	1
		0	0	0
		OR = 0 or 0 produz 0. Qualquer outra combinação produz 1.		
		Exemplo:   00110011 11110000 11110011		
^	XOR	Mask Bit	Input Bit	Output Bit
		1	1	0
		0	1	1
		1	0	1
		0	0	0
		XOR = 1 xor 0 ou 0 xor 1 produz 1. Qualquer outra combinação produz 0.		
		Exemplo: ^00110011 11110000 ===== 11000011		

Anotações

37

# Programação Orientada a Objetos

## Operadores Binários

Operador	Nome	Descrição
<<	Deslocamento para esquerda	Desloca os bits do primeiro operando para a esquerda pelo número de bits especificado pelo segundo operando; preenche a partir da direita com bits 0.
>>	Deslocamento para direita com extensão de sinal	Desloca os bits do primeiro operando para a direita pelo número de bits especificado pelo segundo operando. Se o primeiro operando for negativo, preenche com 1s a partir da esquerda; caso contrário, preenche com 0s a partir da esquerda.
>>>	Deslocamento para direita com extensão de zero	Desloca os bits do primeiro operando para a direita pelo número de bits especificado pelo segundo operando; 0s são inseridos a partir da esquerda.
~	Complemento de um	Para cada número negativo, cada bit de cada número é invertido (zeros são substituídos com um e vice versa). Exemplo: int x = 45 <=> Representação em bits (00101101) ~ x <=> Representação em bits (11010010)

### Manipulação de bits e os operadores sobre bits

#### Exemplos de manipulação do bits:

Right Shift (>>)

$$\begin{aligned} 128 >> 1 &\Leftrightarrow 128/2^1 &= 64 \\ 256 >> 4 &\Leftrightarrow 256/2^4 &= 16 \end{aligned}$$

Left Shift (<<)

$$\begin{aligned} 128 << 1 &\Leftrightarrow 128 * 2^1 &= 256 \\ 16 << 2 &\Leftrightarrow 16 * 2^2 &= 64 \end{aligned}$$

Anotações

38

# Programação Orientada a Objetos

---

## Operadores

Podemos realizar operações de Pré e Pós incremento e de Pré e Pós decremento.

### Exemplos

`x = 10;`      `y = 1 + x`      O valor da variável y é 11 e de x = 11

ou

`x = 10;`      `y = x++`      O valor da variável y é 11 e de x= 11

### Pós-incremento:

`x = 10;`      `y = x + 1;`      O valor da variável y é 11 e de x = 11

ou

`x = 10;`      `y = x++`      O valor da variável y é 10 e de x = 11

Vamos esclarecer antes que haja confusão que para isto precisamos separar a operação `y = x++` em duas fases.

Fase 1:      Nesta fase o valor de x (10) é atribuído para y, logo `y(10) = x(10)`  
`y = x`

Fase 2:      Nesta fase o valor de x (10) é incrementado em 1, logo `x(11)`  
`x++`

Observe que isto ocorre devido a precedência de operadores. Primeiro é feito a atribuição e depois o incremento.

## Operadores de Atribuição

### Exemplo

`op1 = op2;` atribui o valor de op2 para op1  
*não confundir com '=' (comparação)*

OBS: Nunca usar `= =` para comparar variáveis do tipo reference.

### Exemplo

`nome1 = = nome2`

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## String

### A classe String

Objetos String são sequências de caracteres Unicode

#### Exemplo

```
String nome = "Meu nome"
```

#### Principais métodos

**Substituição:** replace

**Busca:** endWiths, startsWith, indexOf e lastIndexOf

**Comparações:** equals, equalsIgnoreCase e compareTo

**Outros:** substring, toLowerCase, toUpperCase, trim, charAt e length

**Concatenação:** concat e operador +

#### Exemplo

O operador + é utilizado para concatenar objetos do tipo String, produzindo uma nova String:

```
String PrimeiroNome = "Antonio";  
String SegundoNome = "Carlos";  
String Nome = PrimeiroNome + SegundoNome
```

#### Exemplo

```
String nome = "Maria";  
if(nome.equals("qualquer nome")){  
    System.out.println("nome = qualquer nome");  
}
```

```
String nome1 = "Maria";  
String nome2 = "maria";  
if(nome1.equalsIgnoreCase(nome2)){  
    System.out.println("Iguais");  
}
```

#### Anotações

---

---

---

---

40



# Programação Orientada a Objetos

---

```
String nome1 = "Maria";
String nome2 = "Joao";
int dif = nome1.compareTo(nome2);
if(dif == 0){
    System.out.println("Iguais");
}
```

A classe String possui métodos estáticos e dinâmicos, como apresentados nos exemplos a seguir:

		Para que serve
(1)	<pre>char letra; String palavra = "Exemplo" letra = palavra.charAt(3)  Resultado → letra = m</pre>	Método dinâmico, que retorna o caracter existente na posição indicada dentro de uma string.
(2)	<pre>String palavra01 = "Maria " String nome; nome = palavra01.concat(" Renata");  Resultado → nome = "Maria Renata"</pre>	Método dinâmico, que retorna uma string produzida pela concatenação de outras duas.
(3)	<pre>int pos; String palavra = "prova"; pos = palavra.indexOf('r');  Resultado → pos = 1</pre>	Método dinâmico, que retorna um número inteiro indicando a posição de um dado caracter dentro de uma string.
(4)	<pre>int pos; String nome = "Jose Geraldo"; pos = nome.indexOf("Ge");  Resultado → pos = 5</pre>	Método dinâmico, que retorna um número inteiro indicando a posição de uma dada substring dentro de uma string.
(5)	<pre>int tamanho; String palavra = "abacaxi"; tamanho = palavra.length();  Resultado → tamanho = 7</pre>	Método dinâmico, que retorna um número inteiro representando o tamanho de uma string (número de caracteres que constituem a string).

Anotações

41

## Programação Orientada a Objetos

(6)	String <b>texto</b> = "Isto e um exemplo"; String nova; nova = <b>texto</b> .substring(7, 9);  Resultado → nova = um	Método dinâmico, que retorna a substring existente em dada posição de uma string.
(7)	String <b>palavra</b> = "Estudo"; String nova; nova = <b>palavra</b> .toLowerCase();  Resultado → nova = estudo	Método dinâmico, que retorna uma nova string, formada pelo conteúdo de uma string dada, com os caracteres convertidos para formato minúsculo.
(8)	String <b>palavra</b> = "Estudo"; String nova; nova = <b>palavra</b> .toUpperCase();  Resultado → nova = ESTUDO	Método dinâmico, que retorna uma nova string, formada pelo conteúdo de uma string dada, com os caracteres convertidos para formato maiúsculo.
(9)	int nro = 17; String nova; Nova = <b>String</b> .valueOf(nro);  Resultado → nova = 17	Método estático que retorna uma string, criada a partir de um dado numérico em formato inteiro.
(10)	float valor = 28.9; String nova; nova = <b>String</b> .valueOf(valor);  Resultado → nova = 28.9	Método estático que retorna uma string, criada a partir de um dado numérico em formato de ponto flutuante.

Observe que nos exemplos (9) e (10) os métodos são chamados por meio da classe, porque são métodos estáticos ou métodos de classe (não requerem instanciamento da classe para serem chamados). Já nos exemplos (1) até (8) a chamada dos métodos é feita por meio de um objeto da classe **String**, porque estes métodos são dinâmicos, ou métodos de instância.

# Programação Orientada a Objetos

---

## StringBuffer

Objetos StringBuffer são uma sequência mutável de caracteres Unicode.

### Construtores:

StringBuffer – Cria um buffer vazio

StringBuffer(int capacidade) – Cria um buffer com a capacidade especificada

StringBuffer(String initialstring) – Cria um buffer contendo uma string informada.

**Principais métodos: append, insert, delete, ...**

### Exemplo

```
public class TestStringBuffer{
    public static void main(String arg[]){
        StringBuffer b1 = new StringBuffer();
        StringBuffer b2, b3;
        b2 = new StringBuffer(25);
        b3 = new StringBuffer("Teste, ");
        /*para exibir o conteúdo é necessário
        usar o método toString */
        System.out.println("b1:" + b1.toString() );
        System.out.println("b2:" + b2.toString() );
        System.out.println("b3:" + b3.toString() );
        b3.append("vamos testar novamente");
        System.out.println("b3:" + b3.toString() );
        b3.insert(0, "A x B ");
        System.out.println("b3:" + b3.toString() );
        b3.delete(0,4);
        System.out.println("b3:" + b3.toString() );
        b2.append("Teste b2");
        System.out.println("b2 Capacidade: " + b2.capacity());
        System.out.println("b2 Tamanho: " + b2.length());
        b2.append(b3);
        System.out.println("b2: " + b2.toString() );
        System.out.println("b2 tamanho: " + b2.length() );
        System.out.println("b2 invertida: " + b2.reverse() );
    }
}
```

### Anotações

---

---

---

---

43

# Programação Orientada a Objetos

## Exemplos de métodos da classe StringBuffer

	Resultado
<pre>StringBuffer frase = new StringBuffer("Isto e um "); frase.append("exemplo");</pre> <p>Resultado → frase = Isto e um exemplo</p>	Método dinâmico que altera o conteúdo de um objeto StringBuffer, adicionando uma string ao final do mesmo.
<pre>char letra; StringBuffer palavra = new StringBuffer("Exemplo"); letra = palavra.charAt(3);</pre> <p>Resultado → letra = m</p>	Método dinâmico, que retorna o caracter existente na posição indicada dentro de um objeto StringBuffer.
<pre>StringBuffer palavra = new StringBuffer("torno"); palavra.insert(4, "ei");</pre> <p>Resultado → palavra = torneio</p>	Método dinâmico que altera o conteúdo de um objeto StringBuffer, inserindo uma string em dada posição do mesmo.
<pre>int nroCaracteres; StringBuffer palavra = new StringBuffer("cadeira"); nroCaracteres = palavra.length( );</pre> <p>Resultado → NroCaracteres = 7</p>	Método dinâmico que retorna um número inteiro indicando o número de caracteres de um objeto StringBuffer.
<pre>StringBuffer palavra = new StringBuffer("mundial"); palavra = palavra.reverse( );</pre> <p>Resultado → palavra = laidnum</p>	Método dinâmico que inverte o conteúdo de um objeto StringBuffer.
<pre>StringBuffer palavra = new StringBuffer("tecer"); palavra.setCharAt(2,'m');</pre> <p>Resultado → Palavra = temer</p>	Método dinâmico que substitui, por outro, o caracter em dada posição de um objeto StringBuffer.

# Programação Orientada a Objetos

---

## Exemplo de utilização de strings da classe StringBuffer

```
import java.lang.*;

public class InverteString
{
    public static String inverte(String origem)
    {
        int i;
        int tamanho = origem.length();
        StringBuffer destino = new StringBuffer(tamanho);

        for (i = (tamanho - 1); i >= 0; i--)
        {
            destino.append(origem.charAt(i));
        }
        return destino.toString();
    }

    public static void main(String p[])
    {
        String textoOriginal, textoInvertido;

        if (p.length == 0)
            textoOriginal = "Sem parâmetros";
        else

            textoOriginal = p[0];

        textoInvertido = InverteString.inverte(textoOriginal);
        System.out.println("Texto original = " + textoOriginal);
        System.out.println("Texto invertido = " + textoInvertido);
    }
}
```

**Anotações**

---

---

---

---

45

# Programação Orientada a Objetos

---

## Classe Math

Classe pertencente ao pacote **java.lang**. Esta classe contém métodos que permitem a execução das principais funções matemáticas como logaritmo, funções trigonométricas, raiz quadrada, etc. Todos os métodos da classe **Math** são métodos de classe (estáticos), portanto esta classe não precisa ser estanciada para que seus métodos possam ser chamados.

É uma classe tipo **final** →

```
public final class Math
```

**final** → não pode ser estendida por outras classes

### Atributos da classe Math

<b>public static final double E</b>	<ul style="list-style-type: none"><li>↳ Número <b>e</b> que é a base dos logaritmos naturais</li><li>↳ Este número é uma constante</li></ul>
<b>public static final double PI</b>	<ul style="list-style-type: none"><li>↳ Número <b>pi</b></li><li>↳ Este número é uma constante</li></ul>

- ↳ O modificador **final** indica que estes valores são constantes
- ↳ O modificador **static** indica que só é gerada uma cópia destes valores para toda a classe, ou seja, são atributos de classe.

---

### Anotações

46

# Programação Orientada a Objetos

---

## Exemplos de métodos da classe Math

	Resultado
<pre>double x = -345.17; double y; y = Math.abs(x);</pre> <p>Resultado → y = 345.17</p>	Método estático que retorna o valor absoluto de um dado número.
<pre>double x = 25; double y = 3; double z; z = Math.IEEEremainder(x,y);</pre> <p>Resultado → z = 1.0</p>	Método estático que retorna o valor do resto da divisão de dois números.
<pre>double x = 625; double y; y = Math.sqrt(x);</pre> <p>Resultado → y = 25</p>	Método estático que retorna o valor da raiz quadrada de um número.
<pre>double x = 25; double y = 3; double z; z = Math.pow(x,y);</pre> <p>Resultado → z = 15625.0</p>	Método estático que retorna o valor da potência de um número por outro.

Anotações

47

# Programação Orientada a Objetos

---

## Classe Integer

Classe pertencente ao pacote **java.lang**. Esta classe é uma extensão da classe abstrata **Number**, portanto implementa seus métodos. A classe **Integer** é um empacotamento do tipo de dado primitivo **int**, necessária, por exemplo, para a conversão de String para o tipo primitivo **int**. Disponibiliza algumas constantes como **MAX\_VALUE** e **MIN\_VALUE**, úteis em cálculos envolvendo dados tipo inteiro. Esta classe fornece constantes e métodos para trabalhar com valores inteiros, como por exemplo, a conversão de inteiro em String e vice-versa.

A classe **Integer** possui métodos estáticos e também métodos dinâmicos

É uma classe tipo final:

```
public final class Integer
```

**final** → não pode ser  
estendida por outras classes



# Programação Orientada a Objetos

---

## Atributos da classe Integer

<b>public static final int MAX_VALUE</b>	↳ Constante que indica o maior valor do tipo de dados primitivo <b>int</b> <b>Exemplo:</b> int x; x = Integer.MAX_VALUE <u>Resultado</u> → x = 2147483647
<b>public static final int MIN_VALUE</b>	↳ Constante que indica o menor valor do tipo de dados primitivo <b>int</b> <b>Exemplo:</b> int x; x = Integer.MIN_VALUE <u>Resultado</u> → x = - 2147483648

## Exemplos de criação de variáveis da classe Integer

Integer x = new Integer(17) <u>Resultado</u> → x = 17	↳ Cria uma variável da classe com o valor de um inteiro (dado primitivo)
Integer x = new Integer("28") <u>Resultado</u> → x = 28	↳ Cria uma variável da classe com o valor obtido a partir da conversão de um objeto tipo String

Anotações

49

# Programação Orientada a Objetos

## Exemplos de métodos da classe Integer

(1)	<pre>Integer x = new Integer(17); double y; y = x.doubleValue();</pre> <p>Resultado → y = 17.0</p> <p><b>OBS:</b> os métodos floatValue( ), longValue( ), intValue( ) e shortValue( ) funcionam da mesma forma que o doubleValue( )</p>	Método dinâmico que converte em double o conteúdo de um objeto Integer.
(2)	<pre>Integer x = new Integer(17); String str; str = x.toString();</pre> <p>Resultado → str = "17"</p>	Método dinâmico que converte em String o conteúdo de um objeto Integer.
(3)	<pre>String str = "35"; Integer x; x = Integer.valueOf(str);</pre> <p>Resultado → x = 35</p>	Método estático que converte em um objeto Integer o conteúdo de um objeto String.
(4)	<pre>int nro; String str = "348"; nro = Integer.parseInt(str);</pre> <p>Resultado → nro = 348</p>	Método estático que converte para o formato primitivo <b>int</b> o conteúdo de um objeto String.

Observe que os exemplos (1) e (2) apresentam métodos dinâmicos da classe Integer (chamados por meio de um objeto desta classe), enquanto que os exemplos (3) e (4) demonstram métodos estáticos (chamados a partir da própria classe).

Anotações

50

# Programação Orientada a Objetos

---

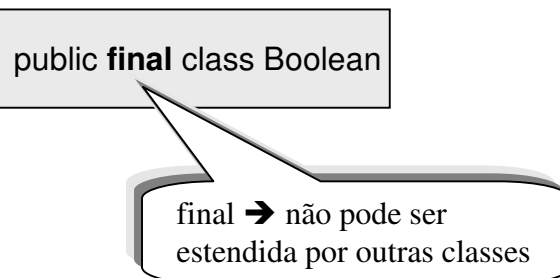
## Classes Long, Short, Byte, Float e Double

Estas classes também pertencem ao pacote **java.lang**, e da mesma forma que a classe **Integer**, são extensões da classe abstrata **Number**. Elas oferecem atributos, construtores e métodos equivalentes aos já apresentados para a classe **Integer**. Também são classes tipo **final**, portanto não podem ser estendidas por outras classes.

### Classe Boolean

Classe pertencente ao pacote **java.lang**, é um empacotamento do tipo de dado primitivo **boolean**. Esta classe fornece constantes e métodos para trabalhar com variáveis booleanas, como por exemplo, a conversão de booleano em String e vice-versa. A classe **Boolean** possui métodos estáticos e também métodos dinâmicos

É uma classe tipo final:



# Programação Orientada a Objetos

---

## Atributos da classe Boolean

<code>public static final Boolean FALSE</code>	↳ Constante que representa o valor primitivo <b>false</b> <b>Exemplo:</b> Boolean.FALSE <b>Resultado</b> → false
<code>public static final Boolean TRUE</code>	↳ Constante que representa o valor primitivo <b>true</b> <b>Exemplo:</b> Boolean.TRUE <b>Resultado</b> → true

## Construtores da classe Boolean

<code>public Boolean(boolean valor)</code>	↳ Cria uma variável da classe Boolean com o conteúdo da variável booleana <b>valor</b> <b>Exemplo:</b> Boolean resposta = new Boolean(false) <b>Resultado</b> → resposta = false
<code>public Boolean(String str)</code>	↳ Cria uma variável da classe com Boolean com o conteúdo <b>true</b> ou <b>false</b> , dependendo do conteúdo da string <b>str</b> : ↳ se o conteúdo de <b>str</b> for igual a <b>true</b> então o novo objeto será criado com o conteúdo <b>true</b> ↳ caso contrário o novo objeto será criado com o conteúdo <b>false</b> <b>Exemplo:</b> String str = "exemplo"; Boolean resp = new Boolean(str); <b>Resultado</b> → resp = false

Anotações

52

# Programação Orientada a Objetos

---

## Classe Character

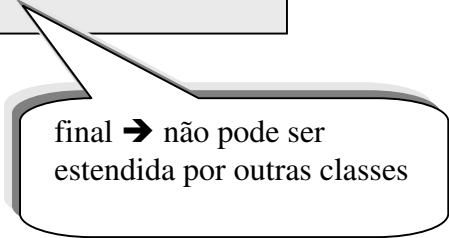
Classe pertencente ao pacote **java.lang**. A classe **Character** é um empacotamento do tipo de dado primitivo **char**.

Os caracteres dentro da linguagem Java são campos de 2 bytes e formam o conjunto de caracteres conhecido como Unicode (codificação introduzida pela linguagem Java para permitir o tratamento de acentuação e outros caracteres como os do alfabeto grego e dos idiomas orientais).

Esta classe fornece constantes e métodos para trabalhar com objetos da classe Character, como por exemplo, a conversão em String, em maiúsculo e minúsculo, etc. A classe **Character** possui métodos estáticos e também métodos dinâmicos.

É uma classe tipo **final**:

```
public final class Character
```



final → não pode ser  
estendida por outras classes

## Objetos X Tipos primitivos de dados

- Todos os tipos primitivos de dados, incluindo os não numéricos (char e boolean) possuem uma classe correspondente no pacote java.lang. Estas classes funcionam como um encapsulamento destes dados primitivos, oferecendo informações adicionais sobre os mesmos, como por exemplo seus limites MIN\_VALUE e MAX\_VALUE.
- As operações aritméticas elementares como soma, subtração, multiplicação, divisão e módulo são aplicáveis apenas aos tipos primitivos e não aos objetos das classes invólucro correspondentes.
- As classes invólucro, oferecem um conjunto de métodos, estáticos e dinâmicos, que implementam funcionalidades como por exemplo a conversão entre tipos primitivos e objetos.
- Os nomes dos tipos primitivos começam com letra minúscula e o das classes invólucro correspondentes começam com letra maiúscula.

Tipo primitivo	Classe
char	Character
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

# Programação Orientada a Objetos

---

## Fluxo de Controle

Java como qualquer outra linguagem de programação, suporta instruções e laços para definir o fluxo de controle. Primeiro vamos discutir as instruções condicionais e depois as instruções de laço.

### Instrução if

Comando if ... else	
<pre>if (expressão) { comando-1;   comando-2;   ...   comando-n; } else { comando-1;   comando-2;   ...   comando-n; }</pre>	<p>⇒ Quando houver apenas um comando dentro do <b>bloco if</b> ou do <b>bloco else</b>, não é necessário o uso das chaves.</p> <p>⇒ Quando houver comandos <b>if</b> aninhados, cada <b>else</b> está relacionado ao <b>if</b> que estiver dentro do mesmo bloco que ele.</p>

Anotações

55

# Programação Orientada a Objetos

---

## Instrução switch

<b>Comando switch</b>  <b>switch</b> (expressão) { <b>case</b> constante-1: bloco de comandos; <b>break</b> ; <b>case</b> constante-2: bloco de comandos; <b>break</b> ; ... <b>default</b> : bloco de comandos; }	<p>↳ O comando <b>switch</b> faz o teste da expressão de seleção contra os valores das constantes indicados nas cláusulas <b>case</b>, até que um valor verdadeiro seja obtido.</p> <p>↳ Se nenhum dos testes produzir um resultado verdadeiro, são executados os comandos do bloco default, se codificados.</p> <p>↳ O comando <b>break</b> é utilizado para forçar a saída do switch. Se for omitido, a execução do programa continua através das cláusulas <b>case</b> seguintes.</p> <p><u><b>Exemplo:</b></u></p> <pre>switch (valor) {     case 5:     case 7:     case 8:         printf (•••)         break;     case 10:     ... }</pre>
---	---

Anotações

56



# Programação Orientada a Objetos

---

## Operadores Ternários

E por fim o operador ternário, Java, oferece uma maneira simples de avaliar uma expressão lógica e executar uma das duas expressões baseadas no resultado.

O operador condicional ternário (? :). É muito parecido com a instrução `iif()` presente em algumas linguagens, Visual Basic, por exemplo.

### Sintaxe

(<expressão boolean>) ? <expressão true> : <expressão false>

ou

variável = (<expressão boolean>) ? <expressão true> : <expressão false>

```
int a = 2;  
int b = 3;  
int c = 4;  
a = b > c ? b : c;
```

É a mesma coisa que:

```
if(b > c)  
    a = b;  
else  
    a = c;
```

## Laços

O que são laços?

Laços são repetições de uma ou mais instruções até que uma condição seja satisfeita. A linguagem Java tem dois tipos de laços: os finitos e os infinitos.

Para os laços finitos a execução está atrelada a satisfação de uma condição, por exemplo:

### Laços

```
while (<boolean-expression>)  
<statements>...
```

```
do  
<statements>...  
while (<boolean-expression>);
```

```
for (<init-stmts>...; <boolean-expression>; <exprs>...)  
<statements>...
```

# Programação Orientada a Objetos

## Instrução for

Comando for	
<pre>for (inicialização; condição; incremento) {     bloco de comandos;     if (condição-2)         break;     bloco de comandos; }</pre> <p><b>Loop eterno</b></p> <pre>for ( ; ; ) {     bloco de comandos;     if (condição)         break;     bloco de comandos; }</pre> <p>☞ Um comando <b>for</b> pode ser controlado por mais de uma variável, e neste caso elas devem ser separadas por vírgulas.</p> <p><b>Exemplo:</b></p> <pre>for (x=1, y=2 ; x+y &lt; 50 ; x++, y++) {     bloco de comandos; }</pre>	<p><b>inicialização</b> → comando de atribuição que define o valor inicial da variável que controla o número de repetições.</p> <p><b>condição</b> → expressão relacional que define até quando as iterações serão executadas. Os comandos só são executados enquanto a condição for verdadeira. Como o teste da condição de fim é feito antes da execução dos comandos, pode acontecer destes comandos nunca serem executados, se a condição for falsa logo no início.</p> <p><b>incremento</b> → expressão que define como a variável de controle do laço deve variar (seu valor pode aumentar ou diminuir).</p>

## Instrução while e do while

Anotações

59

# Programação Orientada a Objetos

---

Comando while	
<pre><b>while</b> (condição) {     bloco de comandos;     if (condição-2)         <b>break</b>;     bloco de comandos; }</pre>	<p><b>condição</b> → pode ser qualquer expressão ou valor que resulte em um verdadeiro ou falso.</p> <p>O laço <b>while</b> é executado <b>enquanto a condição for verdadeira</b>. Quando esta se torna falsa o programa continua no próximo comando após o <b>while</b>.</p> <p>Da mesma forma que acontece com o comando <b>for</b>, também para o <b>while</b> o teste da condição de controle é feito no início do laço, o que significa que se já for falsa os comandos dentro do laço não serão executados.</p>
Comando do ... while	
<pre><b>do</b> {     bloco de comandos;     if (condição-2)         <b>break</b>;     bloco de comandos; } <b>while</b> (condição);</pre>	<p>Sua diferença em relação ao comando <b>while</b> é que o teste da condição de controle é feito no final do laço, o que significa que os comandos dentro do laço são sempre executados pelo menos uma vez.</p>

Anotações

---

---

---

---

60

# Programação Orientada a Objetos

---

## Comandos return e break

Comando return	
<b>return</b> expressão;	Comando usado para encerrar o processamento de uma função, retornando o controle para a função chamadora  <b><u>expressão</u></b> → é opcional, e indica o valor retornado para a função chamadora
Comando break	
<b>break</b> ;	Comando usado para terminar um <b><u>case</u></b> dentro de um comando <b><u>switch</u></b> , ou para forçar a saída dos laços tipo <b><u>for</u></b> , <b><u>while</u></b> ou <b><u>do ... while</u></b>

## Comandos continue e exit

Comando continue	
<b>continue</b> ;	Comando usado dentro dos laços tipo <b><u>for</u></b> , <b><u>while</u></b> ou <b><u>do ... while</u></b> com o objetivo de forçar a passagem para a próxima iteração. (os comandos do laço que ficarem após o continue são pulados e é feita a próxima verificação de condição do laço, podendo este continuar ou não).
Comando exit	
<b>exit</b> (código de retorno);	Comando usado para terminar a execução de um programa e devolver o controle para o sistema operacional.  <b><u>código de retorno</u></b> → é obrigatório. Por convenção é retornado o valor zero (0) quando o programa termina com sucesso, e outros valores quando termina de forma anormal.

Anotações

---

---

---

---

61

# Programação Orientada a Objetos

---

## Arrays

São estruturas de dados capazes de representar uma coleção de variáveis de um mesmo tipo. Todo array é uma variável do tipo Reference e por isto, quando declarada como uma variável local deverá sempre ser inicializada antes de ser utilizada.

### Arrays Unidimensionais

#### Exemplo

```
int [ ] umArray = new int[3];
umArray[0] =1;
umArray[1] = -9;
umArray[2] =0;
int [ ] umArray ={1,-9,0};
```

Para obter o número de elementos de um array utilizamos a propriedade **length**.

```
int [ ] umArray = new int[3];
umArray[0] =1;
umArray[1] = -9;
umArray[2] =0;
System.out.println("tamanho do array =" + umArray.length);
```

### Arrays Bidimensionais

#### Exemplo

```
int [ ] [ ] array1 = new int [3][2];
int array1 [ ] [ ] = new int [3][2];
int [ ] array1[ ] = new int [3][2];
```

#### Inserido valores

```
Array1[0][0] = 1;
Array1[0][1] = 2;
Array1[1][0] = 3;
Array1[1][1] = 4;
Array1[2][0] = 5;
Array1[2][1] = 6;
int Array1[ ][ ]= { {1,2} , {3,4} ,{5,6} };
```

#### Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## Arrays Multidimensionais

Todos os conceitos vistos anteriormente são mantidos para arrays de múltiplas dimensões.

### Exemplo

```
int [ ] [ ] [ ] array1 = new int [2][2][2];
```

---

Anotações

63

# Programação Orientada a Objetos

---

## Método arraycopy

Permite fazer cópias dos dados array de um array para outro.

### Sintaxe

arraycopy(Object origem,  
int IndexOrigem,  
Object destino,  
int IndexDestino,  
int tamanho)

```
public class TestArrayCopy {  
    public static void main(String[] args) {  
        char[] array1 = { 'j', 'a', 'v', 'a', 'l', 'i' };  
        char[] array2 = new char[4];  
        System.arraycopy(array1, 0, array2, 0, 4);  
        System.out.println(new String(array2));  
    }  
}
```

## Método Sort

Para ordenar um array, usamos o método sort da classe java.util.Arrays

### Exemplo

```
import java.util.*;  
public class TestArraySort {  
    public static void main(String[] args) {  
        int[] numero = {190,90,87,1,50,23,11,5,55,2};  
        //Antes da ordenação  
        displayElement(numero);  
        //Depois da ordenação  
        Arrays.sort(numero);  
        displayElement(numero);  
    }  
    public static void displayElement(int[] array){  
        for(int i=0;i < array.length; i++){  
            System.out.println(array[i]);  
        }  
    }  
}
```

**Anotações**

64



## Tratamento de exceções

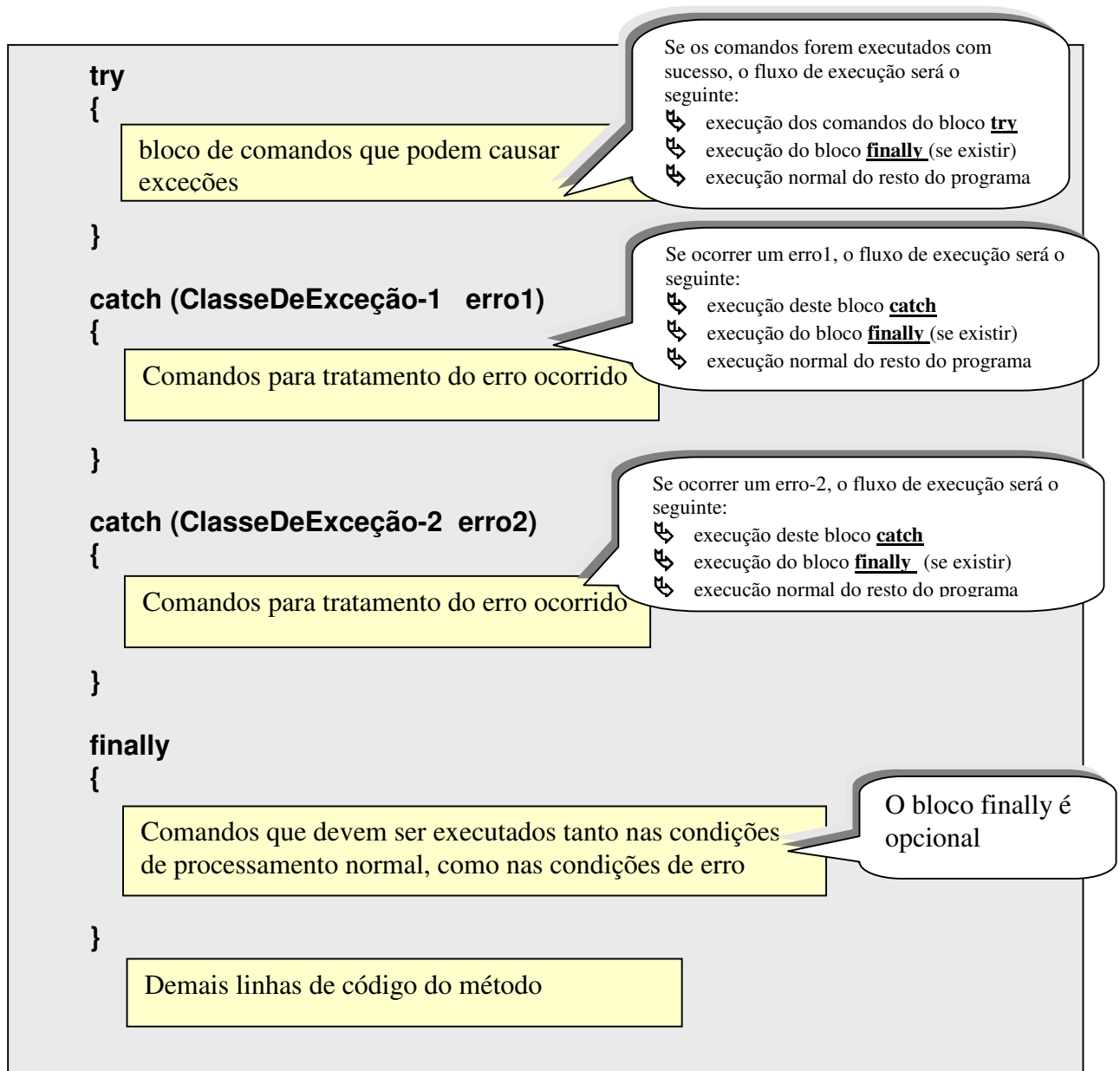
Uma exceção é um erro que pode acontecer em tempo de processamento de um programa. Existem exceções causadas por erro de lógica e outras provenientes de situações inesperadas, como por exemplo um problema no acesso a um dispositivo externo, como uma controladora de disco, uma placa de rede, etc.

Muitas destas exceções podem ser previstas pelo programador, e tratadas por meio de alguma rotina especial, para evitar que o programa seja cancelado de forma anormal.

Dentro dos pacotes do Java existem alguns tipos de exceções já previamente definidas por meio de classes específicas. Estas exceções, pré estabelecidas, fazem parte dos pacotes do Java, e todas elas são sub-classes da classe **Throwable**, que pertence ao pacote **java.lang**.

A detecção e controle de exceções, dentro de um programa Java, deve ser feita pelo programador através da estrutura **try-catch-finally**, conforme mostrado no esquema a seguir:

# Programação Orientada a Objetos



Se ocorrer um erro não previsto pelos blocos catch codificados, o fluxo de execução será o seguinte:

- desvio para o bloco **finally** (se existir)
- **saída** do método **sem** execução do resto do código

Anotações

66

# Programação Orientada a Objetos

---

Todo método que pode provocar algum tipo de exceção em seu processamento indica esta condição por meio da palavra-chave **throws**, em sua documentação, como mostra o exemplo a seguir:

```
public void writeInt(int valor)
    throws IOException
```

Neste exemplo, temos o método **writeInt** do pacote **java.io**, que serve para gravar um valor inteiro em um dispositivo de saída de dados, como um arquivo.

Este método pode causar um tipo de erro tratado pelo Java como **IOException**. Portanto uma chamada deste método deve ser codificada dentro de um bloco **try-catch**, como mostrado abaixo:

```
•
•
int nro;
FileOutputStream arquivo;
arquivo = new FileOutputStream (new File("exemplo.txt"));
•
•
try
{
    for (nro=5;nro<50;nro=nro+5)
    {
        arquivo.writeInt(28);
    }
}
catch (IOException erro)
{
    System.out.println("Erro na gravação do arquivo" +
erro);
}
•
•
```

Além das exceções pré-definidas um programador pode também criar suas próprias exceções, quando necessário. Para tanto deve criá-la quando da definição do método. Esta situação não será abordada por enquanto.

Anotações

67

# Programação Orientada a Objetos

---

## Empacotamento de Classes

É possível trabalhar com vários tipos de dados, inteiros, ponto flutuantes, texto, lógico e etc. Todavia em alguns casos há a necessidade de fazer conversão de um tipo de dado para outro. Bem, começaremos com a conversão de tipos primitivos para tipo referência (objetos), esta operação também é chamada de **Wrapper Class**.

Veja abaixo a tabela que associa os tipos.

<b>boolean</b>	<b>byte</b>	<b>Char</b>	<b>short</b>	<b>int</b>	<b>long</b>	<b>float</b>	<b>double</b>
Boolean	Byte	Character	Short	Integer	Long	Float	Double

### Exemplos

```
public class TestWrapperClass {
    public static void main (String args [ ]){
        int plInteger = 500; // Tipo primitivo
        Integer wInteger = new Integer(plInteger); // Wrapper Class
        int p1 = wInteger.intValue();
        //Conversão de valores:
        String valueStr = "10";
        double d1 = Double.valueOf(valueStr).intValue();
        //ou
        double d2 = Double.parseDouble(valueStr);
    }
}
```

Anotações

68

## Conversão de Tipos

### // Convertendo String em integer

```
int variavel = 42;  
String str = Integer.toString(variavel);  
ou  
int variavel = 42;  
String str = "" + variavel;  
ou  
int variavel = 42;  
String str = String.valueOf(variavel);
```

### // Convertendo String em Long

```
long variavel = Long.parseLong(str);  
ou  
long variavel = Long.valueOf(str).longValue();
```

### // Convertendo String em Double

```
double variavel = Double.valueOf(str).doubleValue();
```

### // Convertendo String em Float

```
float variavel = Float.valueOf(str).floatValue();
```

### // Convertendo Integer em String

```
String str = "14";  
int var = Integer.parseInt(str);  
ou  
String str = "14";  
int var = Integer.parseInt(str);  
ou  
String str = "14";  
int var = Integer.valueOf(str).intValue();
```

# Programação Orientada a Objetos

---

## Mais exemplos

### // Convertendo Integer em código ASCII

```
char c = 'C';  
int i = (int) c;
```

### // Convertendo Double em String

```
String str = Double.toString(i);
```

### // Convertendo Long em String

```
String str = Long.toString(l);
```

### // Convertendo Float em String

```
String str = Float.toString(f);
```

### // Convertendo Decimal em Hexadecimal

```
int i = 12;  
String hexstr = Integer.toString(i, 16);  
ou  
String hexstr = Integer.toHexString(i);
```

### // Convertendo Decimal em Binário:

```
int i = 12;  
String binstr = Integer.toBinaryString(i);
```

### // Convertendo Hexadecimal em Integer:

```
int i = Integer.valueOf("B8DA3", 16).intValue();  
ou  
int i = Integer.parseInt("B8DA3", 16);
```

---

## Anotações

---

---

---

---

70

# Programação Orientada a Objetos

---

## A referência **this**

Na classe `PessoaJuridica`, o uso da palavra **this** é para evitar ambigüidade. Note que no método `setIdade`, o parâmetro chama-se `idade` e o atributo também tem o mesmo nome, ou seja, o mesmo identificador, neste caso o **this** separa o atributo do parâmetro.

O Java associa automaticamente a todas as variáveis e métodos referenciados com a palavra reservada `this`. Em alguns situações torna-se redundante o uso do `this`.

**Aplicação:** - Impedir ambigüidade de nome de variáveis e fazer referência a própria classe.

**Restrição:** Não pode ser aplicada a métodos estáticos, por exemplo, o método `main`.

### Exemplo

*Existem casos em se faz necessário o uso da palavra `this`. Por exemplo, podemos necessitar chamar apenas uma parte do método passando uma instância do argumento do objeto. (Chamar um classe de forma localizada);*

**`Birthday bDay = new Birthday(this);`**

```
public class PessoaJuridica extends Pessoa {
    public PessoaJuridica(){
        super("ACME");
    }
    public static void main(String args[]){
        PessoaJuridica pf = new PessoaJuridica();
        pf.setIdade(10);
        System.out.println("idade: " + pf.getIdade());
    }
    public String getNome(){
        return "";
    }
    public int getIdade()
        { return idade; }
    public void setIdade(int idade)
        { this.idade = idade; }
}
```

Anotações

71

## Coleções

### O que são coleções?

Coleções (também conhecidas como container) é um simples objeto que pode agrupar múltiplos elementos. Coleções são utilizadas para armazenar, recuperar e manipular dados. Os métodos são responsáveis por realizar as operações. A estrutura das coleções inclui uma variedade de recursos que simplificam o desenvolvimento. Essa é uma implementação da noção de reúso de código. Um exemplo de coleção poderia ser uma lista de nomes e telefones.

### Hashtable (API: **Java.util**)

A tabela de hash é uma estrutura de dados que permite procurar os itens armazenados em tabela utilizando uma chave associada. O formato padrão de entrada de dados na tabela é **chave** e **valor**.

Para construir uma tabela de hash em Java, devemos criar um objeto Hashtable, utilizando o construtor Hashtable.

Para adicionar elementos na tabela usaremos o método put, **put(object key, Object value)**. Este método é da classe Hashtable. Para recuperar o elemento usado como *chave*, devemos usar o método get, **get(object key)**. Para remover um elemento usamos o método **remove**.

### Exemplo

```
import java.util.*;
public class TestHashTable{
    public static void main(String Arg[]){
        Hashtable table = new Hashtable();
        table.put("1", "Joao");
        table.put("2", "Jose");
        table.put("3", "Maria");
        table.put("4", "Marta");
        table.put("5", "Pedro");
        table.put("6", "Mateus");
        String find = (String) table.get("4");
        System.out.println(find);
    }
}
```



# Programação Orientada a Objetos

---

## Enumeration (API: `Java.util.Enumeration`)

Retorna uma enumeração de uma coleção (Tabela Hash, por exemplo) especificada.

Principais métodos: **`hasMoreElements`** e **`nextElements`**.

### Exemplo

```
import java.util.Enumeration.*;
import java.util.*;
public class EnumerationExemplo{
    public static void main(String Arg[]){
        Hashtable table = new Hashtable();
        table.put("1", "Joao");
        table.put("2", "Jose");
        table.put("3", "Maria");
        table.put("4", "Marta");
        table.put("5", "Pedro");
        table.put("6", "Mateus");
        Enumeration e = table.elements();
        while(e.hasMoreElements()){
            String valor = (String) e.nextElement();
            System.out.println(valor);
        }
    }
}
```

```
import java.util.Iterator.*;
...
Iterator i = table.values().iterator();
while(i.hasNext()){
    String valor = (String)i.next();
    System.out.println(valor);
}
```

Opcionalmente podemos usar ***Iterator*** que tem a funcionalidade parecida com *Enumeration*

# Programação Orientada a Objetos

---

## Vector

### Exemplo

```
import java.util.*;
public class TestVector{
    public static void main(String Arg[]){
        Vector v = new Vector();           // Declaração do Vector
        Integer y;
        for(int x=0;x<5;x++){
            y = new Integer(x);
            v.addElement(y);
        }

        Object[] objArray = v.toArray();

        for(int x=0;x<5;x++){
            System.out.println(objArray[x]);
        }
    }
}
```

Adiciona valores ao Vector. Note que estes valores são do tipo referência, ou seja, objetos

Converte um Vector para um array.

# Programação Orientada a Objetos

---

## ArrayList ( **API: Java.util**).

ArrayList é array dimensionável da Interface List parecido com Vector. No exemplo abaixo é permitido que os valores sejam duplicados, porém, desordenados. Principais métodos: add, remove, clear, size, isEmpty, get, set, iterator.

### Exemplo

```
import java.util.*;
public class ListExemplo {
    public static void main(String[] args) {
        List list = new ArrayList(); // Declaração do ArrayList

        list.add("um");
        list.add(new Integer(4));
        list.add("terceiro");
        list.add(new Integer(4));
        System.out.println(list);

        Vector v = new Vector();
        Integer y;
        for(int x=0;x<5;x++){
            y = new Integer(x);
            v.addElement(y);
        }

        List listVector = new ArrayList( v );
        System.out.println(listVector);
    }
}
```

Inserção de valores na lista. Veja que estes valores são objetos.

Uma nova coleção.

Uma outra construção para a lista, neste caso ela recebe uma coleção como argumento.

# Programação Orientada a Objetos

---

## HashSet (API: Java.util)

A classe HashSet Implementa a interface Set, ela também tem suporte da Hashtable. Ela é uma coleção que não pode conter elementos duplicados. Principais métodos: add, clear, remove, size, iterator e isEmpty.

### Exemplo

```
import java.util.*;
//Com ordem e sem repetição
public class SetExample {
    public static void main(String[] args) {
        Set set = new HashSet(); // Declaração do HashSet
        set.add("dois");
        set.add("3rd");
        set.add(new Float(11.1F));
        set.add("quarto");
        set.add(new Float(11.1F));
        System.out.println(set); // Imprime os valores da lista
    }
}
```

Inserção de valores na lista. Veja que estes valores são objetos.

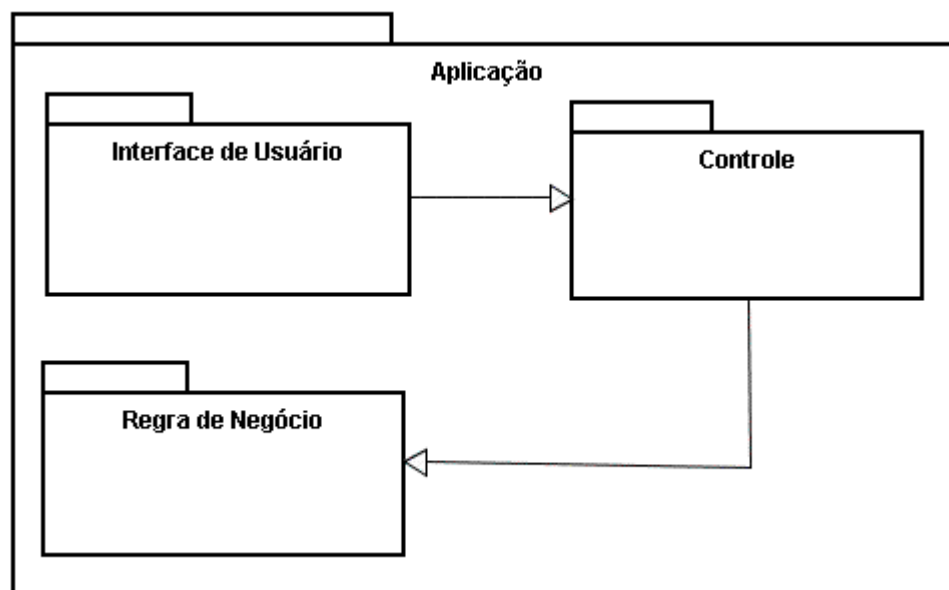
# Programação Orientada a Objetos

---

## Pacotes

A linguagem Java é estruturada em pacotes, estes pacotes contêm classes que por sua vez contêm atributos e métodos. Pacote é forma de organização da linguagem Java, prevenindo de problemas como a repetição de identificadores (nomes) de classes e etc. Podemos usar a estrutura de pacotes para associar classes com semântica semelhante, ou seja, classes que tem objetivo comum. Por exemplo, colocaremos em único pacote todas as classes que se referem a regras de negócios.

### Exemplo



Fisicamente os pacotes tem a estrutura de diretório e subdiretório.

# Programação Orientada a Objetos

---

## Pacotes

### Import

A instrução import faz importação para o arquivo fonte (.java) das classes indicadas, cujo o diretório base deve estar configurado na variável de ambiente: CLASSPATH.

**Sintaxe:** import <classes>;

### Exemplos

```
import java.awt.Button;  
import java.awt.*;
```

### Package

Esta instrução deve ser declarada na primeira linha do programa fonte, esta instrução serve para indicar que as classes compiladas fazem parte do conjunto de classes (*package*), ou sejam um pacote, indicado pela notação path.name (caminho e nome do pacote).

**Sintaxe:** package <path.name>;

```
package mypck;  
public class ClassPkg{  
    public ClassPkg(){  
        System.out.println("Teste de package...");  
    }  
}
```

### Anotações

---

---

---

---

78

# Programação Orientada a Objetos

---

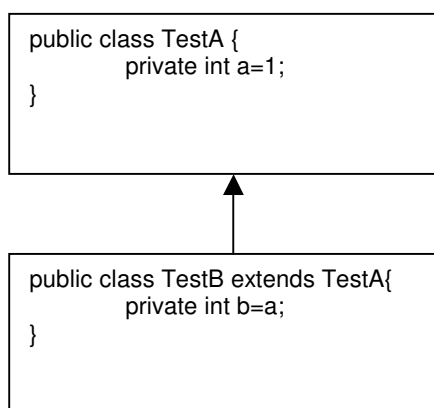
## Acessibilidade

### Acessibilidade ou Visibilidade

Os níveis de controle tem o seguinte papel de tornar um atributo, um método ou classe visível ou não.

#### Exemplo

Se um atributo foi declarado dentro de uma classe chamada TestA e este atributo tem o nível de controle **private**. Somente esta classe poderá fazer acesso a este atributo, ou seja, somente classe TestA o enxergará, todas as demais, não poderão fazer acesso direto a este atributo.



TestB.java:3: a has private access in TestA  
private int b=a;  
^

Para contornar a situação poderíamos fazer duas coisas, a primeira: alterar o nível de controle do atributo a, na classe TestA, para public, desta forma a classe TestB o enxergaria. Todavia, as boas práticas de programação não recomendam fazer isto.

A segunda: é uma maneira mais elegante de resolvermos o problema, podemos criar métodos públicos na classe TestA, por exemplo, getA e setA, aí sim, através destes métodos seria possível manipular o atributo da classe TestA.

*A linguagem Java tem para os atributos e métodos quatro níveis de controles: **public, protected, default e private**.*

*Para classes tem dois níveis: **public** e **default** (também chamado de pacote ou de friendly).*

*Estes níveis de controle são os responsáveis pela acessibilidade ou visibilidade de atributos, e métodos.*

#### Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## Acessibilidade ou Visibilidade

A tabela abaixo demonstra a acessibilidade para cada nível de controle.

<b>Modificador</b>	<b>Mesma Classe</b>	<b>Mesmo Package</b>	<b>SubClasse</b>	<b>Universo</b>
<b>Public</b>	sim	sim	sim	sim
<b>Protected</b>	sim	sim	sim	não
<b>Private</b>	sim	não	não	não
<b>Default</b>	sim	sim	não	não

Anotações

80



# Programação Orientada a Objetos

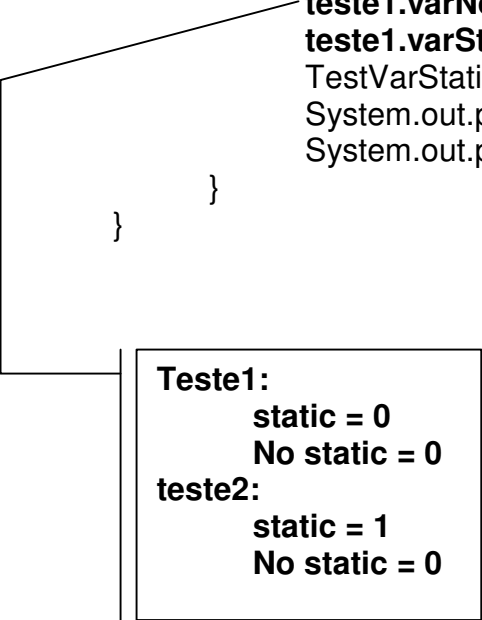
---

## Modificador static

Exemplo de compartilhamento de variável **static**. Apesar de ter dois objetos da classe (teste1 e teste2). Quando fazemos a impressão da variável esta tem o valor que atribuído pelo primeiro objeto teste1.

### Exemplo

```
public class TestVarStatic {  
    private static int varStatic;  
    private int varNoStatic;  
    public static void main(String[] args) {  
        TestVarStatic teste1 = new TestVarStatic();  
        System.out.println("static = " + teste1.varStatic);  
        System.out.println("No static = " + teste1.varNoStatic);  
        teste1.varNoStatic++;  
        teste1.varStatic++;  
        TestVarStatic teste2 = new TestVarStatic();  
        System.out.println("static = " + teste2.varStatic);  
        System.out.println("No static = " + teste2.varNoStatic);  
    }  
}
```



<b>Teste1:</b>	<b>static = 0</b>
	<b>No static = 0</b>
<b>teste2:</b>	<b>static = 1</b>
	<b>No static = 0</b>

# Programação Orientada a Objetos

---

## Modificador static

### Restrições

#### Métodos static

- não pode ter referência **this**.
- não pode ser substituído ("overridden") por um método não static
- pode somente fazer acesso dados static da classe. Ele não pode fazer acesso a não static.
- pode somente chamar métodos static. Ele não pode chamar métodos não static

### Exemplo

```
public class TestMetodoStatic {  
    private static int valor;  
    public TesteMetodoStatic(){  
        valor=0;  
    }  
    public static void main(String[] args) {  
        addSoma(2,2);  
        addSoma(3,2);  
        addSoma(4,2);  
        displayTotal();  
    }  
    public static void addSoma(int a, int b){  
        valor += (a * b);  
    }  
    public static void displayTotal(){  
        System.out.println("Total = " + valor);  
    }  
}
```

### Anotações

82

## Modificador Final (constantes)

Para declarar uma variável, um ou uma classe como *constante* usamos o modificador **final**.

Entretanto, o uso deste modificador deve obedecer a certas restrições como:

- Uma classe constante não pode ter subclasses;
- Um método constante não pode ser sobrescrito;
- O valor para variável constante deve ser definido no momento da declaração ou através de um construtor, para variáveis membro, uma vez atribuído o valor, este não mudará mais.

### Veja o exemplo

```
public final class TestFinalPai{
    private final int VALOR;
    public TestFinalPai(int V){
        VALOR = V;
    }
}

public class TestFinalFilho extends TestFinalPai{
    public TestFinalFilho() {
        super(10);
    }
    public static void main(String args[])
    {
    }
}
```

Quando compilamos a classe um erro é gerado.

*TestFinalFilho.java:1: **cannot inherit from final TestFinalPai***  
*public class TestFinalFilho extends TestFinalPai*

# Programação Orientada a Objetos

---

## Modificador Final (constantes)

### Veja o exemplo

**TestFinal1** - O valor é atribuído através de um construtor, note que a variável é membro da classe.

**TestFinal2** - O valor é atribuído no método, pois, neste caso a variável é um local.

```
public class TestFinal1{
    private final int VALOR;
    public TestFinal1(){
        VALOR = 0;
    }
    public static void main(String args[]){
        TestFinal1 tf= new TestFinal1();
        System.out.println(tf.VALOR);
    }
}

public class TestFinal2{
    public static void main(String args[]){
        final int VAL;
        VAL =0;
        System.out.println(VAL);
    }
}
```

**Convenção, para variáveis constantes (final), o nome da variável deve ser escrito em maiúsculo.**

# Programação Orientada a Objetos

---

## Programação Orientada a Objetos

A metodologia de Orientação a Objetos constitui um dos marcos importantes na evolução mais recente das técnicas de modelagem e desenvolvimento de sistemas de computador, podendo ser considerada um aprimoramento do processo de desenhar sistemas. Seu enfoque fundamental está nas abstrações do mundo real.

Sua proposta é pensar os sistemas como um conjunto cooperativo de objetos. Ela sugere uma modelagem do domínio do problema em termos dos elementos relevantes dentro do contexto estudado, o que é entendido como **abstração** → identificar o que é realmente necessário e descartar o que não é.

Esta nova forma de abordagem para desenho e implementação de software vem sendo largamente apresentada como um meio eficaz para a obtenção de maior qualidade dos sistemas desenvolvidos principalmente no que diz respeito a:

- Integridade, e portanto maior confiabilidade dos dados gerados
- Maior facilidade para detecção e correção de erros
- Maior reusabilidade dos elementos de software produzidos

Os princípios básicos sobre os quais se apoiam as técnicas de Orientação a Objetos são:

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

## Abstração

O princípio da abstração traduz a capacidade de identificação de coisas semelhantes quanto à forma e ao comportamento permitindo assim a organização em classes, dentro do contexto analisado, segundo a essência do problema.

Ela se fundamenta na busca dos aspectos relevantes dentro do domínio do problema, e na omissão daquilo que não seja importante neste contexto, sendo assim um enfoque muito poderoso para a interpretação de problemas complexos.

**Anotações**

85

# Programação Orientada a Objetos

---

A questão central, e portanto o grande desafio na modelagem Orientada a Objetos, está na identificação do conjunto de abstrações que melhor descreve o domínio do problema.

## **As abstrações são representadas pelas classes.**

Uma classe pode ser definida como um modelo que descreve um conjunto de elementos que compartilham os mesmos atributos, operações e relacionamentos. É portanto uma entidade abstrata.

A estruturação das classes é um procedimento que deve ser levado a efeito com muito critério para evitar a criação de classes muito grandes, abrangendo tudo, fator que dificulta sua reutilização em outros sistemas.

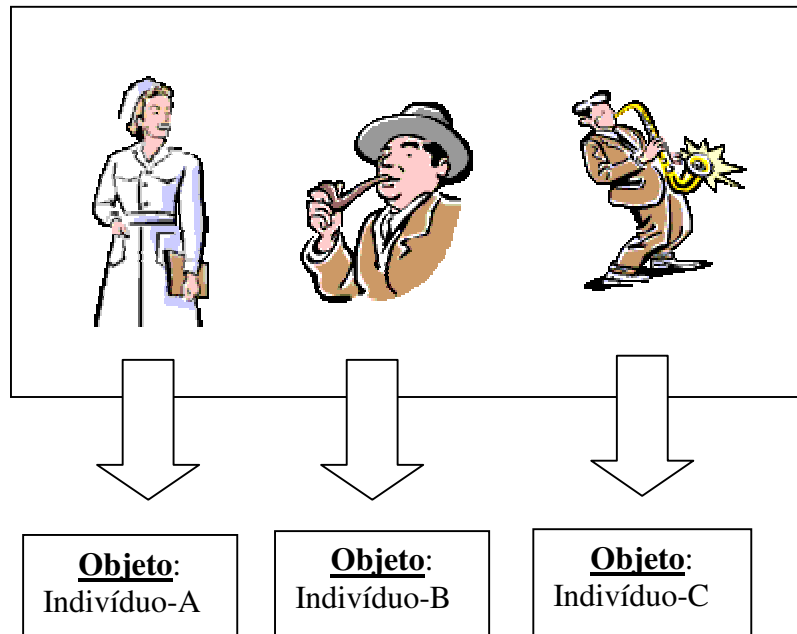
Uma classe deve conter, apenas, os elementos necessários para resolver um aspecto bem definido do sistema.

Um elemento representante de uma classe é uma instância da classe, ou objeto.

Um objeto é uma entidade real, que possui:

- Identidade**      ➤ Característica que o distingue dos demais objetos.
- Estado**          ➤ Representado pelo conjunto de valores de seus atributos em um determinado instante.
- Comportamento** ➤ Reação apresentada em resposta às solicitações feitas por outros objetos com os quais se relaciona.

## Classe Indivíduos



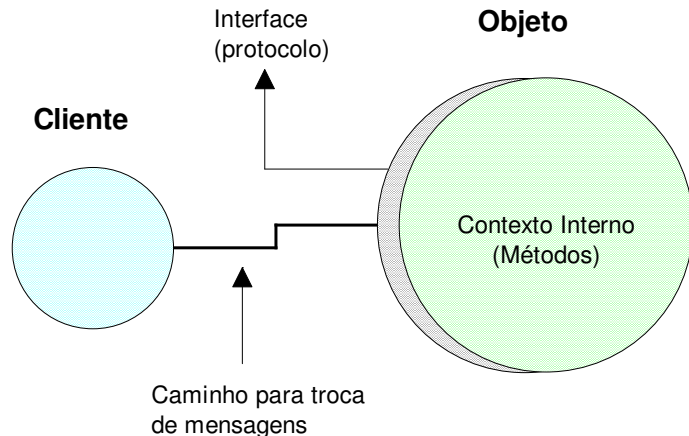
## Encapsulamento

Processo que enfatiza a separação entre os aspectos externos de um objeto (aqueles que devem estar acessíveis aos demais) e seus detalhes internos de implementação.

Através do encapsulamento pode ser estabelecida uma interface bem definida para interação do objeto com o mundo externo (interfaces públicas), isolando seus mecanismos de implementação, que ficam confinados ao próprio objeto.

Desta forma, o encapsulamento garante maior flexibilidade para alterações destes mecanismos (implementação dos métodos), já que este é um aspecto não acessível aos clientes externos.

Todo e qualquer acesso aos métodos do objeto só pode ser conseguido através de sua interface pública.



## Herança

A herança é o mecanismo de derivação através do qual uma classe pode ser construída como extensão de outra. Neste processo, todos os atributos e operações da classe base passam a valer, também, para a classe derivada, podendo esta última definir novos atributos e operações que atendam suas especificidades.

Uma classe derivada pode, também, redefinir operações de sua classe base, o que é conhecido como uma operação de sobrecarga.

O modelo de Orientação a Objetos coloca grande ênfase nas associações entre classes, pois são estas que estabelecem os caminhos para navegação, ou troca de mensagens, entre os objetos que as representam.

Dois tipos especiais de associações têm importância fundamental na modelagem do relacionamento entre classes, tornando possível a estruturação de uma hierarquia:

- Associações de agregação
- Associações de generalização / especialização

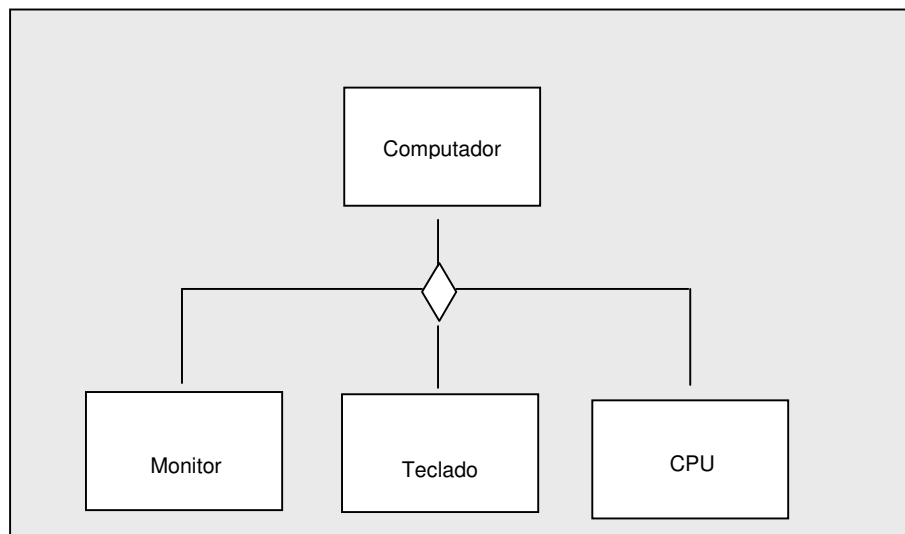


# Programação Orientada a Objetos

---

Uma hierarquia de agregação indica um relacionamento de composição, denotando uma forte dependência entre as classes.

## Hierarquia de Agregação



Por outro lado, as associações de generalização / especialização caracterizam o tipo de hierarquia mais intimamente ligado com o princípio da herança.

Esta hierarquia indica um relacionamento onde acontece o compartilhamento de atributos e de operações entre os vários níveis.

As classes de nível mais alto definem as características comuns, enquanto que as de nível mais baixo incorporam as especializações características de cada uma.

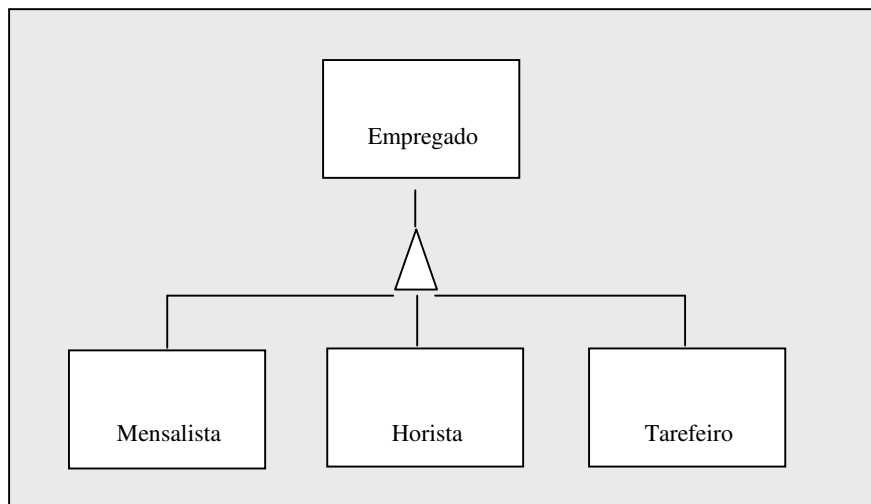
A generalização trata do relacionamento de uma classes com outras, obtidas a partir de seu refinamento.

---

**Anotações**

89

## Hierarquia de Generalização / Especialização



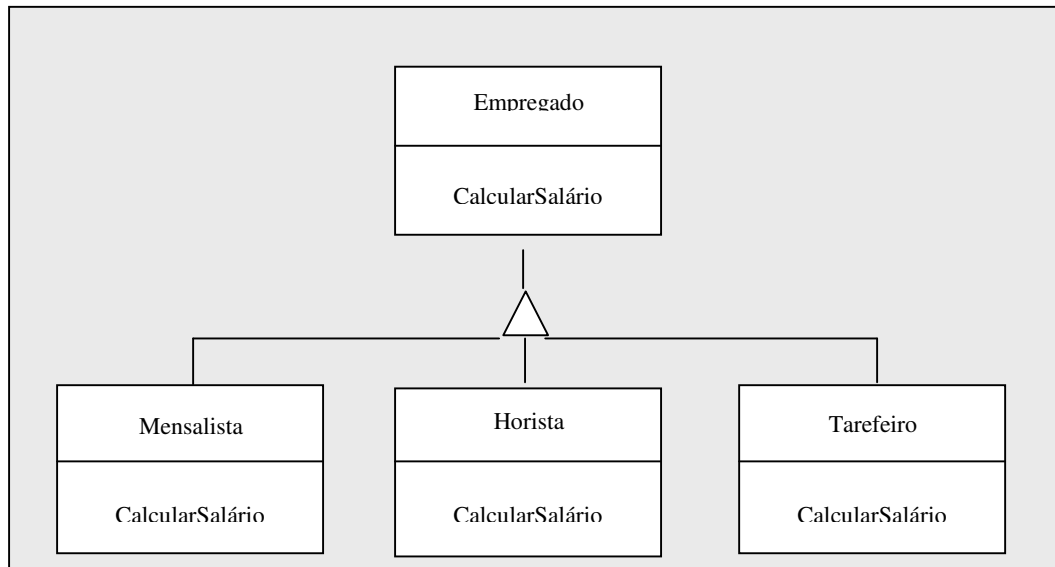
A decomposição em uma hierarquia de classes permite quebrar um problema em outros menores, capturando as redundâncias através da compreensão dos relacionamentos.

## Polimorfismo

Polimorfismo é a característica que garante que um determinado método possa produzir resultados diferentes quando aplicado a objetos pertencentes a classes diferentes, dentro de uma hierarquia de classes.

Através do polimorfismo um mesmo nome de operação pode ser compartilhado ao longo de uma hierarquia de classes, com implementações distintas para cada uma das classes.

No diagrama apresentado a seguir, o método `CalcularSalário` deve ter implementações diferentes para cada uma das classes derivadas da classe base `Empregado`, já que a forma de cálculo do salário é diferente para empregados mensalistas, horistas e tarefeiros.



O encapsulamento e o polimorfismo são condições necessárias para a adequada implementação de uma hierarquia de herança.

# Programação Orientada a Objetos

## Principais conceitos em orientação a objetos

<b>Classe</b>	<ul style="list-style-type: none"><li>↳ Abstração que define um tipo de dados.</li><li>↳ Contém a descrição de um grupo de dados e de funções que atuam sobre estes dados</li></ul>
<b>Objeto ou instância de uma classe</b>	<ul style="list-style-type: none"><li>↳ Uma variável cujo tipo é uma determinada classe</li></ul>
<b>Instanciação</b>	<ul style="list-style-type: none"><li>↳ Processo de criação de um objeto a partir de uma classe</li></ul>
<b>Atributos</b>	<ul style="list-style-type: none"><li>↳ Variáveis pertencentes às classes e que, normalmente, têm acesso restrito, podendo ser manipuladas apenas pelos métodos da própria classe a que pertencem e subclasses desta.</li></ul>
<b>Métodos</b>	<ul style="list-style-type: none"><li>↳ Funções que tratam as variáveis (atributos).</li><li>↳ Podem ou não retornar valores e receber parâmetros.</li><li>↳ Quando não retornam valores devem ser declarados como <b>void</b></li></ul>
<b>Construtores</b>	<ul style="list-style-type: none"><li>↳ Métodos especiais cuja função é criar (alocar memória) e inicializar as instâncias de uma classe.</li><li>↳ Todo construtor deve ter o mesmo nome da classe.</li></ul>
<b>Hierarquia de classes</b>	<ul style="list-style-type: none"><li>↳ Conjunto de classes relacionadas entre si por herança</li></ul>
<b>Herança</b>	<ul style="list-style-type: none"><li>↳ Criação de uma nova classe pela extensão de outra já existente</li></ul>
<b>Superclasse</b>	<ul style="list-style-type: none"><li>↳ Uma classe que é estendida por outra</li></ul>
<b>Subclasse</b>	<ul style="list-style-type: none"><li>↳ Uma classe que estende outra para herdar seus dados e métodos</li></ul>
<b>Classe base</b>	<ul style="list-style-type: none"><li>↳ A classe de nível mais alto em uma hierarquia de classes (da qual todas as outras derivam)</li></ul>
<b>Sobreposição de método</b>	<ul style="list-style-type: none"><li>↳ Redefinição, na subclasse, de um método já definido na superclasse</li></ul>
<b>Encapsulamento</b>	<ul style="list-style-type: none"><li>↳ Agrupamento de dados e funções em um único contexto</li></ul>
<b>Facotes</b>	<ul style="list-style-type: none"><li>↳ Conjunto de classes</li></ul>

Anotações

92

# Programação Orientada a Objetos

---

## Métodos

Métodos são os comportamentos de um objeto. A declaração é feita da seguinte forma:

**< modificador > < tipo de retorno > < nome > ( < lista de argumentos > )  
< bloco >**

< modificador > -> segmento que possui os diferentes tipos de modificações incluindo public, protected, private e default (neste caso não precisamos declarar o modificador).

< tipo de retorno > -> indica o tipo de retorno do método.

< nome > -> nome que identifica o método.

< lista de argumentos > -> todos os valores que serão passados como argumentos.

Método é a implementação de uma operação. As mensagens identificam os métodos a serem executados no objeto receptor. Para chamar um método de um objeto é necessário enviar uma mensagem para ele. Por definição todas as mensagens tem um tipo de retorno, por este motivo em Java, mesmo que método não retorne nenhum valor será necessário usar o tipo de retorno chamado “void” (retorno vazio).

## Exemplos

```
public void somaDias (int dias) { }  
private int somaMes(int mês) { }  
protected String getNome() { }  
int getAge(double id) { }
```

```
public class ContaCorrente {  
    private int conta=0;  
    private double saldo=0;  
    public double getSaldo(){  
        return saldo;  
    }  
    public void setDeposito(int valordeposito){  
        return saldo +=valordeposito;  
    }  
    public void setSaque(double valorsaque){  
        return saldo -=valorsaque;  
    }  
}
```

Anotações

93

## Construtores

### O que são construtores?

Construtores são um tipo especial de método usado para inicializar uma “instance” da classe.

**Toda a classe Java deve ter um Construtor.** Quando não declaramos o “**Construtor default**”, que é inicializado automaticamente pelo Java. Mas existem casos que se faz necessário a declaração explícita dos construtores.

***O Construtor não pode ser herdado. Para chamá-lo a partir de uma subclasse usamos a referência **super**.***

**Para escrever um construtor, devemos seguir algumas regras:**

- 1ª O nome do construtor precisa ser igual ao nome da classe;
- 2ª Não deve ter tipo de retorno;
- 3ª Podemos escrever vários construtores para mesma classe.

### Sintaxe

```
[ <modificador> ] <nome da classe> ([Lista de argumentos]){  
[ <declarações> ]  
}
```

```
public class Mamifero {  
    private int qdepernas;  
    private int idade;  
    public Mamifero(int idade){  
        this.idade = idade;  
    }  
    //Métodos  
}
```

# Programação Orientada a Objetos

---

## Atributos e variáveis

Os **atributos** são pertencentes a classe, eles podem ser do tipo primitivo ou referência (objetos), os seus modificadores podem ser: **public**, **private**, **protected** ou **default**.

O ciclo de vida destes atributos estão vinculados ao ciclo de vida da classe.

### Variáveis Locais

São definidas dentro dos métodos. Elas têm o ciclo de vida vinculado ao ciclo do método, também são chamadas de variáveis temporárias.

### Sintaxe

[<modificador>] <tipo de dado> <nome> [ = <valor inicial>];

### Exemplo

```
public class Disciplina {
    private int cargaHoraria; // atributo
    private String nome;      // atributo
    public Disciplina(String nome, int cargaHoraria){
        this.nome = nome;
        this.cargaHoraria =
            calcCargaHoraria(cargaHoraria);
    }
    public String getNome(){
        return nome;
    }
    public int getCargaHoraria(){
        return cargaHoraria;
    }
    public int calcCargaHoraria(int qdeHoras) {
        int horasPlanejamento = (int) ( qdeHoras * 0.1);
        return cargaHoraria =
            horasPlanejamento + qdeHoras;
    }
}
```

### Anotações

95

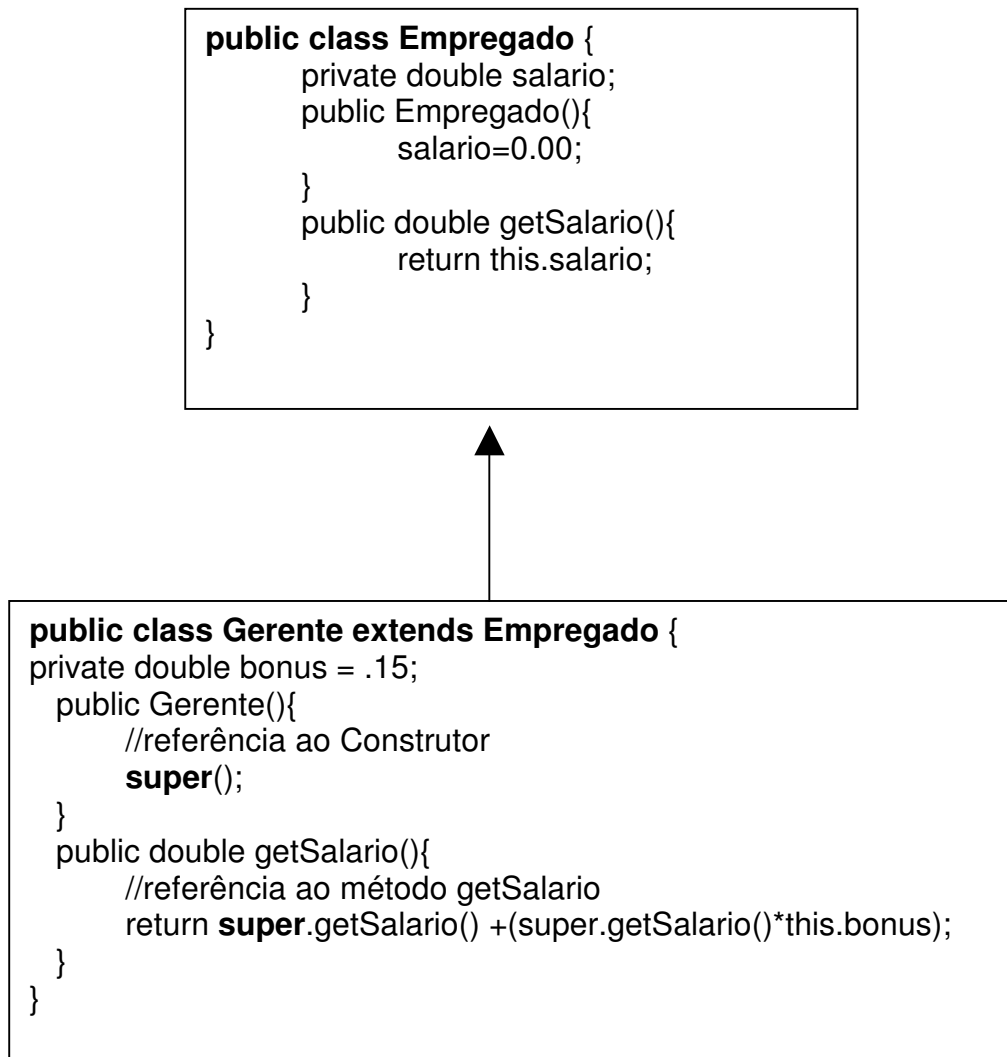
# Programação Orientada a Objetos

---

## A referência super

A palavra `super` é usada para referenciar a super classe (classe pai), na verdade o construtor da classe hierarquicamente superior, podemos usa-lo também para fazer referência aos membros (atributos e métodos), da super classe. Desta forma temos uma extensão do comportamento.

### Exemplo





## Classe Abstrata e Finais

Uma **classe abstrata** não pode ser instanciada, ou seja, não há objetos que possam ser construídos diretamente de sua definição. Por exemplo, a compilação do seguinte trecho de código.

```
abstract class AbsClass {
    public static void main(String[] args) {
        AbsClass obj = new AbsClass();
    }
}
```

geraria a seguinte mensagem de erro:

**AbsClass.java:3: class AbsClass is an abstract class.  
It can't be instantiated.**

**AbsClass obj = new AbsClass();**

**^**

**1 error**

Em geral, classes abstratas definem um conjunto de funcionalidades das quais pelo menos uma está especificada mas não está definida ou seja, contém pelo menos um método abstrato, como em:

```
abstract class AbsClass {
    public abstract int umMetodo();
}
```

Um método abstrato não cria uma definição, mas apenas uma declaração de um método que deverá ser implementado em uma classe derivada. Se esse método não for implementado na classe derivada, esta permanece como uma classe abstrata mesmo que não tenha sido assim declarada explicitamente.

Assim, para que uma classe derivada de uma classe abstrata possa gerar objetos, os métodos abstratos devem ser definidos em classes derivadas:

```
class ConcClass extends AbsClass {
    public int umMetodo() {
        return 0;
    }
}
```

**Anotações**

97

# Programação Orientada a Objetos

---

Uma **classe final**, por outro lado, indica uma classe que não pode ser estendida. Assim, a compilação do arquivo `Reeleicao.java` com o seguinte conteúdo:

```
final class Mandato {  
}  
  
public class Reeleicao extends Mandato {  
}
```

ocasionaria um erro de compilação:

**Exemplos[39] javac Reeleicao.java**

**Reeleicao.java:4: Can't subclass final classes: class Mandato**

**public class Reeleicao extends Mandato {**

**^**

**1 error**

A palavra-chave **final** pode também ser aplicada a métodos e a atributos de uma classe. Um método final não pode ser redefinido em classes derivadas. Um atributo final não pode ter seu valor modificado, ou seja, define valores constantes. Apenas valores de tipos primitivos podem ser utilizados para definir constantes. O valor do atributo deve ser definido no momento da declaração, pois não é permitida nenhuma atribuição em outro momento.

A utilização de final para uma referência a objetos é permitida. Como no caso de constantes, a definição do valor (ou seja, a criação do objeto) também deve ser especificada no momento da declaração. No entanto, é preciso ressaltar que o conteúdo do objeto em geral pode ser modificado apenas a referência é fixa. O mesmo é válido para arranjos.

A partir de Java 1.1, é possível ter atributos de uma classe que sejam final mas não recebem valor na declaração, mas sim nos construtores da classe. (A inicialização deve obrigatoriamente ocorrer em uma das duas formas.) São os chamados *blank finals*, que introduzem um maior grau de flexibilidade na definição de constantes para objetos de uma classe, uma vez que essas podem depender de parâmetros passados para o construtor.

Argumentos de um método que não devem ser modificados podem ser declarados como final, também, na própria lista de parâmetros.

## Interfaces

Java também oferece outra estrutura, denominada interface, com sintaxe similar à de classes mas contendo apenas a especificação da funcionalidade que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada. Uma interface Java é uma classe abstrata para a qual todos os métodos são implicitamente `abstract` e `public`, e todos os atributos são implicitamente `static` e `final`. Em outros termos, uma interface Java implementa uma “classe abstrata pura”.

A sintaxe para a declaração de uma interface é similar àquela para a definição de classes, porém seu corpo define apenas assinaturas de métodos e constantes. Por exemplo, para definir uma interface `Interface1` que declara um método `met1` sem argumentos e sem valor de retorno, a sintaxe é:

```
interface Interface1 {  
    void met1();  
}
```

A diferença entre uma classe abstrata e uma interface Java é que a interface obrigatoriamente não tem um “corpo” associado. Para que uma classe seja abstrata basta que ela seja assim declarada, mas a classe pode incluir atributos de objetos e definição de métodos, públicos ou não. Na interface, apenas métodos públicos podem ser declarados, mas não definidos. Da mesma forma, não é possível definir atributos, apenas constantes públicas.

Enquanto uma classe abstrata é “estendida” (palavra chave `extends`) por classes derivadas, uma interface Java é “implementada” (palavra chave `implements`) por outras classes. Uma interface estabelece uma espécie de contrato que é obedecido por uma classe. Quando uma classe implementa uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe.

Outro uso de interfaces Java é para a definição de constantes que devem ser compartilhadas por diversas classes. Neste caso, a recomendação é implementar interfaces sem métodos, pois as classes que implementarem tais interfaces não precisam tipicamente redefinir nenhum método:

# Programação Orientada a Objetos

---

```
interface Coins {
    int
    PENNY = 1,
    NICKEL = 5,
    DIME = 10,
    QUARTER = 25,
    DOLAR = 100;
}

class SodaMachine implements Coins {
    int price = 3*QUARTER;
    // ...
}
```

**Anotações**

100

# Programação Orientada a Objetos

---

## Utilitários

### JavaDoc (Documentação)

#### Javadoc

É uma utilitário do JDK que gera a documentação dos programas java, geralmente em formato HTML.

Localização JDK1.X/BIN/Javadoc.

**Sintaxe padrão:** javadoc <arquivofonte.java>

Opção	Valor	Descrição
-d	path de saída	Diretório onde será gerado os arquivos HTML
-sourcepath	path	Especifica o diretório raiz dos fontes ou dos Package
-public		Documenta apenas variáveis e métodos públicos
-private		Documenta todas as variáveis e métodos

**Sintaxe:** Javadoc [Opções] package/arquivo < package/arquivo>

**Exemplo:** javadoc MinhasClasses.class

**Tags para Documentação: Usar: /\*\* e \*/**

Tag	Declaração	Class e Interface	Construtor	Método	Atributo
@see	Cria um link com outra página HTML	X	X	X	X
@deprecated	Informa quais métodos estão ultrapassados	X	X	X	X
@author	Nome do Autor	X			
@param	Documenta o parametro	X	X	X	
@throws @Exception	Documenta Exceções		X	X	
@return	Documenta o retorno (valor e tipo)			X	

Anotações

101

## JavaDoc (Documentação)

usage: javadoc [options] [packagenames] [sourcefiles] [classnames]  
[@files]  
-overview <file> Read overview documentation from HTML file  
-public Show only public classes and members  
-protected Show protected/public classes and members  
(default)  
-package Show package/protected/public classes and  
members  
-private Show all classes and members  
-help Display command line options  
-doclet <class> Generate output via alternate doclet  
-docletpath <path> Specify where to find doclet class files  
-1.1 Generate output using JDK 1.1 emulating doclet  
-sourcepath <pathlist> Specify where to find source files  
-classpath <pathlist> Specify where to find user class files  
-bootclasspath <pathlist> Override location of class files loaded  
by the bootstrap class loader  
-extdirs <dirlist> Override location of installed extensions  
-verbose Output messages about what Javadoc is doing  
-locale <name> Locale to be used, e.g. en\_US or en\_US\_WIN  
-encoding <name> Source file encoding name  
-J<flag> Pass <flag> directly to the runtime system

### Provided by Standard doclet:

-d <directory> Destination directory for output files  
-use Create class and package usage pages  
-version Include @version paragraphs  
-author Include @author paragraphs  
-splitindex Split index into one file per letter  
-windowtitle <text> Browser window title for the documentation  
-doctitle <html-code> Include title for the package index(first  
page  
-header <html-code> Include header text for each page  
-footer <html-code> Include footer text for each page  
-bottom <html-code> Include bottom text for each page  
-link <url> Create links to javadoc output at <url>  
-linkoffline <url> <url2> Link to docs at <url> using package list at  
<url2>  
-group <name> <p1>:<p2>.. Group specified packages together in overview  
page  
-nodeprecated Do not include @deprecated information  
-nosince Do not include @since information  
-nodeprecatedlist Do not generate deprecated list  
-notree Do not generate class hierarchy  
-noindex Do not generate index  
-nohelp Do not generate help link  
-nonavbar Do not generate navigation bar  
-serialwarn Generate warning about @serial tag  
-charset <charset> Charset for cross-platform viewing of  
generated documentation.  
-helpfile <file> Include file that help link links to  
-stylesheetfile <path> File to change style of the generated  
documentation  
-docencoding <name> Output encoding name

---

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## Javadoc

Este exemplo exhibe como implementar as tags de documentação que serão usados pelo utilitário Javadoc.

```
import java.util.List;
/**
 * @author YourName
 * @version 2.0
 */
public class DocExemplo {
    /** Declaração e atribuição de x. */
    private int x;
    /**
     * This variable a list of stuff.
     * @see #getStuff()
     */
    private List stuff;
    /**
     * O construtor inicia a variavel x.
     * @param int x
     */
    public DocExemplo(int x) {
        this.x = x;
    }
    /**
     * Este método retorna algum valor.
     * @throws IllegalStateException se nenhum retorno for encontrado
     * @return A lista de valores
     */
    public List getStuff() throws IllegalStateException {
        if ( stuff == null ) {
            throw new java.lang.IllegalStateException("Erro, sem valor");
        }
        return stuff;
    }
}
```

Anotações

103

# Programação Orientada a Objetos

## Jar (Compactação, Agrupamento e Distribuição)

É um utilitário do JDK que faz agrupamento de arquivos em único, um arquivo .jar, geralmente com compressão. Localização JDK1.X/BIN/Jar. É usado também para fazer a distribuição de aplicação.

**Sintaxe:** *jar opções [meta-arq] nome-arquivo-destino [nome-arquivo-entrada]*

Argumento	Descrição
meta-arquivo	Arquivo que contém as informações sobre o arquivo destino gerado. Este argumento é opcional, entretanto um arquivo meta-arquivo é gerado, default, META-INF/MANIFEST.INF
arquivo-destino	Nome do arquivo jar. A extensão .jar não é automática, deve ser Especificada.
arquivo-entrada	Nome dos arquivos a serem agrupados e/ou compactados

Opções	Descrição
c	Cria um novo arquivo
t	Nome do arquivo jar. A extensão .jar não é automática, deve ser Especificada.
x	Extrai todos os arquivos
x <arquivo>	Extrai o arquivo especificado
f	Mostra o conteúdo de um arquivo existente
v	Mostra o status da operação (verbose)
m	Suprime a geração do meta-arquivo
o	Faz apenas o agrupamento, sem compactação. Deve ser utilizado para arquivos jar na variável de ambiente Classpath

### Exemplos

jar cvf Classes.jar ClassA.class ClassB.class ClassC.class

Para ver o conteúdo do arquivo jar, gerado: jar tvf Classes.jar

Para extrair arquivo: Jar xvf Classes.jar

*Obs: a opção f é sempre utilizada em operações com arquivos.*

**Os arquivos Jar podem conter um aplicação inteira, por isso, ele é usado para fazer distribuição de aplicações. Também é bastante usado com componente Javabeans e Applet.**



# Programação Orientada a Objetos

---

## Certificação JAVA

### Capítulo I - Fundamentos da Linguagem

Esse capítulo é bastante significativo para a assimilação dos demais conteúdos, pois ataca todos os conceitos iniciais da linguagem Java, portanto estude-o com bastante cuidado e atenção !

#### Palavras-Chave

Na linguagem Java, 49 são as palavras chaves e você deverá memorizá-las. Não tente fazer como fazíamos no ensino fundamental para decorar a tabuada (como era difícil a do 9, lembra?), essa assimilação será com o tempo, mesmo assim, dê uma olha na lista a seguir, e observe que TODAS as palavras chaves são definidas em letras minúsculas:

byte	short	int	long	char	boolean
double	float	public	private	protected	static
abstract	final	strictfp	transient	native	synchronized
void	class	interface	implements	extends	if
else	do	default	switch	case	break
continue	assert	const	goto	throws	throw
new	catch	try	finally	return	this
package	import	instanceof	while	for	volatile
super					

LEMBRE-SE: null, false, true (Não são palavras chaves, são valores literais!)

Lembrar de todas essas palavras pode ser um pouco complicado, por isso tente entender sua utilização !

#### Tipos primitivos

byte - Inteiro de 8 bits com sinal  
int - Inteiro de 16 bits com sinal  
short - Inteiro de 32 bits com sinal  
long - Inteiro de 64 bits com sinal  
char - Caractere Unicode (16 bits sem sinal)  
float - Ponto flutuante de 32 bits com sinal  
double - Ponto flutuante de 64 bits com sinal  
boolean - Valor indicando true ou false

#### Anotações

105

# Programação Orientada a Objetos

---

## Modificadores de acesso

private - Define que um método ou variável seja acessada somente pela própria classe.

protected - Faz com que uma subclasse acesse um membro da superclasse, mesmo estando em pacotes diferentes.

public - Faz com que um identificador possa ser acessado de qualquer outra classe.

## Modificadores de classe, métodos e variável

abstract - Define uma classe abstrata.

class - Define a implementação de uma classe.

extends - Define qual a hierarquia de classes, quem é a superclasse.

final - Faz com que um identificador não possa ser alterado.

implements - Faz com que uma classe implemente todos os métodos de uma interface.

interface - Define uma interface.

native - Define que o método será escrito em linguagem nativa como C, C++.

new - Instancia um novo objeto na pilha.

static - Define um identificador de classe e não de instância.

strictfp - Define que o método está segundo o padrão IEEE754.

synchronized - Define que um método só poderá ser acessado por uma única thread por vez.

transient - Faz com que uma variável não seja serializada.

volatile - Indica que uma variável pode não ficar sincronizada ao ser usada por threads.

## Controle de Fluxo

break - Faz com que o fluxo seja desviado para o fim do bloco

continue - Muda o curso do fluxo para a próxima iteração do loop

if - Testa o valor lógico de uma condição

else - Indica qual o bloco que deverá ser executado caso o teste feito pelo if seja falso.

default - Bloco que será executado caso nenhuma condição case satisfaça o switch.

switch - Iniciar uma sequência de testes para uma variável a ser testada pelo case.

**Anotações**

106

# Programação Orientada a Objetos

---

case - Testa o valor de uma variável indicada pelo switch

for - Usado para executar um bloco quantas vezes forem necessárias para satisfazer sua condição.

do - Executa um bloco quantas vezes a condição se fizer verdadeira. A condição é testada depois do bloco

while - Executa um bloco quantas vezes a condição se fizer verdadeira. A condição é testada antes do bloco

return - Finaliza a execução de um método, podendo opcionalmente retornar um valor.

instanceof - Testa se um objeto é instância de uma classe qualquer.

## Tratamento de erros

catch - Define o bloco de decisão que executará se por acaso ocorrer no bloco try uma exceção pré-definida.

finally - Bloco que sempre será executado, mesmo que uma exceção seja lançada.

throw - Lança uma exceção.

throws - Indica que um método pode lançar algum tipo de exceção

try - Iniciar um bloco com auditoria

assert - Usado no projeto, testa uma expressão para verificar alternativas para o programador.

## Controle de pacotes

import - Importa uma ou todas as classes de um pacote.

package - Define que a(s) classes farão parte de um pacote.

## Variáveis/Herança

super - Refere-se a superclasse imediata

this - Refere-se a instância do objeto

## Retorno

void - Define que um método não retorna nada

---

### Anotações

---

---

---

---

107

# Programação Orientada a Objetos

---

## Reservadas mas não utilizadas

const - Não use para definir uma variável, use final

goto - Não serve para nada

## Tipo primitivos

Você precisará saber "tudo" sobre os tipos primitivos, suas faixas de valores, valores padrões, conversões implícitas e explícitas, e muito mais. Por isso, prepare-se o seu pesadelo começou! Mas não se preocupe, com um pouco de fé e perseverança, tudo se resolve!

A Java é composta por 8 (oito) tipos primitivos, e como você já deve ter percebido, todos começam com letras minúsculas (pois também são palavras chaves), portanto se você for tiver que responder uma pergunta onde aparece perguntando por exemplo se String é um tipo primitivo, não hesite em responder que NÃO!

Outro conceito importante sobre os tipos primitivos em Java é que todos os tipos numéricos tem sinal. Mas o que isso significa? - deve ter vindo à sua mente! Significa que podem ser negativos ( - ) ou positivos ( + ), ou seja, o tipo int pode ter o numero -1 como também o numero +1. Observe a tabela de valores abaixo:

tipo	bits	fórmula	faixa
byte	8	$-(2)^7 \text{ à } 2^7 - 1$	-128 ~ +127
short	16	$-(2)^{15} \text{ à } 2^{15} - 1$	-32768 a +32767
int	32	$-(2)^{31} \text{ à } 2^{31} - 1$	-2147483648 a +2147483647
long	64	$-(2)^{63} \text{ à } 2^{63} - 1$	-9223372036854775808 a + 9223372036854775807

### Entendendo essa fórmula maluca...

Você deve estar se perguntando, porque deve elevar a 7 (sete) no caso do byte e não a 8 (oito), certo ? 1 Bit é usado para guardar o sinal. E porque só subtrai -1 da faixa positiva ? Porque o zero é incluído no intervalo também.

Se ainda ficou confuso, vamos fazer um teste. Um regra nós sabemos: o tipo byte em Java é representado por oito bits. Se 1 bit é usado para guardar o sinal (+) ou (-), então sobrou 7 bits para representar um número. Se convertermos o número +125 em bits teremos 1111101 e ficaria representado da seguinte forma: 01111101 (observe que o zero inicial indica que é um numero positivo, estranho mas se o primeiro dígito fosse um 1, seria um numero negativo). Agora se

### Anotações

108

# Programação Orientada a Objetos

---

convertermos o numero +128 em bits, teremos 10000000, como esse numero é composto por oito bits não é possível adicionar o bit do sinal portanto o numero 128 positivo não está no intervalo do tipo byte, e está na faixa negativa pois o bit mais significativo a esquerda é 1 que indica sinal negativo. Quais os bits representam o numero 127 ?

## Tipos primitivos (não acabou):

Os demais tipos primitivos são: char, float, double, boolean !

char - É representado por caracter Unicode de 16 bits (sem sinal). Também pode ser representado por um numero inteiro de 16 bits sem sinal, ou seja, pode-se atribuir à uma variavel *char* o valor 100 ou 14555 ou 65535, mas não 65536 ! Pois essa é a faixa de valores da tabela Unicode para representação de caracteres de qualquer idioma. Mas a unica coisa que você precisa saber é a faixa de valores do tipo *char*.  $2^{16} - 1 = 65535$  (valores possíveis).

float - É um tipo numérico ponto flutuante de 32 bits (COM SINAL) !! Apesar de ter um tamanho definido por 32 bits, não é necessário saber a faixa de valores.

double - É um tipo numérico ponto flutuante de 64 bits (COM SINAL) ! Apesar de ter um tamanho definido por 64 bits, não é necessário saber a faixa de valores.

boolean - Um valor que indicado pelos valores literais: true ou false

## Conversões implícitas/expícitas

Você pode sempre que precisar, fazer conversões entre os tipos numéricos, mas uma regra não pode ser quebrada: nunca você poderá converter um tipo de maior valor (bits) em um número de menos valor.

# Programação Orientada a Objetos

---

## Exemplo

```
1. public class Conversao {
2.     public static void main(String[] args) {
3.         int x = 10;
4.         long y = 20;
5.         y = x;           // perfeitamente possível
6.         x = y;           // não é possível
7.         x = (int)y;      // quero correr o risco e deixa eu queto !
8.     }
9. }
```

Na linha 5 houve um conversão implícita perfeitamente possível pois o tipo da variavel y é long, ou seja, maior que o tipo da variavel x.

Na linha 6 houve um tentativa de conversão, mas o compilador não permitira essa operação, pois o tipo long é maior que int.

Na linha 7, uma conversão explícita foi realizada e o compilador gentilmente atendeu a solicitação do programador.

## Literais Inteiros

Um valor literal em Java é um valor escrito no código fonte e identificado como um tipo primitivo como por exemplo:

```
int x = 10;           // literal inteiro
char u = 'k';         // literal char
boolean b = false;    // literal boolean
double d = 9832.11;   // literal double
```

Há três maneiras de representar valores inteiros em Java: octal (base 8), decimal (base 10) e hexadecimal (base 16) ! Será bom você estudar um pouco sobre esses sistemas de numeração, pode-se que os sacanas dos caras que fazem a prova, possa testar você em alguma questão! Mas observe o código abaixo e veja como se representam valores literais em octal e hexadecimal (pois decimal você viu acima):

## Anotações

110

# Programação Orientada a Objetos

---

```
public class Octal {  
    public static void main(String[] args) {  
        int seis = 06; // idem ao decimal 6  
        int sete = 07; // idem ao decimal 7  
        int oito = 010; // idem ao decimal 8  
        int nove = 011; // idem ao decimal 9  
    }  
}
```

Ou seja, se você ver o número 0 antes de um número, saiba que está representando um número octal.

```
public class Hexadecimal {  
    public static void main(String[] args) {  
        int quinze = 0xF; // 15 (decimal)  
        int vinte_e_nove = 0x1D; // 29 (decimal)  
        int vinte_e_nove = 0x1D; // 29 (decimal)  
        int valor = 0xBAFAFA; // 12253946 (decimal)  
        // também pode ser  
        int valor = 0XBaFaFa; // 12253946 (decimal)  
    }  
}
```

Deve vir precedido do sufixo (0x) ! É um dos poucos casos na linguagem em que não importa se as letras forem maiúsculas ou minúsculas. Todos os tipos inteiros literais (tanto octal quanto hexa) são por padrão definidos como int, portanto se ver no exame um questão que atribui a uma variável menor que int (em bits) isso dará erro de compilação. Veja o código a seguir:

```
1. public class ConversaoHexa {  
2.     public static void main(String[] args) {  
3.         int a = 0xbafa; // ok, sem conversão  
4.         long b = 0xffff; // ok, conversão implícita  
5.         long c = 0xffffL; // ok, conversão explícita  
6.         byte d = 0xf; // ok, conversão implícita  
7.         byte e = 0xff; // erro! - 255 não é comportado  
8.         byte f = (int)0xff; // erro! - 255 não é comportado  
9.     }  
10. }
```

# Programação Orientada a Objetos

---

Na linha 6 o compilador sabe que F em hexa equivale a 15, o que pode ser perfeitamente suportado no tipo byte, então ele realiza a conversão.

Na linha 7 o compilador sabe que FF em hexa equivale a +255 o que não pode ser suportado, por isso, erro!

Na linha 8 o compilador só não chinga o programador por conflito de drivers entre seu sistema operacional com sua placa de som, ou seja, mas intimamente ele fala: CARA OLHA O QUE VOCÊ ESTÁ QUERENDO ME FORÇAR A FAZER !!! (%#%\$!#%)

## Literais de ponto flutuante

Um valor literal de ponto flutuante por padrão em Java é definido com double de 64 bits, portanto de você quiser atribuir um valor literal *float* você deverá adicionar o sufixo f no final do valor como o exemplo:

```
1. public class Teste {  
2.     public static void main(String[] args) {  
3.         double a = 9223372036854775807.0;    // ok tipo double  
4.         float b = 2147483647; // ok tipo int para float conversao implicita  
5.         float c = 2147483647.0;              // erro! double -> float  
6.         float d = (float)2147483647.0;       // ok - conversão  
7.         float e = 2147483647.0f;            // ok  
8.     }  
9. }
```

Na linha 4, funciona pois 2147483647 é um literal int e não double! Não tem decimal !

Na linha 5, o compilador reclamará pois 2147483647.0 é um tipo double (o padrão dos literais de ponto flutuante) não pode ser atribuído a uma variável float.

Na linha 6 é feito uma conversão explícita.

Na linha 7 é atribuído um valor float por causa do sufixo f.



# Programação Orientada a Objetos

---

## Literais booleanos

Os valores literais booleanos são compreendidos entre true ou false e só !

```
public class LitBoo {  
    public static void main(String[] args) {  
        boolean a = true;    // ok  
        boolean b = false;   // ok  
        boolean c = 1;       // erro de compilacao  
    }  
}
```

Cuidado que em Java diferentemente de C e outras linguagens não se pode atribuir o valor 0 ou 1 para literais booleanos.

## Literais Caracteres

Os valores literais caracteres são compreendidos com um único caracter entre apóstrofo - se você não sabe o que é apóstrofo, é o mesmo que aspas simples '

```
public class Carac {  
    public static void main(String[] args) {  
        char a = 'a';           // ok  
        char b = '@';           // ok  
        char c = "\u004E";      // refere-se a letra N  
    }  
}
```

Como foi falado, o tipo *char* nada mais é do que um tipo inteiro sem sinal de 16 bits, portanto você poderá atribuir  $2^{16} - 1 = 65535$ . Veja no código abaixo:

```
public class A {  
    public static void main(String[] args) {  
        char a = 65;  
        char b = (char)-65;      // fora do intervalo, precisa de conversão  
        char c = (char)70000;    // fora do intervalo, precisa de conversão  
    }  
}
```

O tipo *char* pode aceitar valores fora de sua faixa, desde haja uma conversão explícita.

# Programação Orientada a Objetos

---

## ARRAY

Um array em Java é um objeto criado na pilha (memória), usado para armazenar e trabalhar com elementos semelhantes por seu tipo. Para que se possa utilizar um array você deverá:

Declarar - Especificar um nome e o tipo do array.

Construir - Informar o tamanho do array, ou seja, numero de elementos.

Inicializar - Atribuir valores aos elementos do array.

### Declarando um array

```
int[] a;           // Recomendado
Thread b[];
String []c;
```

Observe que ainda não se sabe quantos elementos esses array armazenará, ou seja, não sabe qual será o custo para a memória desse array. Nunca coloque a quantidade de elementos do array no passo de declaração, a não ser que você faça tudo em uma única linha (isso será mostrado posteriormente). Se você ver uma questão onde aparece algo como no trecho a seguir, marque sempre erro de compilação.

```
int[3] a;          // só pode mencionar a quantidade de elementos, na construção
```

### Construindo um array

```
int[] a = new int[3];           // Recomendado
Thread b[] = new Thread[1];
String []c = new String[19];
```

Usa-se a palavra new conjugada com o tipo do array. Nunca se esqueça, em Java a contagem dos elementos SEMPRE COMEÇARÁ EM 0 (ZERO), portando uma referência ao elemento a[3] (no array a acima) causará um erro, pois só existem os elementos 0, 1, 2 -> com um total de 3 elementos. Esse passo reserva espaço na memória para os elementos do objeto array, pois somente na construção que a JVM saberá quantos elementos serão composto o array, com isso cria-se um objeto na pilha com espaço necessário para armazenar os elementos do objeto. No passo de construção, todos os elementos são inicializados com seus valores padrão.

### **Anotações**

114

# Programação Orientada a Objetos

---

Veja a tabela a seguir:

TIPO	VALOR PADRAO
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
boolean	false
char	'\u0000'
Object	null

## Inicializando um array

Atribuir valores aos elementos de um objeto array. Quando um array é contruído, seus elementos são automaticamente inicializados com seus valores padrão.

```
int[] x;      // declarado
x = new int[2]; // construindo
x[0] = 10;    // inicializando
x[1] = 20;    // inicializando
```

Observe o seguinte código:

```
1. public class ArrayInicia {
2.     public static void main(String[] args) {
3.         float[] f;
4.         f = new float[1];
5.         System.out.println("valor antes "+f[0]);
6.         f[0] = 9.0;
7.         System.out.println("valor depois "+f[0]);
8.     }
9. }
```

---

**Anotações**

115

## Programação Orientada a Objetos

---

O que será impresso na linha 5 e 7 ??? ( Feche os olhos e responda !!, Não vale olhar...) Se você respondeu 0.0 e 9.0, parabéns por você ser uma pessoa de opinião! Mas infelizmente você errou !!! Lembre-se que todo valor literal ponto flutuante em Java é por padrão double, portanto esse código não compila. Se alterarmos esse código e adicionar o sufixo f na linha 6 => f[0] = 9.0f;;, o resultado seria

0.0 e 9.0, por isso: PRESTE MAIS ATENÇÃO !!!

Os três passos para a utilização de um array: declaração, construção e inicialização podem ser realizados em uma única linha de código. EUREKA !!!

```
boolean[] c = { false, true, false };  
int[] a = {0,1,1,1};  
char[] b = {'a','b','c'};
```

Observe que a palavra chave new não foi utilizada, visto que está implícito o tipo no início, o número de elementos entre as chaves { }. Caso você se depare com uma questão (como o código abaixo) que especifica o número de elementos juntamente com a inicialização na mesma linha, não hesite em marcar a resposta: Erro de compilação !!!

```
int[3] a = {1, 2, 1}; // erro de compilação
```

### Array Multidimensional

Um array multidimensional é um array com mais de uma dimensão (isso é ridículo de dizer!), ou seja, é uma coleção de objetos array dentro de um objeto array. Portanto um array definido como: int[][] i = new int[3][]; nada mais é do que um objeto array i que contém três objetos array (ainda não construído) dentro. (Complicado?)

```
int[][] i = new int[2][];
```

O que isso significa ? O que são os elementos de i ?? Significa que foi criado um objeto na pilha chamado a, e seus elementos ainda não foram contruídos. Para utilizar seus elementos, você deverá construí-los como mostra o código a seguir:

```
i[0] = new int[2]; // construído o elemento 0 do array i  
i[1] = new int[3]; // construído o elemento 1 do array i
```

---

### Anotações

116

# Programação Orientada a Objetos

---

Quantos objetos foram criados na pilha ?

- 1 referenciado por a
- 2 referenciados por a[0] e a[1]

Total de objetos criados: 3 (três)

Agora observe o seguinte código:

```
public class TestArray {  
    public static void main(String[] args) {  
        String s = new String("Kuesley");  
        String[] nomes = { s, null, new String("Kuesley") };  
    }  
}
```

Quantos objetos foram criados na pilha ??

- 1 obj String referenciado por s
- 1 obj array de String referenciado por nomes
- 1 obj String referenciado por nomes[2]

Observe que o elemento 0 é apenas uma referência para s portanto não é criado um novo objeto. O elemento 1 não tem um objeto referenciado, já o elemento 2 é um objeto String.

## Array Anônimo

Como o próprio nome já diz, é um objeto array criado sem a definição de um nome. Imagine que você precise passar um objeto array como parâmetro para um método, e você não queira criar um array, basta passar anonimamente. Veja como no código abaixo:

---

**Anotações**

117

# Programação Orientada a Objetos

---

```
public class A {
    public static void main(String[] args) {
        A obj_a = new A();
        int soma = obj_a.somarArray( new int[] { 0,1,2,3 } );
        System.out.println("Soma do array é: "+soma);
    }
    public int somarArray( int[] a ) {
        int rc = 0;
        for ( int i=0; i < a.length; i++) {
            rc += a[i];
        }
        return rc;
    }
}
```

Observe que não foi criado um objeto array com um identificador específico e passado como parâmetro, foi criado no momento em que se passará o argumento.

Outro exemplo:

```
int[][] numeros = new int[3][];
numeros[0] = new int[10];
numeros[1] = numeros[0];
numeros[2] = new int[] { 0,1,2,3 };
```

LEMBRE-SE: NUNCA ESPECIFIQUE O TAMANHO ENTRE OS COLCHETES, ISSO SERÁ DEDUZIDO DOS ELEMENTOS ENTRE O PAR DE CHAVES !!!

## Array - Observações

Algumas regras devem ser consideradas no uso de array que referenciam objetos!

Dado as classes:

```
class Car implements CarMove { }
class Mazda extends Car { }
class Fusca extends Car { }
class VMax { }
interface CarMove { }
```

## **Anotações**

---

---

---

---

118

# Programação Orientada a Objetos

---

```
1. public class Test {  
2.     public static void main(String[] args) {  
3.         Car[] cars = new Car[3];  
4.         Fusca f = new Fusca();    // instanciando um obj do tipo Fusca  
5.         cars[0] = f;  
6.     }  
7. }
```

Observe que na linha 5, um objeto Fusca foi armazenado em um array do tipo Car. Por que isto é possível ?

Existe uma pergunta que você sempre deverá fazer, para saber se uma classe X pode ser armazenada em um array do tipo Y.

X é membro de Y

Em nosso contexto: Fusca é membro de Car ?

Em outras palavras, Fusca é uma subclasse de Car ?

Se a resposta for positiva, essa atribuição é perfeitamente possível !

Agora observe o seguinte código (baseando-se nas mesmas classes Car, Fusca )

```
1. public class Test {  
2.     public static void main(String[] args) {  
3.         Fusca[] fuscas = new Fusca[3];  
4.         Car c = new Car();  
5.         fuscas[0] = c;  
6.     }  
7. }
```

Isso é possível ?? Se tem dúvida, faça você mesmo e faça o teste e sem preguiça !!!

Aproveite e teste o seguinte:

```
1. public class Test {  
2.     public static void main(String[] args) {  
3.         Car[] c = new Fusca[1];  
4.         Fusca[] f = new Car[1];  
5.     }  
6. }
```

---

**Anotações**

119

# Programação Orientada a Objetos

---

O que acontecerá com o código acima ?

## Mudando um pouco de assunto!

Um objeto array de uma interface, pode receber referencias de instâncias de classes que implementem essa interface.

```
public class Test {  
    public static void main(String[] args) {  
        CarMove[] cm = new CarMove[4];  
        cm[0] = new Car(); // ok! Car implementa CarMov  
        cm[1] = new Fusca(); // ok! Fusca implementa CarMov  
        cm[2] = new Mazda(); // ok! Mazda implementa CarMov  
        cm[3] = new VMax(); // erro de compilação  
    }  
}
```

LEMBRE-SE: Vmax é membro de CarMove ???

Cuidado com detalhes de atribuições tais como:

```
int[] n = new int[10];  
int[] m = new int[] {8,2,1,2,0};  
n = m; // ok! mesmo tipo  
int[][] n = new int[3][];  
int[] m = new int[] {1,2,3};  
n = m; // erro! objetos diferentes
```

A JVM não elimina o objeto n da pilha e substitui pelo valor de m!

Observe também que o array n tem 2 dimensões, enquanto que m tem apenas uma!

Mas o seguinte código seria perfeitamente possível:

```
n[0] = m; // ok!
```

As questões relacionadas com array multidimensionais são esdrúxulas, portanto ESTUDE!

Se achas que estou brincando imagine a seguinte questão ?

---

**Anotações**

120



# Programação Orientada a Objetos

---

```
1. public class Test {
2.     public static void main(String[] args) {
3.         int[][][] nums = new int[2][][];
4.         int[] simple = new int[] {1,2,3,4,5};
5.         nums[0] = new int[1][];
6.         nums[0][0] = new int[10];
7.         nums[1] = new int[3][];
8.         nums[1][0] = simple;
9.         nums[1][1] = new int[] { 3,3,3 };
10.        nums[1][2] = new int[] { 1,2,3 };
11.        ???
12.    }
13. }
```

Qual das linhas abaixo poderão ser colocadas na linha 11 do código acima sem que dê erro de compilação ?? (marque mais de uma)

- a) `nums[1][0] = nums[0][0]`
- b) `nums[1][0] = 10;`
- c) `nums[1][0][3] = 9;`
- d) `nums[1][0][2] = 9;`
- e) `nums[0][0][3] = 9;`
- f) `nums[0][0] = nums[2][1]`

Não é sacanagem, os caras dão questões daí pra pior !!!

## Variáveis de Instância

Declarada no escopo da classe, e tem um valor padrão conforme o seu tipo. (Mesmos valores atribuídos na inicialização de um array estudado anteriormente)

```
public class Book {
    String title;
    public static void main(String[] args) {
        Book b = new Book(); // instanciando a classe
        System.out.println("O titulo é "+b.title);
    }
}
```

Anotações

121

# Programação Orientada a Objetos

---

Resultado: O titulo é null ! String é um objeto e não um tipo primitivo!

Para utilizar um variável primitiva, você SEMPRE terá que inicializá-la, caso contrário o compilador lhe tratará de forma inescrupulosa !

```
public class TheBookonTheTable {
    public static void main(String[] args) {
        int i;
        System.out.println("o valor de i é "+i);
    }
}
```

O código acima gerará um conflito entre o programador e o compilador! Não tente! É irreparável!!!

## Objeto Local

Um Objeto local (variável declarada dentro de um método) tem comportamento distinto de uma variável de instância. Quando um objeto é declarado como membro de uma classe, e uma instância dessa classe é criada, esse membro é inicializado com null (pois em Java null é um valor) e quando é declarado no escopo de um método, o objeto não é inicializado, portanto qualquer ousadia de uso, relações cortadas com o compilador!

```
1. import java.util.*;
2. public class Test {
3.     public static void main(String[] args) {
4.         Date data;
5.         Periodo p = new Periodo();
6.         if (data == null) System.out.print("inicio é nulo"); // erro de compilação
7.         if (p.inicio == null) System.out.print("inicio é nulo"); // ok - membro é
nulo
8.         if (p.fim == null) System.out.print("fim é nulo"); // ok - membro é nulo
9.     }
10. }
11. class Periodo {
12.     Date inicio;
13.     Date fim;
14. }
```

**Anotações**

122

# Programação Orientada a Objetos

---

O compilador mostrará um erro na linha 6, pois o objeto local data não foi inicializado !

## Método main

Você já deve estar cansado de ver as declarações `public static void main(String[] args)`, porém saiba que essa é a chave para a JVM executar uma aplicação em Java. Maiores detalhes sobre modificadores serão vistos no capítulo 2, mas algumas observações devem ser ressaltadas nesse capítulo. Observe:

```
public static void main(String[] args) { ... } // válido - recomendado
static public void main(String[] args) { ... } // válido
public static void main(String[] a) { ... } // válido
public static void main(String [] a) { ... } // válido
public static void main(String a[]) { ... } // válido
public static void main(String a []) { ... } // válido
public static void main([]String args) { ... } // não é válido
```

Mas o que é esse array de String que o método main recebe ?  
Esse array é usado para receber parâmetros da linha de comando, quando um programa Java está sendo executado.

```
c:\>java Test Kuesley Fernandes
```

Nesse caso, dois parâmetros estão sendo passados: Kuesley Fernandes

Portanto com o seguinte código:

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Meu nome é: "+args[0]+args[1]);
    }
}
```

O Resultado seria: Meu nome é Kuesley Fernandes  
E se fosse executado: `c:\>java Test Kuesley`  
Qual seria o resultado ?? TESTE !!!

---

## Anotações

123

# Programação Orientada a Objetos

---

## QUESTÃO

O que fará o seguinte código ?

```
public class Array {  
    int[] a;  
    public static void main(String[] args) {  
        Array obj = new Array();  
        if (obj == null) {  
            System.out.println("ERRO");  
        } else {  
            System.out.println("OK");  
        }  
    }  
}
```

- a) Mostrará ERRO na tela
- b) Mostrará OK na tela
- c) Programa não irá compilar
- d) Uma exceção será lançada
- e) Mostrará null na tela

Anotações

124

## Resumos para certificação - Fundamentos da Linguagem

### Palavras Chave da Linguagem Java

- As palavras chave não podem ser usadas como identificadores (nomes) de classes, métodos, variáveis ou qualquer outro item de seu código;
- Todas as palavras chave começam com letra minúsculas.

### Valores literais e intervalos de todos os tipos primitivos

- Todos os seis tipos de números em Java apresentam sinal, portanto, podem ser positivos e negativos;
- A fórmula  $-2^{(\text{bits}-1)}$  ou  $2^{(\text{bits}-1)} - 1$  é usada para determinar o intervalo de um tipo inteiro;
- Os valores literais são uma apresentação do código fonte para tipos de dados primitivos ou Strings;
- Um tipo char, na verdade, é um inteiro de 16 bits sem sinal;
- Os inteiros podem ser representados como octais(0127), decimais(1254) e hexadecimais(0XCAFE);
- Os literais não podem ter vírgulas;
- Um literal char pode ser representado como um único caractere em aspas simples 'A';
- Um literal char também pode ser representado como um valor Unicode '\u0041';
- Um literal char pode ser representado como um valor inteiro, contanto que o inteiro seja menor do 65536;
- Um literal boolean pode ter o valor true e false;
- Os literais de ponto flutuante são sempre double por padrão; se quiser um float acrescentar F ou f ao literal.

### Declaração, construção e inicialização de arrays

- Os arrays podem conter tipos primitivos ou objetos, mas o array propriamente dito sempre será um objeto;
- Quando declarar um array, os colchetes poderão ficar à esquerda ou à direita do nome da variável;
- Não é considerado válido incluir o tamanho de um array na declaração;
- É preciso incluir o tamanho de um array quando o construir usando new a menos que esteja criando uma array anônimo;
- Os elementos de um array de objetos não são criados automaticamente,

Anotações

125

# Programação Orientada a Objetos

---

- embora elementos de arrays primitivos recebam valores padrão;
- Você receberá um `NullPointerException` se tentar usar elemento de um array de objetos e esse elemento não referenciar a um objeto real;
- Os arrays são indexados começando pelo zero.;
- Receberá um `ArrayIndexOutOfBoundsException` se tentar acessar algo fora do intervalo de um array;
- Os arrays possuem uma variável `length`, que contém a quantidade de elementos do array;
- O último índice que será acessado é sempre uma unidade menor do que o tamanho do array;
- Os arrays multidimensionais são apenas arrays compostos por arrays;
- As dimensões de um array multidimensional podem ter tamanhos diferentes;
- Um array de tipos primitivos aceitará qualquer valor que possa ser promovido implicitamente ao tipo declarado para o array. Por exemplo uma variável `byte` pode ser inserida em um array `int`;
- Um array de objetos pode conter qualquer objeto que passe no teste `É MEMBRO` (ou `instanceof`) aplicado ao tipo declarado para o array;
- Se atribuir um array a uma referência de array já declarada, o array terá que estar na mesma dimensão da referência ao qual for atribuído;

## Usando uma variável ou elemento de array que não tenha sido inicializado ou atribuído

- Quando um array de objetos for instanciado, os objetos do array não serão instanciados automaticamente, mas todas referências receberão valor padrão `null`;
- Quando um array de tipos primitivos for instanciado. Todos os elementos receberão seus valores padrões;
- Exatamente como os elementos de array, as variáveis de instância serão sempre serão inicializadas com um valor padrão;
- As variáveis locais \ automáticas \ de método nunca recebem um valor padrão. Se tentar usar sem inicializá-la, receberá um erro do compilador;
- Variáveis locais e a mesma coisa que automáticas.

## Argumentos de linha de comando para o método main

- Os argumentos de linha de comando são passados para o parâmetros de array de String no método principal;
- O primeiro argumento da linha de comando será o primeiro elemento do parâmetro de array de string do método principal;
- Se nenhum argumento for passado para o método principal, o tamanho do parâmetro de array de String desse método será igual a zero.

# Programação Orientada a Objetos

---

## Capítulo II - Modificadores e Controle de Acesso

Como em todas as linguagens de programação, a acessibilidade a uma classe/método deve seguir algumas regras, e é de extrema necessidade que você as saiba para não vacilar na hora do exame!

### Lista dos modificadores

public	private	protected	abstract
static	final	transient	strictfp
volatile	native	padrão	synchronized

LEMBRE-SE: com exceção ao modificador PADRÃO, todas as demais são palavras chaves em Java.

### Modificadores para as Classes

#### padrão

Um modificador de acesso determina como será a visibilidade de uma classe/método a partir de outras classes ou métodos.

Dado o arquivo Car.java:

```
class Car {  
    // código da classe  
}
```

Observe que não foi definido nenhum modificador para a classe, portanto o modificador de acesso nesse caso é padrão (default ou como já vi em algum lugar na terra - friendly)! O modificador padrão define que a classe só poderá ser acessada por outras classes dentro do mesmo pacote. Uma tentativa de acesso a classe Car a partir de uma classe de outro pacote resultará em uma ofensa ao compilador.



# Programação Orientada a Objetos

---

Dada as classes:

```
Car.java
package carros;
class Car { ... }
```

```
Oficina.java
package oficina;
import carros.*;
class Oficina extends Car { ... } // Erro de compilação
```

```
Conserto.java
package conserto;
import carros.*;
class Conserto {
    public static void main(String[] args) {
        Car c = new Car(); // Erro de compilacao
    }
}
```

Observe que na classe Oficina houve uma tentativa de acesso por herança, mas como a classe Car tem acesso padrão e a classe Oficina está em um pacote diferente do pacote da classe Car, causou erro de compilação. Semelhantemente o erro ocorreu na classe Conserto, apesar de que a forma de tentativa de acesso à classe Car foi através de referencia. Agora observe:

```
Car.java
package carros;
class Car { ... }
```

```
Oficina.java
package carros;
class Oficina extends Car { ... } // ok
```

```
Conserto.java
package carros;
class Conserto {
    public static void main(String[] args) {
        Car c = new Car(); // ok
    }
}
```

**Anotações**

129

# Programação Orientada a Objetos

---

Observe que uma classe com modificar padrão pode ser acessada somente por classes/métodos do mesmo pacote.

## public

O modificador public é o mais liberal, ou seja, faz com que a classe possa ser acessada por qualquer classe independente de estarem ou não no mesmo pacote.

```
Car.java
package carros;
public class Car {
    String getColor() {
        return "yellow";
    }
}
```

```
Oficina.java
package oficina;
import carros.*;
    public class Oficina {
        public static void main(String[] args) {
            Car c = new Car(); // ok
        }
    }
```

Observe que a classe Car é publica e é visível por qualquer classe, portanto muito cuidado ao definir uma classe como pública, pode cair na vista do inimigo.

O que acontecerá com o seguinte código:

```
Car.java
package carros;
public class Car {
    String getColor() {
        return "yellow";
    }
}
```

---

**Anotações**

130

# Programação Orientada a Objetos

---

```
Oficina.java
package oficina;
import carros.*;
public class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        System.out.println("A cor e "+c.getColor());
    }
}
```

- a) Mostrará: A cor e yellow
- b) Erro de compilação
- c) Exceção na hora da execução
- d) Mostrará: A cor e null
- e) Mostrará: A cor e

Caso vc tenha errado, não se preocupe, pois os modificadores aplicados a membros serão discutidos posteriormente.

## abstract

Uma classe definida como abstract não permite criar instâncias dessa classe, ou seja, não podemos ter objetos de uma classe abstrata. O modificador abstract aplica o mesmo conceito de abstração que conhecemos no mundo real, ou seja, parta do pressuposto que uma classe abstrata não sabe qual seja o seu "inteiro" comportamento.

Você deve estar se perguntando, por que então eu usaria uma classe abstrata se não posso instanciá-la ? Esse é um recurso interessantíssimo das linguagens orientadas a objetos, a extensibilidade, você pode definir uma classe Car com comportamentos (métodos) abstratos e deixar que as subclasses definam esses comportamentos de forma diferente, mesmo porque a forma de aceleração de um Mazda RX7 é bem diferente de um Fusca.

```
Car.java
package carros;
public abstract class Car { ... }
```

---

## Anotações

131

# Programação Orientada a Objetos

---

```
Oficina.java
package carros;
public class Oficina {
    public static void main(String[] args) {
        Car c = new Car(); // erro de compilação
    }
}
```

Erro: // carros.Car is abstract; cannot be instantiated

Não se preocupe se você ver uma declaração:

```
abstract public class Car
```

Não é erro de sintaxe colocar o modificar abstract antes do public.

Podemos concluir que uma classe abstrata não está pronta para ser usada como uma classe concreta, ou seja, ainda não sabe "coisas" sobre a classe, que muito provavelmente será implementado em subclasses (a não ser que algum programador resolva criar uma classe abstrata e não usá-la - puro capricho!)

## final

Um classe final não permite que se tenha subclasses, ou seja, não se pode aplicar a herança em uma classe que seja final. Novamente você como é curioso quer saber porque usaria uma classe final ? Pense em um contexto que se tenha uma classe "perfeita" que faz tudo exatamente como você definiu, e não precise que sua classe seja herdada por ninguém, para evitar que Programadores Juniores façam cacas !!! Tem mais um detalhe técnico mais importante que o citado anteriormente: uma classe final, é mais rápida que classes não final, isso porque a máquina virtual sabe que não haverá nenhuma herança (nem pelos próprios caras da Sun) por isso o processo de execução é mais rápido. Um exemplo disso é a classe String.

```
Car.java
package carros;
public final class Car { ... }
```

---

## Anotações

132

# Programação Orientada a Objetos

---

```
Oficina.java
package carros;
public class Oficina extends Car { // erro de compilação
    public static void main(String[] args) {
        Car c = new Car();
    }
}
```

Erro: cannot inherit from final carros.Car

Se você é um cara esperto, já deve ter percebido que o modificador final não deve ser usado com o modificador abstract, visto porque não faz sentido termos uma classe final e abstrata ! Como faríamos a herança sendo que a classe é final !

```
Car.java
package carros;
public abstract final class Car { ... } // erro de compilação
```

Erro: illegal combination of modifiers: abstract and final

## strictfp

Define que membros/variáveis da classe sigam as normas do padrão IEE754 de ponto flutuante. Se você não tiver o que fazer, poderá estudar esse padrão, mas saiba que para o exame não será necessário saber nada sobre o padrão, simplesmente o que o modificador faz. Como seu escopo é a classe, todos os métodos seguiram o mesmo padrão. Você também poderá definir esse modificador para um método específico, mas isso será discutido posteriormente.

```
Car.java
package carros;
public strictfp class Car { ... } // ok
```

Todos os métodos seguiram o padrão IEE754

## MODIFICADORES PARA MÉTODOS

### padrão

Não será necessário falar muito, pois é semelhante ao conceito de classe, ou seja, um método definido como padrão só poderá ser acessado por classes dentro do mesmo pacote.

```
Car.java
package carros;
public class Car {
    String getColor() { método com acesso padrão
    return "red";
    }
}
```

```
Oficina.java
package carros;
public class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        System.out.println("a cor do carro e "+c.getColor()); // ok
    }
}
```

Nenhum problema de acesso pois as classes estão no mesmo pacote

Se definirmos a classe Oficina da seguinte forma:

```
package oficina;
import carros.*;
public class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        System.out.println("a cor do carro e "+c.getColor()); // erro de
        compilação
    }
}
```

Erro: getColor() is not public in carros.Car; cannot be accessed from outside package

### Anotações

134

# Programação Orientada a Objetos

---

## public

Um método público pode ser acessado por qualquer classe em qualquer pacote. É óbvio que o acesso a um método só é permitido se você tiver primeiro acesso à classe, portanto uma tentativa de acesso a um método público de uma classe com acesso padrão não será possível a classes pertencentes a pacotes diferentes da classe que está se desejando o acesso! Se ficou confuso, não se preocupe, vai piorar!

```
Car.java
package carros;
class Car {
    public String getColor() {
        return "red";
    }
}
```

```
Oficina.java
package oficina;
import carros.*;
class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        System.out.println("a cor e "+c.getColor()); // erro de
        compilação
    }
}
```

Erro: carros.Car is not public in carros; cannot be accessed from outside package

Portanto, mesmo que o método seja público (como é o caso de getColor()), a classe (nesse caso Car) também deverá ser visível ao método que está chamando!

## private

Um método private restringe o acesso do método somente à classe que o definiu, ou seja, um método privado só poderá ser acesso dentro da classe que o definiu e ponto final!

---

### Anotações

---

---

---

---

135

# Programação Orientada a Objetos

---

```
Car.java
package carros;
class Car {
    private String getColor() { // Nota 1
        return "red";
    }

    public void imprimirCor() {
        System.out.println( "a cor do carro e "+getColor()); // Nota 2
    }
}

Oficina.java
package carros;
class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        System.out.println("a cor e "+c.getColor()); // Nota 3
        c.imprimirCor(); // Nota 4
    }
}
```

**Nota 1:** O método foi definido como private

**Nota 2:** O acesso ao método getColor está sendo feito dentro da própria classe. Isso não ocasiona erro de compilação!

**Nota 3:** Uma tentativa de acesso a um método private - erro de compilação!

**Nota 4:** Acesso a um método público que acessa um método private, funciona como uma interface entre o método getColor e a classe Oficina, isso é perfeitamente possível.

Esse modificador é o mais restringível, os inimigos tremem quando ele é usado!

## protected

Ele é um pouco mais liberal que o modificador padrão, pois ele permite que um método de uma classe X definida em um pacote PX possa ser acessado por uma classe Y de um pacote PY desde que a classe Y estenda da classe X ( que confusão !!!) Não esquentar isso é bem fácil!

---

## Anotações

136



# Programação Orientada a Objetos

---

```
-- Car.java
package carros;
public class Car {
    protected String getColor() { // Nota 1
        return "red";
    }
}

-- Oficina.java
package oficina;
import carros.*;
class Oficina extends Car { // Nota 2
    public void imprimirCor() {
        System.out.println("a cor e "+getColor()); // Nota 3
    }
}
```

**Nota 1:** O método foi definido com protected

**Nota 2:** A classe Oficina estende de Car - por isso os métodos protegidos podem ser acessados.

**Nota 3:** O método está sendo acessado por herança, mesmo que a classe Oficina esteja definido em pacotes diferentes!

Observe agora:

```
-- Car.java
package carros;
public class Car {
    protected String getColor() {
        return "red";
    }
}

-- Oficina.java
package oficina;
import carros.*;
class Oficina { // Nota 1
    public void imprimirCor() {
        Car c = new Car();
        System.out.println("a cor e "+c.getColor()); // Nota 2
    }
}
```

**Anotações**

137

# Programação Orientada a Objetos

---

**Nota 1:** A classe Oficina não estende de nenhuma superclasse.

**Nota 2:** Há uma tentativa de acesso por referência ao método getColor que é protegido, o que causará um erro de compilação: `getColor() has protected access in carros.Car`

Uma classe que herda um método protegido torna-o private para suas subclasses. Complicou ?

```
-- Car.java
package carros;
public class Car {
    protected String getColor() {
        return "red";
    }
}

-- Passeio.java
package passeio;
import carros.*;
class Passeio extends Car {
    public void imprimirCor() {
        System.out.println("a cor é "+getColor()); // Nota 1
    }
}

-- Mazda.java
package esportivo;
import carros.*;
class Mazda extends Passeio {
    public void mostrarCor() {
        System.out.println("a cor é "+getColor()); // Nota 2
    }
}
```

**Nota 1:** O método pode ser acessado, pois herda de uma classe que o definiu como protegido.

**Nota 2:** Erro de compilação, nessa hierarquia, o direito de acesso da classe Passeio ao método protegido da classe Car, não é outorgado à classe Mazda pois essa herda de Passeio, ou seja, um método protegido se torna private para suas subclasses quando há herança, não permitindo que as subclasses de Passeio herdem os métodos protegidos que herdou.

**Anotações**

138

## abstract

Um método abstrato não implementa nenhuma funcionalidade, somente assina o método e faz com que a primeira subclasse concreta seja obrigada a implementar o método. Uma classe que possua um método abstrato deve obrigatoriamente ser abstrata!

```
-- Car.java
package carros;
public abstract class Car { // Nota 1
    public abstract String getColor(); // Nota 2
}

-- Oficina.java
package oficina;
import carros.*;
public class Oficina extends Car { // Nota 3
    public String getColor() {
        return "red";
    }
}
```

**Nota 1:** A classe Car também teve que ser definida com abstract, uma vez que uma classe contenha um método abstract

**Nota 2:** A definição de um método abstrato não pode ter implementação, veja que nem as chaves foram colocadas, e é assim que deve ser!

**Nota 3:** A classe Oficina é uma classe concreta (não é abstrata) portanto deve implementar todos os métodos abstratos da sua superclasse, se não fizer, o compilador insultará o programador, é melhor não brincar!

# Programação Orientada a Objetos

---

## Erros comuns

### Tentar implementar um método abstrato:

```
-- Car.java
package carros;
public abstract class Car {
    public abstract String getColor() { ... } // Nota 1
}
```

// Nota 1: Erro de compilação: abstract methods cannot have a body

LEMBRE-SE: Um método abstrato deve terminar com ";" e não ter as chaves!

### Não definir a classe como abstrata

```
-- Car.java
package carros;
public class Car {
    public abstract String getColor(); // Nota 1
}
```

// Nota 1: Erro de compilação: carros.Car should be declared abstract;  
it does not define getColor() in carros.Car

### Não implementar os métodos abstratos na primeira subclasse concreta

```
-- Car.java
package carros;
public abstract class Car {
    public abstract String getColor();
    public abstract String getSize();
    public double getPrice() {
        return 24000.00;
    }
}
```

---

## Anotações

140

# Programação Orientada a Objetos

---

```
-- Passeio.java
package carros;
public class Passeio extends Car { // Nota 1
    public String getColor() { // Nota 2
        return "red";
    }
}
```

**Nota 1:** A classe Passeio estende de Car (classe abstrata) por isso, deve implementar todos os métodos abstratos, o que não está acontecendo, se tentarmos compilar essa classe: **carros.Passeio should be declared abstract; it does not define getSize() in carros.Car**

**Nota 2:** O método foi implementado.

A regra é simples: todos os métodos abstratos devem ser implementados na primeira subclasse concreta, se definíssemos a classe Passeio como abstract no caso anterior, não teria problemas de compilação, pois o método abstratos (getSize, getColor) não precisavam ser implementados na classe Passeio e sim na primeira subclasse concreta.

O modificador abstract não pode ser conjugado com modificador private, visto porque um método abstrato deve ser implementado em uma subclasse e um método private não é visível a nenhuma outra classe.

## final

Um método final define que não pode ser sobreposto.

```
-- Car.java
package carros;
public class Car {
    public final String getColor() { // Nota 1
        return "red";
    }
}

class Mazda extends Car {
    public String getColor() { // Nota 2
        return "yellow";
    }
}
```

Anotações

141

# Programação Orientada a Objetos

---

**Nota 1:** Método definido com final, ou seja, não pode ser sobreposto.

**Nota 2:** Tentativa de sobrepor um método final - o que ocasiona um erro de compilação: `getColor() in carros.Mazda cannot override getColor()in carros.Car; overridden method is final`

## static

Um método estático define que esse pode ser executado sem que exista uma instância da classe - um objeto. Você só deve usá-lo se tiver certeza do que está fazendo.

```
-- Car.java
package carros;
public class Car {
    public static String getColor() { // Nota 1
        return "red";
    }
}

-- Oficina.java
package oficina;
import carros.*;
class Oficina {
    public static void main(String[] args) {
        System.out.println("a cor e "+Car.getColor()); // Nota 2
    }
}
```

**Nota 1:** Método definido com static, não há necessidade de um objeto para acessá-lo.

**Nota 2:** Note que não foi preciso criar um objeto para acessar o método getColor

## native

Define que a implementação do método foi escrita em uma linguagem nativa com por exemplo C ou C++.

---

### Anotações

142

# Programação Orientada a Objetos

---

## **strictfp**

Define que os valores de ponto flutuante do método devem seguir o padrão I33754, não se preocupe com esse padrão agora - saiba somente o que o modificar faz.

## **synchronized**

Faz com o método seja acessado por uma thread de cada vez, esse estudo será discutido no capítulo 9, ou seja, quando um método sincronizado estiver sendo executado por uma thread as demais deverão aguardar para iniciar a sua execução.

## **Modificadores para variáveis de instância**

Apesar de muitos acharem que atributos, variável de instância e propriedades são as mesmas coisas, gostaria de deixar bem claro que existem diferenças sutis, mas que não serão relevantes nesse estudo. Por convenção chamaremos de variável de instância por achar que é o termo mais correto que define esse conceito.

## **padrão**

Segue as mesmas regras de um método, ou seja, o acesso só é permitido a classes dentro do mesmo pacote.

```
-- Car.java
package carros;
public class Car {
    double preco; // Nota 1
}

-- Oficina.java
package carros;
class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        c.preco = 13990.00; // Nota 2
    }
}
```

**Anotações**

---

---

---

---

143

# Programação Orientada a Objetos

---

**Nota 1:** Não foi definido nenhum modificador, portanto assume o modificador padrão.

**Nota 2:** Acesso a variável de instância (VI) de uma classe dentro do mesmo pacote.

Observe agora:

```
-- Car.java
package carros;
public class Car {
    double preco; // Nota 1
}

-- Oficina.java
package oficina;
import carros.*;
class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        c.preco = 13990.00; // Nota 2
    }
}
```

**Nota 1:** Não foi definido nenhum modificador, portanto assume o modificador padrão.

**Nota 2:** Acesso não permitido para classes de pacotes diferentes. Erro de compilação: **preco is not public in carros.Car; cannot be accessed from outside package.**

## public

O acesso a uma VI pública não precisa falar muito, segue as mesmas regras de um método. Só uma dica ( sem nenhum ônus) cuidado ao definir um VI pública, pois com isso você estará liberando o acesso a todo o mundo, e isso pode enfraquecer sua classe, essa decisão deve ser bem estudada, e não podemos esquecer do encapsulamento. dentro do mesmo pacote.



# Programação Orientada a Objetos

---

```
-- Car.java
package carros;
public class Car {
    public double preco; // Nota 1
}

-- Oficina.java
package oficina;
import carros.*;
class Oficina {
    public static void main(String[] args) {
        Car c = new Car();
        c.preco = 13990.00; // Nota 2
    }
}
```

**Nota 1:** Variável definida como pública, todo mundo pode acessar.

**Nota 2:** Acesso a uma VI pública, compilação sem nenhum contratempo.

## private

Um variável `private` restringe o acesso somente a própria classe, é o modificador que mais limita o acesso a um membro de classe.

```
-- Car.java
package carros;
class Oficina {
    private String endereco; // Nota 1
}

public class Car {
    public static void main(String[] args) {
        Oficina o = new Oficina();
        System.out.println("o endereco é "+o.endereco); // Nota 2
    }
}
```

**Nota 1:** Variável definida como `private`, acesso somente dentro da própria classe.

**Nota 2:** Tentativa de acesso a uma membro privado, erro de compilação:  
endereco has private access in carros.Oficina

Anotações

145

# Programação Orientada a Objetos

---

O modificador `private` é usado para proteger uma classe do acesso direto a seus membros, com isso podemos garantir uma classe bem segura, e deve ser usado para implementação do conceito: encapsulamento.

## protected

Semelhante aos métodos protegidos, uma variável de instância com esse modificador, limita o acesso à subclasses para classes de outros pacotes.

```
-- Pessoa.java
package p1;
public class Pessoa {
    protected String nome = "Kuesley"; // nota 1
}

-- Diretor.java
package p2;
import p1.*;
public class Diretor extends Pessoa { // subclasse de Pessoa
    public String getNome() {
        return nome; // nota 2
    }
}
```

**Nota 1:** Membro protegido, pode ser acessador por uma subclasse, ou seja, herança.

**Nota 2:** Acesso ao membro herdado, apesar da classe `Diretor` estar em outro pacote, note que `nome` foi definido como `protected`.

Vamos mudar um pouco a classe `Diretor` (definida logo acima) para:

```
-- Diretor.java
package p2;
import p1.*;
public class Diretor { // classe sem herança
    public String getNome() {
        Pessoa p = new Pessoa();
        return p.nome; // nota 2
    }
}
```

Anotações

146

# Programação Orientada a Objetos

---

**Nota 1:** Se tentarmos compilar a classe Diretor o compilador mostrará a mensagem: **nome has protected access in p1.Pessoa**

## final

Uma variável de instância do tipo final é usada para armazenar valores constantes que não serão e nem podem ser alterados (caso alguém resolva cair em tentação) durante o escopo de utilização da classe.

Você deve estar se perguntando, porque usar variáveis que nunca serão alteradas ? E se no meio do contexto da classe eu precisar alterar o valor de uma variável final ? ESQUEÇA !! Uma vez inicializada uma variável com esse modificador nunca mais poderá ser alterada. Imagine que você tenha uma classe chamada Pessoa e resolva arbitrariamente definir uma altura e peso ideal - para qualquer pessoa os valores são idênticos e não podem ser alterados, pronto, taí uma utilização para uma variável final. Talvez não tenha sido muito feliz no exemplo, mas vamos vê-lo na prática.

```
-- Modelo.java
package p1;
public class Modelo {
    public final int ALTURA_IDEAL_CM = 175;
    public final int PESO_IDEAL_KG = 75;
}
```

Como os valores de ALTURA\_IDEAL\_CM e PESO\_IDEAL\_KG são finais, você poderá implementar métodos que usem esses valores para calcular por exemplo se uma pessoa, está ou não fora de forma, sem que algum engraçadinho fora de forma aumente o PESO\_IDEAL\_KG para 100 kilos. Qualquer tentativa de mudança de uma variável final resultará em alerta do compilador.

O código abaixo resultará em um erro de compilação:

```
package p2;
import p1.*;
public class Diretor {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        p.PESO_IDEAL_KG = 100;
    }
}
```

**Anotações**

147

# Programação Orientada a Objetos

---

Erro: cannot assign a value to final variable PESO\_IDEAL\_KG

Como estamos falando de uma variável de instância e já discutimos no capítulo 1 sobre variáveis que são inicializadas automaticamente, saiba que uma VI deve ser inicializada EXPLÍCITAMENTE pelo programador e não esperar que o compilador faça isso por você!

Você pode realizar essa inicialização de duas maneiras: a primeira junto com a declaração como em nosso exemplo anterior e a segundo é no método construtor!

Primeira Forma:

```
package p1;
public class Pessoa {
    public final int ALTURA_IDEAL_CM = 175;
    public final int PESO_IDEAL_KG = 75;
    public Pessoa() { ... } // método construtor
}
```

Segunda Forma:

```
package p1;
public class Pessoa {
    public final int ALTURA_IDEAL_CM;
    public final int PESO_IDEAL_KG;
    public Pessoa() {
        ALTURA_IDEAL_CM = 75; // definido no método construtor
        PESO_IDEAL_KG = 75;
    }
}
```

O compilador é paciente e aguarda que você inicialize até que o método construtor seja concluído, caso você tente esquecer ou tentado enganá-lo, aguarde as consequências.

```
package p1;
public class Pessoa {
    public final int ALTURA_IDEAL_CM;
    public final int PESO_IDEAL_KG;
    public Pessoa() {
        ALTURA_IDEAL_CM = 75; // Nota 1
    }
}
```

Anotações

148

# Programação Orientada a Objetos

---

**Nota 1:** Observe que somente o membro ALTURA\_IDEAL\_CM foi inicializado no construtor, e como o membro PESO\_IDEAL\_KG não foi inicializado explicitamente na declaração, o compilador mostrará a seguinte mensagem: **variable PESO\_IDEAL\_KG might not have been initialized**

*Portanto não conte com a idéia de inicialização automática mesmo que sejam tipos primitivos para variáveis com modificador final*

Uma interface também pode conter variáveis de instância, e mesmo que não sejam definida explicitamente como final, ele assim será!

-- FuncionarioPublico.java

```
package p1;
public interface FuncionarioPublico {
    int tempoEstabilidade = 2; // Nota 1
}
```

-- ProcuradorEstado.java

```
package p2;
import p1.*;
public class ProcuradorEstado implements FuncionarioPublico {
    public ProcuradorEstado() {
        tempoEstabilidade = 10; // Nota 2
    }
}
```

**Nota 1:** Membro de uma interface implicitamente é public e final, observe que não foi preciso definí-lo como public para se ter o acesso a outra classe mesmo estando em outro pacote!

**Nota 2:** Tentativa de alterar um membro de uma interface, como é final, o compilador mostrará a seguinte mensagem de erro: **cannot assign a value to final variable tempoEstabilidade**

# Programação Orientada a Objetos

---

## static

Esse modificador é muito simples de se entender! Você como é um cara esperto sabe a diferença de uma classe e um objeto certo ? (Se sua resposta for negativa, procure saber antes de prosseguir!) Uma variável de instância com esse modificador é compartilhada com todas as instâncias (objetos) dessa classe, ou seja, se você tiver uma classe com um atributo estático você não terá uma cópia desse atributo para cada objeto e sim uma única cópia para todos, portanto uma alteração do valor desse atributo em qualquer dos objetos, causará efeitos sobre esse atributos para todas as instância. Por isso, um membro estático é um membro da classe e não do objeto!

```
-- Pessoa.java
package p1;
public class Pessoa {
    public static int count = 0;           // Nota 1
    private String nome;                  // Nota 2

    public Pessoa(String n) {              // Nota 3
        nome = n;                         // Nota 4
        count += 1;                        // Nota 5
    }
    public static int getCount() {         // Nota 6
        return count;
    }
}

-- Clientes.java
package p2;
import p1.*;
public class Clientes {
    public static void main(String[] args) {
        System.out.println("Antes: "+Pessoa.getCount());
        Pessoa p1 = new Pessoa("Kuesley");
        Pessoa p2 = new Pessoa("Jose");
        System.out.println("Depois: "+Pessoa.getCount());
    }
}
```

Anotações

150

# Programação Orientada a Objetos

---

**Nota 1:** Observe que foi criada uma variável de instância `count` estático, ou seja, é único para todas as instâncias dessa classe!

**Nota 2:** A VI `nome` não é `static` portando, cada instância terá um nome distinto.

**Nota 3:** A classe `Pessoa` tem um modificador parametrizado!

**Nota 4:** Atribui à VI `'nome'` o valor informado na instanciação do objeto

**Nota 5:** Cada vez que o construtor é chamado, é acrescentado 1 à variável `count`

**Nota 6:** Método que retorna o valor da variável `count`. Talvez você esteja se perguntando mas porque esse método é `static` ? Imagine que você precise chamar o método `getCount` sem ter nenhuma instância de `Pessoa`, não há problema algum em nosso exemplo tirarmos o modificador `static` do método `getCount`, mas você não poderá chamá-lo sem haver pelo menos 1 instância ! Observe também que foi usado o nome da classe para chamar os métodos e não o objetos (`'Pessoa.getCount()'`), apesar de que não haveria problema pois a JVM saberia que se trata de um método `static` e se reportaria a classe para saber as informações sobre a variável `count`.

Observe a definição das seguintes classes:

```
-- Pessoa.java
package p1;
public class Pessoa {
    public static int count = 0;
    private String nome;

    public Pessoa(String n) {
        nome = n;
        count += 1;
    }
    public int getCount() {                // Removido o modificador static
        return count;
    }
}

-- Clientes.java
package p2;
import p1.*;
public class Clientes {
    public static void main(String[] args) {
        Pessoa p1 = new Pessoa("Kuesley");
        Pessoa p2 = new Pessoa("Jose");
        System.out.println("Depois p1: "+p1.getCount()); // Nota 1
```

Anotações

151

# Programação Orientada a Objetos

---

```
System.out.println("Depois p2: "+p2.getCount()); // Nota 2
System.out.println("Depois Pessoa: "+Pessoa.getCount()); //
                                                    Nota 3
    }
}
// Nota 1/2: Ambos retornarão 2 no método getCount()
// Nota 3: Erro de compilação: non-static method getCount() cannot be
referenced from a static context
```

OBSERVAÇÃO: Você NUNCA poderá referenciar de um contexto estático, uma variável ou método não estático! Apesar de que isso já foi falado anteriormente, no estudo dos métodos estáticos, mas sempre é bom lembrarmos, pois os caras das provas de certificação gostam de pegar no nosso pé quando se trata de contexto estático!

Observe:

```
package p1;
public class Clientes {
    int count = 0;
    public static void main(String[] args) {
        System.out.println(""+count);
    }
}
```

**Erro: non-static variable count cannot be referenced from a static context**

Se você quiser enganar o compilador da seguinte forma:

```
package p1;
public class Clientes {
    int count = 0;
    public void imprimirLista() {
        System.out.println(""+count);
    }
    public static void main(String[] args) {
        imprimirLista(); // Nota 1
    }
}
```

Nota 1: O compilador saberá que está tentando ser enganado e xingará: **Erro: non-static variable count cannot be referenced from a static context**

Anotações

152



## transient

Esse modificador indica a JVM para não esquentar a cabeça com as variáveis transient quando for realizar a serialização de um objeto! Calma! Calma! Calma! Se você não sabe o que é serializar um objeto, não se preocupe, por enquanto a Sun não está exigindo que você tenha esse conhecimento - apesar de ser um dos recursos mais brilhantes da linguagem Java, e é usado para a programação em ambientes distribuídos! Por isso, apresse seus estudos!

Mesmo assim, gostaria que você tivesse uma idéia do que é serializar um objeto, pois assim acredito que fique mais fácil a compreensão desse modificador.

Imagine que você esteja desenvolvendo uma aplicação em um ambiente de rede e precise que um objeto Pessoa seja enviado para uma outra máquina da rede, e lá continue o processo com esse objeto, esse processo é conhecido como serialização, ou seja, transformar um objeto em uma sequência de bytes e enviá-lo para outra máquina (consequentemente, outra JVM) que deserializará essa sequência de bytes, obtendo o objeto como na máquina de origem! Quem disse que não há milagres na informática!!!

```
public class Pessoa {  
    public String nome;  
    public String endereco;  
    transient public Image Foto; // Nota 1  
}
```

**Nota 1:** Imagine que sua conexão de rede seja muito ruim! E você não precise ficar transportando a foto da pessoa na rede, com o modificador transient o membro Foto não será serializado!

## volatile

Quando uma variável de instância com esse modificador é alterada, a thread deverá sincronizar sua cópia com a cópia principal.

Uma analogia a esse modificador é o caso dos programadores que quando alteram um projeto devem sempre alterar a documentação para que todos os demais membros da equipe estejam atualizados!

## Programação Orientada a Objetos

---

Para encerrarmos o estudo sobre modificadores, veja abaixo uma tabela que mostra os modificadores e onde podem ser usados:

modificador	classe	método	var instância	var local
padrão	sim	sim	sim	não
public	sim	sim	sim	não
protected	não	sim	sim	não
private	não	sim	sim	não
static	não	sim	sim	não
final	sim	sim	sim	sim
abstract	sim	sim	não	não
Strictfp	sim	sim	não	não
synchronized	não	sim	não	não
Transient	não	não	sim	não
native	não	sim	não	não
volatile	não	não	sim	não

Anotações

154

## Implementação de Interface

Você terá que saber algumas regras sobre interface para se certificar, e deverá saber de "cor e salteado" (nunca tinha citado isso) como funciona a interface Runnable ! Uma interface é uma espécie de contrato que uma classe deve fazer e cumprir para que todos fiquem felizes! Veja como é a implementação de uma interface:

```
public interface FuncoesPublicas {  
    void assinarPonto();  
    void executarResponsabilidade();  
}
```

Uma interface nada mais é que uma classe totalmente abstrata, ou seja, só existe as assinaturas de métodos!

Apesar de não terem sido definidos como abstract e public, convencionalmente os métodos de uma interface o são. Portanto o compilador os enxerga:

```
public interface FuncoesPublicas {  
    public abstract void assinarPonto();  
    public abstract void executarResponsabilidade();  
}
```

Apesar de não existir herança múltipla em Java para classe em Java, as interfaces podem herdar de múltiplas interface, veja o código a seguir:

```
public interface FuncaoAnimal {  
    void nascer();  
}  
  
interface FuncaoAve extends FuncaoAnimal {  
    void voar();  
}  
  
interface FuncaoReptil extends FuncaoAnimal {  
    void rastejar();  
}  
  
interface FuncaoCobraVoadora extends FuncaoAve, FuncaoReptil { // Nota1  
}
```

# Programação Orientada a Objetos

---

**Nota 1:** Um interface pode herdar mais de uma interface! Não pense em fazer isso com uma classe!

Apesar de que uma classe pode implementar várias interfaces:

```
class CobraViva implements FuncaoAnimal, Runnable { ... }
```

Você deve saber algumas regras quanto á declaração de constante de interface:

```
public interface FuncoesPublicas {  
    int tempoServico = 0;  
}
```

Apesar de não estarem explicitamente definido mas a variável tempoServico é:

```
public  
final  
static
```

## CONSIDERAÇÕES FINAIS

### Modificadores de acesso

Os modificadores de acesso: (padrão, public, private e protected) nunca poderão ser combinados !

### Métodos

- Nunca poderá ser definido como (transient, volative);
- Um método nunca poderá ser abstract e final;
- Um método nunca poderá ser abstract e strictfp;
- Um método nunca poderá ser abstract e native;
- Um método nunca poderá ser abstract e synchronized;
- Um método nunca poderá ser abstract e private;
- Um método final nunca poderá ser sobreposto;
- Um método abstrato nunca poderá ser implementado;
- Se um método for abstrato a classe também será.

Anotações

156

# Programação Orientada a Objetos

---

## Variável de Instância

- Pode ter qualquer um dos quatro modificadores de acesso;
- Podem ser (volatile, transient, final, static);
- Não pode ser (abstract, native, synchronized, strictfp) .

## Interfaces

- Uma interface nunca herda uma classe;
- Uma classe nunca herda uma interface;
- Uma interface nunca implementa outra interface;
- Uma classe nunca implementa outra classe;
- Uma interface pode herdar várias interfaces;
- Uma variável de uma interface sempre será implicitamente: (public, final, static);
- Um método de uma interface sempre será (public, abstract).

Lembre-se da Interface: java.lang Runnable

Ela só tem um método: public void run();

## Resumos para certificação - Modificadores e Controle de Acesso

### Modificadores de Acesso a Classe

- Há três tipos de modificadores: public, protected, private;
- Há quatro níveis de acesso: public, protected, default, private;
- As classes podem ser públicas ou padrão;
- A visibilidade das classes gira em torno de se o código de uma classe pode:
  - ❑ Criar uma instância da classes;
  - ❑ Estender (ou criar subclasses) outra classe;
  - ❑ Acessar métodos e variáveis de outra classe;
- Uma classe com acesso padrão só pode ser detectada por classes do mesmo pacote;
- Uma classe com acesso público pode ser detectada por classes de todos os pacotes.

### Modificadores de classes (não referentes a acesso)

- As classes também podem ser alteradas com final, abstract ou strictfp;
- Uma classe não pode ser final e abstract;
- Uma classe final não pode ter subclasses;
- Uma classe abstrata não pode ser instanciada;
- Uma classe com um método abstrato significa que a classe inteira deve ser abstrata;
- A primeira classe concreta a estender uma classe abstrata terá que implementar todos os métodos abstratos;

### Modificadores de acesso a membros

- Os métodos e as variáveis de instância (não locais) são conhecidos como membros;
- Os membros podem usar todos os quatro níveis de acesso: public, protected, default e private;
- O acesso aos membros se dá de duas formas:
  - ❑ O código de uma classe só pode acessar um membro de outra classe;
  - ❑ Uma subclasse pode herdar membros de sua superclasse;
- Se uma classe não puder ser acessar, seus membros também não poderão;

Anotações

158

# Programação Orientada a Objetos

---

- A visibilidade da classe deve ser determinada antes da dos membros;
- Os membros públicos podem ser acessados por todas as outras classes, mesmo em pacotes diferentes;
- Se um membro da superclasse for público, a subclasse herdará – independente do pacote;
- Os membros acessados sem o ponto(.) têm que pertencer à mesma classe ou as superclasses;
- `this.aMethod()` é o mesmo que simplesmente chamar o método `aMethod()`;
- Os membros privados só podem ser acessados por um código da mesma classe;
- Os membros privados não ficam visíveis para a subclasses, portanto não podem ser herdados;
- Os membros padrão e protegido só diferem quando subclasses estão envolvidas;
  - Os membros padrão só podem ser acessados por outra classe do mesmo pacote;
  - Os membros protegidos podem ser acessados por outras classes do mesmo pacote, além de por subclasses, independentes do pacote;
- Protegido = pacote + filho (filhos significando subclasses);
- Por subclasses externas ao pacote, o membro protegido só pode ser acessado através da herança: uma subclasse externa ao pacote não pode acessar membro protegido usando a referência a uma instância de superclasse(em outras palavras, a herança é o único mecanismo para uma subclasse externa ao pacote acessar um membro protegido de sua superclasse);
- Um membro protegido herdado por uma subclasse de outro pacote não pode ser acessado por nenhuma outra classe do pacote da subclasse, exceto pelas próprias subclasses dessa).

## Variáveis Locais

- As declarações de variáveis locais(de método, automáticas, de pilha) não podem ter modificadores de acesso;
- `final` é o único modificador disponível para variáveis locais;
- As variáveis locais não recebem valores padrão, portanto, devem ser inicializadas antes do uso.

## Outros Modificadores - membros

- Os métodos finais não podem ser sobrepostos em uma subclasse;
- Os métodos abstratos foram declarados, com uma assinatura e tipo de retorno, mas não foram implementados;

Anotações

159

# Programação Orientada a Objetos

---

- Os métodos abstratos terminam com um ponto e vírgula e não com chaves;
- Há três maneiras de identificar um método não abstrato:
  - ❑ O método não é marcado como abstrato;
  - ❑ O método possui chaves;
  - ❑ O método possui um código entre as chaves.
- A primeira classe não abstrata(concreta) a estender uma classe abstrata deve implementar todos os métodos abstratos dessa;
- Os métodos abstratos devem ser implementados por uma subclasse, portanto, têm que poder ser herdados. Por essa razão:
  - ❑ Os métodos abstratos não podem ser privados;
  - ❑ Os métodos abstratos não podem ser finais;
- O modificador synchronized só é aplicado a métodos;
- Os métodos sincronizados podem ter qualquer controle de acesso e também serem marcados como final.
- Os métodos sincronizados não podem ser abstratos;
- Outros Modificadores - membros
- O modificador strictfp só é aplicado a classes e métodos;
- O modificador native só é aplicado a métodos;
- As variáveis de instância podem:
  - ❑ Ter qualquer tipo de acesso;
  - ❑ Serem marcadas como final ou transient.
- As variáveis de instância não podem ser declaradas como abstract, synchronized, native ou strictfp.
- É válido declarar uma variável local como o mesmo nome da variável de instância; isso é chamado de sombreamento;
- As variáveis finais apresentam as seguintes propriedades:
  - ❑ Não podem ser reinicializadas, uma vez que tiverem um valor atribuído;
  - ❑ As variáveis de referência finais não podem referenciar um objeto diferente se já tiverem um objeto atribuído a elas.
- As variáveis de referência finais devem ser inicializadas antes que a execução do construtor seja concluída;
- Não existem objetos finais. Uma referência a objeto marcada como final não significa que o objeto propriamente dito seja inalterável;
- O modificador transient e volatile só pode ser aplicado a variáveis de instância;

## Variáveis e métodos estáticos

- Não são associados a nenhuma instância específica da classe;
- Não é necessária a existência da instância da classe para que membros estáticos dessas sejam usados;

Anotações

160



# Programação Orientada a Objetos

---

- Só haverá uma cópia da variável estática por classe e todas as instâncias a compartilharão;
- As variáveis estáticas recebem o mesmos valores padrão das variáveis de instância;
- Um método estático como o `main()` não pode acessar uma variável não estática (de instância);
- Os membros estáticos são acessados através do nome da classe: `ClassName.metodo()`;
- Os métodos estáticos não podem ser sobrepostos;

## Regras de declaração

- Um arquivo de código fonte só pode ter uma classe pública;
- Se o arquivo de código fonte tiver uma classe pública, seu nome deverá coincidir com o nome dessa classe;
- O arquivo só pode ter uma instrução de pacote, porém várias de importação;
- A instrução de pacote (se houver) deve ficar na primeira linha do arquivo de código fonte;
- A instrução de importação (se houver) deve vir depois do pacote e antes da declaração de classe;
- Se não houver declaração de pacote, as instruções de importação terão que ser as primeiras do arquivo de código fonte;
- As instruções de pacote e de importação são aplicadas a todas as classes do arquivo;
- O arquivo pode ter mais de uma classe não pública;
- Os arquivos que não tiverem classes públicas não apresentam restrições de nomeação;
- As classes poderão ser listadas em qualquer ordem no arquivo;
- As instruções de importação só fornecem um atalho na digitação do nome totalmente qualificado da classe;
- As instruções de importação não causam impacto no desempenho e não aumentarão o tamanho do seu código fonte;
- Se você usar uma classe de um pacote diferente, mas não importá-la, terá que empregar seu nome totalmente qualificado em todos os locais onde for utilizar no código;
- As instruções de importação podem coexistir com nomes totalmente qualificados de classes em um arquivo de código fonte;
- As instruções de importação que terminarem com `.*`; importaram todas as classes do pacote;

Anotações

161

# Programação Orientada a Objetos

---

- As instruções de importação que terminarem com ; importaram uma única classe;
- Você precisa usar nomes totalmente qualificados quando tiver classes diferentes de pacotes distintos, com o mesmo nome de classe, uma instrução de importação não será explícita o suficiente.

## Propriedades de main( )

- Deve ser marcado com static;
- Deve ser marcado como void;
- Deve ter um único argumento de array String, o nome do argumento é flexível, mas a convenção é args;
- Para fins do exame, adote a asserção de que o método main( ) deve ser público;
- Declarações inapropriadas do método main( ) ou a falta do método main( ) causará erro de tempo de execução e não na compilação;
- Na declaração de main( ), a ordem de public e static pode ser alterada, e args pode ser renomeado;
- Outros métodos de sobreposição chamados main( ) podem existir e ser válidos na classe, mas se nenhum deles tiver a assinatura do método main() principal, então, o JVM poderá usar essa classe para iniciar a execução de seu aplicativo.

## Java.lang.Runnable

- Você precisa conhecer a interface java.lang.Runnable de cor, ela só tem um método a ser implementado: public void run{ }.

## Implementando Interfaces

- As interfaces são contratos que definem o que a classe poderá fazer, mas não dizem nada sobre a maneira pela qual ela deverá fazê-lo;
- As interfaces podem ser implementadas por qualquer classe, de qualquer árvore de herança;
- A interface é como a classe 100% abstrata, e será implicitamente abstrata caso você digite ou não o modificador abstract na declaração;
- Uma interface só pode ter métodos abstratos, nenhum método concreto é permitido;
- As interfaces são por padrão públicas e abstratas – a declaração explícita desses modificadores é opcional;

Anotações

162

# Programação Orientada a Objetos

---

- As interfaces podem ter constantes, que são sempre implicitamente public, static e final;
- As declarações da constante de interface como public, static e final são opcionais em qualquer combinação;
- Uma classe de implementação não abstrata válida terá as propriedades a seguir:
  - ❑ Fornecerá implementações concretas de todos os métodos da interface;
  - ❑ Deve seguir todas as regras de sobreposição válidas para os métodos que implementa;
  - ❑ Não deve declarar nenhuma exceção nova do método de implementação;
  - ❑ Não deve declarar nenhuma exceção que seja mais abrangente do que declaradas no método de interface;
  - ❑ Pode declarar exceções de tempo de execução em qualquer implementação de métodos de interface, independente do que constar na declaração da interface;
  - ❑ Deve manter a assinatura e o tipo de retorno exatos dos métodos que implementa (mas não precisa declarar as exceções da interface).
- Uma classe que tiver implementando interface pode ela ser abstrata;
- A classe só pode estender uma classe (sem herança múltipla), porém pode implementar várias;
- As interfaces podem estender uma ou mais interfaces;
- As interfaces não podem estender uma classe ou implementar uma classe ou interface;
- Quando fizer o exame, verifique se as declarações de interface e classe são válidas antes de verificar outras lógicas do código.

# Programação Orientada a Objetos

---

## Capítulo III - Operadores e atribuições

### Introdução

A compreensão desse capítulo é muito importante pois trata de um assunto essencial em qualquer linguagem de programação, as diversas formas de se atribuir e manipular valores de uma variável. Além de ser um dos elementos fundamentais de uma linguagem de programação, as variáveis tem um papel importante na geração de programas e suas peculiaridades serão abordadas nesse capítulo (pelo menos é o que esperamos) !

A forma mais elementar de se atribuir um valor a uma variável é:

```
x = 0;
```

Mas não espere que no exame, haja perguntas com esse desprezo de complexidade, muito pelo contrário, o que normalmente você não utiliza ou nunca utilizou com certeza estarão lá (naquela bendita prova)!!!

```
byte x = 10;
```

Lembra-se do capítulo 1 ? Que na linha anterior 10 é um valor literal *int*, composto por 32 bits e que um tipo *byte* só tem capacidade para armazenar 8 bits. Você deve estar se perguntando e o que acontece com o resto dos bits ? Calma, para todas essas perguntas em algum lugar no universo haverá uma resposta. Nesse caso o compilador é generoso e executa a conversão implicitamente para você. Portanto uma variável nada mais é do que um repositório de bits para a representação de algum valor. O que está armazenado na variável x ? Se você respondeu 00001010 está no caminho certo para o sucesso de sua certificação ! Como o compilador realizou a conversão de *int* para *byte*, os 24 bits de diferença foram desprezados.

```
byte x = 19; // ok
byte y = x;  // ok
byte z = x + y; // deveria aceitar, mas necessita de uma conversão explícita.
```

Toda operação envolvendo dois tipos inteiros sejam eles (byte e short, long e byte) retornaram um tipo int .

---

### Anotações

164

# Programação Orientada a Objetos

---

Falando em Objetos:

```
Button b = new Button();
```

O que b armazena ? Se você pensou em dizer que b armazena um objeto do tipo Button, sua resposta está errada ! b armazena um conjunto de bits usados para acessar (referenciar) o objeto na memória. Portanto, podemos dizer a rigor que b não é um objeto do tipo Button e sim uma forma (através de uma sequência de bits) usada para acessar esse objeto na memória.

Falando um pouco sobre atribuições de número primitivos

Atribuição	Observação
byte a = 19;	byte a = (byte)19;
byte b = 128;	// Não é possível sem uma conversão explícita
byte c = (byte)128;	// Ok

O valor 128 não pode ser atribuído à variável b por ultrapassar a capacidade em bits do tipo byte - uma conversão se faz necessária. Mas devemos ressaltar um acontecimento muito interessante nesse caso. Qual o valor da variável c da atribuição anterior ?

Entenda o que acontece:

```
0000 0000 0000 0000 0000 0000 1000 0000  <-->  +128
```

Como o compilador despreza os primeiros 24 bits (pois o tipo byte só comporta 8 bits) ficamos com:

```
1000 0000
```

Se você é um cara esperto sabe que o bits da extrema esquerda deverá ser usado para armazenar o sinal (nesse caso negativo), como temos um número negativo agora desejamos saber qual é ! Existe uma regra que você deverá se familiarizar que é usada para descobrir o valor de um número negativo em bits:

---

**Anotações**

165

# Programação Orientada a Objetos

---

Primeiro inverte todos os bits do número, assim temos:

0111 1111

Transforme esse número em decimal e acrescente 1

$127 + 1 = 128$

Aplique o bit do sinal e terá o valor: -128 !

Exercite isso com o número 140, 147 !

Você deverá obter: -116, -109

## Atribuição de Tipos Primitivos

Atribuições elementares não precisam ser discutidas, pois você já deve estar familiarizado, agora um regrinha deve ser lembrada para sempre: toda atribuição que é resultada de uma operação (seja ela qual for: soma, divisão, deslocamento de bits....) envolvendo tipos inteiros (byte, short, int, long) SEMPRE RESULTARÁ EM UM TIPO INT !!!

### Portanto

```
byte x = 1;           // ok
byte y = 2 + 3;       // ok
byte z = x + y;        // deveria aceitar pois 6 cabe perfeitamente em um tipo byte,
                        porém é necessário uma conversão explícita
byte z = (byte)x + y; // agora sim
```

## Deslocamento de Bits

Deslocar bits em número e alterar sua sequência de armazenamento e só poderá ser feito em TIPOS INTEIROS !!

```
>>   deslocamento de bits à direita com sinal
<<   deslocamento de bits à esquerda com sinal
>>>  deslocamento de bits à direita sem sinal
```

### Anotações

166

# Programação Orientada a Objetos

---

Deslocamento a direita com sinal ( >> )

```
int x = 16;  
x = x >> 2;
```

A variável x estará armazenando ao final da linha de código anterior o valor 4 ! O que o código anterior faz é simplesmente descolar dois bits para a direita, assim temos:

x antes do deslocamento

0000 0000 0000 0000 0000 0000 0001 0000 (base 2)  
16 (base 10)

x depois do deslocamento

0000 0000 0000 0000 0000 0000 0000 0100 (base 2)  
4 (base 10)

O sinal sempre será mantido com esse operador. Uma regra bem simples é que quando se desloca à direita é o mesmo que aplicar a seguinte regra matemática:

Para o caso anterior:

Fórmula: 16 dividido por 2 elevado a x (onde x é a quantidade de bits a deslocar)

Deslocamento a esquerda com sinal ( << )

```
int x = 16;  
x = x << 2;
```

x antes do deslocamento

0000 0000 0000 0000 0000 0000 0001 0000 (base 2)  
16 (base 10)

x depois do deslocamento

0000 0000 0000 0000 0000 0000 0100 0000 (base 2)  
64 (base 10)

---

**Anotações**

167

# Programação Orientada a Objetos

---

Outra regrinha que você já deve ter deduzido é:

Fórmula: 16 multiplicado por 2 elevado a x (onde x é a quantidade de bits a deslocar)

Observe o comportamento em caso de número negativos:

```
int x = -200;  
x <<= 3;
```

O resultado de x será: -1600

Note que o sinal é mantido, observe o deslocamento dos bits:

1000 0000 0000 0000 0000 0000 1100 1000 equivale a -200

Após o deslocamento:

1000 0000 0000 0000 0000 0110 0100 0000 equivale a -1600

Deslocamento a direita sem sinal ( >>> )

Nesse caso o bit do sinal não é mantido, sendo preenchido com zero no bit da extrema esquerda, portanto exceto em um caso particular que logo logo explicaremos, todo numero deslocado com esse sinal (mesmo que seja um número negativo) fica positivo.

```
int x = 200;  
x >>>= 2;
```

O valor de x será 50 após o deslocamento, veja porque:

0000 0000 0000 0000 0000 0000 1100 1000 equivale a 200

Após o deslocamento:

0000 0000 0000 0000 0000 0000 0011 0010 equivale a 50

---

**Anotações**

168



# Programação Orientada a Objetos

---

Agora vamos examinar um número negativo:

```
int x = -200;  
x >>>= 2;
```

O valor de x será após o deslocamento, veja porque:

1000 0000 0000 0000 0000 0000 1100 1000    equivale a -200

Após o deslocamento:

0010 0000 0000 0000 0000 0000 0011 0010    equivale a 1073741774

Existe uma particularidade que você deve estar atento no caso de quando for tentado a deslocar uma quantidade superior a capacidade do tamanho. Por exemplo, o que aconteceria se você tentar deslocar 35 bits em uma variável do tipo `int` que comporta 32 ? Ou até mesmo 9 em uma variável `byte` ? Se isso acontecer o compilador fará uma cálculo nuclear para descobrir como deslocar:

```
int x = 90 >> 35;
```

Veja que no exemplo anterior, há um tentativa de deslocar 35 bits em um tipo `int` ! O compilador fará uma divisão entre a 35 quantidade de bits a deslocar e 32 que é a capacidade do tipo em questão e o resto dessa divisão será a quantidade deslocada, nesse caso: 3 !!

```
byte y = (byte) 8 >> 12;
```

12 % 8 = 4 (quatro bits a deslocar)

Isso nos levanta um caso particular. E se for um múltiplo de 8 (no caso do tipo `byte`), por exemplo, tente deslocar 16 ou 32 em um tipo `byte`, quantos bits serão deslocados ? Exatamente nenhum pois o resto da divisão será 0 (nenhum bit) será deslocado. Portanto nem sempre o deslocamento de bits com o operador ( `>>>` ) será positivo, ou seja, tente deslocar 32 bits ou 64 bits em um tipo `int` que armazena o valor -300 e terá o mesmo valor -300 !! Talvez você não esteja lembrado, mas um número se diz múltiplo de outro se o resto entre sua divisão for 0, veja o exemplo:

---

**Anotações**

169

# Programação Orientada a Objetos

---

64 é múltiplo de 32 pois,  $64 / 32 = 2$  e resta 0

32 é múltiplo de 32 pois,  $32 / 32 = 1$  e resta 0 (evidentemente)

Não se irrite, foi só pra lembrar!!!

Portanto nem sempre o deslocamento com o operador (  $\gg$  ) resulta em um número positivo !!!

## Atribuição de Objetos

Vamos trabalhar com as seguintes classes:

```
class Carro {  
    public double preco = 0;  
}
```

```
class Audi extends Carro {  
}
```

```
1.  public class Teste {  
2.      public static void main(String[] args) {  
3.          Carro a = new Carro();  
4.          Audi b = new Audi();  
5.          Carro c = new Audi();  
6.          Audi d = new Carro();  
7.      }  
8.  }
```

Qual das linhas anteriores estão incorretas e não permitiriam que o programa fosse compilado ?

- a) 1, 3
- b) 2, 3
- c) 5
- d) 6
- e) o código será compilado

Não se esqueça da regra: X é membro de Y ??

**Anotações**

170

# Programação Orientada a Objetos

---

## Passagem de Parâmetro em Java é por Valor

Muito discussão existe quando o assunto é a forma como a Java passa um variável (primitiva ou referência) para um método, mas acalme-se estamos aqui para descomplicar sua vida e lhe dar um nova oportunidade!!

Quando uma variável é passada para um método, SEMPRE SEMPRE E SEMPRE será passado um cópia dos seus bits!!

### Variável Primitiva

```
public class Teste {  
    public static void main(String[] args) {  
        byte x = 10;  
        System.out.println("X antes: "+x);  
        altera(x);  
        System.out.println("X depois: "+x);  
    }  
    public static void altera(byte a) {  
        a = 2;  
    }  
}
```

No caso anterior o que é passado para o método *altera* é: 00001010 que nada mais é que o valor decimal 10, porém observe que no escopo do método *altera* o valor recebido é alterado e quando o fluxo retorna ao método *main* o valor da variável *x* está intacta. Isso é fácil de entender, você deve pensar erroneamente que se o método *altera* modificasse o valor de *x*, então *x* teria sido passado como referência ! Essa forma não é correta de se pensar, pense simplesmente que o método *altera* recebeu um cópia dos bits da variável *x* que era: 00001010.

Portanto o resultado do código acima é:

X antes: 10  
X depois: 10

---

### Anotações

171

# Programação Orientada a Objetos

---

## Variável Referência

Agora o bicho pega!

Uma variável de referência nada mais é que um repositório de bits que representam um forma de se acessar um objeto na memória. A variável é uma coisa o objeto que está na memória é outra, tente colocar isso na sua cabeça!

```
public class Teste {  
    public static void main(String[] args) {  
        Carro c = new Carro();  
        c.preco = 13990.00;  
        System.out.println("preço antes: "+c.preco);  
        anula(c);  
        System.out.println("preço depois: "+c.preco);  
    }  
    public static void anula(Carro p) {  
        p = null;  
    }  
}
```

### Resultado

preço antes: 13990.0

preço depois: 13990.0

No trecho de código anterior acontece um fenômeno muito interessante e que nos dá a idéia de como funciona toda essa parafernália:

### Premissas

- c é um variável de referência a um objeto do tipo Carro;
- o método anula, recebe uma variável p do tipo Carro;
- o método anula, diz para a variável p não referenciar nada (null);
- se a variável c fosse passada como referência, quando o fluxo retornasse ao método main, c estaria referenciando null;
- se houver uma alteração do objeto que a variável p está referenciando, essa alteração também será visível à variável c (quando o fluxo retornar é claro) veja o trecho a seguir:

### Anotações

---

---

---

---

172

# Programação Orientada a Objetos

---

```
public class Teste {  
    public static void main(String[] args) {  
        Carro c = new Carro();  
        c.preco = 13990.00;  
        System.out.println("preço antes: "+c.preco);  
        aumento(c);  
        System.out.println("preço depois: "+c.preco);  
    }  
    public static void aumento(Carro p) {  
        p.preco *= 1.05;  
    }  
}
```

## Resultado

preço antes: 13990.0  
preço depois: 14689.5

Talvez você esteja se perguntando porque é então que o valor foi alterado se em Java um variável é passada como valor ??? ERRADO, o que é passado como valor são os bits que referenciam o objeto, portanto p no método *aumento* referencia exatamente o mesmo objeto que c referencia no método main, ou você acha que a Java sai criando objetos na memória assim sem mais nem menos ?

## Operadores Bit a Bit

Existem três operadores bit a bit:

&      e  
|      ou inclusivo  
^      u exclusivo

# Programação Orientada a Objetos

---

## Aplicando o operador &

O operador & compara dois bits e será um se ambos o forem ! Calma que voce verá um exemplo :

```
int x = 10 & 9;
```

Convertendo para bits os números 10 e nove temos:

```
1010
1001
-----
1000
```

O Resultado em decimal é 8 !

## Aplicando o operador | ou inclusivo

```
int x = 10 | 9;
```

```
1010
1001
-----
1011
```

Resultado: 11

### 3.5.3 - Aplicando o operador ^ ou exclusivo

```
int x = 10 ^ 9;
```

```
1010
1001
-----
0011
```

Resultado: 3

---

**Anotações**

174

# Programação Orientada a Objetos

---

## Operador Bit Complementar

Esse operador é unário, ou seja, envolve somente uma variável e deve ser usada quando se deseja inverter todos os bits de um número use o operador ( ~ )

```
int x = 8;           // 1000
x = ~x;             // Passa a valer 0111
```

## Operador Condicional

O operador condicional é um operador ternário, ou seja, três operandos (óbvio) e deve ser usado quando se necessita realizar uma condição em uma única linha. Veja o código abaixo:

```
int tamanho = 19;
String texto = (tamanho >= 10) ? "Maior ou igual a 10" : "Menor que 10";
System.out.println(texto);
```

Não precisa nem perguntar o que vai sair na tela que todos já sabem !!!  
Mas para os que ainda ficaram em dúvida, lá vai: o primeiro operando é a condição, ou seja, se ela for verdadeira (*true*) atribui o valor da segunda à variável *texto* em sendo falsa (*false*) atribui o valor da terceira! Mais um exemplo:

```
class Teste {
    public static void main(String[] args) {
        int x = 10 ^ 8;
        int y = ~x;
        System.out.println(( x > y ) ? "x é maior que y" : "x é menor que y");
    }
}
```

Não tente fazer algo do tipo:

```
( x > y ) ? System.out.println("x é maior que y") : System.out.println("x é menor que y");
```

Um operador condicional só pode ser usado para atribuir valores, ou seja, para ser usado quando se necessita fazer uma atribuição condicional....

# Programação Orientada a Objetos

---

Mas algo assim é perfeitamente possível:

```
public class Teste {  
    public static void main(String[] args) {  
        int x = 10 ^ 8;  
        int y = ~x;  
        If ( ((x>y)?x:y) > 10 ) {  
            System.out.println("x é maior que b e é maior que 10");  
        } else {  
            System.out.println("x é maior que b mas não é maior que 10");  
        }  
    }  
}
```

O que o trecho de código irá fazer ?

- a) imprimirá " x é maior que b e é maior que 10"
- b) imprimirá "x é maior que b mas não é maior que 10"
- c) Não imprimirá nenhum resultado
- d) Erro de compilação
- e) Uma exceção será lançada

## Operador instanceof

Operador binário (dois operandos) que é utilizado para saber se um objeto é instância de uma classe.

```
String nome = "kuesley";  
if (nome instanceof String) {  
    System.out.println("nome é do tipo String");  
} else {  
    System.out.println("nome não é do tipo String");  
}
```

---

**Anotações**

176



# Programação Orientada a Objetos

---

Você também poderá compara subclasse veja o exemplo a seguir:

```
public class Veiculo { }

public class Audi extends Veiculo { }

public class Teste {
    public static void main(String[] args) {
        Audi a = new Audi();
        Veiculo v = new Veiculo();
        if (a instanceof Audi) System.out.println("a é do tipo Audi");
        if (v instanceof Veiculo) System.out.println("v é do tipo Veiculo");
        if (v instanceof Audi) System.out.println("v é do tipo Audi");
        if (a instanceof Veiculo) System.out.println("a é do tipo Veículo");
    }
}
```

Nesse código será exibido:

```
a é do tipo Audi      // a que é instância da classe Audi é membro de Audi (isso é
óbvio)
v é do tipo Veiculo   // v que é instância da classe Veiculo é membro de Veiculo
(óbvio)
a é do tipo Veiculo   // a que é uma instância da classe Audi é membro de veículo,
pois a classe Audi é uma sub-classe de Veiculo!
```

Isso porque aplica-se sempre aquela regra: x É MEMBRO y ???

Esse operador também pode ser aplicado as interfaces, ou seja, uma classe é membro de uma *interface* sempre que ela a implementa.

```
public interface Acao { }
public class Veiculo implements Acao { }
public class Audi extends Veiculo { }
public class Moto { }
```

# Programação Orientada a Objetos

---

```
public class Teste {  
    public static void main(String[] args) {  
        Audi a = new Audi();  
        Veiculo v = new Veiculo();  
        Moto m = new Moto();  
        If (a instanceof Acao) System.out.println("a é membro de Acao");  
        if (v instanceof Acao) System.out.println("v é membro de Acao");  
        if (m instanceof Acao) System.out.println("m é membro de Acao");  
    }  
}
```

Resultado do código anterior:

a é membro de Acao  
v é membro de Acao

Lembra-se do primeiro capítulo quando estudamos os arrays, e foi dito que um array é um objeto, agora comprovaremos essa afirmação para que não fique dúvida em sua cabeça.

```
public class Teste {  
    public static void main(String[] args) {  
        boolean c = (new int[] { 1,2,3 } instanceof Object);  
        System.out.println(c);  
    }  
}
```

O resultado será true.

Foi criado um array anônimo (estudado no capítulo 1) e verificado se é um tipo Object e atribui-se a variável *c* o resultado *true*

Importante: Você poderá se deparar com questões que abordam se um objeto é de um tipo de classe que esteja fora da hierarquia, por exemplo:

```
class Veiculo { }  
class Audi extends Veiculo { }
```

---

**Anotações**

178

# Programação Orientada a Objetos

---

```
public class Teste {
    public static void main(String[] args) {
        Audi a = new Audi();
        Veiculo v = new Veiculo();
        boolean b1 = (a instanceof Audi); // ok - b1 é true
        boolean b2 = (a instanceof Veiculo); // ok - b2 é true
        boolean b3 = (v instanceof Audi); // ok - b3 é false
        boolean b4 = (a instanceof String); // erro de compilação
    }
}
```

## Sombreando variáveis

Uma área interessante da Java é a forma como trata os sobreamentos de variáveis. Talvez você esteja se perguntando "o que é sobrear uma variável ???", pense em uma classe com um membro inteiro chamado *tamanho*, imagine ainda que você crie um método e internamente você também crie um variável local chamada *tamanho*, o que acontecerá ? A classe pirará ? Não a Java é esperta e sabe tratar isso, veja como:

```
public class Teste {
    static int tamanho = 0;
    public static void main(String[] args) {
        tamanho = 9;
        System.out.println("Antes: "+tamanho);
        crescer(2);
        System.out.println("Depois: "+tamanho);
    }
    public static void crescer(int tamanho) {
        tamanho = tamanho + 1;
    }
}
```

O resultado do código anterior será:

Antes: 9  
Depois: 9

A variável *tamanho* dentro do escopo do método que está sendo usada, não é a mesma declarada como membro da classe, agora se alterarmos um pouco o

**Anotações**

179

# Programação Orientada a Objetos

---

código como segue teremos um resultado diferente:

```
public class Teste {
    int tamanho = 0;
    public static void main(String[] args) {
        Teste t = new Teste();
        t.tamanho = 9;
        System.out.println("Antes: "+t.tamanho);
        t.crescer(2);
        System.out.println("Depois: "+t.tamanho);
    }
    public void crescer(int tamanho) {
        this.tamanho = tamanho + 1;
    }
}
```

O resultado será:

Antes: 9  
Depois: 3

OBS: VOCÊ NÃO PODE SIMPLEMENTE COLOCAR A PALAVRINHA *this* no método *crescer*, porque uma variável *static* nunca poderá ser referenciado por um contexto não-estático, e a palavra *this* quer dizer que é de um objeto !!!

O mesmo comportamento tem os objetos, mas preste bastante atenção pois estive conversando com o pessoal da Sun e eles me falaram que para a sua prova eles vão caprichar em questões que tratar esse assunto.

## Operadores matemáticos

Não precisa falar muito sobre esse operadores, pois sei que você é bom em matemática e apesar de ter errados algumas respostas quando a professora da 4ª série lhe perguntava.

Uma observação que é importante ser ressaltada é que toda operação envolvendo números inteiros resultarão em um tipo *int*, mesmo sendo *byte* \* *short*, ou *byte* / *int*. Como você sabe não é possível realizar divisão por zero para números inteiros, mas em número de pontos flutuantes isso é possível podendo retornar um 0 positivo ou um 0 negativo.

**Anotações**

180

# Programação Orientada a Objetos

---

Os operadores são:

Unários: -- ++

Binários: + - / \* %

```
int x = 10 + 10;
```

```
int y = 10 / 0; // Uma exceção será lançada !
```

```
java.lang.ArithmeticException
```

## Divisão por zero

Veja o código a seguir:

```
public class Teste {  
    public static void main(String[] args) {  
        double sal = 140 / 0;  
        System.out.println("Salario: "+sal);  
    }  
}
```

O resultado será uma exceção `java.lang.ArithmeticException`

Pois observe que 140 é um *inteiro* e não um *double*, por isso a exceção. Agora observe o código a seguir:

```
public class Teste {  
    public static void main(String[] args) {  
        double sal = -140.0 / 0;  
        System.out.println("Salario: "+sal);  
    }  
}
```

Resultado: Salario: -infinity (isso mesmo infinito negativo)

# Programação Orientada a Objetos

---

Olhe o comportamento estranho da Java quando os dois operadores forem zero

```
public class Teste {  
    public static void main(String[] args) {  
        double sal = -0.0 / 0;  
        System.out.println("Salario: "+sal);  
    }  
}
```

Resultado: Salario: NaN (atenha-se a saber o que resulta e não o porquê)

## Incremento e Decremento

Os operados unários incremento e decremento são bastante utilizados e devem ser aprendidos para a obtenção da tão sonhada certificação, veja como funciona:

### Operador pré-fixado

```
int x = 10;  
System.out.println("x é "+(++x));
```

Não tente fazer em casa esse trecho sem os parenteses " System.out.println("x é "++x);" o compilador pira !!!

É o equivalente a dizer: "Por favor, incremente 1 a variável x depois exiba o valor da variável, continue o seu trabalho"

O resultado será: x é 11

### Operador pós-fixado

```
int x = 9;  
System.out.println(""+x--);  
System.out.println("o valor final de x é "+x);
```

O resultado será: 9  
o valor final de x é 8

## Resumos para certificação - Operadores Java

- O resultado da execução da maioria dos operadores é um boolean ou um valor numérico;
- As variáveis são apenas depósitos de bits com tipo designado;
- Os bits de uma variável de referência representam uma maneira de acessar um objeto;
- Os bits de uma variável de referência sem sinal é null;
- Há 12 tipos de atribuições: =, \*=, +=, /=, -=, <=, >, <<, >>, << >>=, &=, ^=, |=;
- As expressões numéricas sempre resultam em um valor, pelo menos do tamanho do tipo int – nunca menor;
- Os números de ponto flutuante são implicitamente double(64 bits);
- A compactação de um tipo primitivo trunca os bits de ordem superior;
- O complemento de dois significa: inverte todos os bits e adicione 1.
- As atribuições compostas( por exemplo, +=) executam a conversão automática.

## Variáveis de referência

- Ao criarmos um novo objeto, por exemplo, Button b = new Button(), três coisas ocorrerão:
  - ❑ Será criada uma variável de referência chamada b, do tipo Button;
  - ❑ Será criado um novo objeto Button;
  - ❑ A variável de referência b apontará para o objeto Button.
- As variáveis de referência podem apontar para as subclasses do tipo declarado, mas não para as superclasses;

## Referências do objeto String

- Os objetos String são inalteráveis, não podem ser modificados;
- Quando você usar uma variável de referência String para alterar uma String;
  - ❑ Uma nova string será criada (a string existente é imutável);
  - ❑ A variável de referência apontará para a nova string;

# Programação Orientada a Objetos

---

## Operadores de comparação

- Os operadores de comparação sempre resultam em um valor booleano(true ou false);
- Há quatro operadores de comparação <,>,<=,>=;
- Quando compara caracteres, a linguagem java usa o valor ASCII ou Unicode, como valor numérico.

## Operador instanceof

- instanceof só é usado com variáveis de referência e verifica se o objeto é de um tipo específico;
- O operador instanceof só pode ser usado para testar objetos (ou valores null) confrontando-os com tipos de classes que estejam na mesma hierarquia da classe;
- Para interfaces, um objeto será “de um tipo” se alguma de suas superclasses implementarem a interface em questão.

## Operadores de igualdade

- Quatro tipos de itens podem ser testados: números, caracteres, booleanos, variáveis de referência;
- Há dois tipos de operadores de igualdade: == e !=.

## Operadores Aritméticos

- Há quatro tipos de operadores principais: adição, subtração, multiplicação e divisão;
- O operador de resto retorna o resto da divisão.
- Quando números de ponto flutuante são divididos por zero, retornam infinitos positivo ou negativo, exceto quando o dividendo também é zero, caso em que você receberá o valor NaN;
- Quando o operador de resto executar a divisão de um ponto flutuante por zero, não causará uma exceção de tempo de execução;
- Quando inteiros são divididos por zero, uma exceção ArithmeticException de tempo de execução é lançada;

Anotações

184



# Programação Orientada a Objetos

---

## Operadores de concatenação de strings

- Se um dos operandos for uma String, o operador + concatenará os operandos;
- Se os dois operandos forem numéricos, o operador + somará os operandos;

## Operadores de acréscimo / decréscimo

- O operador pré fixado será executado antes do valor ser usado na expressão;
- O operador pós fixado será executado depois que o valor ser usado na expressão;
- Em qualquer expressão, os dois operandos são avaliados integralmente antes que o operador seja aplicado;
- O valor das variáveis finais não pode ser aumentado ou diminuído.

## Operadores de deslocamento

- Há três operadores de deslocamento: >>, <<, >>>; os dois primeiros tem sinal, o último não;
- Os operadores de deslocamento só podem ser usados em tipos inteiros;
- Os operadores de deslocamento podem funcionar em todas as bases dos inteiros(octal, decimal e hexadecimal).
- Excetos nos casos raros de deslocamento de um int por um espaço múltiplo de 32, ou de um long por um espaço múltiplo de 64(esses deslocamentos não resultarão em alterações nos valores originais); os bits são preenchidos de forma a seguir:
  - << preencher os bits da direita com zeros;
  - >> preencher os bits da esquerda com o valor do bit original (bit da extrema esquerda);
  - >>> preencher os bits da esquerda com zeros (números negativos se tornarão positivos).
- Todos os operandos do deslocamento de bits são promovidos à pelo menos um tipo int;
- Em deslocamentos de tipos > 32 ou tipos long > 64, o valor do deslocamento será o resto deixada da divisão do operando direito por 32 ou 64, respectivamente;
  - Deslocar  $x \gg 4$  é exatamente o mesmo que dizer  $x / 2^4$ ;
  - Deslocar  $x \ll 3$  é o mesmo que dizer  $x * 2^3$ .

Anotações

185

# Programação Orientada a Objetos

---

## Operadores bit a bit

- Há três operadores bit a bit - &, ^, | e o operador de bit a bit complementar ~;
- O operador & configura o bit com 1 se os dois operandos estiverem configurados com 1;
- O operador ^ configura o bit com 1 se apenas 1 bit do operando estiver configurado com 1;
- O operador | configura o bit com 1 se pelo menos um bit do operando estiverem configurados com 1;
- O operador ~ inverterá o valor de cada bit do único operando.

## Ternário (Operador Condicional)

- Retorna um entre dois valores baseando-se caso uma expressão booleana é verdadeira ou falsa;
- O valor depois do símbolo ? Significa “*se verdadeiro* retorne”;
- O valor depois do símbolo : “*se falso* retorne”;

## Conversão

- A conversão implícita (não escrita em código) ocorre em uma transformação de ampliação;
- A conversão explícita (a conversão é escrita) é necessária quando se deseja uma transformação de compactação;
- Converter um número de ponto flutuante em tipo inteiro fará com que todos os dígitos à direita do ponto decimal sejam perdidos(truncados);
- A conversão de compactação podem causar perda de dados – os bits mais importantes (da extrema esquerda) podem ser perdidos.

## Operadores Lógicos

- Há quatro tipos de operadores lógicos: &, |, &&, ||;
- Os operadores lógicos trabalham com duas expressões que devem resultar em valores booleanos;
- Os operadores && e & retornaram true somente os dois operandos forem verdadeiros;
- Os operadores || e | retornarão true somente se um ou os dois operandos

## Anotações

186

# Programação Orientada a Objetos

---

forem verdadeiros;

- Os operadores && e || são conhecidos como operadores de abreviação;
- O operador && não avaliará o operando direito se o esquerdo for falso;
- O operador && não avaliará o operando direito se o esquerdo for falso;
- O operador || não avaliará o operando direito se o esquerdo for verdadeiro;
- Os operadores & e | sempre avaliam os dois operandos.

## Passando variáveis para os métodos

- Os métodos podem usar tipos primitivos e / ou referência a objetos como argumento;
- Os argumentos dos métodos são sempre cópias – de variáveis primitivas ou de referência;
- Os argumentos dos métodos nunca são objetos reais(podem ser referências a objetos);
- Na prática, o argumento primitivo é uma cópia totalmente disvinculada do tipo primitivo original;
- Na prática, o argumento de referência é outra cópia de uma referência ao objeto original.

## Capítulo IV - Controle de fluxo, exceções e assertivas

### Introdução

O controle do fluxo é um elemento essencial em qualquer linguagem de programação. Mesmo com os paradigmas da programação orientada a objetos, controlar o fluxo de um algoritmo é uma necessidade imprescindível. Quem já desenvolveu software em linguagens estruturadas como clipper, cobol, c entre outras, sabe que se esse recurso não existisse, um algoritmo não poderia ser condicionado e testado para prever por exemplo, eventos inerentes a falhas em um recurso ou dispositivo do computador, ou até mesmo se um usuário entrou com um valor correto em um campo de entrada de dados.

### if

Esse comando é mais conhecido que o ditador alemão, pois testa se uma condição é verdadeira, em sendo, executa o bloco seguinte à sua escrita, veja o exemplo:

```
if ( 2 > 1) {  
....  
}
```

Fácil isso ? Pode crer que no exame não cairá questões que abordem essa complexidade, ou melhor essa simplicidade. Veja agora o trecho a seguir :

```
1. byte x = 10;  
2. if (x<0) {  
3.     if (x == 11) {  
4.         System.out.println("x é igual 11");  
5.     }  
6.     else if (x==x--) {  
7.         System.out.println("x é menor que 11");  
8.     }}  
9. else  
10.    System.out.println("x não é menor que 0");
```

# Programação Orientada a Objetos

---

O que resultará do seguinte código ? Complicou um pouquinho ??

Se passasemos agora para o mundo compilatório (nem sei se isso existe), mas encarne o compilador agora.

A primeira coisa que ele vai checar é se na linha 2 o x é menor que 0 ? - como é uma condição falsa, iremos para a linha 9 pois esse else é referente ao if (x<0) (pode contar as chaves) e exibirá a mensagem "x não é menor que 0" .

Veja agora o trecho a seguir, apesar de muito parecido, existe uma diferença do código anterior:

```
1. byte x = 10;
2. if (x<0)
3. if (x == 11) {
4.     System.out.println("x é igual 11");
5. }
6. else if (x==x--) {
7.     System.out.println("x é menor que 11");
8. }
9. else
10.    System.out.println("x não é menor que 0");
```

O que resultará esse código ? Se você respondeu NADA ! Está correto ! Vamos entender - como (x<0) é uma condição falsa e não existe else para esse if nenhum resultado será gerado - nesse caso o else da linha 9 ficou pertencente ao conjunto de condições iniciados pelo if da linha 3, tanto é que se as condições das linhas 3 e 6 não forem verdadeiras o else será executado como mostra o código a seguir:

Com sua licença, uma observação, o if da linha 2 também foi alterado senão, não entrará na demais condições:

```
1. byte x = 10;
2. if (x>0)
3.     if (x == 11) {
4.         System.out.println("x é igual 11");
5.     }
6. else if (x==--x) {
7.     System.out.println("x é menor que 11");
8. }
9. else
10.    System.out.println("x não é menor que 0");
```

**Anotações**

189

# Programação Orientada a Objetos

---

Acho que isso já deve te alertar para as pegadinhas que os crápulas tentarão te colocar para saber o seu conhecimento quanto a sequencia do fluxo que um algoritmo deve obedecer, observe tambem que a indentação ou récuo não foi colocada para confundir mais ainda sua cabeça e pode ter certeza que eles tem muito mais para lhe apresentar, por isso ESTUDE !

if com atribuição

```
boolean b = false;
if (b = true) { System.out.println("b é verdadeiro") }
else { System.out.println("b é false") }
```

Quando tem as chaves tudo fica mais fácil ! O que o código acima resultará ?

- a) Erro de compilação !
- b) Uma exceção será lançada
- c) false
- d) true
- e) nenhum resultado

Loko isso não ! O que acontece é que b sendo uma variável boolean o compilador entende que (b = true) é uma atribuição à variável b e depois faz a comparação, mas não tente fazer isso com qualquer outro tipo de variável, veja a seguir:

```
byte x = 10;
if (x = 7) { ... } // erro de compilação
else { ... }
```

Agora, assim é permitido:

```
byte x = 10;
if (x == 7) { ... } // ok
else { ... }
```

## switch

Esse comando também é muito utilizado para checagem de valores em uma variável,mas existem algumas particularidades que devem ser observadas, veja sua sintaxe:

**Anotações**

190

# Programação Orientada a Objetos

---

O switch só poderá ser aplicado a variáveis: int short byte char  
Portanto qualquer tentativa de utilizar em outras variáveis como: boolean, long, double, float ou um objeto resultará em erro de compilação !!!

```
byte x = 3;
switch (x) {
    case 1 :
        System.out.println("x vale 1");
        break;
    case 2:
        System.out.println("x vale 2");
        break;
    default:
        System.out.println("x é maior que 2");
        break;
}
```

O uso das chaves entre os cases é opcional pois não faz nenhuma diferença, veja o código:

```
byte x = 3;
switch (x) {
    case 1 : {
        System.out.println("x vale 1");
        break;
    }
    case 2:
        System.out.println("x vale 2");
        break;
    default:
        System.out.println("x é maior que 2");
        break;
}
```

O uso do break é necessário, se for omitido o fluxo seguirá normalmente para o outro case, como você pode ver no trecho a seguir:

# Programação Orientada a Objetos

---

```
byte x = 2;
switch (x) {
    case 1 :
        System.out.println("x vale 1");
        System.out.println("x vale 1");
        break;
    case 2:
        System.out.println("x vale 2");
    default:
        System.out.println("x é maior que 2");
        System.out.println("x é maior que 2");
        break;
}
```

O código a seguir resultará :

```
x vale 2
x é maior que 2
x é maior que 2
```

Pos entrará no segundo case e por não existir o break, também seguirá para o default !

Provavelmente você se deparará com questões que abordam o uso do switch de forma inválida, como por exemplo:

## Não colocar a variável entre parênteses

```
int x = 10;
switch x {    // Erro de compilação: '(' expected
    case 1:
        ...
        break;
    case 2:
        ...
        break;
}
```

---

**Anotações**

192



# Programação Orientada a Objetos

---

## Uso de duas opções case iguais

```
int x = 10;
switch (x) {
    case 1: { ... }
    case 2: { ... }
    case 1: { ... } // Erro de compilação: duplicate case label
}
```

## Uso de variável inválida

```
long x = 100;
switch (x) { // Erro de compilação: possible loss of precision
    case 1: { ... }
    case 2: { ... }
}
```

## Uso de objeto

```
String x = new String("teste");
switch (x) { // Erro de compilação: incompatible types
    case "aaa": { .... }
}
```

## Uso de variável não final

```
final int op1 = 1;
final int op2 = 2;
int op3 = 3;
int opcao = 2;

switch (opcao) {
    case op1: { ... } // ok, op1 é final
    case op2: { ... } // ok, op2 é final
    case op3: { ... } // Erro de compilação: constant expression required
    default: { ... }
}
```

## Anotações

---

---

---

---

193

# Programação Orientada a Objetos

---

Bom acho que o switch é isso, mas vamos retificar nossa linha de pensamento para ver se você entendeu tudo direitinho. Observe o trecho de código a seguir:

```
int x = 1;
switch (x) {
    case 1: System.out.println("1");
    default: System.out.println("default");
    case 2: System.out.println("2");
    case 3: System.out.println("3");
    case 4: System.out.println("4");
}
```

O que resultará ?

1  
default  
2  
3  
4

Lembre-se sempre, uma vez que um case for verdadeiro, todos subsequentes serão se nenhuma palavra chave break for encontrada.

Agora observe esse outro código:

```
int x = 3;
switch (x) {
    case 1: System.out.println("1");
    default: System.out.println("default");
    case 2: System.out.println("2");
    case 3: System.out.println("3");
    case 4: System.out.println("4");
}
```

O que resultará ?

3  
4

Observe que como a condição default está antes da primeira condição verdadeira (case 3), ela não é executada.

**Anotações**

194

# Programação Orientada a Objetos

---

## Extremamente importante saber

```
public static void main(String[] args) {  
    final int x = 2;  
    for (int i = 0; i < 2; i++) {  
        switch (i) {  
            case x-1: System.out.print("1 ");  
            default: System.out.print("def ");  
            case x: System.out.print("2 "); break;  
            case x+1: System.out.print("3 ");  
        }  
    }  
}
```

O que esse código resultará ?

- a) Não será compilado
- b) def 2 def 2 1
- c) def 2 1 def 2
- d) def 2 1 def 1
- e) def 1 2 def 1

Agora observe o mesmo código alterando simplesmente a localização da palavra *default*.

```
public static void main(String[] args) {  
    final int x = 2;  
    for (int i = 0; i < 4; i++) {  
        switch (i) {  
            case x-1: System.out.print("1 ");  
            case x: System.out.print("2 "); break;  
            case x+1: System.out.print("3 ");  
            default: System.out.print("def ");  
        }  
    }  
}
```

O resultado será: def 1 2 2 3 def

**Anotações**

195

# Programação Orientada a Objetos

---

## Entendendo o primeiro código

O mais interessante e que deve ser ressaltado é como o switch faz suas comparações, vamos tentar entender. Em nosso primeiro código sabemos que ocorrem duas iterações - isso todos concordam. O switch vai ao primeiro case - (case x-1), ou seja, se 0 (valor de i) é igual a 1, não é o caso, vai ao próximo que em nosso caso é um default, como é um default o switch pergunta: "Existe algum case abaixo que atenda a minha necessidade (i=0) ????" - e uma voz do além "Não senhor, nenhum caso que atenda essa condição", bom nesse caso o switch executa o default exibindo a palavra def na tela e como não existe break, executa também o case x imprimindo o valor 2 onde encontra o break e para a execução do switch. Na segunda iteração o i vale 1 portanto na primeira condição case x -1 é atendida imprimindo 1 na tela, em seguida def consecutivamente o número 2, onde a palavra break é encontrada encerrando o switch.

## Falando um pouco em loops

### while

Quando você desejar executar um bloco uma quantidade desconhecida de vezes, esse é comando normalmente utilizado, veja sua sintaxe:

```
int x = 10;
while (x==10) {
    System.out.println("entrou");
    x++;
}
```

Resultado desse código:

entrou

Esse código será executado somente uma vez, visto que na segunda iteração o valor de x estará valendo 11, o que resultará em falso na condição do while, isso não dá uma dica de que uma iteração while pode não acontecer se a condição testada primeiramente for falsa, como você poderá ver o código a seguir:

---

### Anotações

196

# Programação Orientada a Objetos

---

```
int x = 11;
while (x==10) {
    System.out.println("entrou");
    x++;
}
```

Nenhum resultado será exibido.

## do-while

Outro comando de iteração que pode ser usado é do-while, veja sua sintaxe:

```
int x = 0;
do {
    System.out.println(""+x);
    x++;
} while (x < 1);
```

Resultado:

0

Esse comando SEMPRE será executado pelo menos uma vez, pois a condição é verificada após a iteração.

## for

O comando for é utilizado normalmente quando você sabe quantas vezes você precisa fazer as iterações. Apesar de que ele também pode ser usado exatamente como o while mas não é uma boa prática de programação.

```
for ( int i = 0; i < 10; i++ ) {
    System.out.print(""+i);
}
```

Resultado: 0123456789

---

## Anotações

197

# Programação Orientada a Objetos

---

O comando for é composto por três expressões:

- a) declaração e inicialização
- b) expressão condicional
- c) expressão de iteração

## a) declaração e inicialização

É executado uma única vez antes da primeira iteração, normalmente se usa para criar e inicializar um variável para controlar as iterações - você pode inicializar outras variáveis nessa parte como mostra o código a seguir:

```
int s = 9;
for ( int i = 0, s = 10; i < 10; i++ ) {
    System.out.println(""+(s+i));
}
```

Observe que a variavel s foi redefinida com o valor 10 no bloco de inicialização (separado por vírgula). O escopo de uma variável criada nessa parte é somente dentro do escopo do comando for não tente usar uma variável como i em nosso exemplo anterior após o término do comando for que você magoará o compilador.

```
for ( int i = 0; i < 10; i++ ) {
    System.out.println(""+i);
}
System.out.println(" apos o for: "+i);// erro de compilação: cannot resolve symbol
```

## b) expressão condicional

Só pode haver UMA expressão condicional (não tente separar condições por vírgula que você terá um erro do compilador), será testada antes de cada iteração.

```
for (int i = 0; i < 1; i++) {
    System.out.println(""+i);
}
```

# Programação Orientada a Objetos

---

Nesse exemplo uma iteração será executado pois  $i$  vale 0 e antes de executar a primeira iteração a JVM testa se  $i < 1$  ou seja se 0 é menor que 1, como é true, executa a iteração. Já na segunda iteração a "expressão de iteração" será executada incrementando 1 a variável  $i$  portanto  $i$  valerá 1, o que tornará a condição falsa, por isso esse código resultará:

0

Você poderá ter uma condição composta como mostra o código a seguir:

```
for (int i=0; i < 10 | (i%2) == 0; i++) {  
    System.out.println(""+i);  
}
```

Mas lembre-se, quando essa condição for false o loop é encerrado, e não pense que vai para a próxima iteração, funciona como o while, mesmo que exista uma condição como o código anterior  $i < 10$ .

## c) expressão de iteração

Essa expressão é executada sempre após a execução de uma iteração, ou seja, quando um bloco termina, a expressão de iteração é executada. Por isso o código a seguir é executado pelo menos uma vez:

```
for (int i = 0; i < 1; i++) {  
    System.out.println(""+i);  
}
```

Porque se a expressão (  $i++$  ) fosse executada antes da execução da primeira iteração, nenhum resultado seria mostrado na tela.

# Programação Orientada a Objetos

---

## break

Essa palavra chave dentro de uma iteração seja ela (for, while, do) faz com que o fluxo será reportado para a primeira linha após o bloco de iteração como você pode ver no código a seguir:

```
int x = 0;
for (int i = 0; i < 10; i++ ) {
    x = i;
    if ( i > 3 ) break;
}
System.out.println("ultimo valor de i: "+x);
```

Esse código resultará:

ultimo valor de i: 4

## continue

Interrompe a iteração atual indica para iniciar a próxima iteração. Para exemplificar vamos imaginar um algoritmo onde se deseja lista todos os números ímpares menores que 10:

```
for (int i = 0; i < 10; i++ ) {
    if ((i%2)==0) continue;
    System.out.print(""+i);
}
```

O código acima resultará:

13579

Não vamos discutir se esse é o melhor algoritmo para esse problema mas é uma solução! Saiba que todas as linhas após o *continue* (que não é o nosso caso) não seria executada.

Funciona exatamente igual para os comandos de iteração: ( *do* e *while* )

---

## Anotações

200



# Programação Orientada a Objetos

---

## Outras formas de se utilizar o comando for

Você não é obrigado a definir nenhuma das três expressões, podendo usar o comando for da seguinte forma:

```
for ( ; ; ) {  
    ...  
}
```

Esse código comporta-se exatamente igual ao:

```
while (true) {  
    ....  
}
```

Saiba que palavra break deverá estar contida dentro de qualquer um desses blocos, caso contrário isso causará um loop infinito, talvez não seja infinito porque seu computador será desligado por orientação da ANAEL (lembra-se do APAGÃO) !

```
for (int i=0; i < 10; ++i) {  
    System.out.println(""+i);  
    continue;  
}
```

Quantas iterações serão executadas com esse código ?

- a) erro de compilação
- b) uma exceção será lançada
- c) 10
- d) 9
- e) loop infinito

Explicação: A palavra-chave *continue* nesse código não faz nada de mais além do que o próprio comando já faria, visto que ele não está em nenhuma condição e também não existe nenhum código abaixo dele, portanto, é natural que mesmo sem ter o *continue* o comando já passaria para a próxima iteração.

**Anotações**

201

# Programação Orientada a Objetos

---

## loops rotulados

Apesar de ser um assunto crítico, os loops rotulados devem ser entendidos para a obtenção da sua certificação, mas não posso deixar de expressar minha opinião pessoal, é um recurso (se assim me permitem dizer) feio como prática de programação, lembrem-se daquelas linguagens que usavam goto (como basic, etc.) ou arquivos de lote (.bat), pois então, não use essa prática - espero que nenhum programador basic ou de arquivos de lote do DOS leia isso. Veja como criar um loop rotulado:

```
int x=0;
for (int i=0; i < 4; i++) {
    :foo
    for (int j=0; j <= 4; j++ ) {
        if (i == j) {
            System.out.println(i+"x"+j);
            break foo;
        }
    }
}
```

Esse código resultará:

```
0x0
1x1
2x2
3x3
```

Não precisa falar muito sobre o que aconteceu com esse código, pois essa seria uma ação normal mesmo se não existisse o rótulo no loop interno ( for j ), mas se você precisar cancelar as iterações do for exterior ( for i ) ???

## Trabalhando com exceções

Esse recurso é muito importante para a garantia de um código seguro, ou seja, trabalhar com exceções em uma linguagem é uma forma de tentar honrar seu programa contra erros que inconvenientemente aparecem se serem chamados.

---

**Anotações**

202

# Programação Orientada a Objetos

---

```
try {  
    // primeira linha vigiada  
}  
catch (Exception e) {  
    // primeira linha que será executada caso haja um exceção do tipo Exception  
}  
finally {  
    // bloco que será executado, havendo ou não uma exceção (sempre!)  
}
```

## try

O bloco try inicia uma região vigiada, ou seja, qualquer exceção que ocorrer nesse trecho modificará o fluxo do programa para um bloco catch (se houver) e/ou finally (se houver). Um bloco try deve obrigatoriamente ter ou um catch ou um finally, pode ter os dois, mas nunca um bloco try sozinho.

### Válido

```
try {  
}  
catch (MyException e) {  
}
```

### Válido

```
try {  
}  
finally {  
}
```

### Inválido

```
try {  
}
```

Erro: 'try' without 'catch' or 'finally'

---

## Anotações

203

# Programação Orientada a Objetos

---

## catch

Bloco que define uma exceção e sua ação caso venha ocorrer. Você poderá ter mais de um catch em um bloco try.

```
try {  
    // algoritmo  
}  
catch (MyException e) {  
    // solução caso ocorra  
}
```

Nesse caso MyException é a exceção que pode acontecer, não que seja um desejo ardente do coração do programador, mas é uma cautela !

## finally

Bloco que será executado após o try ou catch, ou seja, se um bloco try tiver um subbloco finally, este sempre será executado, independente se ocorrerem exceções ou não. Talvez você tenha se perguntado, "porque não coloque o código do bloco finally no try ou no catch ?" como mostra o pseudo-código a seguir:

Algo assim você deve ter pensado:

```
try {  
    1. Aloca memória para abertura de um arquivo  
    2. Abre um arquivo  
    3. Imprime arquivo na porta da impressora  
    4. Fecha arquivo  
    5. Desaloca memória  
}  
catch (ArquivoNaoExisteException e1) {  
    1. Desaloca Memória  
    2. Mostra mensagem que arquivo não foi encontrado  
}  
catch (PortadaImpressoraNaoResponde e2) {  
    1. Desaloca Memória  
    2. Fecha Arquivo  
}
```

---

## Anotações

---

---

---

---

204

# Programação Orientada a Objetos

---

Observe que o código de desalocar memória está redundante (tá!! sei que poderia alocar somente se abrisse o arquivo, mas não vamos complicar)  
Veja como assim fica mais bonito:

```
try {  
    1. Aloca memória para abertura de um arquivo  
    2. Abre o arquivo  
    3. Imprime arquivo na porta da impressora  
    4. Fecha arquivo  
    5. Desaloca memória  
}  
catch (ArquivoNaoExisteException e1) {  
    1. Mostra mensagem que arquivo não foi encontrado  
}  
catch (PortadaImpressoraNaoResponde e2) {  
    2. Fecha Arquivo  
}  
finally {  
    1. Desaloca memória  
}
```

## Hierarquia de classes de exceção

Pode parecer estranho, mas existe uma diferença singular entre Erro e Exceção. Por mais que você tenha achado esquisito essa colocação, saiba que um erro é algo que não tem "remédio" e exceção é um evento que deveria acontecer normalmente mas por algum motivo de força maior não conseguiu se completar.

Erro

- ❖ Falta de memória do computador

Exceção

- ❖ Tentar consultar um coluna em um banco de dados que não exista
- ❖ Tentar abrir um arquivo que foi apagado
- ❖ Tentar trocar informações com um outro computador que por algum motivo for desligado

Isso nos leva a seguinte conclusão:

**Anotações**

205

# Programação Orientada a Objetos

---

Todas as subclasses de Exception (com exceção as subclasses RuntimeException) são exceções e devem ser tratadas !!!

Todos os erros provenientes da classe Error ou RuntimeException são erros e não precisam ser tratados !!!

Você deve estar se perguntando, "pô! mas tudo não são erros ?" ! São, mas somente as exceções precisam ser tratadas ! Vamos ver alguns código a seguir que o ajudará a entender esses conceitos.

## Lançando uma exceção:

```
1. public class Erros {  
2.     public static void main(String[] args) {  
3.         metodoDoMal();  
4.     }  
5.     public static void metodoDoMal() {  
6.         throw new IOException("eu fiz um erro");  
7.     }  
8. }
```

Esse código lança uma exceção dentro do *metodoDoMal()*. Normalmente você não precisará lançar exceções pois Murphy se encarregará de fazer isso pra você, porém imagine uma situação em que você precise gerar um exceção. Apesar da sintaxe está correta na geração da exceção, esse código não será compilado. Quem está tratando a exceção IOException que foi gerada no método *metodoDoMal()* ?? Se uma exceção ocorrer ela deve ser tratada (se não for Error ou RuntimeException é claro), e existem duas maneiras de se fazer isso:

# Programação Orientada a Objetos

---

## Usando um bloco try/catch

O código poderia ser remediado da seguinte forma:

```
1. public class Erros {
2.     public static void main(String[] args) {
3.         metodoDoMal();
4.     }
5.     public static void metodoDoMal() {
6.         try {
7.             throw new IOException("eu fiz um erro");
8.         }
9.         catch (IOException e) {
10.            // código que solucionará o problema
11.        }
12.    }
13. }
```

## Usando a propagação de exceção através de métodos

```
1. public class Erros {
2.     public static void main(String[] args) {
3.         metodoDoMal();
4.     }
5.     public static void metodoDoMal() throws IOException {
6.         throw new IOException("eu fiz um erro");
7.     }
8. }
```

Esse código também não será compilado, pois o compilador é esperto e sabe que uma exceção pode ocorrer em `metodoDoMal` (pois foi declarado em `throws IOException`) e está sendo propagado para o método mais que não está tratando, pois nenhum bloco `try` existe no método chamado. Talvez isso tenha complicado um pouco mais olhe o código a seguir:

# Programação Orientada a Objetos

---

```
import java.io.*;
import java.awt.print.*;

public class Teste {
    public static void main(String[] args) {
        try {
            metodoDoMal();
        }
        catch (IOException e) {
        }
        catch (PrinterException p) {
        }
    }
    static void metodoDoMal() throws IOException, PrinterException {
        metodoPiorAinda();
    }

    static void metodoPiorAinda() throws PrinterException {
    }
}
```

Isso é o que chamamos de pilha de método, mas acalme-se é fácil. O método *main()* chama o método *metodoDoMal()* que por sua vez chama o *metodoPiorAinda()*, criando um pilha de métodos

## Pilha

MetodoPiorAinda() throws PrinterException

metodoDoMal() throws IOException, PrinterException

main(String[] args)

Observe que o método *MetodoPiorAinda()* declara a exceção *PrinterException*, isso que dizer que essa exceção poderá ocorrer dentro desse método. Como essa exceção não está sendo tratada em nenhum bloco try/catch, ela deverá propagada para o método superior na pilha "*metodoDoMal()*", Se nesse método não houver um tratamento com try/catch ou propagação através de throws na cabeça do método, haverá um erro de compilação (pode testar, tire a declaração *PrinterException* de *metodoDoMal()* que esse código não será compilado).

**Anotações**

208



## Programação Orientada a Objetos

---

Quando o *metodoDoMal()* for encerrado, seguindo a ordem da pilha o fluxo será retornando para o método *main*, se esse método não tratar as exceções declaradas em *metodoDoMal()*, o código também não será compilado (tente tirar qualquer um dos blocos *catch*'s que o código será extinguido do planeta).

Você terá que saber a diferente entre um exceção declarada e não-declarada. Lembre-se sempre que os erros (subclasses de *Error* e *RuntimeException*) não precisam ser tratadas como mostra o código a seguir:

```
public class Teste {  
    public static void main(String[] args) {  
        metodoDoMal();  
    }  
    static void metodoDoMal() throws Error {  
    }  
}
```

Observe que o método *metodoDoMal* declara/propaga uma exceção (erro) e o método chamado (*main*) não trata e nem propaga.

## Assertivas

O uso de assertivas é um recurso adicionado somente na versão 1.4 que permite ao programador testar a consistência de seu código. Você deverá sempre usar assertiva quando quiser testar se uma variável está em um dado momento com um valor apropriado, se não gerar um `AssertionError`, isso em tempo de desenvolvimento, embora quando o programa for distribuído e entregue ao cliente, nenhuma anormalidade ocorrerá visto que as assertivas nunca são ativadas por padrão. Considere o pensamento de que as assertivas foram incrementadas para auxiliar ao programador gerar código mais robustos, mas nunca poderão alterar o comportamento do programa.

```
private int foo(int i) {  
    assert (i>0);  
    return i * i;  
}
```

Observe que o código anterior nada mais faz do que testar se o valor de `i` passado como parâmetro é maior que 0 se for o programa não faz nada, agora se o valor de `i` não for maior que 0, então um `AssertionError` será lançada - com isso o programador poderá saber que em algum lugar algum engraçadinho está passando um valor incorreto para o método `foo`. Mas nenhuma consequência isso teria se o código fosse entregue, visto que nenhum resultado seria afetado no cliente.

Existem duas maneiras de se usar assertivas:

muito simples

```
assert ( i < 0 );
```

simples

```
assert ( i > 0 ) : "Valor do i é "+i;
```

A segunda forma, incrementa o valor após os dois pontos na mensagem de erro de `AssertionError`, com isso você poderá depurar seu código com mais detalhes.

# Programação Orientada a Objetos

---

Você precisará saber a diferença entre usar uma assertiva de forma válida e de forma apropriada ou correta, visto que, válida está relacionada com o código ser compilado, mas nem sempre é a maneira como deve ser.

## Assertivas - uso incorreto/inapropriado

### 1) Nunca manipula um AssertionError

Não use um catch e manipula um erro de assertiva

### 2) Não use assertiva para validar argumentos em métodos públicos

Você não pode garantir nada em métodos público, portanto usar assertiva nesse caso, não é uma boa prática

### 3) Não use assertiva para validar argumento da linha de comando

Isso é uma particularidade de um método público, mas segue a mesma regra

### 4) Não use assertivas que possam causar efeitos colaterais

Você não pode garantir que as assertivas sempre serão executadas, portanto o seu programa deve executar independentemente das assertivas, e nunca de forma diferente simplesmente porque as assertivas foram ativadas.

## Uso Apropriado/Correto

### 1) Use assertiva para validar argumentos em métodos privados

Como a visibilidade é privada, você consegue detectar se alguém (você) tá fazendo caca.

### 2) Use assertiva sem condição em um bloco que pressuma que nunca seja alcançado.

Se você tem um bloco que nunca seria alcançado, você pode usar uma assert false, pois assim saberia se em algum momento esse bloco está sendo alcançado.

---

## Anotações

211

# Programação Orientada a Objetos

---

3) Lançar um `AssertionError` explicitamente.

4) Se um bloco `switch` não tiver uma instrução *default*, adicionar uma assertiva é considerado uma boa prática.

## Considerações finais

- Mesmo que um código tenha sido compilado com o uso de assertivas, quando executar, vc deverá explicitamente forçar a utilização, pois por padrão as assertivas não são ativadas;
- Assertivas não são compiladas nem executadas por padrão;
- É possível instruir a JVM desativar assertivas para uma classe e ativar para um dado pacote em uma mesma linha de comando;

## flags

```
java -ea          java -enableassertion  // habilita assertion
java -da          java -disableassertion // desabilita assertion
java -ea:br.com.Atimo //habilita para o pacote br.com.Atimo
java -ea -das      // habilita assertion em âmbito geral e desabilita nas classes do sistema
java -esa          // habilita nas classes do sistema
```

## Resumos para certificação - Instrução if e switch

- A instrução if deve ter as expressões enfeixadas por pelo menos um par de parênteses;
- O único argumento válido para uma instrução if é um valor booleano, portanto o teste if só pode ser inserido em uma expressão que tenha como resultado um valor ou uma variável booleana;
- Cuidado com atribuições booleanas (=) que podem ser confundidas com testes de igualdade booleanos (==);
  - ❑ `boolean x = false;`
  - ❑ `If(x = true){}`
- As chaves serão opcionais para o bloco if que tiverem só uma instrução condicional. Porém cuidado com os recuos enganosos;
- As instruções switch podem avaliar somente tipos byte, short, int e char. Você pode escrever:
  - ❑ `long s = 30;`
  - ❑ `Switch(s){}`
- O argumento de case deve ser uma variável literal ou final. Você não pode ter uma instrução case que inclua uma variável que não seja final ou um intervalo de valores;
- Se a condição de uma instrução switch coincidir com o valor de uma instrução case, todo o código da instrução switch que tiver após a instrução case coincidente, será executado até que uma instrução break ou o final da instrução switch seja alcançado. Em outras palavras, a instrução case coincidente será apenas o ponto de entrada para o bloco case, porém a menos que haja uma instrução break, a instrução case coincidente não será o único código do bloco case a ser executado;
- A palavra chave default deve ser usada em uma instrução switch se você quiser executar um código quando nenhum dos valores das instruções case coincidir com o valor condicional;
- O bloco padrão pode ser inserido em qualquer local do bloco switch, portanto, se nenhuma instrução case apresentar correspondência, o bloco default será executado, e se ele não tiver uma instrução break, continuará a ser executado (passagem completa) até o final da instrução switch ou até que a instrução break seja encontrada.

# Programação Orientada a Objetos

---

## Escrevendo código usando loops

- Uma instrução `for` não requer nenhum argumento na declaração, mas possui três partes: a declaração e / ou inicialização, a avaliação booleana e a expressão de iteração;
- Se uma variável `for` incrementada ou avaliada dentro de um loop `for`, terá que ser declarada antes do loop ou dentro da declaração do loop `for`;
- Uma variável declarada (e não inicializada) dentro da declaração do loop `for` não poderá ser acessada fora do loop (em outras palavras, o código abaixo do loop `for` não poderá usar a variável);
- Você pode inicializar mais de uma variável na primeira parte da declaração do loop `for`, cada inicialização de variável tem que ser separada por vírgula;
- Você não pode usar um número (antiga estrutura das linguagens do estilo C) ou outro item que não resulte em um valor booleano como a condição de uma instrução `if` ou estrutura de loop. Não pode por exemplo, escrever: ***if (x)*** a menos que `x` seja uma variável booleana;
- O loop `do – while` entrará no corpo do loop pelo menos uma vez, mesmo se a condição do teste não for atendida.

## Usando o `break` e `continue`

- Uma instrução não rotulada fará com que a interação atual da estrutura de loop mais interna seja interrompida e a linha de código posterior ao loop seja executada;
- Uma instrução `continue` não rotulada fará com a iteração do loop mais interno seja interrompida e a condição desse loop seja verificada, se ela for atendida, o loop será executado novamente;
- Se a instrução `break` e `continue` for rotulada, ela fará com que uma ação semelhante ocorra no loop rotulado e não no loop interno;
- Se uma instrução `continue` for usada em um loop `for`, a instrução de iteração será executada e a condição verificada novamente.

---

## Anotações

214

## Capturando uma exceção

- As exceções vêm em duas versões: as verificadas e não verificadas;
- As exceções capturadas incluem todos os subtipos de `Exception`, excluindo as classes que estendem `RuntimeException`;
- As exceções verificadas estão sujeitas à regras de manipulação ou declaração, qualquer método que puder lançar uma exceção verificada (incluindo os métodos que chamem outros métodos capazes de realizar essa tarefa) deve declará-la usando a palavra chave `throws` ou manipulá-la com um bloco `try/catch` apropriado;
- Os subtipos de `Error` ou `RuntimeException` não são verificados, portanto, o compilador não impõem a regra de manipulação ou declaração. Você pode manipulá-los e também declará-los, mas o compilador não se importará se uma ou outra coisa ocorreu;
- Se você usar o bloco `finally`, ele sempre será chamado, independente de uma exceção do bloco `try` correspondente ser ou não lançada e de uma exceção lançada ser ou não capturada;
- A única exceção à regra do bloco `finally` ser sempre chamado será quando o JVM for encerrado. Isso poderá acontecer se o código do bloco `try` ou `catch` chamar `System.exit()`, situação na qual o JVM não iniciará seu bloco `finally`;
- Só porque `finally` será chamado não significa que será concluído. O código do bloco `finally` pode ele próprio lançar uma exceção ou emitir um `System.exit()`;
- As exceções não capturadas serão propagadas para baixo na pilha de chamadas, a partir do método em que a exceção foi lançada e terminando no primeiro método que tiver um bloco `catch` correspondente a esse tipo de exceção ou com o encerramento do JVM (que ocorrerá se a exceção chegar a `main()` e esse método não passar a exceção declarando-a);
- Você pode criar suas próprias exceções, geralmente estendendo `Exception` ou um de seus subtipos. Ela será considerada uma exceção verificada e o compilador forçará a aplicação da regra de manipulação ou declaração e essa exceção;

# Programação Orientada a Objetos

---

- Todos os blocos catch devem ser ordenados do mais específico ao mais geral. Por exemplo, se você tiver uma cláusula catch tanto para IOException quanto para Exception, terá que inserir o bloco catch de IOException primeiro(em ordem, de cima para baixo do seu código). Do contrário, a exceção IOException seria capturada pelo bloco catch (Exception e), porque um argumento catch pode capturar a exceção especificada ou qualquer um de seus subtipos. O compilador o impedirá de definir cláusulas catch que nunca poderão ser alcançadas (uma vez que ele consegue detectar que a exceção mais específica será capturada primeiro pelo bloco catch mais genérico).

## Trabalhando com o mecanismo de assertivas

- As assertivas lhe fornecerão uma maneira de testar suas suposições durante o desenvolvimento e a depuração;
- As assertivas geralmente são ativadas durante o teste, mas desativadas durante a distribuição;
- Você pode usar assert como palavra chave(na versão 1.4) ou um identificador, mas não como ambos. Para compilar códigos antigos que usem assert como identificador(por exemplo, o nome de um método), use o flag de linha de comando `-source 1.3` em javac.
- As assertivas são desativadas no tempo de execução por padrão. Para ativá-las, use o flag de linha de comando `-ea` ou `enableassertions`;
- Você pode desativar as assertivas seletivamente usando o flag `-da` ou `disableassertions`;
- Se você ativar ou desativar assertivas usando flag sem nenhum argumento, estará ativando ou desativando-as em âmbito geral. É possível combinar switches de ativação e desativação para ter as assertivas ativadas em algumas classes e/ou pacotes, mas não em outros;
- Você pode ativar ou desativar assertivas nas classes do sistema como flag `-esa` ou `-das`;
- Você pode ativar ou desativar assertivas por classes, usando a sintaxe a seguir: ***Java -ea -da:MyClass TestClass***

Anotações

216



## Programação Orientada a Objetos

---

- Você pode ativar e desativar assertivas por pacote, e qualquer pacote que especificar também incluirá todos os subpacotes(pacotes abaixo na hierarquia do diretório);
- Não use as assertivas para lidar com argumentos de métodos públicos;
- Não use expressões assertivas que causem efeitos colaterais. A exceção das assertivas não é sempre garantida, portanto, não será bom ter um comportamento que se altere dependendo das assertivas estarem ativadas;
- Use assertivas – mesmo em métodos públicos – para garantir que um bloco de código específico nunca seja alcançado. Você pode usar: ***assert false;*** para o método que nunca será alcançado de modo que um erro de assertiva seja lançado imediatamente se a instrução assert for executada;
- Não use expressões assertivas que possam causar efeitos colaterais.

# Programação Orientada a Objetos

---

## Capítulo V - Orientação a Objetos, sobreposição e substituição, construtores e tipos de retorno

### Escapsulamento, relacionamentos É-UM e TEM-UM

Escapsular - tem a função básica de proteger uma classe de "coisas estranhas", ou seja, mantê-la de forma mais protegida contra o uso indevido de certos usuários. Vamos ver o seguinte código para entender isso:

```
public class Carro {  
    private float motor = 1.0f;  
    public setMotor( int value ) {  
        this.motor = value;  
    }  
    public float getMotor() {  
        return this.motor;  
    }  
}
```

O código acima mostra a forma mais trivial de se proteger os membros de uma classe, ou seja, mantenha seus atributos com o modificador *private* (com isso o acesso se restringe a classe), e crie métodos set's e get's para modificar e acessar seus membros - os padrões mundiais dizem que você deve nominar seus métodos da forma acima setNomeDoAtributo e getNomeDoAtributo.

Talvez você deve esta se perguntando: "pô, mas se eu pude setar diretamente a variável motor, porque eu não posso acessá-la diretamente ???" Esse é um raciocínio muito bom, mas se não o teve não se preocupe, vamos entender. Imagine que a classe acima foi criada liberando o acesso direto a seus membros, dois anos após a sua criação, descobre-se que ninguém pode setar o valor de motor inferior a 1.0 (mesmo porque um carro com motor inferior a isso, nem pode ser considerado um carro!), qual a solução eminente ? Varrer todos os código que utilizam essa classe e verificar se existe erro, caso exista, conserta-se! Bom se o tempo de produção não é um diferencial em sua equipe de desenvolvimento, eu outras empresa ou lugares, as pessoas não querem perder muito tempo, com essa mudanças, pois se a classe for criada da forma acima, basta alterar o método setMotor, fazer uma condição e consertar somente ali ! Viu como o encapsulamento é uma idéia interessante !

Anotações

218

# Programação Orientada a Objetos

---

## O relacionamento É-UM refere-se a herança

Quando se vê uma questão que se indaga se X É-UM Y, isso quer dizer se X é uma subclasse, subtipo, estendeu de Y. Veja o código:

```
public class Carro {}

public class Mazda extends Carro {}
```

Se você se deparar com uma questão, Mazda É-UM Carro, pode ter certeza que sim ! Agora do inverso não ! Carro É-UM Mazda !

## TEM-UM referencia outra classe

Quando se tem uma questão: X TEM-UM Y, quer dizer que X tem uma referencia a Y, através de um atributo, vejamos um exemplo:

```
public class Motor { }
public class Pneu { }
public class Carro {
    private Motor motor;
    private Pneu[] pneu;
}
public class Mazda extends Carro { }
```

## Asserções corretas

Carro TEM-UM Motor  
Carro TEM-UM Pneu  
Mazda É-UM Carro

# Programação Orientada a Objetos

---

## Substituição de método (override)

Substituir um método como o próprio nome já diz, significa cancelar um método de uma superclasse em uma subclasse dando-o um outro sentido, ou tecnicamente, um outro código - portando infere-se que eu só posso substituir um método se houver herança caso contrário não existe substituição de método, vejamos o código:

```
public class Empregado {  
    public double getSalario() {}  
}  
  
public class Horista extends Empregado {  
    public double getSalario() {  
        // código que retorna salário em horas  
    }  
}  
  
public class Mensalista extends Empregado {  
    public double getSalario() {  
        // código que retorna salário mensal  
    }  
}
```

Observe que o método *getSalario* nas subclasses têm comportamento distintos nas subclasses - isso devido ao fato de ter sido substituído nas subclasses. Observe ainda que o método nas subclasses se manteve idêntico ao método da superclasse no que se refere a assinatura do método (lista de argumentos) - se essa assinatura tivesse sido alterada, não seria substituição (override) e assim sobreposição ou sobrecarga (overload) - muita atenção com isso, mas estudaremos isso mais a frente.

```
public class Empregado {  
    public double getSalario() {}  
    public double getSalario() {}  
}
```

---

**Anotações**

220

# Programação Orientada a Objetos

---

Qualquer tentativa de redefinição de método na mesma classe, você passará por uma vergonha incrível pelo compilador, não tente! O que define um método é sua assinatura (lista de argumento) - portanto mesmo que tente enganá-lo como o código a seguir, assuma as consequências:

```
public class Empregado {  
    public double getSalario() { }  
    public float getSalario() { }  
}
```

Erro de compilação: getSalary() is already defined in Empregado

Lembre-se sempre: O método para ser substituído deve ter a mesma assinatura (lista de argumentos) na subclasse, caso contrário, não é substituição e sim sobreposição.

## Modificando o acesso de um método substituído

Você poderá alterar a visibilidade de um método substituído, mas nunca para uma visibilidade inferior a definida na superclasse, ou seja, se você tiver um método na superclasse com o modificador *protected* você não poderá substituí-lo por um método com modificador *private* (mais restritivo) mas para um método público por exemplo, sem nenhum problema.

A ordem é:

private  
padrão  
protected  
public

Como podem ser substituídos:

private	private, padrão, protected e public
padrão	padrão, protected e public
protected	protected, public
public	public

# Programação Orientada a Objetos

---

Vejamos o código:

```
abstract class Empregado {  
    protected abstract double getSalario();  
}
```

```
abstract class Horista extends Empregado {  
    public abstract double getSalario();  
}
```

Veja que a classe Horista tem um método mais abrangente que o método da superclasse. Recordando ao capítulo 2, lembre-se que a primeira subclasse concreta de Horista ou Empregado deverá substituir o método getSalario() !

Uma tentativa de restringir mais o código do método causa erro de compilação, como veremos o código a seguir:

```
abstract class Empregado {  
    protected abstract double getSalario();  
}
```

```
abstract class Horista extends Empregado {  
    private abstract double getSalario();  
}
```

Erro de compilação: getSalario() in Horista cannot override getSalario() in Employee; attempting to assign weaker access privileges; was protected !

Não deve lançar exceções verificadas novas ou mais abrangentes

# Programação Orientada a Objetos

---

## Exceções mais abrangentes

Já discutimos o que são exceções no capítulo 4, e com certeza não precisaremos relembrar (torcemos pra isso - pois só foi um capítulo atrás). Uma exceção mais abrangente é uma exceção que pode capturar mais erros, vejamos o código:

```
abstract class Empregado {  
    abstract public double calcularComissao() throws ClassNotFoundException;  
}
```

```
abstract class Horista extends Empregado {  
    abstract public double calcularComissao() throws Exception;  
}
```

Na hierarquia de classes de exceções, `ClassNotFoundException` é subclasse de `Exception`, ou seja, `Exception` é mais abrangente que `ClassNotFoundException`, portanto qualquer tentativa de compilar o código acima causará um erro de compilação: `calcularComissao() in Horista cannot override calcularComissao() in Empregado; overridden method does not throw java.lang.Exception !`

## Exceções verificadas novas

```
abstract class Empregado {  
    abstract public double calcularComissao() throws ClassNotFoundException;  
}
```

```
abstract class Horista extends Empregado {  
    abstract public double calcularComissao() throws IllegalAccessException,  
    RuntimeException;  
}
```

O código acima não compila pois `IllegalAccessException` é uma exceção verificada e não tinha sido definida no método da superclasse, enquanto que a exceção `RuntimeException` não é verificada e não causará nenhum efeito, se tirarmos a declaração `IllegalAccessException` do método da subclasse o código acima será compilado normalmente. Você precisará saber o que é uma exceção verificada e não verificada, acrescentar exceção verificada em um sub-método gera um erro de compilação, enquanto que acrescentar uma exceção não verificada não causa nenhum efeito colateral. Se tiver alguma dúvida quanto a exceção, revise o capítulo 4.

**Anotações**

223

# Programação Orientada a Objetos

---

## Métodos finais não podem ser substituídos

Revisando o capítulo 2, um método final nunca poderá ser substituído, veja o código:

```
class Empregado {  
    private double salario;  
    final public double getSalario() {  
        return this.salario;  
    }  
}
```

```
class Horista extends Empregado {  
    public double getSalario() {  
        return -1;  
    }  
}
```

Erro de compilação: getSalario() in Horista cannot override getSalario() in Empregado; overridden method is final



# Programação Orientada a Objetos

---

## Uma subclasse usará `super.MetodoNome` para executar o método substituído pela subclasse

Isso é muito simples, examinando o código a seguir:

```
class Empregado {
    public double getSalario() {
        return -1;
    }
}

class Horista extends Empregado {
    public double getSalario() {
        return -2;
    }
    public void calcular() {
        System.out.println("Salario da superclasse: "+super.getSalario());
        System.out.println("Salario da subclasse: "+this.getSalario());
    }
    public static void main(String[] args) {
        Horista h = new Horista();
        h.calcular();
    }
}
```

O resultado será:

Salario da superclasse: -1.0  
Salario da subclasse: -2.0

# Programação Orientada a Objetos

---

## Sobreposição/Sobrecarga de método (overload)

### Métodos sobrepostos devem alterar a lista de argumentos

Para sobrepor um método, sua assinatura deve ser diferente, pois a identificação de um método é sua assinatura (lista de argumentos), analogicamente igual ao ser humano. Vejamos o código:

```
class Empregado {  
    public void calcularSalario() { }  
    public void calcularSalario(int value) { }  
    public void calcularSalario(long value) { }  
}
```

Observe que o método `calcularSalario` foi definido de três formas, mas não pense assim! Pois não é o mesmo método e sim três métodos distintos, com assinaturas diferentes. Isso é muito importante pois imagine que você queira definir a lógica de negócio dos métodos somente no método sem argumentos `calcularSalario()`, fazendo com que os demais somente o chamem - com isso você eliminaria a redundância de código que muitas vezes nos é incomodada.

### Tipos de retornos diferentes desde que a assinatura também seja

Nenhum problema você terá com o código abaixo, visto que as assinaturas são diferentes, portanto são métodos sobrepostos.

```
class Empregado {  
    public void calcularSalario() { }  
    public double calcularSalario(int value) {  
        return 2.0;  
    }  
    public double calcularSalario(long value) {  
        return 1.0;  
    }  
}
```

Não confunda com métodos substituídos que não podem ter assinaturas diferentes !

# Programação Orientada a Objetos

---

## Podem ter modificadores de acesso diferentes

Como são métodos distintos, cada um pode ter um modificador de acesso.

```
class Empregado {  
    public void calcularSalario() { }  
    protected void calcularSalario(int value) { }  
    private void calcularSalario(long value) { }  
    void calcularSalario(short value) { }  
}
```

Observe que os métodos acima tem quatro modificadores diferentes.

## Podem lançar exceções diferentes

Nenhum problema você terá se lançar exceções (verificadas ou não) em métodos sobrecarregados, mas repito novamente, não confunda com os métodos substituídos, pois as regras são outras.

```
class Empregado {  
    public void calcularSalario() throws CloneNotSupportedException { }  
    protected void calcularSalario(int value) throws IllegalAccessException { }  
    private void calcularSalario(long value) throws Exception { }  
    void calcularSalario(short value) { }  
}
```

## Métodos de uma superclasse podem ser sobrepostos em um subclasse

Um método de uma superclasse pode perfeitamente ser sobreposto em um subclasse como você pode ver no código a seguir:

```
abstract class Empregado {  
    abstract public void calcularSalario();  
}  
  
abstract class Horista extends Empregado {  
    abstract protected double calcularSalario(int value) throws Exception;  
}
```

**Anotações**

---

---

---

---

227

# Programação Orientada a Objetos

---

Observe também que fomos além, ou seja, aplicamos todas as regras de métodos sobreposto no método da classe *Horista* - mudamos seu modificador (*protected*), mudamos o retorno (*double*), acrescentamos uma exceção verificada, isso tudo em função da alteração da assinatura (*int value*). Com isso concluímos que um método sobreposto é um outro método, apesar de ter o mesmo nome, mas analogicamente quantos Josés conhecemos ? O que os identifica burocraticamente são suas assinaturas !

## O polimorfismo é aplicado à substituição e não a sobreposição

Essa afirmação pode parecer um pouco intrigante, mas é muito simples. Primeiro, vamos entender o que é polimorfismo. Segundo a maioria das bibliografias, polimorfismo é a capacidade que um método tem de responder de várias formas (como o próprio sentido etimológico da palavra). Bom, como aprendemos (pelo menos é o que esperamos), a sobreposição tem a função de permitir que você defina métodos distintos, ou seja, novos métodos, como você deve ter observado, na definição de polimorfismo acima, a palavra "um" foi destacada, para enfatizar bem, que é o mesmo método, único - portanto isso só pode ser aplicado à substituição. Vejamos o código abaixo para entender isso na prática:

```
abstract class Empregado {  
    protected double salario;  
    public abstract double getSalario();  
}  
  
class Horista extends Empregado {  
    public double getSalario() {  
        return this.salario/220;  
    }  
}  
  
class Mensalista extends Empregado {  
    public double getSalario() {  
        return this.salario;  
    }  
}
```

# Programação Orientada a Objetos

---

Observe que o método *getSalario* foi substituído (não sobreposto, a assinatura continuou idêntica), nas subclasses, mas quando você executar esse código verá que o método *getSalario* responderá de forma distinta quando chamado o da classe *Horista* e/ou da classe *Mensalista*. Talvez você possa estar se perguntando, "mas são métodos redefinidos, dois métodos distintos" - até podemos concordar em partes, mas existe uma mágica que você verá no código abaixo que encerrará essa questão, vejamos:

```
abstract class Empregado {
    protected double salario;
    public abstract double getSalario();
    public void setSalario(double value) {
        this.salario = value;
    }
}

class Horista extends Empregado {
    public double getSalario() {
        return this.salario/220;
    }
}

class Mensalista extends Empregado {
    public double getSalario() {
        return this.salario;
    }
}

public class Polimorfismo {
    public static void main(String[] args) {
        Empregado e;
        Horista h = new Horista();
        Mensalista m = new Mensalista();
        h.setSalario( 2240.00 );
        m.setSalario( 2240.00 );
        e = h;
        System.out.println( "Salario em horas: "+e.getSalario() );
        e = m;
        System.out.println( "Salario em horas: "+e.getSalario() );
    }
}
```

**Anotações**

229

# Programação Orientada a Objetos

---

Observe que na classe *Polimorfismo* o método *e.getSalario* foi chamado duas vezes no entanto o resultado desse código será:

Salario em horas: 10.181818181818182

Salario em horas: 2240.0

Existe muita discussão sobre esse assunto, mas não se preocupe, absorva o que achar correto, tenha opinião!

## **Você não pode criar um novo objeto sem chamar o seu construtor**

Para criar uma nova instância de uma classe (um objeto) você deverá chamar o seu construtor. Existe uma particularidade definida por um padrão de projeto chamada Singleton que trata isso um pouco diferente, mas não vamos entrar nesse estudo agora, visto que não é um objetivo para quem está almejando a certificação. A sintaxe para se criar um objeto é:

```
ClassName x = new ClassName();
```

Onde,

ClassName - é o nome da classe ou tipo

x - será o nome do objeto

## **Toda superclasse da árvore de herança de um objeto chamará um construtor**

Mesmo que você não defina um construtor padrão, o compilador se encarregará de fazer isso para você, independente do nível de herança de sua classe, saiba que esta sempre chama um construtor de sua superclasse imediata. Tá, sei, você está se perguntando, "pô, mas e se minha classe não herdar de ninguém ?" Toda classe a exceção da Object herda de alguma classe, se não herdar de ninguém o compilador coloca como superclasse a class java.lang.Object !

# Programação Orientada a Objetos

---

## Toda classe, mesmo as classes abstratas, tem pelo menos um construtor

Mesmo que você não defina um construtor para a sua classe, o compilador o fará, mesmo que sua classe seja abstrata, entenda porque. Se tivermos a hierarquia de classe a seguir:

```
abstract class Pessoa { }
```

```
abstract class PessoaFisica extends Pessoa { }
```

```
abstract class Empregado extends PessoaFisica { }
```

Nenhum construtor foi definido para essas classes, mas o compilador (como é camarada), modificou suas classes que ficaram assim:

```
abstract class Pessoa {  
    public Pessoa() {  
        super();  
    }  
}
```

```
abstract class PessoaFisica extends Pessoa {  
    public PessoaFisica() {  
        super();  
    }  
}
```

```
abstract class Empregado extends PessoaFisica {  
    public Empregado() {  
        super();  
    }  
}
```

Você pode achar estranho, criar construtores para classes abstratas, visto que, nunca serão instanciadas, mas isso é feito para respeitar a hierarquia da pilha de construtores.

# Programação Orientada a Objetos

---

## Construtores devem ter o mesmo nome da classe

Qualquer tentativa de definir um construtor com um nome diferente do nome da classe, gerará erro de compilação !

```
class Emp {  
    protected double salario;  
  
    public emp() { // nota 1  
        super();  
    }  
  
    public double getSalario() {  
        return this.salario;  
    }  
  
    public void setSalario(double value) {  
        this.salario = value;  
    }  
}
```

**nota 1:** erro: "invalid method declaration; return type required"

O erro se deu porque houve uma tentativa de definição do construtor como um nome diferente do nome da classe, o compilador subentendeu que isso seria a definição de um método, como não foi especificado nenhum retorno (mesmo que fosse void) gerou um erro de compilação.



# Programação Orientada a Objetos

---

## **Construtores não podem ter tipo retornos**

Qualquer tentativa de especificar um tipo de retorno a um construtor, automaticamente será considerado um método com o mesmo nome da classe.

```
class Emp {  
    protected double salario;  
  
    public void Emp() {  
    }  
  
    public double getSalario() {  
        return this.salario;  
    }  
  
    public void setSalario(double value) {  
        this.salario = value;  
    }  
}
```

Nesse caso *Emp()* não é um construtor e sim um método com o nome idêntico ao nome da classe.

## **Construtores podem ter qualquer modificador de acesso**

Um construtor poderá ser definido com qualquer modificador de acesso, inclusive `private` !

```
class Emp {  
    protected double salario;  
    private Emp() {  
    }  
    public double getSalario() {  
        return this.salario;  
    }  
    public void setSalario(double value) {  
        this.salario = value;  
    }  
}
```

**Anotações**

233

# Programação Orientada a Objetos

---

## Se nenhum construtor for criado o compilador criará um padrão

Se definirmos uma classe da seguinte forma:

```
class Emp {  
    protected double salario;  
    public double getSalario() {  
        return this.salario;  
    }  
    public void setSalario(double value) {  
        this.salario = value;  
    }  
}
```

Onde está seu construtor ? Não foi especificado pelo programador, porém o compilador o criará, deixando a classe compilada da seguinte forma:

```
class Emp {  
    protected double salario;  
    private Emp() {  
        super();  
    }  
    public double getSalario() {  
        return this.salario;  
    }  
  
    public void setSalario(double value) {  
        this.salario = value;  
    }  
}
```

Não pense que ele irá alterar o seu arquivo .java, ele só faz isso quando gerá o .class pois se você tiver um construtor padrão, o compilador não se entromete e não gera nenhum construtor.

# Programação Orientada a Objetos

---

## A primeira chamada de um construtor

Como boa prática de programação, um construtor deve primeiramente chamar um construtor sobrepostos com a palavra `this()` ou um construtor da superclasse imediata com a palavra chave `super()`, se não for definida explicitamente no construtor, o compilador terá o capricho de fazê-lo:

```
class Emp {
    protected double salario;
    Emp(double value) {
        super();
        setSalario( value );
    }

    Emp() {
        this(0);
    }

    public double getSalario() {
        return this.salario;
    }

    public void setSalario(double value) {
        this.salario = value;
    }
}
```

Muito interessante o que acontece nesse código, veja que existem dois modificadores, um que recebe um argumento (*double value*) e outro que não recebe nenhum argumento, ou seja, o programador que deseja utilizar essa classe pode instanciar essa classe de duas formas, esse exemplo mostra bem um bom caso de utilização para a sobreposição de métodos (mesmo que sejam construtores). Observe que o código de definição inicial do salário está sendo feito somente no construtor parametrizado pois quando o usuário chama um construtor sem argumento, esse chama o construtor sobreposto passando o valor 0 usando a palavra `this`. Com isso, podemos concluir que de qualquer forma que essa classe for instanciada, o construtor parametrizado será executado e consequentemente uma chamada a super construtor da superclasse também o será.

---

## Anotações

235

# Programação Orientada a Objetos

---

## O compilador adicionará uma chamada super

Se você definir um construtor em sua classe, o compilador não se afoitará e adicionará para você, mas como já foi falado varias vezes, um construtor deverá chamar sempre o construtor da sua superclasse (a exceção de uma chamada a um construtor sobreposto), isso porque o construtor da classe Object sempre deverá ser executado - provavelmente ele faz coisas maravilhosas que ainda não conhecemos, mas sempre, sempre mesmo, esse deverá ser executado, por isso vejamos o código a seguir:

```
class Emp {
    protected double salario;
    Emp(double value) {
        setSalario( value );
    }

    Emp() {
        this(0);
    }

    public double getSalario() {
        return this.salario;
    }

    public void setSalario(double value) {
        this.salario = value;
    }
}
```

O código de Object será executado nesse caso ? Claro que não ! Você não escreveu nenhuma chama super em nenhum construtor. Mas vamos nos atentar aos detalhes. Sabemos que essa classe é composta por dois construtores, se o usuário instancia da seguinte forma: `Emp e = new Emp(240);` - o objeto seria criado, setado o salário e não passaria pelo construtor sem parâmetros, pois foi chamado diretamente o construtor parametrizado, agora vejamos um outro caso, se a instanciação ocorresse da seguinte forma: `Emp e = new Emp();` - nenhum parâmetro foi passado, portanto o construtor que será executado será obviamente o construtor sem parametros que chama o construtor parametrizado através da palavrinha `this`, que seta o valor da variável `salario`. Seguindo o fluxo, podemos observar que sempre o construtor parametrizado será executado, independente das formas como fora instanciado a classe, portanto a inserção da chamada ao

**Anotações**

236

## Programação Orientada a Objetos

---

construtor superior imediato deverá ser nesse caso no construtor parametrizado como mostra o código abaixo, mesmo porque qualquer tentativa de inserir uma chamada a `super` no construtor sem parametro ocasionaria um erro de compilação.

```
class Emp {
    protected double salario;
    Emp(double value) {
        super();
        setSalario( value );
    }
    Emp() {
        this(0);
    }
    public double getSalario() {
        return this.salario;
    }
    public void setSalario(double value) {
        this.salario = value;
    }
}
```

Assim geraria um erro de compilação:

```
class Emp {
    protected double salario;
    Emp(double value) {
        super();
        setSalario( value );
    }
    Emp() {
        super();
        this(0); // Erro
    }
    public double getSalario() {
        return this.salario;
    }
    public void setSalario(double value) {
        this.salario = value;
    }
}
```

Erro de compilação: call to this must be first statement in constructor

**Anotações**

237

# Programação Orientada a Objetos

---

## As variáveis de instâncias só podem ser acessados após o construtor ser encerrados

Nenhuma referência a variáveis de instância (do objeto) pode ser feita antes do construtor ser chamado, como mostra o código abaixo:

```
public class Teste {  
    public static void main(String args[]) {  
        Emp e;  
        e.setSalario( 240 ); // Erro 1  
    }  
}
```

Erro de compilação 1: variable e might not have been initialized

```
class Emp {  
    protected double salario;  
    Emp(double value) {  
        super();  
        setSalario( value );  
    }  
    Emp() {  
        this(0);  
    }  
  
    public double getSalario() {  
        return this.salario;  
    }  
  
    public void setSalario(double value) {  
        this.salario = value;  
    }  
}
```

Se você lembra bem do capítulo 2 deve constatar isso, devido ao fato de existirem as variáveis estáticas ! Claro que não, as variáveis estáticas não são variáveis de instância (de objeto) e sim da classe !

# Programação Orientada a Objetos

---

## As classe abstratas tem construtores que são chamados quando as classe concretas são instanciadas

Mesmo que não as classes abstratas nunca serão instanciadas, seus construtores são executadas na pilha de execução pelas suas subclasses concretas, por isso que o compilador acrescenta caso você não os tenha acrescentado.

## Interfaces não tem construtores

Como as interfaces tem outro papel na orientação a objetos, elas não possuem construtores.

## Construtores nunca são herdados

Isso nos leva a concluir que nunca poderão ser substituídos.

## Um construtor não pode ser chamado por um método da classe

Os construtores nunca poderão ser chamados de métodos de uma classe, como mostra o código abaixo:

```
class Emp {  
    private double salario;  
    public void calcular() {  
        super(0);    // Erro  
    }  
}
```

Erro de compilação: call to super must be first statement in constructor

# Programação Orientada a Objetos

---

Somente através de um outro construtor:

```
class Emp {
    private double salario;
    Emp(double value) {
        super();
        setSalario( value );
    }
    Emp() {
        this(0);      // ok - um construtor pode chamar outro
    }
    public double getSalario() {
        return this.salario;
    }
    public void setSalario(double value) {
        this.salario = value;
    }
}
```

## Regras para chamadas a construtores

**Deve ser a primeira instrução de um construtor**

```
class Emp {
    private double salario;
    Emp(double value) {
        setSalario( value );
        super();      // Erro
    }
    Emp() {
        this(0);
    }
    public double getSalario() {
        return this.salario;
    }
    public void setSalario(double value) {
        this.salario = value;
    }
}
```

Erro de compilação: call to super must be first statement in constructor

**Anotações**

240



# Programação Orientada a Objetos

---

## A assinatura determina qual construtor será chamado

Como é permitido a sobreposição de construtores, a assinatura é que vai determinar qual o construtor será chamado.

## Um construtor pode chamar outro

Para evitar redundância de código, você pode criar vários construtores com assinaturas diferentes mas definir somente um para iniciar seu objeto realmente, onde os demais construtores somente o chamem, como mostra o código a seguir:

```
class Emp {
    private double salario;
    Emp(double value) {
        super();
        setSalario( value );
    }
    Emp() {
        this(0);
    }
    public void setSalario(double value) {
        this.salario = value;
    }
}
```

No exemplo acima, o construtor sem parâmetro está chamando o construtor na própria classe (através da palavra *this*) que está chamando o construtor de sua superclasse (nesse caso *Object*) e depois setando o valor de variável de instância *salario*. Note que a atribuição à variável *salario* poderia ser feita diretamente no código do construtor *Emp()*, mas se resolvêssemos adicionar algum código nos construtores dessa classe, o deveríamos fazer em dois ou mais lugares, o que causaria uma certa redundância de código, no caso acima não, o código que precisa se alterado é somente do construtor *Emp(double value)*, pois os demais (nesse caso um) o chamam.

# Programação Orientada a Objetos

---

## this e super não podem no mesmo construtor

Você nunca poderá ter em um único construtor duas chamadas uma a `this()` e outra a `super()`. Ou você tem um ou outro, qualquer código escrito de forma que discorde dessa linha de pensamento é considerado errado, e o compilador não permitirá tal façanha.

## Tipos de retorno

### Métodos sobrepostos podem alterar seu retorno.

Observe que no código abaixo houve uma sobrecarga de método, por isso o retorno pode ser alterado sem nenhum problema.

```
abstract class Empregado {  
    abstract public int calcularReajuste(int value);  
    abstract public double calcularReajusta(double value);  
}
```

Não tente fazer isso com substituição de métodos como mostra o código a seguir:

```
abstract class Empregado {  
    public abstract double getSalario();  
}  
  
class Horista extends Empregado {  
    public float getSalario() { // Erro  
        return 240f;  
    }  
}
```

Erro de compilação: `getSalario() in Horista cannot override getSalario() in Empregado; attempting to use incompatible return type`

# Programação Orientada a Objetos

---

## null é um valor aceitável para métodos que retornem objetos

```
abstract class Empregado {
    public abstract String getNome();
}

class Horista extends Empregado {
    public String getNome() {
        return null;
    }
}
```

Nenhum problema no código acima.

## Retorno de tipos primitivos

Quando um método retornar um tipo primitivo, qualquer valor que poderá implicitamente convertido para o tipo do retorno, poderá ser retornado sem nenhum problema.

```
abstract class Empregado {
    public abstract double getSalario();
}

class Horista extends Empregado {
    public double getSalario() {
        float rc = 240.0f;
        return rc;
    }
}
```

Veja que no código acima, o retorno do método é um tipo double, no entanto o que está sendo retornado é um float, como essa conversão é implícita, esse código é compilado e executado sem nenhum problema.

# Programação Orientada a Objetos

---

Agora, vamos observar o seguinte código:

```
abstract class Empregado {
    public abstract float getSalario();
}

class Horista extends Empregado {
    public float getSalario() {
        double rc = 240.0;
        return rc;    // Erro
    }
}
```

Erro de compilação: possible loss of precision

## Tipo de retorno void

Um método pode não retornar nada (void), você poderá perfeitamente criar um método que não tenha nenhum retorno, basta colocar a palavra void no seu retorno, como veremos no código abaixo:

```
public class Empregado {
    public void calcularSalario() { }
}
```

O método acima não retorna nenhum valor, apesar de processar algum código, você ainda poderá inserir um return (sem nenhum valor) no corpo do método se quiser encerrar a sua execução antes do fluxo normal, como o código abaixo:

```
public class Empregado {
    public void calcularSalario() {
        if ( this.count <= 0 ) return;
        // faz os cálculos
    }
}
```

O código acima, testa se a variável count é menor ou igual a 0, se for encerra o método ! É fora de nosso foco questionar se isso é um código bem feito ou não, claro que não faríamos algo assim, mais a questão é a utilização da palavra return no meio do corpo de um método e que não retorna nada (void).

**Anotações**

244

# Programação Orientada a Objetos

---

Observe o código abaixo:

```
public class Empregado {  
    public void calcularSalario() {  
        if (count <= 0) return -1; // Erro  
        // faz os cálculos  
    }  
}
```

Erro de compilação: cannot return a value from method whose result type is void

Qualquer tentativa de retornar um valor em um método com o tipo de retorno void, causará um erro de compilação.

## Retornando classes

Se um método retorna uma classe X e Y é uma subclasse de X, Y pode ser retornado sem maiores problemas nesse método.

Vejamos:

```
abstract class Empregado {  
}  
  
class Horista extends Empregado {  
    public Empregado getObject() {  
        return this;  
    }  
}
```

Observe que o retorno de getObject() é uma classe Empregado e o que está sendo retornado no métodos é this que refere-se a um objeto da classe Horista, e o código acima não seria nenhum problema para o grande compilador !

Agora vamos a parte mais interessante desse capítulo:

Qual é o número desse capítulo 5 !

Quantas vezes o Brasil é campeão: 5 !

---

**Anotações**

245

## Resumos para certificação - Encapsulamento, relacionamento É-UM e TEM-UM

- O objetivo do encapsulamento é ocultar a implementação existente por trás de uma interface(ou API);
- O código encapsulado oferece dois recursos:
  - ❑ As variáveis de instância são mantidas protegidas(geralmente com o modificador private);
  - ❑ Os métodos capturadores e configuradores fornecem acesso a variáveis de instância.
- O relacionamento É-UM se refere à herança;
- O relacionamento É-UM é representado pela palavra chave extends;
- É-UM, herda de, é derivado de, e, é um subtipo de, são todas expressões equivalentes;
- TEM-UM significa que a instância de uma classe “tem uma” referência à instância de outra classe.

## Substituição e Sobreposição

- Os métodos pode ser substituídos ou sobrepostos; os construtores podem ser sobrepostos, mas não substituídos;
- Os métodos abstratos devem ser substituídos pela primeira subclasse concreta(não abstrata);
- No que diz respeito ao método substituído, o método novo:
  - ❑ Deve ter a mesma lista de argumento;
  - ❑ Deve ter o mesmo tipo de retorno;
  - ❑ Não deve ter modificador de acesso mais restritivo;
  - ❑ Pode ter modificador de acesso menos restritivo;

Anotações

246

- ❑ Não pode lançar exceções verificadas novas ou mais abrangentes;
- ❑ Pode lançar exceções mais restritivas ou menos abrangentes, ou qualquer exceção não verificada;
- ❑ Os métodos finais não podem ser substituídos;
- ❑ Só os métodos herdados podem ser substituídos;
- ❑ Uma subclasse usará `super.overrideMethodName()` para chamar a versão da superclasse de um método substituído;
- ❑ Sobreposição significa reutilizar o mesmo nome de método, mas com argumentos diferentes;
- ❑ Os métodos sobrepostos:
  - Devem ter listas de argumentos diferentes;
  - Podem ter tipos de retorno diferentes, contanto que as listas de argumentos também sejam diferentes;
  - Podem ter modificadores de acesso diferentes;
  - Podem lançar exceções diferentes.
- Os métodos de uma superclasse podem ser sobrepostos em uma subclasse.
- O polimorfismo é aplicável a substituição e não a sobreposição;
- O tipo de objeto determina que o método substituído será usado no tempo de execução;
- O tipo de referência determina que o método sobreposto será usado em tempo de compilação.

# Programação Orientada a Objetos

---

## Instanciação e Construtores

- Os objetos são construídos:
  - ❑ Você não pode criar um novo objeto sem chamar um construtor;
  - ❑ Toda superclasse da árvore de herança de um objeto chamará um construtor.
  - ❑ Toda classe, mesmo as classes abstratas, tem pelo menos um construtor;
- Os construtores ter o mesmo nome da classe;
- Os construtores não têm tipo de retorno. Se houver, então, será simplesmente um método como o mesmo nome da classe e não um construtor;
- A instanciação do construtor ocorrerá da maneira a seguir:
  - ❑ O construtor chamará o construtor de sua superclasse, que chamará o construtor de sua superclasse e assim por diante, até alcançar o construtor de Object;
  - ❑ O construtor de Object será executado, retornando em seguida ao construtor chamador, que será processado até sua conclusão, e por sua vez retornará ao construtor que o chamou, dando prosseguimento a essa sequência até a execução do construtor da instância que tiver sendo criada;
- Os construtores podem usar qualquer modificador de acesso (até private).
- O compilador criará um construtor padrão se você não criar nenhum construtor em sua classe;
- O construtor padrão é um construtor sem argumentos com uma chamada também sem argumentos a `super()`;
- O compilador adicionará uma chamada a `super()` se você não o fizer, a menos que já tenha inserido uma chamada a `this()`;
- Os métodos e variáveis de instância só podem ser acessados depois de o

Anotações

248



# Programação Orientada a Objetos

---

construtor da superclasse ser executado;

- As classes abstratas possuem construtores que são chamados quando a subclasse concreta é instanciada;
- As interfaces não tem construtores;
- Se sua superclasse não tiver um construtor sem argumentos, você terá que criar um construtor e inserir uma chamada a `super()` com argumentos que coincidam com os do construtor da superclasse.
- Os construtores nunca são herdados, portanto, não podem ser substituídos;
- Um construtor só pode ser chamado diretamente por outro construtor (usando uma chamada a `super()` ou `this()`);
- Questões relacionadas a `this()`:
  - Só podem aparecer como a primeira instrução do construtor;
  - A lista de argumentos determina que construtor sobreposto será chamado;
  - Os construtores podem chamar outros construtores que podem chamar ainda outros e assim por diante, mas, cedo ou tarde é melhor que um deles chame `super()` ou a pilha excederá o limite;
  - `This()` e `super()` não podem ficar no mesmo construtor. Você pode ter um ou outro, mas nunca os dois.

## Tipos de retorno

- Os métodos sobrepostos podem alterar os tipos de retorno, os substituídos não podem ;
- Os tipos de retorno de referência a objetos podem aceitar null como um valor retornado;
- O array é um tipo de retorno válido como valor a ser declarado ou retornado;
- Para métodos com tipos de retorno primitivos, qualquer valor que possa ser implicitamente convertido no tipo do retorno poderá ser retornado;

**Anotações**

249

## Programação Orientada a Objetos

---

- Um tipo void não pode retornar nada, mas é permitido não retornar nada. Você pode inserir return em qualquer método com tipo de retorno void para sair dele antes que termine. Porém não é permitido não retornar nada de um método com um tipo de retorno que não seja void;
- Para métodos com referência a um objeto como tipo de retorno, uma subclasse desse tipo poder ser retornada.

Anotações

250

# Programação Orientada a Objetos

---

## Capítulo VI - java.lang - a classe Math, Strings e Wrappers

### String

Objetos String são imutáveis, porém as variáveis de referências não.

Essa afirmação pode ser contenciosa mais não é. Quando um objeto String é criado na memória, ele não poderá ser alterado, vejamos o código:

```
String s1 = "abc";  
String s2 = s1;  
s1 = "abcd";
```

Observe que ambas as variáveis de referências apontam (ops! isso não é C, referenciam) para o mesmo objeto na memória, se listássemos o valor das variáveis s1 e s2 obteríamos os resultados:

```
s1 -> "abcd"  
s2 -> "abc"
```

Note que a string "abc" não foi alterada, pois s1 esta agora referenciando um outro objeto "abcd" ! Enquanto que s2 continua referenciando "abc".

Verbalmente esse código diz o seguinte: "JVM, por favor, crie uma string no pool de strings (depois detalharemos sobre isso) com o valor "abc" obrigado.". Agora, s2 guarda o endereço físico da mesma string "abc" (nenhum string foi criada no pool), na terceira linha, soará uma voz: "Crie uma string "abcd" (veja que não foi alterada a primeira string) onde s1 a referenciará, obrigado.". Com isso podemos concluir que s2 continua referenciando a string "abc", mas que a referencia s1 foi alterada e a string "abc" não.

Se um nova String for criada e não for atribuída a nenhuma variável de referência, ela ficará perdida.

O consumo de memória é um dos pontos críticos para que um programa seja bem sucedido, o uso incorreto da "escassa memória" (a não ser que seu programa vá rodar em um equipamento com 32 gb e memória) pode gerar falhas irreparáveis em um programa, com essa preocupação a Sun (criadora da especificação Java se você não sabe), criou um pool de String que armazena todas as strings coincidentes em um programa Java, ou seja, se você tiver duas variáveis que referenciam uma string "kuesley é lindo" saiba que você só terá uma string no

**Anotações**

251

# Programação Orientada a Objetos

---

pool, porém duas referências e, é por esta razão que as strings são inalteráveis. Mas se você criar um string e não especificar nenhuma variável de referência ela ficará perdida - o que os criadores da linguagens não gostarão nem um pouquinho se olharem para o seu código, pois você estaria desprezando todo o esforço deles em otimizar o uso da memória. Vejamos um exemplo prático disso:

```
String s1 = "abc";  
String s2 = "abcd" + "e";
```

Saiba que o código acima criará no pool 4 strings, sendo que duas ficaram perdidas ("abcd","e"), pois somente as string "abc" e "abcde" estão sendo referenciadas.

Se você redirecionar a referencia de uma String para outra String, o objeto String anterior poderá ser perdido

```
String s1 = "abcd";  
s1 = "xuxu";
```

O que o código acaba de fazer é criar uma string no pool "abcd" e referenciá-lo por s1, depois criar um nova string "xuxu" e referenciá-lo pela mesma variável s1, portanto, a string "abcd" ficará perdida no pool.

## O método substring

O método que erroneamente foi definido com substring - pois deveria ser subString, recebe dois argumentos do tipo inteiro, onde define a posição inicial da string (começando do 0) e onde deve terminar a string (começando de 1), isso quer dizer que:

```
"abcdefghijlmnop".substring(0,3); // resulta em: abc
```

Pois o primeiro argumento é considerado o início em 0 e o segundo o início é considerado em 1, lembre-se disso.

```
String s = "kuesley e lindo";  
System.out.println(s.substring(8,15));
```

Resultado: "e lindo"

---

## Anotações

252

# Programação Orientada a Objetos

---

## A classe String é final

Nenhum método da classe String pode ser substituído, visto que a classe é final.

## Métodos da String

**concat** - Adiciona uma string a outra, porém não altera a string em que o método está sendo executado:

```
String s = "teste";  
s.concat("nao mudou");  
System.out.println(s);      // Resultado "teste"
```

Note que nenhuma alteração foi efetuada na string s. Se desejássemos alterar s, o código deveria ser da seguinte forma:

```
String s = "teste";  
s = s.concat("nao mudou");  
System.out.println(s);      // Resultado "teste nao mudou"
```

**equalsIgnoreCase** - testa se uma string é igual a outra ignorando a diferença entre letras maiúsculas e minúsculas:

```
String sl = "teste";  
String su = "TESTE";  
System.out.println(sl.equals(su));      // resultado: false  
System.out.println(sl.equalsIgnoreCase(su)); // resultado: true
```

**length** - Obtém o tamanho da string.

```
String nome = "angeline jolie";  
System.out.println(nome.length());      // resultado: 14
```

**replace** - Substitui os caracteres de uma string.

```
String texto = "Est4mos 4qui p4r4 test4r";  
texto = texto.replace('4','a');  
System.out.println(texto); // resultado: Estamos aqui para testar
```

---

## Anotações

253

# Programação Orientada a Objetos

---

**substring** - Extrai uma string de outra.

```
String texto = "0123456789";  
System.out.println(texto.substring(0,3)); // resultado: 012
```

Já falamos anteriormente, mas queremos ser chatos mesmo, o primeiro argumento considerada a string iniciando na posição 0, enquanto que o segundo em 1 - por isso que o resultado anterior é "012".

Se for especificado um valor inválido para qualquer um dos dois argumento, uma exceção será lançada.

```
String texto = "0123456789";  
System.out.println(texto.substring(0,11)); // exceção
```

Uma exceção será lançada: java.lang.StringIndexOutOfBoundsException: String index out of range: 11

**toLowerCase** - Muda todas as letras que estiverem maiúscula para letra minúscula.

```
String s = "AKJDJE";  
s.toLowerCase();  
System.out.println(s);
```

Qual o resultado do trecho de código acima ?

- a) Erro de compilação
- b) Exceção
- c) AKJDJE
- d) Akjdje
- e) akjdje

Errou ??? Não esqueça que a string são imutáveis. Portanto s não está sendo reatribuído.

**toUpperCase** - Processo inverso do toLowerCase, ou seja, transforma em maiúscula todas as letras que estiverem minúscula.

```
String s = "teste de caixa alta";  
s = s.toUpperCase();  
System.out.println(s); // resultado: TESTE DE CAIXA ALTA
```

**Anotações**

254

# Programação Orientada a Objetos

---

**trim** - Retira espaços das extremidades de uma string.

```
String s = " tem um ";  
s = s.trim()  
System.out.println(">"+s.trim()+"");    // resultado: *tem um*
```

**toString** - retorna o valor da string. (método inútil : ) )

Como a classe String é derivada de Object esse método é substituído na classe String retornando o valor da String propriamente dita.

```
String s = "mesma coisa";  
System.out.print(s.toString());    // resultado: mesma coisa
```

**equals** - compara o valor de uma string com outra informada em argumento.

```
String s1 = "texto";  
String s2 = "texto";  
System.out.println(s1 == s2);        // resultado 1: true  
System.out.println(s1.equals(s2));    // resultado 2: true  
s1 = new String("texto");  
s2 = new String("texto");  
System.out.println(s1 == s2);        // resultado 3: false  
System.out.println(s1.equals(s2));    // resultado 4: true
```

O resultado 1 é true em função da string "texto" ser criada uma única vez e ser referenciado pelos objetos s1 e s2. O resultado 2 é verdadeiro pois o método equals está comparando os valores. Já o resultado 3 é falso, pois como os objetos foram criados usando a palavra chave new você terá dois objetos na memória apesar de terem os mesmos valores, já o resultado 4 é true pois o método equals sempre irá comparar os valores, como ambos os objetos tem os mesmos valores o resultado é true. Com isso podemos concluir que, o operador == compara se os variáveis de referência estão referenciando o mesmo objeto, se for o caso retorna true caso contrário retorna falso, enquanto que o método equals da classe String sempre irá comparar os valores (não pense que esse método funciona da mesma forma para a classe StringBuffer, isso é o oitavo pecado capital).

# Programação Orientada a Objetos

---

## StringBuffer

Diferente da classe String, a classe StringBuffer pode sofrer alteração.

```
StringBuffer sb = new StringBuffer("testando");  
sb.append(" se mudou");  
System.out.println(sb);
```

Resultado: testando se mudou

Observe que o valor de sb foi alterado mesmo que não tenha sido atribuído à sb, ou seja, o método append atuou sobre sb.

## StringBuffer - métodos importantes

**append** - esse método adiciona uma string ao StringBuffer

```
StringBuffer sb = new StringBuffer("abc");  
sb.append("def");  
System.out.println(sb);    // Resultado: abcdef
```

**insert** - insere uma string em um StringBuffer, começando em 0 (zero)

```
StringBuffer sb = new StringBuffer("abc");  
sb.insert(1,"x");  
System.out.println(sb);    // Resultado: axbc
```

Se um valor inválido for inserido no primeiro argumento, uma exceção será lançada.

Como você pode conferir no exemplo abaixo:

```
StringBuffer sb = StringBuffer sb = new StringBuffer("abc");  
sb.insert(4,"x");  
System.out.println(sb);
```

Um exceção será lançada: `StringIndexOutOfBoundsException`, não esqueça que a posição tem início em 0 (zero).

---

## Anotações

256



# Programação Orientada a Objetos

---

Uma observação vale ser ressaltada, visto que você poderá se deparar com uma questão como:

```
StringBuffer sb = new StringBuffer("abc");
sb.insert(3,"x");
System.out.println(sb);
```

Você pode pensar, "bom se posição inicia em 0 (zero), então o limite para o primeiro argumento é 2, pois é o índice maior dessa string, mas o método deve possibilitar uma inserção após a letra "c" então qual seria o índice ??? 3 - exatamente. Nenhum problema ocorrerá no código acima, e o resultado seria: "abcx"

**reverse** - inverte todos os caracteres da StringBuffer.

```
StringBuffer sb = new StringBuffer("kuesley");
sb.reverse();
System.out.println(sb);    // resultado: yelseuk
```

**toString** - retorna o valor do objeto StringBuffer, esse método é herdado de Object.

```
StringBuffer sb = new StringBuffer("kuesley");
sb.reverse();
System.out.println(sb);    // resultado: yelseuk
System.out.println(sb.toString()); // resultado: yelseuk
```

Exatamente o mesmo resultado do próprio objeto.

## equals

O método equals da classe StringBuffer não é substituído, isso significa que ele não compara valores, portanto uma questão com essa consideração pode confundir, vejamos:

```
StringBuffer sb1 = new StringBuffer("kuesley");
StringBuffer sb2 = new StringBuffer("kuesley");
System.out.println( sb1.equals(sb2) );    // resultado: false
```

Apesar de conterem o mesmo valor, o resultado será false pois é comparado se

**Anotações**

257

# Programação Orientada a Objetos

---

as variáveis, referenciam o mesmo objeto na memória, e como existem 2 objetos distintos na memória, o resultado é false.

## Encadeamento de métodos

Você poderá se deparar também com questão que encadenham métodos como:

```
StringBuffer sb = new StringBuffer("kuesley");  
System.out.println(sb.insert(sb.length()," é lindo").reverse().toString());
```

Qual seria o resultado ?

- a) Uma exceção será lançada
- b) Erro de compilação
- c) KUESLEY É LINDO
- d) kiesley é lindo
- e) onil é yelseuk

Observe que os métodos foram executados da ESQUERDA para a DIREITA exatamente como se lê, ou seja, o primeiro método que foi executado foi o insert alterando o objeto sb, mudando a StringBuffer para "kuesley é lindo", depois foi invertido a string "odnil é yelseuk" depois o método toString que, aqui pra nós, não faz nada !!!

## Usando a classe Java.lang.Math

### abs

Retorna o valor absoluto de um argumento, veja suas assinaturas:

```
public static int abs( int value )  
public static long abs( long value )  
public static float abs(float value )  
public static double abs(double value )
```

Observe que ele é sobreposto para aceitar qualquer tipo primitivo a exceção de double e float. O resultado sempre será um número positivo a exceção de dois casos, mas como ainda não sei, não colocarei aqui, mas tem !

Agora já tô sabendo, Hhehehe

Se o valor informado for menor que Integer.MIN\_VALUE ou Long.MIN\_VALUE veja o exemplo a seguir:

```
System.out.println("Valor absoluto para "+Math.abs(-2147483648));  
// Resultado: -2147483648
```

```
System.out.println("Valor absoluto para "+Math.abs(-9223372036854775808L));  
// Resultado: -9223372036854775808L
```

```
System.out.println("Valor absoluto para "+Math.abs(-2147483647));  
// Resultado: 2147483647
```

```
System.out.println("Valor absoluto para "+Math.abs(-9223372036854775807L));  
// Resultado: -9223372036854775807
```

Onde:

```
Integer.MIN_VALUE -> -2147483648  
Long.MIN_VALUE -> -9223372036854775808
```

Note que quando o valor informado for igual ao valor de Integer.MIN\_VALUE e Long.MIN\_VALUE a função não retorna o valor absoluto, isso é só para complicar nossa vida.

### Anotações

259

# Programação Orientada a Objetos

---

## ceil

Retorna o número "ponto flutuante inteiro" superior mais próximo.

```
public static float ceil(float value)
public static double ceil(double value)
```

```
System.out.println(Math.ceil(3.8));    // Resultado: 4.0
System.out.println(Math.ceil(-3.3));   // Resultado: -3.0
```

## floor

Retorna o número ponto flutuante inteiro inferior mais próximo.

```
public static float floor(float value)
public static double floor(double value)
```

```
System.out.println(Math.floor(-10.9)); // Resultado: -11.0
System.out.println(Math.floor(12.2));  // Resultado: 12.0
```

## max

Esse método retorna o maior número entre dois informados, sua assinatura de métodos é:

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

Observe que não existe assinatura para os tipos byte, short porque esse podem implicitamente ser convertidos para int.

```
System.out.println("O maior entre -8 e -9 é: "+Math.max(-8,-9)); // Resultado: -8
System.out.println("O maior entre -8 e 9 é: "+Math.max(-8.0,9)); // Resultado: 9.0
```

Por mais que não exista um método com as assinaturas (double, int) seguindo as regras de valores literais da Java, o valor 9 é convertido implicitamente para double por isso o resultado é: 9.0 um tipo double.

## Anotações

260

# Programação Orientada a Objetos

---

## min

Esse método retorna o menor número entre dois informados, suas assinaturas são:

```
public static int min(int a, int b )
public static long min(long a, long b )
public static float min(float a, float b )
public static double min(double a, double b )
```

Observe que não existe assinatura para os tipos byte, short porque esse podem implicitamente ser convertidos para int.

```
System.out.println("O menor entre -8 e -9 é: "+Math.max(-8,-9)); // Resultado: -9
System.out.println("O menor entre 8 e -9 é: "+Math.max(8,-9.0)); // Resultado:
8.0
```

Por mais que não exista um método com as assinaturas (int, double) seguindo as regras de valores literais da Java, o valor 8 é convertido implicitamente para double por isso o resultado é: 8.0 um tipo double.

## round

Arredonda um numero ponto flutuante recebido como argumento, veja suas assinaturas:

```
public static int round(float a)
public static long round(double a)
```

Note que como sua função é arredondar, deve retornar um tipo inteiro, porém como o argumento pode ser um float (32 bits) ou um double (64 bits) o retorno deve suportar essa quantidade de bits por isso o tipo de retorno muda conforme o argumento de entrada.

# Programação Orientada a Objetos

---

```
class Teste {  
    public static void main(String args[]) {  
        System.out.println(Math.round(9.4));           // Resultado: 9  
        System.out.println(Math.round(9.5));           // Resultado: 10  
        System.out.println(Math.round(9.9));           // Resultado: 10  
        System.out.println(Math.round(-9.9));          // Resultado: -10  
    }  
}
```

Para entender, vamos dividir em duas regras:

## 1) Número positivo:

Arredondando um número positivo: o algoritmo soma ao número informado 0.5 e trunca o número, como podemos ver a seguir:

3.9	+	0.5	=	4.4	// Resultado: 4
3.5	+	0.5	=	4.0	// Resultado: 4
3.4	+	0.5	=	3.9	// Resultado: 3

Funciona exatamente como os métodos `Math.ceil` (se o número decimal for maior ou igual a 0.5) e `Math.floor` (se o número decimal for menor que 0.5) !

## 2) Número negativo:

-3.9	+	0.5	=	-4.4	// Resultado: -4
-3.4	+	0.5	=	-3.9	// Resultado: -3
-3.5	+	0.5	=	-4.0	(deveria), mas o resultado é: -3

Pode parecer estranho o fato de que no primeiro caso o resultado foi -4 onde o esperado fosse -3, porém nesse caso o `round` ignora o sinal, ou seja, trabalha com números absolutos - porém como os criados da linguagem java gostam de complicar, existe um exceção no caso de número negativo quando o valor decimal do número for 0.5 o número será arredondado para cima.

Vejamos exemplos:

```
Math.round(-3.5) // Resultado: -3  
Math.round(3.5)  // Resultado: 4
```

```
Math.round(-3.4) // Resultado: -3  
Math.round(-3.6) // Resultado: -4
```

---

## Anotações

---

---

---

---

262

# Programação Orientada a Objetos

---

## random

O método Math.random retorna um número aleatório entre 0.0 e menor que 1.0.

```
public static double random()
```

```
System.out.println( Math.random() );    // Resultado: qualquer valor entre 0.0  
(inclusive) e menor que 1.0
```

## sin

Retorna o seno de um ângulo, para o exame você terá que saber como calcular o seno de qualquer espécie de ângulo, precisará ser um matemático e físico (calma, não desista, só estava brincando) ! Você não precisará saber como se calcula nenhum ângulo, basta saber que recebe um argumento que é um valor de um ângulo em radiano.

```
public static double sin(double a)
```

## cos

Calcula o co-seno de um ângulo.

```
public static double cos(double a)
```

## tan

Retorna a tangente de um ângulo.

```
public static double tan(double a)
```

Lembre-se que esse três últimos métodos recebe um valor de grau em radiano, não tente passar por exemplo 90" como parâmetro que você poderá ser surpreendido com um resultado: prédio no chão !!! (os engenheiros que me perdoe)

---

## Anotações

263

# Programação Orientada a Objetos

---

## **sqrt**

Retorna a raiz quadrada de um número.

```
public static double sqrt(double a)
```

```
System.out.println(Math.sqrt(9)); // Resultado: 3
```

Como você é um cara esperto, deve estar se perguntando: "E se tentar passar um número negativo para obter a raiz quadrada ? " - aahah, eu sabia que você questionaria isso. Como todos sabem, pelo menos uma grande maioria, tá bom, algumas pessoas, ou melhor um grupo bem seleto, ufa!, sabe que não existe raiz quadrada de um número negativo, portanto se você passar -9 para esse método, você obterá um resultado NaN (Not a Number - ou Não é um número).

## **toDegrees**

Retorna um valor de um ângulo em graus, para isso você deve passar um ângulo em radiano.

```
public static double toDegrees(double a)
```

```
Math.toDegrees(Math.PI * 2.0) // Retorna: 360.0
```

## **toRadians**

Retorna em radiano um ângulo informado em graus.

```
public static double toRadians(double a)
```

Exemplo:

Calculando o seno de um ângulo

```
System.out.println("O seno de 90° é: "+Math.sin(Math.toRadians(90.0)));  
// Resultado: 1.0
```

Note que usamos dois métodos encadeados para que o cálculo fosse realizado,

## **Anotações**

264



# Programação Orientada a Objetos

---

visto que o método `sin` recebe um valor em radiano e por isso usamos o método `toRadians` para converter 90.0 (em graus) para radiano.

Algumas observações sobre a classe `Math`

```
double x;
```

```
float p_i = Float.POSITIVE_INFINITY;
```

```
double n_i = Double.NEGATIVE_INFINITY;
```

```
double n_a_n = Double.NaN;
```

```
if ( n_a_n != n_a_n ) System.out.println("NaN é diferente de NaN");  
// Será ecoada, pois NaN não é igual a nada inclusive a NaN
```

```
if (Double.isNaN(n_a_n)) System.out.println("é um NaN");  
// resultado: é um NaN
```

```
x = Math.sqrt(n_i);  
// Alerta geral ! será atribuído NaN para x
```

```
if (Double.isNaN(x)) System.out.println( "x é um NaN");  
// Resultado: x é um NaN
```

```
System.out.println( 32 / 0 );  
// Resultado: java.lang.ArithmeticException
```

```
System.out.println( 32.0 / 0.0 );  
// Resultado: Infinity
```

```
System.out.println( -32.0 / 0.0 );  
// Resultado: -Infinity
```

```
System.out.println( 32.0 / -0.0 );  
// Resultado: -Infinity
```

```
System.out.println( -32.0 / -0 );  
// Resultado: -Infinity
```

```
System.out.println( 32.0 / -0 );
```

**Anotações**

265

# Programação Orientada a Objetos

---

// Resultado: Infinity

```
System.out.println( -32.0 / -0.0 );
```

// Resultado: Infinity (ops! jogo de sinal, vamos entender isso mais abaixo)

É bom ressaltar uma regra ou melhor exceção quando envolvemos números ponto-flutuante em Java.

Observe o seguinte programa e responda o que acontece quando tentamos executar o seguinte código:

```
1. public class Test {  
2.     public static void main(String[] args) {  
3.         System.out.println( " -32.0 / -0.0 -> " + -32.0 / -0.0 );  
4.         System.out.println( " -32.0 / 0.0 -> " + -32.0 / 0.0 );  
5.         System.out.println( " 32.0 / 0.0 -> " + 32.0 / 0.0 );  
6.         System.out.println( " 32.0 / -0.0 -> " + 32.0 / -0.0 );  
7.         System.out.println( " -32 / -0.0 -> " + -32 / -0.0 );  
8.         System.out.println( " 32 / -0.0 -> " + 32 / -0.0 );  
9.         System.out.println( " -32 / 0.0 -> " + -32 / 0.0 );  
10.        System.out.println( " 32 / 0.0 -> " + 32 / 0.0 );  
11.        System.out.println( " -32.0 / 0 -> " + -32.0 / 0 );  
12.        System.out.println( " -32.0 / -0 -> " + -32.0 / -0 );  
13.        System.out.println( " 32.0 / 0 -> " + 32.0 / 0 );  
14.        System.out.println( " 32.0 / -0 -> " + 32.0 / -0 );  
15.        System.out.println( " -32 / -0 -> " + -32 / -0 );  
16.        System.out.println( " -32 / 0 -> " + -32 / 0 );  
17.        System.out.println( " 32 / -0 -> " + 32 / -0 );  
18.        System.out.println( " 32 / 0 -> " + 32 / 0 );  
19.        System.out.println( "" );  
20.    }  
21. }
```

- a) Erro de compilação
- b) java.lang.ArithmeticException na linha 17
- c) Mostrará os valores fazendo jogo de sinal para as divisões
- d) Erro de compilação na linha 19
- e) java.lang.ArithmeticException na linha 15

---

**Anotações**

266

# Programação Orientada a Objetos

---

Entendendo como Java trata a divisão por 0 (zero)

Para ajudar entender as regras abaixo, você precisará saber que:

$$\begin{array}{r} 86 \overline{) 2} \\ \underline{43} \\ 0 \end{array}$$

onde:

86 é o divisor

2 é o dividendo

43 é o quociente

0 é o resto

Quando o divisor e o dividendo forem números inteiros:

Se o divisor for 0, será lançada uma exceção.

```
System.out.println( 32 / 0 );      // exceção
System.out.println( -32 / -0 );    // exceção
```

Quando o divisor é 0.0:

## SEMPRE HAVERÁ A TROCA DE SINAIS

```
System.out.println( 32.0 / 0.0 ); // resultado: INFINITY
System.out.println( -32.0 / 0.0 ); // resultado: -INFINITY
System.out.println( 32.0 / -0.0 ); // resultado: -INFINITY
System.out.println( -32.0 / -0.0 ); // resultado: INFINITY
System.out.println( 32 / 0.0 ); // resultado: INFINITY
System.out.println( -32 / 0.0 ); // resultado: -INFINITY
System.out.println( 32 / -0.0 ); // resultado: -INFINITY
System.out.println( -32 / -0.0 ); // resultado: INFINITY
```

Quando o divisor for 0:

**Anotações**

267

# Programação Orientada a Objetos

---

## NUNCA HAVERÁ A TROCA DE SINAIS, SEMPRE SERÁ MANTIDO O SINAL DO DIVIDENDO

```
System.out.println( 32.0 / 0 ); // resultado: INFINITY
System.out.println( -32.0 / 0 ); // resultado: -INFINITY
System.out.println( 32.0 / -0 ); // resultado: INFINITY
System.out.println( -32.0 / -0 ); // resultado: -INFINITY
```

Quando o divisor for diferente de 0 and 0.0

## SEMPRE HAVERÁ JOGO DE SINAL

```
System.out.println( -32.0 / -1 ); // resultado: 32.0
System.out.println( -32.0 / 1 ); // resultado: -32.0
System.out.println( 32.0 / -1 ); // resultado: -32.0
System.out.println( 32 / -1 ); // resultado: -32
System.out.println( -32 / 1 ); // resultado: 32
System.out.println( 32 / 1 ); // resultado: 32
```

Relembrando...

Você pode ser pego de surpresa durante o exame quando se deparar com questões envolvendo operações matemáticas e tipos primitivos, porém para vencermos os crápulas do exame, vamos entender algumas regrinhas:

1) Regra geral: Já foi discutido no capítulo 3 mas vale a pena relembrar, toda operação envolvendo numeros inteiro SEMPRE retornará um tipo int.

Vejamos:

```
public class Test {
    public static void main(String[] args) {
        byte b = 127; // tamanho 8 bits
        short s = 127; // tamanho 16 bits
        short x = s * b; // deveria funcionar, mas dá erro de compilação
        System.out.println(x);
    }
}
```

Se tentarmos compilar o programa anterior receberíamos um insulto por parte do

**Anotações**

268

## Programação Orientada a Objetos

---

compilador, pois como já dissemos, todo resultado de uma operação entre tipos inteiros SEMPRE resultará em int e um tipo short não pode suportar um int sem conversão explícita.

```
public class Test {  
    public static void main(String[] args) {  
        byte b = 127;           // tamanho 8 bits  
        short s = 127;          // tamanho 16 bits  
        short x = (short)s * b; // deveria funcionar, mas também dá erro  
de compilação  
        System.out.println(x);  
    }  
}
```

Esse código não funcionará, pode confirmar, pois o compilador não consegue saber os valores das variáveis em tempo de compilação e com isso, não pode efetuar a conversão explícita !

Vale também ressaltar que se a variável x fosse long, o código seria compilado, pois um tipo int pode perfeitamente ser atribuídos à uma variável do tipo long.

## Usando as classes Wrappers

As classes wrappers tem o papel de encapsular os tipos primitivos para a possibilidade de operações como: conversões, mudança de bases decimais, e algumas operação que somente a objetos é permitido, como por exemplo, trabalhar com conjuntos (que será abordado no capítulo seguinte).

Todo tipo primitivo tem uma classe wrapper correspondente, vejamos:

<b>tipo primitivo</b>	<b>classe wrapper</b>
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean

Note que os nomes das classes wrapper tem o mesmo nome dos tipos primitivos, a exceção de Integer e Character, e isso não teve nenhuma graça por parte dos desenvolvedores.

Para criar um objeto podemos fazer da seguinte forma:

```
Integer a = new Integer("10");  
Integer b = new Integer(10);
```

Note que as classes tem os construtores sobrecarregados, ou seja, aceitam o valor String e o valor int no caso anterior, vejamos outro exemplo:

```
Float f1 = new Float("14.0f");  
Float f2 = new Float(14.0f);  
Boolean b1 = new Boolean("TRUE");  
Boolean b2 = new Boolean(true)
```

No caso da classe Boolean, você poderá instanciar um objeto passando uma string TRUE ou FALSE, independente se estas forem maiúsculas ou minúsculas.

# Programação Orientada a Objetos

---

## Métodos importantes

valueOf()  
xxxValue()  
parseXxx()

Os métodos acima devem ser compreendidos para o sucesso nas respostas sobre as classes Wrappers.

### O método valueOf()

É um método estático que quase todas classes wrapper tem que retorna o objeto wrapper a classe relacionada. Esse método tem uma funcionalidade muito interessante, pois pode converter números em bases diferentes, vejamos:

```
Integer i1 = Integer.valueOf("89");
```

Retorna um objeto Integer na base decimal.

```
Integer i2 = Integer.valueOf("89",2);
```

Tentativa de retornar um objeto Integer, porém nesse caso foi especificado a base 2 (binário) como "89" nunca foi e nunca será um número binário, essa linha gerará um exceção `java.lang.NumberFormatException`.

Não pense em usar:

```
Float.valueOf("89.0",2) ou  
Double.valueOf("89.0",2)
```

que você terá um erro de compilação, pois as classes Float e Double não tem esse método com essa assinatura.

```
Float f1 = Float.valueOf("10f");  
Float f2 = Float.valueOf("10.0f");
```

Porém as linhas acima seriam compiladas sem nenhum problema.

---

## Anotações

---

---

---

---

271

# Programação Orientada a Objetos

---

**NÃO ESQUEÇA:** Não pense que esse método é sobrecarregado como os construtores permitindo que você passe o tipo primitivo diretamente:

```
Float f1 = Float.valueOf(10f);
```

Isso seria repulsivo ao compilador.

## O método xxxValue()

Assinatura do método:

```
public int intValue()  
public byte byteValue()  
public short shortValue()  
public long longValue()  
public float floatValue()  
public double doubleValue()
```

Observe que não são métodos estáticos.

Esse método tem a função de realizar conversões dentro das classes wrapper, pois como dissemos é uma característica dessas classes:

```
public class TestWrappers {  
    public static void main(String[] args) {  
        Float w = Float.valueOf("89f");  
        int i = w.intValue();  
        System.out.println(w.toString());  
        System.out.println(i);  
    }  
}
```

O programa acima está fazendo uma conversão do valor que o objeto w está armazenando e convertendo para o tipo primitivo int. Poderíamos converter para qualquer outro, vejamos:

```
byte b = w.byteValue();  
short s = w.shortValue();
```



# Programação Orientada a Objetos

---

## O método estático `parseXxx()`

Outra forma de se converter é usando os métodos `parse's`, vejamos um exemplo bem simples para entender como isso funciona:

Imaginemos que você não precise criar um objeto wrapper necessariamente para realizar uma conversão, você pode usar o método `parseXxx` para realizar tal tarefa:

```
int i = Integer.parseInt("10");
```

A string passada como parâmetro deve corresponder a um tipo primitivo `int`, senão uma exceção será lançada como podemos observar no exemplo a seguir:

```
int i = Integer.parseInt("10f");
```

Nesse caso uma exceção será lançada: `java.lang.NumberFormatException`

## O método `toString()`

O método `toString()` tem uma funcionalidade semelhante as vistas até aqui. E além do mais as classes Wrapper tem um método estático `toString()` sobrecarregado, ou seja, se você precisar transformar uma variável `int` em `String`, você pode usar:

```
int i = 10;
```

Opção 1:

```
String s = "" + i;
```

Opção 2:

```
String s = new Integer(i).toString();
```

Opção 3:

```
String s = Integer.toString(i);
```

Você concordará comigo que a opção três é a mais elegante, visto que não cria nenhum objeto na memória, e a opção 1 não vou nem me estressar em definir aquilo. (mas acreditem eu já usei )

---

### Anotações

273

# Programação Orientada a Objetos

---

Você também poderá chamar o método `toString` do objeto:

```
Float f = Float.valueOf("10.0");  
System.out.println(f.toString());
```

## Convertendo bases com `toString()`

As classes `Integer` e `Long` tem dois métodos estáticos que podem ser usados para realizar conversão de bases decimais, vejamos:

Classe `Integer`:

```
public static String toString(int i, int radix)
```

Classe `Long`:

```
public static String toString(long i, int radix)
```

Você poderá realizar conversões entre bases da seguinte forma:

```
String s1 = Integer.toString(256,16);  
String s2 = Integer.toString(256,2);
```

O código acima retorna para as variáveis `s1` e `s2`, os valores correspondentes a 256 em Hexa (base 16) e em Binário (Base 2).

Para encerrarmos esse estudo, ainda temos os métodos estáticos nomeados:

Classe `Integer`:

```
public static String toHexString(int i)  
public static String toBinaryString(int i)  
public static String toOctalString(int i)
```

Classe `Long`:

```
public static String toHexString(long i)  
public static String toBinaryString(long i)  
public static String toOctalString(long i)
```

# Programação Orientada a Objetos

---

Exemplo de uso:

```
String s1 = Integer.toHexString(123);
```

O código acima transforma o inteiro 123 em Hexadecimal.

Métodos que lançam a exceção `NumberFormatException`

- Todos os construtores das classes wrapper numéricas;
- os métodos estáticos `valueOf`;
- os métodos estáticos `toString` de `Integer` e `Long`;
- os métodos `parseXxx`.

## Resumos para certificação - Usando a classe `java.lang.String`

- Os objetos `String` são imutáveis, porém as variáveis de referência `String` não;
- Se você criar uma nova `String` sem atribuí-la, ela ficará perdida para seu programa;
- Se você redirecionar a referência a uma `String` para uma nova `String`, o objeto `String` anterior pode ser perdido;
- Os métodos `String` usam índices iniciados em zero, exceto para o segundo argumento de `substring`;
- A classe `String` é final – seus métodos não podem ser sobrescritos;
- Quando uma `String` literal for encontrada pelo JVM, ela será adicionada ao pool;
- As `Strings` têm um método chamado `length()` e os arrays um atributo chamado `length`;
- Os `StringBuffers` são mutáveis – eles podem ser alterados sem a criação de um novo objeto.
- Os métodos `String Buffers` atuam sobre o objeto chamador, mas os objetos podem ser alterados sem a atribuição explícita na instrução;
- O método `equals()` de `StringBuffers` não é sobrescrito, ele não compara valores;
- Em todas as seções, lembre-se que métodos encadeados são avaliados da esquerda para a direita.

## Usando uma Classe `java.lang.Math`

- O método `abs()` é sobreposto para usar um tipo `int`, `long`, `float` ou `double`;
- O método `abs()` pode retornar um valor negativo se o argumento for o menor `int` ou `long` igual ao valor `Integer.MIN_VALUE` ou `Long.MIN_VALUE`, respectivamente;
- O método `max()` é sobreposto para usar argumentos `int`, `float`, `long` ou `double`;
- O método `min()` é sobreposto para usar argumentos `int`, `float`, `long` ou `double`;
- O método `random()` retorna um `double` maior que 0.0 e menor que 1.0;
- O método `random()` não usa nenhum argumento;
- Os métodos `ceil()`, `floor()` e `round()` retornarão os números de ponto flutuante equivalentes aos inteiros que receberem; `ceil()` e `floor()` retornaram `doubles`; e o `round()` retornará um tipo `float` se tiver recebido um `int`, ou um `double` se tiver recebido um `long`;
- O método `round()` é sobreposto para usar um tipo `float` ou `double`;
- Os métodos `sin()`, `cos()` e `tan()` usam ângulos do tipo `double` em radianos;
- O método `sqrt()` pode retornar `NaN` se o argumento que usar for `NaN` ou menor que zero;
- Os números de ponto flutuante podem ser divididos por zero(0.0) se causar erro; o resultado será infinito positivo;
- `NaN` não é igual a nada, nem a ele próprio.

# Programação Orientada a Objetos

---

## Usando Wrappers

- As classes wrappers estão relacionadas aos tipos primitivos;
- Os wrappers tem duas funções principais:
  - Encapsular tipos primitivos para que possam ser manipuladas como objetos;
  - Fornecer métodos utilitários para os tipos primitivos(geralmente conversões).
- Exceto em Character e Integer, os nomes das classes wrappers são iguais ao dos tipos primitivos, começando com maiúsculas;
- Os construtores wrapper pode usar uma String ou num tipo primitivo, exceto para Character, que pode usar um tipo char;
- Um objeto boolean não pode ser usado como um tipo primitivo booleano;
- As três famílias de métodos mais importantes são:
  - xxxValue() – não usa argumentos, retorna um primitivo;
  - parseXxx() – Usa uma String, retorna um tipo primitivo, é estático e lança exceção NFE.
  - ValueOf() – Usa uma String, retorna um objeto encapsulado, é estático e lança exceção NFE.
- O argumento radical referencia bases(normalmente) diferentes de 10; a base binária tem radical 2, octal = 8 e hex = 16;
- Use == para comparar variáveis tipo primitivas;
- Use == para determinar se duas variáveis de referência apontam para o mesmo objeto;
- O operador == compara padrões de bits, primitivos ou de referência;
- Use equals() para determinar se dois objetos são significativamente equivalentes;

Anotações

278

# Programação Orientada a Objetos

---

- As classes String e Wrapper substituem equals() para verificar valores;
- O método equals da classe StringBuffer não é substituído, ele usa o operador == sub-repeticionalmente;
- O compilador não permitirá o uso do operador == se as classes não tiverem na mesma hierarquia;
- Os objetos wrapper não passarão no teste do método equals() se estiverem em classes diferentes.

Anotações

279

## Capítulo VI - Objetos e conjuntos

### O método equals

Esse método é definido na classe Object, portanto toda classe o terá por herança. Sua assinatura é:

```
public boolean equals(Object o)
```

Qualquer tentativa de substituição desse método escrita de forma diferente, não será evidentemente uma substituição e sim uma sobrecarga de método. Você deve prestar bastante atenção nesses conceitos que agora serão explicados, pois são fundamentais para absorção e compreensão desse capítulo. Você já deve saber que o método equals pode funcionar de forma diferente em cada classe, e isso vai depender de como ele foi definido, por exemplo, nas classes Wrappers, é possível saber se um valor 5 inteiro que está armazenado em um objeto x é igual a um objeto y que também armazena o valor 5 (desde que sejam da mesma classe como Integer), isso devido ao fato de que o método equals foi substituído nas classes Wrappers, fazendo com que retorne true quando o valor armazenado pelas classes Wrappers forem idênticos, a exceção é claro, se o tipo for diferente. Vejamos:

```
Integer i = new Integer("5");  
Long l = new Long("5");  
if ( i.equals(l)) System.out.println("igual");  
else System.out.println("diferente");
```

// Resultado: diferente

Agora quando comparamos instâncias distintos de duas classes iguais:

```
Integer i = new Integer("5");  
Integer j = new Integer("5");  
if ( i.equals(l)) System.out.println("igual");  
else System.out.println("diferente");
```

// Resultado: igual

Apesar de serem objetos distintos, tem o mesmo valor.



# Programação Orientada a Objetos

---

Se usarmos o operador ==

```
Integer i = new Integer("5");
Integer j = new Integer("5");
if (i == j) System.out.println("igual");
else System.out.println("diferente");
```

// Resultado: diferente

Apesar dos valores serem identicos, o resultado do teste (==) é falso, pois esse testa se os objetos apontam para o mesmo objeto na memória.

Quando substituir o método equals

Para iniciarmos essa discussão, precisamos saber que o método equals na classe Object funciona da mesma forma como o operador ==, portanto quando não substituimos o método equals em uma classe o retorno só será true quando ambos objetos apontarem para o mesmo objeto na memória, vejamos:

```
class Brasil {
    public static void main(String[] args) {
        Brasil penta = new Brasil();
        Brasil melhor = new Brasil();
        Brasil temp = penta;
        int i = 0;
        if (penta.equals(melhor)) ++i;
        if (melhor.equals(penta)) ++i;
        if (temp.equals(penta)) ++i;
        if (temp.equals(melhor)) ++i;
        System.out.print(i);
    }
}
```

// Resultado: 1

Note que somente uma vez a condição é satisfeita, no caso em que temp == penta!

# Programação Orientada a Objetos

---

Como a Java lhe dá os poderes do mundo, você pode alterar esse comportamento. Como ? Mudando a JVM ? Não ! É só substituir o método equals na classe (tira o calçado para falar esse nome) Brasil, mas para isso é preciso saber quando e porque mudar o método equals !

Essa é uma decisão que deve ser tomada medindo todas as consequências. Você deve primeiro responder uma pergunta básica para saber se precisará ou não mudar o método equals. Pergunte se é preciso em alguma situação identificar dois objetos distintos como iguais. Ficou confuso, calma, calma ! Voce tem dois objetos A e B, e adiciona os mesmos em um conjunto (estudaremos conjuntos mas afrente) agora você precisa realizar uma pesquisa afim de saber se o objeto C está no conjunto - pressuponha que os atributos de C e B são semelhantes, se o métodos equals da classe dos objetos citados não for modificado, você nunca obterá true na condição B.equals(C), mesmo sendo classes iguais por sua semântica.

Então a decisão de mudar o método equals de uma classe está intimamente relacionada com a necessidade de em uma pesquisa se saber se um objeto está na lista pesquisa ou não.

## Entendendo o uso de hashCode

Outro conceito que precisamos entender agora é, onde a utilização do código hash influência no uso de conjuntos ? Não se preocupe, apesar de parecer estranho é um conceito fácil de se entender.

Vamos imaginar uma sala com inúmeras caixas postais, o PAULO tem uma caixa com número 385, a ANA tem uma com o número 208, PEDRO usa a de número 378 e assim por diante. Toda vez que chega uma correspondência (do PAULO por exemplo), sabe-se qual o número da caixa postal através de um cálculo em função do seu nome, por exemplo, a soma dos códigos ASCII como podemos observar a seguir:

P     -> 80  
A     -> 65  
U     -> 85  
L     -> 76  
O     -> 79

HASH       -> 385

---

## Anotações

282

## Programação Orientada a Objetos

---

Não tire sarro desse cálculo, esse exemplo de código hash é apropriado porém ineficiente (a quem diga absurdo) e isso veremos mais tarde. Com esse cálculo foi possível obter o número da caixa postal do PAULO. Agora vamos analisar como ficaria a pesquisa, ou seja, quando o PAULO chegar na empresa de correspondência e quiser saber se existe alguma correspondência para ele, o atendente perguntaria seu nome e faria o mesmo cálculo para descobrir qual o número da caixa postal, obtendo o número 297, assim o atendente se reportaria à cx no. 297 e recuperaria todas as suas correspondências. Esse processo é muito indicado quando se deseja extrema velocidade no processo de pesquisa. Se você é um cara esperto já deve ter notado um "erro" em nosso algoritmo de cálculo do código hash, pois seguindo esse cálculo qual seria o número da caixa postal para a IVONE. aha com certeza o PAULO pegaria as correspondências da IVONE e vice-versa ! NÃO ! Não pense assim, o processo de pesquisa usando código hash é um processo em duas fases, a primeira é saber em qual depósito se encontra as informações, ou seja, em qual caixa postal está as correspondências, a segunda é verificar cada item da caixa postal comparando para saber se é um item da pessoa desejada. O que não pode acontecer é criar um algoritmo que coloque todas as correspondências em um ou dois depósitos (caixa postal) assim o seu processo de pesquisa ficaria ineficiente, ruim, ridículo, absurdo! Mas correto ! MAS LENTO !

Talvez o exemplo de correspondência não foi muito convincente, visto que uma caixa postal é um depósito de correspondências de uma única pessoa, mas como a empresa de correspondência é nossa, e os clientes não tem chave, convencionamos dessa forma - só quem pega as correspondências é um funcionário da empresa. Imagine que nosso algoritmo fosse contar as letras do nome para saber o número da caixa postal, a seguinte lista mostraria a bagunça que ficaria:

NOME	DEPOSITO (CP)
JOAO	4
ANA	3
PEDRO	5
AMY	3
MARIA	5
JOSE	4
ENIO	4
JOAQUIM	7
CAIO	4

Note que cada nome com 4 (quatro) dígitos o número da caixa postal será o

**Anotações**

283

# Programação Orientada a Objetos

---

mesmo, ficaria uma bagunça, o processo de busca seria muito lento, portanto, quando for criar um algoritmo de cálculo para código hash (espalhamento), faça-o de forma inteligente, usando números complexos, matrizes, limites, derivadas e funções !!! Só para termos uma idéia, cálculos de hash dão assunto para teses de doutorado (isso eu li no livro) !

## Método equals versus hashCode

Você já deve ter notado a relação entre os métodos equals e hashCode, pois em uma pesquisa, você precisará verificar na segunda fase se um objeto X é igual a um item do conjunto - que nada mais é que um objeto Y. Se você não substituir o métodos equals você não terá essa possibilidade, visto que o método equals da classe Object retorna true se ambos os objetos apontarem (ops! isso não é C) referenciarem o mesmo objeto na memória. Memorize isso: para se utilizar uma classe em um conjunto, ela deve ter o método equals substituído.

### Contratos do método equals

- 1) Reflexivo - Para qualquer valor, x.equals() sempre será true;
- 2) Simétrico - Para qualquer valor de x e y, x.equals(y) é true, se e somente se y.equals(x) também for.
- 3) Transitivo - Para qualquer valor de x, y e z, se x.equals(y) for verdadeiro e y.equals(z) também for, então x.equals(z) também será verdadeiro.
- 4) Consistente - Todas as chamadas para x.equals(y) retornará consistentemente true ou false, até que haja uma mudança em algum dos objetos, caso contrário o resultado será sempre o mesmo.
- 5) Se x referenciar algum objeto, então x.equals(null) deverá retornar falso. Obviamente que uma tentativa de chamar um método equals para um objeto null, lançará um exceção.

### Contrato do método hashCode

Anotações

284

# Programação Orientada a Objetos

---

- 1) Sempre que um método hashCode for chamado, ele deverá resultar no mesmo código hash consistentemente (caso nenhum valor usado nas comparações for alterado).
- 2) Se dois objetos forem iguais, considerados pela chamada ao métodos equals, portanto o código hash deverá ser o mesmo para ambos os objetos.
- 3) Não é obrigado que dado dois objetos distintos, tenham códigos hash distintos.
- 4) Não é aconselhável usar atributos transient's no cálculo do código hash, visto que após o processo de deserialização, o método equals poder produzir um resultado diferente de quando o objeto foi serializado, portanto é uma situação perigosa.

Esse contratos devem ser respeitados pois você poderá alterar (substituir) o método equals, portanto siga sempre essas leis.

## **Substituindo os métodos equals e hashCode**

```
public class Pessoa {  
  
    private String CPF;  
  
    public int hashCode() {  
        return 5;  
    }  
  
    public boolean equals(Object o) {  
        if (o instanceof Pessoa) || (o.CPF == this.CPF) return true;  
        else return false;  
    }  
  
}
```

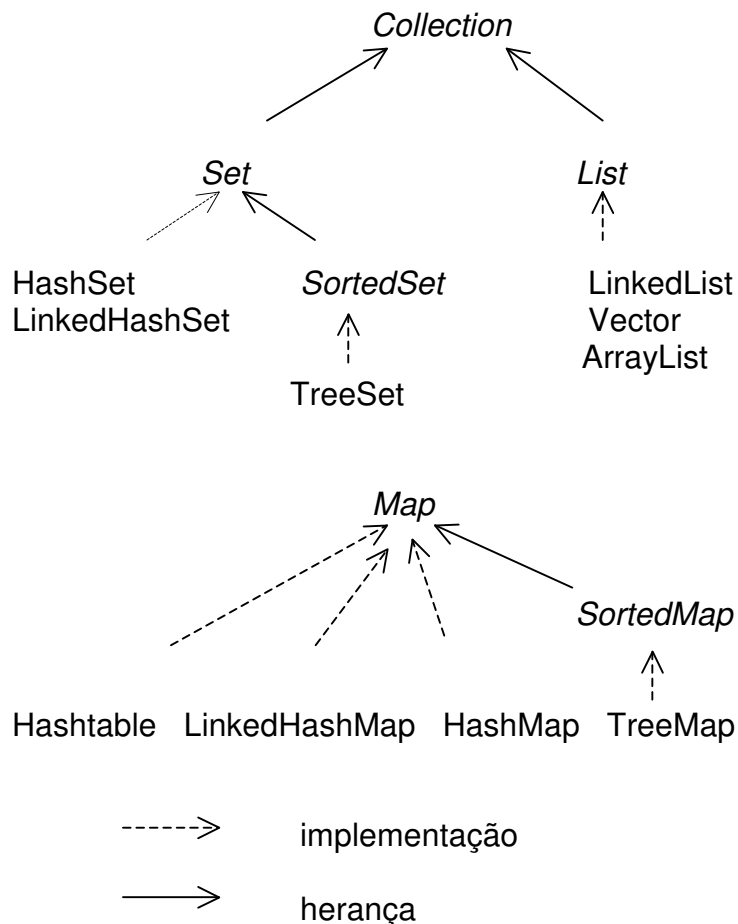
## Coleção de Dados

O uso de coleção de dados é uma necessidade quando precisamos de trabalhar com dados. Já vimos como comparar objetos afim de saber se são equivalentes ou não, agora vamos entender seus repositórios e como funcionam.

Existem classes e interfaces que precisaremos entender, se listassêmos aqui, já ficaria muito nebuloso, por isso vamos por parte. Podemos dividir os conjuntos em três categorias:

- Listas
- Conjuntos
- Mapas

Vejamos a hierarquia de interfaces/classes de coleção de dados em Java:



# Programação Orientada a Objetos

---

Os nomes em *itálico* são interfaces, os demais são classes.

Pode parecer confuso agora, mas depois ficará tudo muito claro !

## Ordem e classificação

Nesse primeiro instante, precisamos entender dois conceitos fundamentais nas coleções, que são: ordem e classificação de conjuntos. Abaixo podemos vislumbrar uma definição completa desses conceitos:

**ordenada** - Uma classe é ordenada se pode ser iterada pelos seus elementos em uma ordem específica, através de um índice ou por exemplo pela ordem de inserção.

**classificada** - Uma classe classificada, quando seus elementos estão classificados por algum critério, como por exemplo, em ordem alfabética, crescente ou cronológica etc. Toda classe classificada é ordenada, já uma classe ordenada pode não ser classificada.

## List

As classes que implementam a interface *List*, relevam o índice, com isso podemos inserir um item no meio de uma lista. Como você já deve ter percebido, as classes que implementam a interface *List* são ordenadas por meio de um índice, isso permite o acesso a um elemento que se encontra no meio da lista, através de seu índice. É uma espécie de sequência para armazenamento de objetos.

**ArrayList** - É um estrutura de dados que tem com base um array. É isso mesmo! Um *ArrayList* nada mais é que um array que pode ser alterado. Sua estrutura interna (pode conferir) é baseada em um *Array* com um tamanho inicial e deve ser especificado na criação do objeto - se nenhum valor for especificado a classe assumirá 10 (sei lá porquê). Com isso é possível tem uma ordem em *ArrayList*, pois o índice é o identifica os elementos. Vejamos suas principais características:

- Acesso sequencial / aleatório extremamente rápido.
- Em função do índice o acesso a um elemento no meio da lista é uma operação extremamente rápida, como já sabemos é o mesmo que recuperar um item de um vetor.

## Anotações

---

287

# Programação Orientada a Objetos

---

- Inserção é também extremamente rápida
- Vale uma ressalva nessa situação, visto que uma ArrayList cresce a medida que os itens vão sendo inseridos.

## Exemplo

```
import java.util.*;
public class TestList {
    public static void main(String[] args) {
        long inicio, fim;
        int n = 3000000;
        inicio = System.currentTimeMillis();
        ArrayList array = new ArrayList();
        for (int i = 0; i < n; i++) {
            array.add(new Integer(i));
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo inserir: " + (fim - inicio)/1000.000 );
        inicio = System.currentTimeMillis();
        Iterator o = array.iterator();
        while (o.hasNext()) {
            Integer x = (Integer)o.next();
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo iterar: " + (fim - inicio)/1000.000 );
    }
}
```

Resultado do programa acima:

Tempo inserir: 2.563

Tempo iterar: 0.172

Note que houve uma demora relativamente maior no processo de inserção, porém observe que foram inseridos três milhões de objetos que foram inseridos - acredito que o tempo consumido nos dois processos foi muito bem aproveitado pela classe.

---

## Anotações

288



# Programação Orientada a Objetos

---

A classe `ArrayList` tem um construtor sobrecarregado que recebe um argumento onde pode-se definir a capacidade da estrutura, ou seja, se for especificado 1000, a classe irá reservar 1000 endereços nulos para o preenchimento dos dados, isso evita a realocação constante de seu array.

**Vector** - Tem as mesmas características de `ArrayList`, porém seus métodos são sincronizados. Se aplicarmos o mesmo teste anterior notaremos uma diferença na inserção, vejamos:

```
import java.util.*;
public class TestVector {
    public static void main(String[] args) {
        long inicio, fim;
        int n = 60000;
        int j = 0;

        inicio = System.currentTimeMillis();
        Vector array = new Vector(0,1);
        for (int i = 0; i < n; i++) {
            array.add(new Integer(i));
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo inserir: " + (fim - inicio)/1000.000 );

        inicio = System.currentTimeMillis();
        Iterator o = array.iterator();
        while (o.hasNext()) {
            Integer x = (Integer)o.next();
            j++;
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo iterar: " + (fim - inicio)/1000.000 );
    }
}
```

Resultado do programa acima:

Tempo inserir: 109.938

Tempo iterar: 0.015

Observe que o tempo de inserção foi extremamente superior ao de `ArrayList`

**Anotações**

289

# Programação Orientada a Objetos

---

mesmo com uma quantidade 50 vezes inferior, portanto a inserção usando a classe Vector é muito lenta, você provavelmente nunca precisará usá-la, visto que sua única diferença em relação à classe ArrayList é que seus métodos são sincronizados, e esse recurso você pode conseguir se utilizar métodos estáticos da classe java.util.Collections.

Só pra não ficar nenhuma dúvida, se o mesmo programa sobre a classe Vector fosse especificado 3 milhões com em ArrayList, seria necessário um tempo equivalente a  $50 * 109.938 = 5496,9$  s o que é equivalente a 91,615 minutos.

**LinkedList** - A diferença entre LinkedList e ArrayList é que os elementos de LinkedList são duplamente encadeados entre si. Isso é essencial quando se deseja implementar uma fila ou pilha. Por causa da duplicidade de encadeamento, é possível por exemplo inserir no final da lista, no início sem a necessidade da realocação do array, visto que cada nó tem uma referência para seu sucessor e seu predecessor, logicamente que isso faz com que o processo de inserção seja um pouco mais lento, pois a cada objeto inserido é registrado o "endereço dos vizinhos" consequentemente uma lista duplamente encadeada é bem maior que uma lista simples, vejamos:

```
import java.util.*;
public class TestVector {
    public static void main(String[] args) {
        long inicio, fim, j;
        j = 0;
        inicio = System.currentTimeMillis();
        LinkedList array = new LinkedList();
        for (int i = 0; i < 1500000; i++) {
            array.add(new Integer(i));
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo inserir: " + (fim - inicio)/1000.000 );
        inicio = System.currentTimeMillis();
        Iterator o = array.iterator();
        while (o.hasNext()) {
            Integer x = (Integer)o.next();
            j++;
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo iterar: " + (fim - inicio)/1000.000 );
    }
}
```

Anotações

290

# Programação Orientada a Objetos

---

Resultado:

Tempo inserir: 2.485

Tempo iterar: 0.109

Note que a metade dos itens do primeiro exemplo foi inserido e no entanto o tempo gasto foi praticamente o dobro.

**Resumindo as listas** - O que precisamos saber é que o índice em uma lista é relevante, toda lista é ordenada, ou seja, podemos iterar em uma ordem específica, seja ela pela ordem de inserção ou pela ordem do índice. A não se esqueça que não existe lista classificada !

## Set

Voltemos a 5a. série do ensino médio:

$$A = \{ 0, 1, 2 \}$$
$$B = \{ 1, 2, 3 \}$$
$$A \cup B = \{ 0, 1, 2, 3 \}$$

Note que os elementos repetidos {1,2} foram suprimidos de um dos conjuntos para não haver repetição. Isso você já deveria saber, pois deve ter estudado no início do ensino médio. Bom o que precisamos saber agora é que toda classe que implementa a interface Set: **NÃO ACEITA DUPLICATAS !!!** Esse conceito é fundamental para entendermos as classes Set's !

Como tocamos em um assunto essencial, vejamos como a classe funciona:

X -> Testa o método equals

Y -> Pega o código hash

Se X ou Y for falso o objeto é inserido.

Vejamos um caso que sempre será inserido: Se o método equals não for substituído todos os elementos serão inseridos, a menos que você tente inserir o mesmo objeto duas vezes, vejamos:

---

**Anotações**

291

# Programação Orientada a Objetos

---

```
public class Test {  
    public static void main(String[] args) {  
        HashSet conjunto = new HashSet();  
        A x, y, z;  
        x = new A();  
        y = new A();  
        x.a = 10;  
        y.a = 20;  
        z = x;  
        conjunto.add(x);  
        conjunto.add(y);  
        conjunto.add(z);  
    }  
}  
  
class A {  
    int a = 0;  
}
```

A pergunta é: Quantos objetos eu tenho nesse conjunto ??? Acertou se você respondeu dois, vamos tentar entender: note que só existem dois objetos distintos: x e y o z é uma referência a x, portanto, quando o método add foi chamado para o caso de z foi calculado o código hash (digamos que deu 7621), a segunda questão que a classe faz é saber se existe algum objeto no depósito 7621 que seja igual a z, como sabemos (já estudamos isso) que o código hash de dois objetos igual sempre será o mesmo, nesse caso x e z, podemos pressupor que o objeto x está também no depósito 7621, conseqüentemente o método equals deve (pelo menos é o que esperamos) retornar true, portanto o objeto z não será inserido no conjunto. Talvez você esteja rindo desse exemplo, mais vamos complicar um pouco:

# Programação Orientada a Objetos

---

```
import java.util.*;
public class Teste {
    public static void main(String[] args) {
        HashSet conjunto = new HashSet();
        A x, y, z;
        x = new A();
        y = new A();
        z = new A();
        x.a = 10;
        y.a = 20;
        z.a = 10;
        conjunto.add(x);
        conjunto.add(y);
        conjunto.add(z);
    }
}

class A {
    int a = 0;
    public int hashCode() {
        return 10;
    }
}
```

Note que nesse caso, substituímos o método `hashCode`, apesar de ser uma substituição ridícula, ela funciona. Seguindo o mesmo raciocínio do exemplo anterior, quantos elementos foram inseridos ? Tentamos entender: cada vez que o método `add` é chamado, a classe `HashSet` chama o método `hashCode` para tentar encontrar um objeto equivalente, bom nesse caso, o `hashCode` está retornando 10, com isso podemos saber o endereço de um possível elemento já inserido, porém o método `equals` que prevalece nesse caso é o da classe `Object` (que você deve lembrar - retorna `true` se ambas as referências forem para os mesmos objetos) e o método `equals` nesse caso irá retornar `false`, com isso mesmo que se tenha a intenção de serem semelhantes os objetos `x` e `z` não o são, visto que para isso será necessário substituir também o método `equals`, vejamos no próximo exemplo:

# Programação Orientada a Objetos

---

```
import java.util.*;
public class Teste {
    public static void main(String[] args) {
        HashSet conjunto = new HashSet();
        A x, y, z;
        x = new A();
        y = new A();
        z = new A();
        x.a = 10;
        y.a = 20;
        z.a = 10;
        conjunto.add(x);
        conjunto.add(y);
        conjunto.add(z);
        Iterator o = conjunto.iterator();
        while (o.hasNext()) {
            A azinho = (A)o.next();
            System.out.println( azinho.a );
        }
    }
}

class A {
    int a = 0;
    public int hashCode() {
        return 10;
    }
    public boolean equals(Object o) {
        boolean rc = false;
        if (o == null) {
            rc = false;
        }
        else {
            if ((o instanceof A) && ((A)o).a == this.a) {
                rc = true;
            }
        }
        return rc;
    }
}
```

**Anotações**

294

# Programação Orientada a Objetos

---

Resultado:  
20  
10

Nesse caso, alteramos a classe A e determinar que quando o membro a tiver o mesmo valor será considerada igual (semelhantes as classes Wrappers) - por isso que nesse caso foram inseridos somente dois objetos no conjunto.

Bom agora que acreditamos que ficou claro essa questão da unicidade dos elementos nas classes Set's, vejamos suas implementações usadas.

**HashSet** - é um conjunto não-ordenado e não-classificado, utiliza o código hash do elemento que está sendo inserido para saber em qual depósito deve armazenar, com isso podemos notar que um cálculo de código hash ineficiente é a morte para a boa performance durante o processo de inserção/recuperação. Nunca use essa classe quando precisar de uma ordem na iteração.

**LinkedHashSet** - é um conjunto ordenado e duplamente encadeado, com isso podemos iterar pelos seus elementos em uma ordem, sempre use essa classe quando precisar de ordem na iteração, porém saiba que é um pouco mais lenta que HashSet na inserção visto que leva mais tempo para registrar os vizinhos (elemento posterior e inferior).

**TreeSet** - essa é uma das duas classes do framework de coleção da api Java que é classificada e ordenada - essa classificação se dá pela ordem natural da classe que está sendo inserida (isso pode ser alterado mas foge do nosso escopo), vejamos algumas classificações naturais para as classes mais usadas:

Classe	Ordem Natural
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical
Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	system-dependent lexicographic on pathname.
String	lexicographic
Date	chronological

**Anotações**

295

# Programação Orientada a Objetos

---

## Map

As classes que implementam a interface Map tem funcionalidades distintas da aprendida até aqui, vejamos porquê. Aprendemos que em uma lista o índice é relevante para se pesquisar ou até mesmo inserir um objeto no meio no fim ou em qualquer posição da lista, já nas classes Set's, a unicidade é a característica fundamental dessas classes sendo necessário uma correta relação dos métodos equals e hashCode para seu funcionamento, já as classes Map's faz um mapeamento de chave X valor, ou seja, você tem um objeto chave para um objeto valor. Vejamos um exemplo de seu uso:

```
import java.util.*;
public class TestMap {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        map.put( "um", new Integer(1) );
        map.put( "dois", new Integer(2) );
        map.put( "tres", new Integer(3) );
        map.put( "quatro", new Integer(4) );
    }
}
```

Note que um objeto (nesse caso String) é mapeado para cada um valor do conjunto, portanto nas classes Map's a unicidade da chave é relevante, se você tentar inserir um item como podemos ver abaixo, você não terá um novo item no conjunto pois a chave é idêntica, o valor somente é substituído.

```
map.put( "dois", new Float(30.0f) );
```

Podemos concluir que a chave do conjunto Map é um Objeto de uma classe Set, ou seja, a chave deve ser única, o processo para saber se a chave está no conjunto é idêntica ao processo explicado nas classes Set's, com isso, uma boa implementação dos métodos hashCode e equals é imprescindível para a boa performance desse conjunto.



# Programação Orientada a Objetos

---

Vejamos isso na prática:

```
import java.util.*;
```

```
1. public class TestMap {  
2.     public static void main(String[] args) {  
3.         HashMap map = new HashMap();  
4.         map.put( "um", new Integer(1) );  
5.         map.put( "dois", new Integer(2) );  
6.         map.put( "tres", new Integer(3) );  
7.         System.out.println( "Antes: "+map.size());  
8.         map.put( "um", new Float(1f) );  
9.         System.out.println( "Depois: "+map.size());  
10.    }  
11. }
```

Resultado:

Antes: 3

Depois: 3

Note que mesmo adicionando um objeto diferente na linha 8, o resultado do tamanho do conjunto não foi alterado. Fica subitamente entendido que se a classe usada na chave (nesse caso String) não substituir os métodos equals e hashCode, todo objeto será adicionado no conjunto, o que não faz muito sentido, portanto sempre utilize um objeto de uma classe onde seus métodos equals e hashCode tenham sido substituídos.

# Programação Orientada a Objetos

---

**HashMap** - é um conjunto Map não-ordenado e não classificado. Permite a existência de chave e valores nulos. Vejamos um exemplo esdrúxulo, porém funcional:

```
import java.util.*;

public class TestMap {
    public static void main(String[] args) {
        HashMap map = new HashMap();
        String um, dois;
        um = null;
        dois = null;
        map.put( um, new Integer(1) );
        map.put( dois, new Integer(2) );
        System.out.println( "Tamanho: "+map.size());
    }
}
```

O resultado será: Tamanho 1

**Hashtable** - essa classe é equivalente à HashMap com a diferença que seus métodos são sincronizados, e, além disso, não permite valores/chaves nulos.

```
import java.util.*;

public class TestMap {
    public static void main(String[] args) {
        Hashtable map = new Hashtable();
        String um = null;
        String dois = null;
        map.put( "um", um );
        map.put( "dois", dois );
        System.out.println( "Tamanho: "+map.size());
    }
}
```

Erro:

```
java.lang.NullPointerException
at java.util.Hashtable.put(Hashtable.java:386)
at TestMap.main(TestMap.java:8)
Exception in thread "main"
```

**Anotações**

298

# Programação Orientada a Objetos

---

Só pra relaxar: Note que a nomenclatura de nomes da Java para essa classe foi esquecida, pois o t em Hashtable deveria ser maiúsculo.

**LinkedHashMap** - é uma versão ordenada (ordem de inserção/acesso) da interface Map, embora seja um pouco mais lento para inserção e remoção, visto que deve sempre registrar seus sucessor e predecessor, porém a iteração é bem rápida, e, isso tudo por ser uma classe duplamente encadeada.

**TreeMap** - é a outra classe da Framework Collection Java que é classificada e consequentemente ordenada, vejamos uma aplicação dessa classe:

```
import java.util.*;
public class TestMap {
    public static void main(String[] args) {
        TreeMap map = new TreeMap();
        long inicio, fim;
        inicio = System.currentTimeMillis();
        int n = 500000;
        for (int i=0; i < n; i++) {
            map.put( "um"+i, new Integer(i) );
        }
        fim = System.currentTimeMillis();
        System.out.println( "Tempo inserir: " + (fim - inicio)/1000.000 );
    }
}
```

Resultado: Tempo inserir: 3.937

Porém o resultado estaria em ordem (nesse caso lexicographic) pois se trata de uma classe String.

# Programação Orientada a Objetos

---

Bom isso encerrado o estudo das classes de coleções que a Framework Java nos oferece, vejamos uma tabela resumindo todas:

Classe	Map	Conjunto	Lista	Ordenada	Classificada
HashMap	X			nao	nao
Hashtable	X			nao	nao
TreeMap	X			sim	sim
LinkedHashMap	X			sim	nao
HashSet		X		nao	nao
TreeSet		X		sim	sim
LinkedHashSet		X		sim	nao
ArrayList			X	sim	nao
Vector			X	sim	nao
LinkedList			X	sim	nao

## Coleta de Lixo

A memória sempre foi o "calcanhar de aquiles" para as linguagens de programação. Uma linguagem de programação se diz tipada, por poder definir tipos de dados, por exemplo, uma variável inteira/integer é um tipo, cada linguagem de programação tem uma representação para o tipo inteiro, em pascal um tipo inteiro corresponde a um intervalo que provavelmente não equivale o mesmo intervalo em Java, e essas definições são essencialmente para o melhor aproveitamento da memória, pois com a definição dos tipos podemos (os programas podem) estimar o uso de memória, saber quanto um programa ou uma rotina vai usar, enfim a memória é um elemento essencial e primordial para o sucesso de qualquer programa, por isso devemos saber aproveitá-la.

O gerenciamento da memória não é uma tarefa fácil, saber quando um objeto está pronto para ser dilacerado (sempre quis usar essa palavra), anular uma referência, re-referenciar um objeto, sugerir a execução do coletor de lixo, todos esses conceitos serão entendidos (pelo menos é o que esperamos) a partir de agora.

## Como o coletor de lixo funciona

Você poderá encontrar dezenas de teorias de como o coletor funciona, mas não caia nessa, pode ser que sim ou que não, ou sei lá ! Mas vamos entender uma coisa: um programa Java roda simultaneamente vários processos ou vamos chamar segmentos (não gosto muito desse nome, prefiro chamar thread), e um segmento pode estar ou não ativo, e um objeto está pronto para ser extirpado quando nenhum segmento não pode alcançar esse objeto, simples não ?

Anotações

300

# Programação Orientada a Objetos

---

Você pode dar uma mão para o coletor de lixo, porém não caia na idéia de que pode chamá-lo quando bem entender, você pode sugerir: hein tá na hora da limpeza, mas não é garantido que ele venha. Portanto se você se deparar com uma questão perguntando qual a maneira mais precisa de se executar o coletor de lixo ? Não hesite em marcar: IMPOSSÍVEL !!!

Anulando uma referência:

```
1. public class Test {  
2.     public static void main(String[] args) {  
3.         StringBuffer b = new StringBuffer("hello");  
4.         System.out.println(b);  
5.         b = null;  
6.     }  
7. }
```

A partir da linha 5, o objeto b não referencia mais nada da memória, portanto o coletor pode muito bem eliminar a String "hello" que deverá estar ocupando lugar. Isso é muito gentil por parte dos programadores, anular uma referência sempre que não venha mais precisar do objeto.

Vejamos outro exemplo:

```
1. public class Test {  
2.     public static void main(String[] args) {  
3.         StringBuffer sb1 = new StringBuffer("hello");  
4.         StringBuffer sb2 = new StringBuffer("my friendly");  
5.         System.out.println( sb1 );  
6.         sb1 = sb2;  
7.     }  
8. }
```

Após a execução da linha 6, a string "hello" ficará perdida na memória, estando disponível para o coletor de lixo executar o seu trabalho.

# Programação Orientada a Objetos

---

## Objetos dentro de um método

Todos os objetos criados localmente dentro de um método estarão qualificados para a coleta após a execução do método. Vejamos um exemplo:

```
public void show() {  
    String s1 = "s1";  
    String s2 = "s2";  
    System.out.println(s1+s2);  
}
```

Após a execução do métodos acima, os dois objetos s1 e s2 estarão qualificados para a coleta, agora vejamos um outro exemplo:

```
public String show() {  
    String s1 = "s1";  
    System.out.println(s1+s2);  
    return s1;  
}
```

Note que o objeto s1 está sendo utilizado no retorno do método, portanto esse objeto não pode ser coletar, mesmo porque, ele ainda está sendo referenciado.

## Isolando uma referência

Vamos complicar um pouco as referências para ver o que acontece com os objetos:

```
public class Test {  
    Test t;  
    public static void main(String[] args) {  
        Test t2 = new Test();  
        Test t3 = new Test();  
        Test t4 = new Test();  
        t2.t = t3;  
        t3.t = t4;  
        t4.t = t2;  
        t2 = null;  
        t3 = null;  
        t4 = null;  
    }  
}
```

**Anotações**

---

---

---

---

302

# Programação Orientada a Objetos

---

O que achas que acontece nesse caso ?? Calma vamos entender:

Apenas três objetos foram criados na memória: t2, t3, t4.

Quando os objetos t2, t3 e t4 foram anulados, tornaram-se qualificados para a coleta, mesmo existindo outras referências para esses objetos.

## Sugerindo a coleta

```
import java.util.Date;
public class Memory {
    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("Memoria total: "+rt.totalMemory());
        System.out.println("Memoria Antes: "+rt.freeMemory());
        Date d = null;
        for (int i=0; i < 5000; i++) {
            d = new Date();
            d = null;
        }
        System.out.println("Memoria Depois: "+rt.freeMemory());
        rt.gc();
        System.out.println("Memoria Final: "+rt.freeMemory());
    }
}
```

## Capítulo V - Classes internas

### Onde deve ser usada e para que serve ?

Antes de qualquer introdução à esse assunto, vamos fazer uma explanação sobre as classes internas, onde e quando devem ser usadas, visto que esse assunto é um tanto contundente no que se refere a reutilização de código, extensabilidade e escalabilidade de um código sob o ótica da OO.

### Um exemplo oficial

Definir uma classe interna ou aninhada é um recurso interessante adicionado na versão 1.1 da Java, porém deve ser bem cauteloso o seu uso visto que se assim não o for, com certeza teremos redundância de código, classes extremamente complicadas de ser entendidas e com seu uso limitado, portanto saiba decidir seu uso. Vejamos um exemplo usado na própria framework de coleção Java, as classes Map herdam de uma superclasse chamada AbstractList, para seu funcionamento ela usa um objeto de uma classe interna chamado Entry e seu escopo está limitado à classe AbstractList, com um funcionamento totalmente peculiar à AbstractList, não teria muito sentido, criar uma classe externa para Entry, visto que sua utilização se limita à AbstractList. Definir classes especialista é um conhecimento extremamente necessário no tocante à orientação a objeto e nesse caso a classe Entry nada mais é do que um membro (pois é assim que também pode ser chamadas as classes internas) de AbstractList.

As classes internas são divididas em:

- Classe estática aninhada (top-level class)
  - ❑ Classe interna comum
  - ❑ Classe interna local de método
  - ❑ Classe interna anônima



# Programação Orientada a Objetos

---

## Classes estáticas

Entenderemos agora o que são as classes estáticas que também podem ser chamadas como classe de nível superior ou top-level class, entretanto no fritar dos ovos é tudo a mesma coisa.

Uma classe estática não tem acesso aos membros da instância da classe encapsulada, somente os membros estáticos, tem a mesma visibilidade de uma classe externa.

### Modificadores que podem ser atribuídos à uma classe interna estática:

static - obrigatório é claro

protected

public

private

abstract

final

\* nunca abstract e final simultâneamente é claro!

```
public class Ex03 {  
    static int CountStatic = 0;  
    int CountNonStatic = 0;  
  
    public static class Inner {  
    }  
}
```

Você NUNCA poderia referencia o membro CountNonStatic dentro da classe Inner, uma vez que esse membro não é estático. Isso geraria um erro de compilação como podemos observar a seguir:

```
public class Ex03 {  
    static int CountStatic = 0;  
    int CountNonStatic = 0;  
  
    public static class Inner {  
        public void doInner() {  
            System.out.println( CountNonStatic );  
        }  
    }  
}
```

**Anotações**

305

# Programação Orientada a Objetos

---

O erro acima é básico pra quem já está no capítulo 8 desse guia, mas vou mostrar o erro que o compilador gera:

Erro: non-static variable CountNonStatic cannot be referenced from a static context.

Agora analisaremos o código a seguir:

```
public class Outer {
    static private int CountStatic = 0;
    int CountNonStatic = 0;

    public static class Inner {
        public void doInner() {
            System.out.println( CountStatic );
        }
    }
}
```

Já o código acima compila sem nenhum problema, visto que CountStatic é um membro estático da classe e pode ser acessado sem que haja uma instância de Outer.

Você deve estar se perguntando: "Oxente, mas porquê então criar uma classe estática (top-level class) se ela se comporta da mesma forma que uma classe externa ?" - Essa questão é natural na cabeça de qualquer um. A resposta nem sempre, vejamos um explicação lógica para tal.

O que nós aprendemos no capítulo 2 sobre o modificador private ? Que um membro com modificador private só pode ser acesso de dentro da própria classe certo ? Mentira era tudo mentira ! Calma é só uma brincadeira. Note que no código anterior o membro CountStatic tem o modificador private, e mesmo assim está sendo acessado de uma classe que se comporta como classe externa porém não deixa de ser interna.

O que uma classe de nível superior têm de diferentes das classes externas no tocante à relacionamento com sua classe encapsulada é um acesso direto aos membros independentes de sua modificador de acesso. A seguir temos um exemplo prático do us desse tipo de classe:

---

## Anotações

306

# Programação Orientada a Objetos

---

```
import java.util.*;
public class Week {
    private int weeknr;
    private int year;
    public Week(int weeknr, int year) {
        this.weeknr = weeknr;
        this.year = year;
    }
    public Iterator getDays() {
        return new Daylterator(this);
    }
    public int getWeeknr() {
        return weeknr;
    }
    public int getYear() {
        return year;
    }
    public static class Daylterator implements Iterator {
        private int index = 0;
        private Calendar cal = null;

        Daylterator (Week aWeek) {
            cal = new GregorianCalendar();
            cal.clear();
            cal.set(Calendar.YEAR, aWeek.getYear());
            cal.set(Calendar.WEEK_OF_YEAR, aWeek.getWeeknr());
        }
        public boolean hasNext() {
            return index < 7;
        }
        public Object next() {
            cal.set(Calendar.DAY_OF_WEEK, index++);
            return cal.getTime();
        }
        public void remove() {
            // not implemented
        }
    }
}
```

**Anotações**

307

# Programação Orientada a Objetos

---

```
public static void main(String[] args) {
    // list the days of the week
    if (args.length < 2) {
        System.err.println("Usage: java Week <weeknr> year>");
        System.exit(1);
    } else {
        try {
            int weeknr = Integer.parseInt(args[0]);
            int year = Integer.parseInt(args[1]);
            Week wk = new Week(weeknr, year);
            for (Iterator i=wk.getDays();i.hasNext();) {
                System.err.println(i.next());
            }
        } catch (NumberFormatException x) {
            System.err.println("Illegal week or year");
        }
    }
}
```

Entender o código acima o fará abrir os horizontes para o uso de classes estáticas.

## Classe interna comum

O estudo de classes internas não segue uma orientação explícita da Sun, porém é comum se deparar com questões ao longo do exame, e você precisará saber algumas regras para não vacilar (errar mesmo) alguma questão pertinentes à esse assunto.

Você NUNCA poderá ter uma instância de uma classe interna sem que haja uma instância de uma classe externa, Vejamos:

```
public class Outer {
    class Inner {
    }
}
```

---

**Anotações**

308

## Programação Orientada a Objetos

---

Fixe bem isso: "Qualquer tentativa de instanciação da classe Inner sem que haja um OBJETO do tipo Outer, não funciona e entenderemos isso no que segue". Podemos inferir a partir dessa asserção que um método estático de Outer nunca poderá instanciar uma classe Inner, visto que um método estático pode ser acessado sem um objeto propriamente dito e isso viola a regra definida anteriormente. Portanto:

```
public class Outer {  
    public static void main(String[] args) {  
        Inner i = new Inner();  
    }  
  
    class Inner {  
    }  
}
```

O código anterior causa erro de compilação, visto que o método main é estático e pode ser chamado sem que haja uma instância de Outer, portanto erro: "Uma instância de Inner só poderá existir a partir sua classe externa" - mudando um pouco o código obteríamos um êxito na compilação:

```
public class Outer {  
    public static void main(String[] args) {  
        Outer o = new Outer();  
        Inner i = o.new Inner();  
    }  
  
    class Inner {  
    }  
}
```

Compilação executado sem nenhum erro !

# Programação Orientada a Objetos

---

## Classes internas e membros externos

Uma classe interna como já foi falado, pode ser chamada de membro de classe da classe externa, portanto se é um membro, ter visibilidade para qualquer membro da classe externa, isso mesmo, qualquer membro da classe externa pode ser acesso pela classe interna, vejamos:

```
public class Outer {
    private int x = 0;
    public static void main(String[] args) {
        Outer o = new Outer();
        Inner i = o.new Inner();
        o.print();
    }
    public void print() {
        System.out.println("x before "+x);
        Inner i = new Inner();
        i.print();
    }
    class Inner {
        public void print() {
            x++;
            System.out.println("x after: "+x);
        }
    }
}
```

Note que o membro x que é privado foi acesso por Inner sem nenhum erro. O código acima resulta em:

```
x before 0
x after: 1
```

# Programação Orientada a Objetos

---

## Instanciando um objeto interno fora da classe externa

Por mais que isso possa parecer estranho, mas é possível obter uma instância de uma classe interna fora de sua classe externa, vejamos o código que segue:

```
public class TestOuter {  
    public static void main(String[] args) {  
        Outer o = new Outer();  
        Outer.Inner i = o.new Inner();  
    }  
}  
class Outer {  
    class Inner { }}
```

O código acima compila e é válido, pois para se obter uma instância Inner se informa sua classe externa - Outer.

Isso é muito interessante pode ser usado também para deixar o acoplamento de classes em um nível baixo - mas tome muito cuidado com isso!

## Referenciando a classe externa de dentro da classe interna

Ter uma referência da classe externa de dentro da classe interna pode parecer estranho, porém, às vezes, é necessário. Imaginemos uma situação em que um método da classe interna precise passar como referência a classe externa, viu ? Por mais que a classe interna tem acesso à todos os membros da classe externa, pode ser que em um caso atípico, você precise disso, ou acha que estamos colocando isso aqui por ser notório ?

# Programação Orientada a Objetos

---

```
public class TestOuter {
    public static void main(String[] args) {
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
        i.see();
    }
}
class Outer {
    private int x = 10;
    class Inner {
        public void see() {
            System.out.println(x);
            System.out.println(this);    // nota 1
            System.out.println(Outer.this); // nota 2
        }
    }
}
```

Nota 1: Observe que a palavra chave this foi usada, nesse caso ele é uma referência ao objeto de Inner !

Nota 2: Note que this foi usado com o nome da classe externa, portanto, esse é o objeto da classe externa.

## Modificadores aplicados as classes internas comuns

Como já foi falado mais tenho certeza que sua displicência o fez esquecer : ) - calma, calma é só um teste de paciência ! - uma classe interna é um membro da classe externa, portanto os modificadores a seguir podem ser aplicados à uma classe interna, seguindo as mesmas regras do capítulo 2:

final  
abstract  
public  
private  
protected  
static - com uma exceção que estudaremos mais adiante.  
strictfp

---

**Anotações**

312



# Programação Orientada a Objetos

---

## Classe interna local de método

Até agora vimos como uma classe pode ser criada dentro de outra, com escopo em toda a classe, pois bem, agora vamos reduzir esse escopo, isso mesmo, definir uma classe com escopo de método, vejamos:

```
public class Outer {  
    public void see() {  
        class Inner { }  
    }  
}
```

Note que a classe Inner foi criada dentro do método see(), apesar de não fazer nada pois nenhuma instância foi criada de Inner dentro do método see.

## Modificador padrão

Uma classe de método tem o modificador private por padrão, visto que não pode ser instanciada de nenhum outro local senão o método que encapsula a classe, vejamos um exemplo de uso:

```
public class TestOuter {  
    public static void main(String[] args) {  
        Outer o = new Outer();  
        o.see();  
    }  
}  
class Outer {  
    private int x = 10;  
    public void see() {  
        System.out.println("before "+x);  
        class Inner {  
            public Inner() {  
                x = 0;  
            }  
        }  
        System.out.println("after "+x);  
    }  
}
```

**Anotações**

313

## Programação Orientada a Objetos

---

Note que a classe Inner foi criada no meio do método, e isso é perfeitamente aceitável pelo compilador, só não tente instanciá-la antes de criar, pois isso seria um insulto ao compilador. O código acima resulta em:

```
before 10
after 10
```

A variável x não foi inicializada com 0 pois o construtor de Inner não foi chamado, vejamos o exemplo a seguir:

```
public class TestOuter {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.see();
    }
}
class Outer {
    private int x = 10;
    public void see() {
        System.out.println("before "+x);
        class Inner {
            public Inner() {
                x = 0;
            }
        }
        Inner i = new Inner();
        System.out.println("after "+x);
    }
}
```

Resultado:

```
before 10
after 0
```

Não existe forma no mundo Java para instanciar uma classe interna de método fora do próprio método, se conhecerem não esqueçam de me avisar. Tentei mostrar um exemplo de instanciação inválida mas não fui criativo o suficiente, vejamos algo:

---

**Anotações**

314

# Programação Orientada a Objetos

---

```
public class TestOuter {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.see();
    }
}

class Outer {
    private int x = 10;
    Inner x; // ERRO A CLASSE INNER NAO FOI DECLARADA
    public void see() {
        System.out.println("before "+x);
        class Inner {
            public Inner() {
                x = 0;
            }
        }
        Inner i = new Inner();
        System.out.println("after "+x);
    }
}
```

ou

```
public class TestOuter {
    public static void main(String[] args) {
        Outer o = new Outer();
        System.out.println(o.see().new Inner().s); // SERIA O CÚMULO
    }
}

class Outer {
    private int x = 10;
    public void see() {
        System.out.println("before "+x);
        class Inner {
            public String s = "string inner";
            public Inner() {
                x = 0;
            }
        }
    }
}
```

**Anotações**

315

# Programação Orientada a Objetos

---

```
        Inner i = new Inner();
        System.out.println("after "+x);
    }
}
```

## Modificadores aplicados à classe interna de método

Os únicos modificadores aplicados a essa classe são:

**abstract**  
**final**

Nunca os dois ao mesmo tempo é claro! Pesquisamos muito para saber qual seria o sentido de criar uma classe de método abstrata. Se alguém descobrir esse enigma não esqueça de me avisar...

PODE ESPERAR QUE NA PROVA PODE CAIR ALGO ASSIM, MAS NÃO VACILEM !

## Usando as variáveis automática de métodos

Uma classe interna de método não pode referenciar as variáveis locais de método, por incrível que pareça ! Você deve estar desgrendando-se, mas não pode ! E existe uma explicação plausível para isso. Imagina que uma classe interna de método chame um método e precise passar como parâmetro a própria classe, como um método da classe interna pode referenciar uma variável automática após a execução do método, uma vez que esse sendo finalizado, suas variáveis são destruídas. Você pode estar se perguntando: "Oxente, mas como a classe interna precisará de uma variável local após a execução do método ?" Calma, imagine que o objeto da classe interna, que está no heap foi passado para um outro método, e esse tem sua execução mesmo após o método ter sido finalizado. Vejamos um exemplo inválido:

**Anotações**

316

# Programação Orientada a Objetos

---

```
public class TestOuter {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.see();
    }
}

class Outer {
    private int x = 10;
    public void see() {
        int y = 5;
        System.out.println("before "+x);
        class Inner {
            public String s = "string inner";
            public Inner() {
                x = y; // nota 1
            }
        }
        Inner i = new Inner();
        System.out.println("after "+x);
    }
}
```

Nota 1: Note que a variável local `y` não pode ser referenciada dentro da classe interna de método, isso causaria um erro de compilação: *local variable y is accessed from within inner class; needs to be declared final*

Mas para aumentar a complexidade e o número de regras da linguagem Java, existe uma exceção para esse caso: Se a variável for `final` poderá ser referenciada (sei que você já tinha traduzido a mensagem de erro acima!), vejamos:

```
public class TestOuter {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.see();
    }
}
```

# Programação Orientada a Objetos

---

```
class Outer {  
    private int x = 10;  
    public void see() {  
        final int y = 5;  
        System.out.println("before "+x);  
        class Inner {  
            public String s = "string inner";  
            public Inner() {  
                x = y;  
            }  
        }  
        Inner i = new Inner();  
        System.out.println("after "+x);  
    }  
}
```

Resultado:

before 10  
after 5

## Modificadores de acesso

As mesmas regras de variáveis locais se aplicam as classes internas de métodos. Com isso podemos lembrar facilmente quais são os únicos modificadores aplicáveis às variáveis locais ?

- a) public, private, protected, padrão
- b) final, abstract, static
- c) static, final, protected
- d) abstract, final
- e) todos os modificadores

---

Anotações

318

# Programação Orientada a Objetos

---

## Classes internas anônimas

Preste bastante atenção nesse tópico, pois você pode se confundir quando se deparar com questões que envolvem classes anônimas. Vejamos sua sintaxe:

```
class Car {  
    public void run() { ... }  
}  
  
class Gol {  
  
    Car car = new Car() {  
        public void break() { ... }  
    };  
}
```

Observe que a classe Car foi criada sem nenhuma anormalidade, porém a classe Vehicle foi criada com uma instância de Car, e se você for uma camarada esperto, notou que não houve um (;) (ponto-e-vírgula) após a declaração do membro car, ao contrário, foi criada uma instância de uma classe anônima (pois não foi definido um nome - que é uma subclasse de Car) com um novo método chamado break. A priori, pode parecer complicado, mas isso tudo nadamais é do que, uma herança em local exclusivo (ou pra complicar mesmo), ou seja, somente nesse ponto preciso de redefinir a classe X. Vejamos um outro exemplo para entender melhor:

```
class Empregado {  
    public void trabalhar() {  
        System.out.println("trabalhar");  
    }  
}  
  
class QuadroFuncionario {  
    Empregado mgr = new Empregado() {  
        public void trabalhar() {  
            System.out.println("mandar");  
        }  
    };  
  
    Empregado peao = new Empregado() {  
        public void trabalhar() {
```

**Anotações**

319

# Programação Orientada a Objetos

---

```
        System.out.println("executar");
    }
};
}
```

Note que estamos realizando uma criação de métodos polimórficos para os objetos `peao` e `mgr` onde ambos estendem de `Empregado`.

Você é capaz de descobrir qual o erro da seguinte listagem ?

```
class Empregado {
    public void trabalhar() {
        System.out.println("trabalhar");
    }
}

class QuadroFuncionario {
    Empregado mgr = new Empregado() {
        public void trabalhar() {
            System.out.println("mandar");
        }

        public void demite() {
            System.out.println("demite");
        }
    };

    public void work() {
        mgr.trabalhar();
        mgr.demite();
    }
}
```

Observe que `mgr` é uma instância de `Empregado`, o qual define o métodos `trabalhar`, porém há uma tentativa de execução do método (que realmente foi definido na classe anônima) `mgr.demite()`, que o compilador acusa erro de ignorância, ou seja, o compilador só irá conhecer os métodos definidos na classe pai - qualquer tentativa de execução ou chamada de um método não existente, causará um erro de compilação: *cannot resolve symbol*

**Anotações**

320



# Programação Orientada a Objetos

---

Não podemos esquecer...

```
public class Test {
    static public void enclosingMethod(final String arg1, int arg2) {
        final String local = "A local final variable";
        String nonfinal = "A local non-final variable";
        Object obj = new Object() {
            public String toString() {
                return local + "," + arg1;
            }
        };

        System.out.println(obj.toString());
    }

    public static void main(String[] args) {
        enclosingMethod("fim", 0);
    }
}
```

Esse código é perfeitamente compilado pois a classe anônima que está sendo criada dentro do método `enclosingMethod` referencia a variável local que é definida como `final` e `arg1` que também é um argumento com modificador `final`. Qualquer tentativa de referencia `arg2` ou `nonfinal` dentro do método acima, causará erro de compilação, se estiver duvidando por tentar.

Se você realmente precisar de referenciar uma variável local dentro de uma classe anônima podemos dar uma dica, provavelmente você nunca precisará mas, só pra não perder a viagem: use array. Vejamos um exemplo:

```
public class Ex02 {
    public static void main(String[] args) {
        final int[] age = new int[1];
        System.out.println("Before: "+age[0]);
        Pessoa p = new Pessoa() {
            public void getAge() {
                age[0] = 25;
            }
        };
        p.getAge();
        System.out.println("After: "+age[0]);
    }
}
```

**Anotações**

321

# Programação Orientada a Objetos

---

```
    }  
}  
abstract class Pessoa {  
    abstract void getAge();  
}
```

Se você está lembrado do que estudamos nos capítulos iniciais, vai lembrar que o array é final seus elementos não. Portanto o resultado desse código será:

Before: 0  
After: 25

## Implementando uma interface anonimamente

Você também poderá se deparar com o código a seguir:

```
public class Ex04 {  
    public static void main(String[] args) {  
        acao( new Evento() {  
            public void clicar() {  
                System.out.println("clique");  
            }  
            public void arrastar() {  
                System.out.println("arrastou");  
            }  
        });  
    }  
    public static void acao(Evento e) {  
        e.clicar();  
        e.arrastar();  
    }  
}  
interface Evento {  
    public abstract void clicar();  
    public abstract void arrastar();  
}
```

---

**Anotações**

322

# Programação Orientada a Objetos

---

Você deve estar se perguntando, como pode passar para um método uma instância de uma interface ? Por mais que possa parecer isso, não é o que está acontecendo, na verdade estamos criando uma classe anônima que implementa a interface Evento, tanto é que a nova classe teve que implementar todos os métodos de Eventos (clicar e arrastar), se um desses dois métodos não forem implementados na classe anônima um erro de compilação seria exibido. Pode parecer estranho, mas não é ! O que criamos aqui foi um implementador de classe como algumas literaturas assim o definem.

Mesmo que esse assunto de classes anônimas possa parecer diferentes as regras de herança, polimorfismo se aplicam, ou seja, uma tentativa de criar uma classe anônima que herda de uma classe abstrata deve obrigatoriamente implementar todos os seus métodos, isso porque você não pode definir uma classe anônima abstrata.

```
public class Ex04 {
    public static void main(String[] args) {
        acao( new Evento() {
            public void clicar() {
                System.out.println("clizou");
            }
            public void arrastar() {
                System.out.println("arrastou");
            }
        });
    }
    public static void acao(Evento e) {
        e.clicar();
        e.arrastar();
    }
}
interface Evento {
    public abstract void clicar();
    public abstract void arrastar();
}
```

# Programação Orientada a Objetos

---

Vamos mudar um pouco o código acima:

```
public class Ex05 {
    public static void main(String[] args) {
        acao( new Evento() {
            public void clicar() {
                System.out.println("clizou");
            }
        });
    }
    public static void acao(Evento e) {
        e.clicar();
        e.arrastar();
    }
}
abstract class Evento {
    public abstract void clicar();
    public abstract void arrastar();
}
```

O código acima gerará um erro de compilação, pois o método arrastar não foi implementado na nova classe (isso seria perfeitamente possível se a nova classe fosse abstrata) porém como é uma classe concreta deve implementá-lo.

## Classes finais não podem ser anônimas

O título anterior já falou tudo o que precisava ser falado, mas vamos ser menos sucinto. Aprendemos algum dia de nossa vida que uma classe marcada com o modificador final nunca poderia ser estendida, conseqüentemente nunca poderíamos ter uma classe anônimas a partir de uma classe final, pois uma classe anônima nada mais é do que uma herança de uma outra classe onde a subclasse não tem nome. Vejamos isso na prática:

# Programação Orientada a Objetos

---

```
public class Ex06 {
    public static void main(String[] args) {
        acao( new Boy() {
            public String get() {
                System.out.println("boy anonymous");
            }
        });
    }
    public static void acao(Boy e) {
        System.out.println(e.get());
    }
}
final class Boy {
    public String get() {
        return "boy";
    }
}
```

O código acima gerará o seguinte erro de compilação:

Erro: cannot inherit from final Boy

Bom não podemos esquecer das demais regras envolvendo classes e herança, como a visibilidade para com classes de outros pacotes, enfim, se tiver alguma dúvida quanto a isso volte ao capítulo 2.

## Um código real para esse caso...

Vamos ver um uso prático desse tipo de classe, inclusive vamos utilizar o mesmo exemplo do início do capítulo quando falávamos de classes estáticas, modificaremos um pouco o código exibido anteriormente:

```
import java.util.*;
public class WeekAnonymous {
    private int weeknr;
    private int year;
    public WeekAnonymous(int weeknr, int year) {
        this.weeknr = weeknr;
        this.year = year;
    }
}
```

**Anotações**

325

```
public Iterator getDays() {
    return new Iterator() {
        private int index = 0;
        private Calendar cal = null;
        private Calendar getCalendar () {
            if (cal == null) {
                cal = new GregorianCalendar();
                cal.clear();
                cal.set(Calendar.YEAR, year);
                cal.set(Calendar.WEEK_OF_YEAR, weeknr);
            }
            return cal;
        }
        public boolean hasNext() {
            return index < 7;
        }
        public Object next() {
            getCalendar().set(Calendar.DAY_OF_WEEK, index++);
            return getCalendar().getTime();
        }
        public void remove() {
            // not implemented
        }
    };
}

public static void main(String[] args) {
    // list the days of the week
    if (args.length < 2) {
        System.err.println("Usage: java Week <weeknr> year");
        System.exit(1);
    } else {
        try {
            int weeknr = Integer.parseInt(args[0]);
            int year = Integer.parseInt(args[1]);
            Week wk = new Week(weeknr, year);
            for (Iterator i=wk.getDays();i.hasNext();) {
                System.err.println(i.next());
            }
        } catch (NumberFormatException x) {
            System.err.println("Illegal week or year");
        }
    }
}
```

## Programação Orientada a Objetos

---

```
    }  
  }  
}
```

Note que o código acima não criou uma classe estática como no início do capítulo, nesse caso foi criada uma classe anônima para resolver o mesmo problema, com isso podemos inferir que a utilização de classes internas é um assunto extremamente relativo, podemos ser utilizado de diversas maneiras.

Esses exemplos foram tirados do guia de certificação da Sun !

Para encerrar esse assunto de classes anônimas, gostaríamos de explicar que o uso de classes anônimas são frequentemente utilizadas em desenvolvimento de ambientes gráficos quando se usa AWT ou Swing principalmente para tratamento de eventos.

## Capítulo IX - Threads

### Uma visão sobre Threads

Antes de qualquer estudo mais aprofundado sobre threads, gostaríamos de deixar bem claro alguns pontos importantes para o sucesso da absorção do conteúdo desse capítulo.

Com o crescimento do poder de processamento dos computadores modernos, e as inovações dos sistemas operacionais para criar um ambiente de trabalho amigável junto aos seus usuários, uma série de inovações foram acrescentadas, e, um recurso que ganhou espaço e tem se mostrado bastante interessante é o processamento paralelo, onde um processo pode se dividir em inúmeros processos de dimensões menores para resolver problemas distintos. Calma que não é mágica. O que acontece na verdade é o seguinte: em um ambiente monoprocessado, um processo maior que é responsável pelos subprocessos divide o tempo do processador entre esses subprocessos, ou seja, começa a executar o subprocesso 01, após algum tempo (esse tempo depende da JVM e SO) esse processo fica aguardando sua vez de continuar sendo executado, a partir daí o processador está livre para execução de um outro processo, até que novamente o subprocesso 01 volta a ser executado, até que por algum motivo ele termina. Esse recurso é bastante intrigante pois permite-nos resolver problemas que levaríamos um tempo bem superior, uma vez que sem esse poder de escalonar o processador, o processo começaria e terminaria para assim liberar o processador a iniciar a execução de um novo processo.

Em java, temos duas alternativas para implementar o recurso de multi-thread:

- a) Estendendo da Classe Thread
- b) Implementando a Interface Runnable



# Programação Orientada a Objetos

---

Vejamos um exemplo para cada método:

a) Estendendo de java.lang.Thread:

```
public class Execucao {
    public static void main(String[] args) {
        Proc p = new Proc();
        p.start();
        while (true) {
            System.out.println("thread main executando");
        }
    }
}

class Proc extends Thread {
    public void run() {
        while (true) {
            System.out.println("thread executando");
        }
    }
}
```

b) Implementando a interface Runnable

```
public class Execucao {
    public static void main(String[] args) {
        Proc p = new Proc();
        Thread t = new Thread(p);
        t.start();
        while (true) {
            System.out.println("thread main executando");
        }
    }
}

class Proc implements Runnable {
    public void run() {
        while (true) {
            System.out.println("thread executando");
        }
    }
}
```

**Anotações**

329

# Programação Orientada a Objetos

---

Por mais que os códigos acima possam não ter muito sentido, mais posso lhe garantir que funcionam. O método main é uma thread e instancia um objeto p que também é instancia de uma classe que estende de Thread. O método start() chamado, informa ao escalonador de thread que coloque na fila de execução a Thread p, mas não pense que a thread p vai executar imediatamente quando o método start foi chamado, somente sinaliza o escalonador (hei eu sou p e sou thread, me execute assim que puder, obrigado).

## O que uma thread executa

Você já deve ter notado que nos exemplos anteriores o código da Thread fica no método run, porém mostramos duas formas de se executar um thread, então vamos entender. Quando uma classe herda de Thread e implementa seu método abstrato run, o código que se encontrar dentro é executado. Já quando uma classe implementa Runnable e implementa o método run, o método que é executado é o método da classe que implementa a interface Runnable. Você nunca poderá chamar diretamente o método run de uma classe Thread, pois assim, o processador não será dividido entre os diversos processos, você deverá chamar o método start para que o escalonador saiba da existência da Thread, e deixe que o método run seja executado pelo escalonador. Se você não chamar o método start, sua thread nunca será executada.

A assinatura do método run é:

```
public void run()
```

Note que o método não lança nenhuma exceção checada, isso significa que você não poderá fazer.

# Programação Orientada a Objetos

---

## Métodos importantes da classe Thread

Os métodos a seguir precisam ser decorados, lembrados na hora do exame:

`run()` - é o código que a thread executará.

`start()` - sinaliza à JVM que a thread pode ser executada, mas saiba que essa execução não é garantida quando esse método é chamado, e isso pode depender da JVM.

`isAlive()` - volta true se a thread está sendo executada e ainda não terminou.

`sleep()` - suspende a execução da thread por um tempo determinado;

`yield()` - torna o estado de uma thread executável para que thread com prioridades equivalentes possam ser processadas, isso será estudando mais adiante;

`currentThread()` - é um método estático da classe Thread que volta qual a thread que está sendo executada.

`getName()` - volta o nome da Thread, você pode especificar o nome de uma Thread com o método `setName()` ou na construção da mesma, pois existe os construtores sobrecarregados.

## Métodos herdados de `java.lang.Object` relacionados com Threads

Os métodos a seguir pertencem à classe Object, porém estão relacionados com a programação Multi-Threading:

```
public final void wait()  
public final void notify()  
public final void notifyAll()
```

Examinaremos os métodos acima citados, porém antes de qualquer coisa precisamos entender os estados de uma thread, e é isso que vamos entender a partir de agora.

# Programação Orientada a Objetos

---

## Estados dos segmentos

**Novo** - estado que uma thread fica no momento de sua instanciação, antes da chamada do método start();

**Executável** - estado em que a thread fica disponível para ser executada e no aguardo do escalonador de thread, esperando a sua vez de se executar;

**Execução** - Momento em que a thread está executando, está operando, trabalhando, pagando seu salário e sua execução continua até que por algum motivo (que veremos brevemente) esse momento é interrompido;

**Espera/Bloqueio/Suspensão** - esse estado pode ser dar por inúmeros motivos, e vamos tentar esplanar alguns. Uma thread em seu grande momento de glória (sua execução) pode se bloquear por que algum recurso ou objeto não está disponível, por isso seu estado pode ficar bloqueado, até que esse recurso/objeto esteja disponível novamente assim seu estado torna-se executável, ou então, uma thread pode ficar suspensa porque o programador definiu um tempo de espera, assim que esse tempo expirar essa thread volta ao estado executável para continuar seus serviços;

**Inativo** - a partir do momento em que o método run() foi concluído, a thread se tornará inativa, porém ainda existirá o objeto na memória, somente não como uma linha de execução, e não poderá novamente se estartado, ou seja, qualquer tentativa de chamada do método start() após a conclusão do métodos run(), uma exceção será lançada e não duvide;

## Impedindo uma thread de ser executada

### Suspendendo uma thread

Pense em uma situação em que você tenha 10 Thread rodando simultâneamente (sei, eu fui redundante agora), porém uma delas você deseja tarefa realizada, ela aguarde 5 minutos para prosseguir suas tarefas, por mais que isso pareça loucura, mas é possível, você poderá usar o método sleep para tornar uma thread no modo suspenso até que esse tempo expire, vejamos sua sintaxe:

---

### Anotações

---

---

---

---

332

# Programação Orientada a Objetos

---

```
try {  
    sleep(5*60*1000);  
}  
catch (InterruptedException ex) {  
}
```

O código acima, faz com que a thread espere por 5 minutos até voltar ao estado Executável, porém é uma alternativa quando se pretende executar as thread de forma mais organizada, apesar de que lembre-se sempre: VOCÊ NUNCA TERÁ GARANTIAS QUANTO A ORDEM DA EXECUÇÃO DAS THREAD, POIS ISSO VAI DEPENDER DO ESCALONADOR !

Qualquer pergunta com relação à tentativa de garantir um ordem, ou sequencia de execução das thread, não hesite em marcar a alternativa que diz que não existe garantias para tal tarefa.

Não pense que o tempo que for especificado no método sleep será exatamente o tempo que a thread ficará sem executar, a única garantia que teremos é que esse será o tempo mínimo, porém após o seu término, seu estado passa para Executável e não Execução.

Tenha em mente que o método sleep é um método estático, portanto você não poderá chamar o método sleep de um thread x ou y, somente da thread que estiver em estado de execução.

Vejamos um exemplo prático:

```
public class Exec11 {  
    public static void main(String args[]) {  
        Contador c1 = new Contador();  
        c1.setQtde(10);  
        c1.setName("t001");  
        c1.start();  
        Contador c2 = new Contador();  
        c2.setQtde(15);  
        c2.setName("t002");  
        c2.start();  
    }  
}
```

---

**Anotações**

333

# Programação Orientada a Objetos

---

```
class Contador extends Thread {
    private int qtde = 0;
    public void run() {
        for (int i=0; i <= 100; i++) {
            if ((i%qtde) == 0) {
                System.out.println(Thread.currentThread().getName()+"> "+i);
            }
            try {
                sleep(500);
            }
            catch (InterruptedException ex) {
            }
        }
    }

    public void setQtde(int value) {
        this.qtde = value;
        if (this.qtde == 0) this.qtde = 10;
    }
}
```

Por mais que tenha se tentado estabelecer uma ordem, se você executar o código acima irá perceber que não há garantias de ordem na execução.

## Prioridades

A prioridade também é assunto contundente, pois cada JVM define a sua maneira de escolher a thread que passará do estado de executável para execução. Pois a especificação Java não define nada a respeito. Por isso não confie em prioridades para garantir uma execução sincronizadas de suas threads, use-as para melhorar a eficiência de suas tarefas. Para definir uma prioridade para uma thread você usará o método

```
public void setPriority(xxx)
```

onde xxx - é um número inteiro compreendido entre 1 e 10

A prioridade padrão é 5, ou seja, se não for definida nenhuma prioridade, será assumido o valor 5 para a thread.

Você poderá cair em uma situação em que poderá ter várias thread com a mesma

**Anotações**

334

# Programação Orientada a Objetos

---

prioridade, e desejar que thread que estiver executando dê lugar para outras serem processadas, e para isso existe o método `yield()` que faz justamente isso, torna o estado de execução da thread atual para executável e dá o lugar para as demais thread de prioridades semelhantes. Não obstante a isso, não garante que a thread que vai ser executada, não seja a mesma que acabou de sair do estado de execução, portanto esse método também não garante fazer o que se propõe.

## Hierarquia de Threads

Imagine que você tenha uma thread Y que só poderá ser executada quando a thread X concluir sua tarefa, você poderá ligar uma thread à outra, usando o método `join()`, vamos entender isso a partir de agora:

Examinemos o código a seguir:

```
public class Exec12 {
    public static void main(String args[]) {
        try {
            Contador c1 = new Contador();
            c1.setQtde(10);
            c1.setName("t001");
            c1.start();
            c1.join();
            for (int i=0; i <= 100; i++) {
                if ((i%5) == 0) {
                    System.out.println(Thread.currentThread().getName()+"> "+i);
                }
            }
        }
        catch (InterruptedException e) {
        }
    }
}
```

```
class Contador extends Thread {
```

**Anotações**

335

# Programação Orientada a Objetos

---

```
private int qtde = 0;
public void run() {
    for (int i=0; i <= 100; i++) {
        if ((i%qtde) == 0) {
            System.out.println(Thread.currentThread().getName()+"> "+i);
        }
        try {
            sleep(500);
        }
        catch (InterruptedException ex) {
        }
    }
}

public void setQtde(int value) {
    this.qtde = value;
    if (this.qtde == 0) this.qtde = 10;
}
}
```

Resultado do código acima:

```
t001> 0
t001> 10
t001> 20
t001> 30
t001> 40
t001> 50
t001> 60
t001> 70
t001> 80
t001> 90
t001> 100
main> 0
main> 5
main> 10
main> 15
main> 20
main> 25
main> 30
main> 35
```

**Anotações**

336



## Programação Orientada a Objetos

---

```
main> 40  
main> 45  
main> 50  
main> 55  
main> 60  
main> 65  
main> 70  
main> 75  
main> 80  
main> 85  
main> 90  
main> 95  
main> 100
```

O exemplo acima garante que a thread main só será executada quando a Thread t001 estiver inativa.

Vamos complicar um pouco.... O método join é sobrecarregado e pode receber um valor long que corresponde à quantidade millissegundos que a thread main (em nosso caso) anterior espera até que t001 se concluir, se a mesma não ficar inativa no tempo informado, nada mais é garantido, ou seja, o escalonador poderá iniciar a execução de ambas as threads. Vamos alterar um pouco o código e observar os resultados:

# Programação Orientada a Objetos

---

```
public class Exec13 {
    public static void main(String args[]) {
        try {
            Contador c1 = new Contador();
            c1.setQtde(10);
            c1.setName("t001");
            c1.start();
            c1.join(5000);
            for (int i=0; i <= 100; i++) {
                if ((i%5) == 0) {
                    System.out.println(Thread.currentThread().getName()+"> "+i);
                }
            }
        }
        catch (InterruptedException e) {
        }
    }
}

class Contador extends Thread {
    private int qtde = 0;
    public void run() {
        for (int i=0; i <= 100; i++) {
            if ((i%qtde) == 0) {
                System.out.println(Thread.currentThread().getName()+"> "+i);
            }
            try {
                sleep(250);
            }
            catch (InterruptedException ex) {
            }
        }
    }

    public void setQtde(int value) {
        this.qtde = value;
        if (this.qtde == 0) this.qtde = 10;
    }
}
```

**Anotações**

338

# Programação Orientada a Objetos

---

## Resultado

```
t001> 0
t001> 10
main> 0
main> 5
main> 10
main> 15
main> 20
main> 25
main> 30
main> 35
main> 40
main> 45
main> 50
main> 55
main> 60
main> 65
main> 70
main> 75
main> 80
main> 85
main> 90
main> 95
main> 100
t001> 20
t001> 30
t001> 40
t001> 50
t001> 60
t001> 70
t001> 80
t001> 90
t001> 100
```

Note que a thread main só esperou pelo tempo de 5000 (5 segundos) a partir daí ela começou sua execução.

## Anotações

339

# Programação Orientada a Objetos

---

## Sincronização

O assunto agora é um pouco mais complicado do que o estudado até agora, pois trata de como duas ou mais threads podem compartilhar o mesmo objeto, ou seja, quais são os riscos que corremos quando dois objetos podem ser vistos simultaneamente.

*Cenário:*

Imaginemos um processo de compra on-line pela Internet, onde inúmeras pessoas podem consultar os itens disponíveis em estoque e realizar seus pedidos. Pois bem, como não queremos causar situações indigestas com nossos clientes, precisamos garantir com seus pedidos sejam faturados corretamente. Bom onde queremos chegar ? Imagine que temos 5 aparelhos celulares S55 SIEMENS em nosso estoque e que foi lançado uma promoção desse aparelho e 200 pessoas estão dispostas a entrar no tapa por um aparelho, bem temos que garantir que esse processo seja concretizado sem maiores problemas. (tudo isso foi dito para se esplanar a situação... ) Vejamos como resolver esse problema:

```
public class PedidoCompra {  
    public static void main(String[] args) {  
        Produto p = new Produto(5);  
        Thread[] t = new Thread[15];  
        for (int i=0; i < t.length; i++) {  
            t[i] = new Thread(p);  
            t[i].setName("Cliente: "+i);  
            t[i].start();  
        }  
    }  
}
```

**Anotações**

340

# Programação Orientada a Objetos

---

```
class Produto implements Runnable {
    private int estoque = 5;

    public void run() {
        try {
            for (int i=0; i<2; i++) {
                efetuarPedido();
            }
        }
        catch (Exception ex) {
        }
    }

    public void efetuarPedido() {
        try {
            if (this.estoque > 0) {
                System.out.println("Pedido faturado para o cliente
                "+Thread.currentThread().getName());
                Thread.sleep(250);
                this.estoque--;
            } else {
                System.out.println("Não tem estoque para o cliente
                "+Thread.currentThread().getName());
            }
        }
        catch (Exception ex) {
        }
    }

    public Produto(int value) {
        this.estoque = value;
    }
}
```

Não tentar vender o programa acima para alguma loja que você será escurraçado!

O código de efetuar pedido, sempre efetuará o pedido tendo ou não estoque, note que na saída houve 10 cliente que efetuaram seus pedidos com estoque estourado:

**Anotações**

341

## Programação Orientada a Objetos

---

Pedido faturado para o cliente Cliente: 0  
Pedido faturado para o cliente Cliente: 1  
Pedido faturado para o cliente Cliente: 2  
Pedido faturado para o cliente Cliente: 3  
Pedido faturado para o cliente Cliente: 5  
Pedido faturado para o cliente Cliente: 6  
Pedido faturado para o cliente Cliente: 8  
Pedido faturado para o cliente Cliente: 9  
Pedido faturado para o cliente Cliente: 10  
Pedido faturado para o cliente Cliente: 11  
Pedido faturado para o cliente Cliente: 4  
Pedido faturado para o cliente Cliente: 7  
Pedido faturado para o cliente Cliente: 12  
Pedido faturado para o cliente Cliente: 13  
Pedido faturado para o cliente Cliente: 14  
Pedido faturado para o cliente Cliente: 0  
Pedido faturado para o cliente Cliente: 1  
Pedido faturado para o cliente Cliente: 2  
Pedido faturado para o cliente Cliente: 3  
Nao tem estoque para o cliente Cliente: 5  
Nao tem estoque para o cliente Cliente: 6  
Nao tem estoque para o cliente Cliente: 8  
Nao tem estoque para o cliente Cliente: 9  
Nao tem estoque para o cliente Cliente: 11  
Nao tem estoque para o cliente Cliente: 10  
Nao tem estoque para o cliente Cliente: 4  
Nao tem estoque para o cliente Cliente: 12  
Nao tem estoque para o cliente Cliente: 7  
Nao tem estoque para o cliente Cliente: 13  
Nao tem estoque para o cliente Cliente: 14

O que queremos é não permitir que haja faturamento caso o estoque esteja negativo.

Pelo resultado não é muito difícil deduzir o que aconteceu nesse processamento - embora você possa executar e obter outro resultado. Observe que todos os pedidos só foram efetuados porque há no método efetuarPedido uma suspensão da execução das thread para sua concretização, ou seja, até o momento em que a thread volta após sua suspensão, o pedido ainda não foi efetuado, com isso, outros clientes podem efetuar seus pedidos - quando esse ciclo se repetir para os 5 primeiros clientes, aí sim, não será mais possível concretizar pedidos, pois o estoque do item se tornou 0. Porém não foi exatamente isso que aconteceu em

**Anotações**

342

# Programação Orientada a Objetos

---

nosso exemplo anterior. E você é capaz de descobrir porque ?

A grande sacada do programa anterior é suspender a thread por um tempo para que antes de concluir sua operação, dando lugar as outras, fazendo com que o estoque fique negativo.

Mas existe uma solução para que esse "erro" seja revertido, ou seja, não permitir que dois clientes possam concluir seus pedidos ao mesmo tempo.

Vamos alterar o código anterior, e acrescentar o modificar synchronized que não permite com que 2 thread executem o mesmo método ao mesmo tempo, ou seja, as demais thread ficam esperando até que a thread em execução conclua seu processamento para assim iniciar o seu.

## Resultado

Pedido faturado para o cliente Cliente: 0  
Pedido faturado para o cliente Cliente: 1  
Pedido faturado para o cliente Cliente: 2  
Pedido faturado para o cliente Cliente: 3  
Pedido faturado para o cliente Cliente: 4  
Nao tem estoque para o cliente Cliente: 5  
Nao tem estoque para o cliente Cliente: 6  
Nao tem estoque para o cliente Cliente: 7  
Nao tem estoque para o cliente Cliente: 8  
Nao tem estoque para o cliente Cliente: 10  
Nao tem estoque para o cliente Cliente: 11  
Nao tem estoque para o cliente Cliente: 12  
Nao tem estoque para o cliente Cliente: 13  
Nao tem estoque para o cliente Cliente: 14  
Nao tem estoque para o cliente Cliente: 9  
Nao tem estoque para o cliente Cliente: 0  
Nao tem estoque para o cliente Cliente: 1  
Nao tem estoque para o cliente Cliente: 2  
Nao tem estoque para o cliente Cliente: 3  
Nao tem estoque para o cliente Cliente: 4  
Nao tem estoque para o cliente Cliente: 5  
Nao tem estoque para o cliente Cliente: 6  
Nao tem estoque para o cliente Cliente: 7  
Nao tem estoque para o cliente Cliente: 8  
Nao tem estoque para o cliente Cliente: 10

## Anotações

---

---

---

---

343

# Programação Orientada a Objetos

---

Nao tem estoque para o cliente Cliente: 11  
Nao tem estoque para o cliente Cliente: 12  
Nao tem estoque para o cliente Cliente: 13  
Nao tem estoque para o cliente Cliente: 14  
Nao tem estoque para o cliente Cliente: 9

Note que a saída agora melhorou ! E houve faturamento somente dos 5 itens !

Pensamento Entrópico: Fazer condições e checagens para evitar que o estoque fique negativo ! Lembre-se sempre, você nunca terá certeza da ordem da execução das thread, ou seja, uma thread pode começar (tendo estoque 5) e terminar quando o estoque já acabou, outras threads foram mais rapidas.....

Outro exemplo:

```
public class Test {  
    public static void main(String[] args) {  
        Creditorio c1 = new Creditorio();  
        Creditorio c2 = new Creditorio();  
        Thread t1 = new Thread(c1);  
        t1.setName("t1");  
        Thread t2 = new Thread(c2);  
        t2.setName("t2");  
        t1.start();  
        t2.start();  
    }  
}  
  
class Creditorio implements Runnable {  
    public void run() {  
        Cliente c = Cliente.getInstance();  
        System.out.println("Iniciando transacao de crediario...  
"+Thread.currentThread().getName()+" - "+c.toString());  
        c.analisarFicha();  
    }  
}
```

```
class Cliente {
```

**Anotações**

344



# Programação Orientada a Objetos

---

```
private static Cliente singleton;
public static Cliente getInstance() {
    if (singleton == null) singleton = new Cliente();
    return singleton;
}

synchronized void analisarFicha() {
    try {
        Thread.sleep(500);
        System.out.println("Analisando ficha...."
            + Thread.currentThread().getName());
        Thread.sleep(500);
        liberarFicha();
    }
    catch (Exception e) {
    }
}

synchronized void liberarFicha() {
    try {
        Thread.sleep(500);
        System.out.println("Liberando ficha...."
            + Thread.currentThread().getName());
    }
    catch (Exception e) {
    }
}
}
```

Vejamos um exemplo em que dois métodos são sincronizados e um não, e observe que pelo resultado nenhum problema há quando se chama um método sem sincroniza mesmo no caso do objeto estar bloqueado:

---

**Anotações**

345

# Programação Orientada a Objetos

---

```
public class Test {
    public static void main(String[] args) {
        Credario c1 = new Credario();
        Credario c2 = new Credario();
        Negativacao n1 = new Negativacao();
        Thread t1 = new Thread(c1);
        t1.setName("t1");
        Thread t2 = new Thread(c2);
        t2.setName("t2");
        Thread t3 = new Thread(n1);
        t3.setName("t3");
        t1.start();
        t2.start();
        t3.start();
    }
}

class Credario implements Runnable {
    public void run() {
        Cliente c = Cliente.getInstance();
        System.out.println("Iniciando transacao de credario...
        "+Thread.currentThread().getName()+" - "+c.toString());
        c.analisarFicha();
    }
}

class Negativacao implements Runnable {
    public void run() {
        Cliente c = Cliente.getInstance();
        System.out.println("Iniciando transacao de negativacao...
        "+Thread.currentThread().getName()+" - "+c.toString());
        c.negativar();
    }
}
```

# Programação Orientada a Objetos

---

```
class Cliente {
    private static Cliente singleton;
    public static Cliente getInstance() {
        if (singleton == null) singleton = new Cliente();
        return singleton;
    }

    synchronized void analisarFicha() {
        try {
            Thread.sleep(500);
            System.out.println("Analisando ficha...."
                + Thread.currentThread().getName());
            Thread.sleep(500);
            liberarFicha();
        }
        catch (Exception e) {
        }
    }

    synchronized void liberarFicha() {
        try {
            Thread.sleep(500);
            System.out.println("Liberando ficha...."
                + Thread.currentThread().getName());
        }
        catch (Exception e) {
        }
    }

    void negativar() {
        try {
            Thread.sleep(500);
            System.out.println("Negativando ficha...."
                + Thread.currentThread().getName());
        }
        catch (Exception e) {
        }
    }
}
```

**Anotações**

---

---

---

---

347

# Programação Orientada a Objetos

---

## Resultado

Iniciando transacao de crediario... t2 - Cliente@ba34f2  
Iniciando transacao de negativacao... t3 - Cliente@ba34f2  
Iniciando transacao de crediario... t1 - Cliente@ba34f2  
Analisando ficha.... t2  
Negativando ficha.... t3  
Liberando ficha.... t2  
Analisando ficha.... t1  
Liberando ficha.... t1

Note que o processo de negativação se processou mesmo não terminando o processo de crediário.

Agora vamos sincronizar o método sincronização para ver o que acontece:

Iniciando transacao de crediario... t1 - Cliente@ba34f2  
Iniciando transacao de negativacao... t3 - Cliente@ba34f2  
Iniciando transacao de crediario... t2 - Cliente@ba34f2  
Analisando ficha.... t1  
Liberando ficha.... t1  
Negativando ficha.... t3  
Analisando ficha.... t2  
Liberando ficha.... t2

Note que agora o processo de nativação (método negativar) só se procedeu após o término do processo de liberação. Isso nos leva a pensar em uma situação inesperada: o impasse..... vejamos no item seguinte:

## Deadlock (impasse)

Essa é uma situação em que duas thread ficam esperando um bloqueio, ou seja, a thread A fica aguardando o bloqueio que está sob bloqueio da thread B e a thread B está aguardando o bloqueio que está em posse da thread A, uma situação assim ocasionará em um travamento do sistema. Embora não seja uma situação comum, quando ocorre o sistema fica paralizado e além disso não é muito fácil encontrar o problema.

## Anotações

---

---

---

---

348

# Programação Orientada a Objetos

---

O código a seguir mostra um exemplo bem simples de um deadlock:

```
public class TestDeadLock {
    public static void main(String[] args) {
        Crediario c1 = new Crediario();
        Negativacao n1 = new Negativacao();
        Thread t1 = new Thread(c1);
        t1.setName("t1");
        Thread t2 = new Thread(n1);
        t2.setName("t2");
        t1.start();
        t2.start();
    }
}

class Crediario implements Runnable {
    public void run() {
        Class1.m1();
    }
}

class Negativacao implements Runnable {
    public void run() {
        Class2.m1();
    }
}
```

# Programação Orientada a Objetos

---

```
class Class1 {
    static synchronized void m1() {
        try {
            Thread.sleep(500);
            System.out.println("executando class1 m1");
            Class2.m2();
        }
        catch (Exception e) {
        }
    }

    static synchronized void m2() {
        try {
            Thread.sleep(500);
            System.out.println("executando class1 m2");
        }
        catch (Exception e) {
        }
    }
}

class Class2 {
    static synchronized void m1() {
        try {
            Thread.sleep(500);
            System.out.println("executando class2 m1");
            Class1.m2();
        }
        catch (Exception e) {
        }
    }

    static synchronized void m2() {
        try {
            Thread.sleep(500);
            System.out.println("executando class2 m2");
        }
        catch (Exception e) {
        }
    }
}
```

**Anotações**

350

# Programação Orientada a Objetos

---

## Interação entre segmentos

Eventualmente você precisará notificar um segmento que B que o segmento A finalizou sua tarefa, fazendo com que o segmento B possa entrar em estado de executável para se candidatar a executar.

A classe Object define três métodos para realização dessa interação, são eles:

```
wait()
notify()
notifyAll()
```

Só um lembrete: como esses métodos são definidos na classe Object, todos os demais objetos tem esses métodos, até mesmo a classe Thread, portanto não vacile.....

Eles SEMPRE devem ser chamados de um contexto sincronizados.

Um exemplo bem prático para esse recurso pode ser visto no trecho abaixo, pois uma thread de leitura aguarda a criação de um arquivo que é gerado por outra thread, vejamos o código:

```
import java.io.*;
public class InterThread {
    public static void main(String[] args) {
        try {
            Arquivo arq = new Arquivo(new File("saida.txt"));
            Thread[] a = new Thread[20];
            for (int i=0; i < a.length; i++) {
                a[i] = new Thread(new Leitura(arq));
                a[i].setName( ""+i);
                a[i].start();
            }
            Thread b = new Thread( new Gravacao(arq) );
            b.start();
            b.join();
            System.out.println("Processo finalizado...");
        }
        catch (Exception ex) {}
    }
}
```

**Anotações**

351

# Programação Orientada a Objetos

---

```
class Gravacao implements Runnable {
    Arquivo arq;
    public Gravacao(Arquivo value) {
        arq = value;
    }
    public void run() {
        arq.gravar();
    }
}
```

```
class Leitura implements Runnable {
    Arquivo arq;
    public Leitura(Arquivo value) {
        arq = value;
    }
    public void run() {
        arq.ler();
    }
}
```

```
class Arquivo {
    File file;

    public Arquivo(File value) {
        file = value;
    }

    synchronized public void ler() {
        try {
            if (!file.exists()){
                wait();
            }
            System.out.print( "thread#
            "+Thread.currentThread().getName() + ">>> ");
            if (file.exists()){
                FileInputStream fis = new FileInputStream(file);
                int in;
                while ((in=fis.read())!=-1) {
                    System.out.print((char)in);
                }
            }
        }
    }
}
```

**Anotações**

352



```
        fis.close();
    }
}
catch (Exception e) {
    e.printStackTrace();
}
}

synchronized public void gravar() {
    try {
        if (!file.exists()){
            FileOutputStream fos = new FileOutputStream(file);
            for (int i=0; i < 5; i++) {
                fos.write( "linha "+i).getBytes() );
            }
            fos.write("\n".getBytes());
            fos.close();
            System.out.print("Entrou no notify");
            notify();
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

Como você pode notar, o código acima dispara 10 thread para leitura de um arquivo que muito provavelmente nem tenha sido criado, portanto as thread entram no estado wait até que a thread crie o arquivo liberando para a leitura.

# Programação Orientada a Objetos

---

## Exercícios Teóricos

- 1) O Java foi criado a partir de quais linguagens de programação?
- 2) Quais as principais razões que levaram os engenheiros da Sun a desenvolver uma nova linguagem de programação?
- 3) Por que um sistema escrito em Java pode rodar em qualquer plataforma?
- 4) Marque com “X” na alternativa correta. Para rodar uma aplicação Java, por mais simples que seja, é necessário possuir uma Java Virtual Machine.  
( ) Verdadeiro ( ) Falso
- 5) Qual o principal papel do JCP (Java Community Process)?
- 6) Por quem é formado o JCP?
- 7) Marque “V” para verdadeiro ou “F” para falso.
  - a. ( ) A Sun, como dona da tecnologia poderá mudar os rumos do Java a qualquer tempo.
  - b. ( ) Para a Sun fazer uma alteração no Java será necessário se submeter ao JCP.
  - c. ( ) O JCP é o responsável por defender os interesses da indústria da comunidade Java e da Sun.
- 8) Como está estruturada a plataforma Java?
- 9) O que é Java?
- 10) Marque com um “X” na alternativa correta.  
É possível compilar um código Java para uma plataforma específica?  
( ) Verdadeiro ( ) Falso
- 11) Quais as principais características do Java?
- 12) Qual a função do Garbage Collector?
- 13) Quais são as três tecnologias Java para desenvolvimento de aplicativos?

**Anotações**

354

# Programação Orientada a Objetos

---

- 14) Quais são as convenções estabelecidas para a declaração de classes, métodos e variáveis?
- 15) O Java é compilado ou interpretado?
- 16) Para que serve a Java Virtual Machine?
- 17) Como inserimos comentários em um código Java?
- 18) Um dos principais motivos que contribuíram para o desenvolvimento da linguagem Java foi:
- a. ☐ O nome da linguagem.
  - b. ☐ O desenvolvimento da Internet.
  - c. ☐ A linguagem ser relativamente simples.
  - d. ☐ O desempenho da linguagem em termos de velocidade.
- 19) Por que o aspecto da utilização do Java em multiplataforma é muito importante para os programadores?
- 20) Qual das características seguintes não diz respeito a linguagem Java:
- a. ☐ Pode ser executada em qualquer computador, independente de existir uma máquina virtual java instalada.
  - b. ☐ É uma linguagem compilada e interpretada.
  - c. ☐ O desempenho dos aplicativos escritos em Java, com relação à velocidade de execução, é inferior à maioria das linguagens de programação.
  - d. ☐ É uma linguagem com um bom nível de segurança.
- 21) A sequência de desenvolvimento de um programa em Java é:
- a. ☐ Compilação, digitação e execução.
  - b. ☐ Digitação, execução e compilação.
  - c. ☐ Digitação, compilação e execução.
  - d. ☐ Digitação, execução e testes de funcionamento.
- 22) Qual a principal característica que distingue a plataforma Java das demais existentes?

**Anotações**

355

# Programação Orientada a Objetos

---

- 23) Para a linguagem Java, as variáveis PATH e CLASSPATH correspondem a:
- a. ☐ Variáveis usadas em um programa Java.
  - b. ☐ Uma variável de ambiente e um caminho para a execução dos programas Java.
  - c. ☐ Um caminho para encontrar as classes e um caminho para encontrar os aplicativos da linguagem Java.
  - d. ☐ Um caminho para encontrar os aplicativos e um caminho para encontrar as classes da linguagem Java.
- 24) Qual a diferença entre uma variável do tipo primitivo e uma variável do tipo reference?
- 25) Quais são os tipos primitivos da linguagem Java?
- 26) O que são variáveis locais?
- 27) O que é um array?
- 28) Como um array unidimensional pode ser declarado?
- 29) Todo array declarado como uma variável local deve ser inicializado.
- ☐ Verdadeiro      ☐ Falso
- 30) Todo array é uma variável do tipo:
- ☐ Primitivo      ☐ Reference
- 31) Como obter o tamanho de um array?
- 32) Como um array Bidimensional pode ser declarado?
- 33) Qual a função da classe Math?
- 34) Qual a função da classe String?
- 35) Qual método da classe String retorna o tamanho da String?
- 36) Qual método da classe String converte qualquer tipo de dados em String?

**Anotações**

356

# Programação Orientada a Objetos

---

37) Explique os seguintes conceitos :

Objeto:

Classe:

Herança:

Mensagem:

Atributo:

Método:

Polimorfismo:

38) Demonstre graficamente uma classe.

39) Construa em java uma classe Conta Corrente com os seguintes atributos e métodos:

**Atributos**

Nome do Correntista

Saldo

Limite do cheque especial

**Métodos**

getNome

getSaldo

setLimite

Anotações

357

# Programação Orientada a Objetos

---

## Exercícios Práticos

- 1) Crie um programa que recebe três nomes quaisquer por meio da linha de execução do programa, e os imprima na tela da seguinte maneira: o primeiro e o último nome serão impressos na primeira linha um após o outro, o outro nome (segundo) será impresso na segunda linha.
  
- 2) Faça um programa que receba a quantidade e o valor de três produtos, no seguinte formato: quantidade1 valor1 quantidade2 valor2 quantidade3 valor3. O programa deve calcular esses valores seguindo a fórmula  $\text{total} = \text{quantidade1} \times \text{valor1} + \text{quantidade2} \times \text{valor2} + \text{quantidade3} \times \text{valor3}$ . O valor total deve ser apresentado no final da execução.
  
- 3) Crie um programa que receba a largura e o comprimento de um lote de terra e mostre a área total existente.
  
- 4) Crie um programa que receba quatro valores quaisquer e mostre a média, somatório entre eles e o resto da divisão do somatório por cada um dos valores.
  
- 5) Faça um aplicativo que receba três valores inteiros na linha de comando e mostre o maior dentre eles.
  
- 6) Faça um programa que apresente o total da soma dos cem primeiros números inteiros ( $1+2+3+\dots+99+100$ ).
  
- 7) Faça um aplicativo que calcule o produto dos inteiros ímpares de 1 a 15 e exiba o resultado na tela.
  
- 8) Crie uma classe que gere um numero aleatório entre os valores máximo e mínimo recebidos do usuário na linha de comando.
  
- 9) Crie um aplicativo que receba uma frase e mostre-a de forma invertida.

# Programação Orientada a Objetos

---

10) Crie um aplicativo que mostre o efeito abaixo:

```
J
Ja
Jav
Java
Jav
Ja
J
```

11) Crie uma classe que leia um parâmetro passado na linha de comando no seguinte formato: dd/mm/aaaa. Desta maneira, a classe devera ser executada como java Exe04 11/09/2001. A saída gerada por essa execução deve ser a impressão separada do dia, do mês e do ano - utilizando apenas os métodos da classe String.

12) Queremos imprimir uma tabela de números entre 100 e 200 seguindo o seguinte formato:

- Se o numero for divisível por 2 imprimir:  
<numero> e divisível por 2
  - Se o numero for divisível por 3 imprimir:  
<numero> e divisível por 3
  - Se o numero for divisível por 2 e 3 imprimir:  
<numero> e divisível por 2 e divisível por 3 e portanto e divisível por 6
- Utilize o controle de fluxo for para navegar do limite inferior ate o superior.

13) Quantos números divisíveis por 6 existem entre 14 e 130? Comprove utilizando um laço while.

14) Uma escola precisa de um programa que controle a média das notas dos alunos de cada classe e a média das notas de todos os alunos da escola. Sabendo que essa escola possui 3 classes com 5 alunos em cada classe, gerando um total de 15 notas, crie um programa que receba as notas de cada aluno de cada classe e no final apresente a média de cada classe e a média da escola em geral.

**Anotações**

359

## Programação Orientada a Objetos

---

15) Uma empresa quer transmitir dados por telefone, mas está preocupada com a possibilidade de seus telefones estarem grampeados. Todos seus dados são transmitidos como inteiros de quatro dígitos. Eles pedem para você escrever um programa que criptografará seus dados de modo que estes possam ser transmitidos mais seguramente. Seu aplicativo deve ler um inteiro de quatro dígitos inserido pelo usuário na linha de comando e criptografá-lo como segue: substitua cada dígito por (a soma deste dígito mais 1). Então troque o primeiro dígito pelo terceiro e troque o segundo pelo quarto. A seguir imprima o inteiro criptografado. Escreva um aplicativo separado que recebe como entrada um inteiro de quatro dígitos criptografado e o descriptografa para formar o número original.

16) O fatorial de um número inteiro não negativo  $n$  é escrito como  $n!$  (pronuncia-se fatorial de  $n$ ) e é definido como segue:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1 \text{ (para valores de } n \text{ maiores que ou iguais a } 1\text{)}$$

e

$$n! = 1 \text{ (para } n=0\text{)}$$

Por exemplo:  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , o que dá 120.

17) Escreva um aplicativo que lê um inteiro não negativo via linha de comando, computa e imprima seu fatorial.

18) A série de Fibonacci 0,1,1,2,3,5,8,13,21.....  
inicia com 0 e 1 e tem a prioridade de que cada número de Fibonacci subsequente é a soma dos dois anteriores que o procedem.

19) Escreva um aplicativo que recebe a entrada do número de vezes que deve ocorrer a série.

20) Escreva um aplicativo que recebe entradas de texto e envia o texto para saída com letras em maiúsculas e em minúsculas.

21) Faça um aplicativo que verifique se uma palavra é um palíndromo. Ex: Ana.

---

**Anotações**

360



## Programação Orientada a Objetos

---

22) Nesta classe foram declaradas algumas variáveis para armazenar dados relativos a uma pessoa:

*nome, dataNascimento, rg, sexo (o char M para o sexo MASCULINO e o F para FEMININO) e salario*

Nosso objetivo é imprimir o texto correto levando em consideração os dados armazenados nas variáveis, para isto iremos utilizar o condicional if.

### INSTRUÇÕES:

- a) Criar uma variável do tipo boolean para armazenar se a pessoa é CLT (true) ou não (false)
- b) Vamos utilizar o condicional if para atender as seguintes especificações:
  - Quando o sexo for masculino (M) imprimir:  
O Senhor <nome da pessoa> é portador do rg...
  - Quando o sexo for feminino (F) imprimir:  
A Senhora <nome da pessoa> é portadora do rg ..
  - Quando o sexo for inválido (diferente de F ou M) imprimir:  
O Senhor(a) <nome da pessoa> é portador(a)
  - Quando a pessoa for CLT imprimir:  
está registrado com o salário de R\$ <salario>
  - Quando a pessoa for Autônoma imprimir:  
foi contratado pelo valor de R\$ <salario>

```
public class ControleFluxo03 {  
    public static void main(String[] args) {  
        String nome;  
        String dataNascimento;  
        String rg;  
        char sexo; // UTILIZE 'M' para MASCULINO e 'F' para FEMININO  
        float salario;  
        nome = "Manuel da Silva";  
        dataNascimento = "22/04/1980";  
        rg = "29345432";  
        sexo = 'M';  
        salario = 2500.00f;  
    }  
}
```

Anotações

361

# Programação Orientada a Objetos

---

23) Considere o aplicativo chamado **Administracao** e a classe **Populacao**, apresentados a seguir e codifique as questões de 1 a 4:

```
public class Populacao{  
    private int pop[ ][ ];  
    public int estados, municipios;
```

- 1) Codificar neste quadro o construtor, da classe, que recebe como parâmetros o número de estados e o número de municípios, e cria a matriz de populações.

```
public void atualizarPopulacao(int i, int j, int populacao){  
    if (i>=0 && i<4 && j>=0 && j<5 && populacao > 0)  
        pop[i][j] = populacao;  
}
```

- 2) Codificar neste quadro o método que determina a população média de um dado estado.

```
}
```

Anotações

362

# Programação Orientada a Objetos

---

```
public class Administracao{  
    public static void main (String p[ ]){
```

3) Declarar variáveis

```
        for (i=0; i<4; i++){  
            for (j=0; j<5; j++){  
                n = Integer.parseInt(JOptionPane.showInputDialog  
                    ("Informe a população da cidade " +  
                     String.valueOf(j+1) + "º do estado " +  
                     String.valueOf(i+1)));  
                pop.atualizarPopulacao(i, j, n);  
            }  
        }  
    }
```

4) Codificar, neste espaço, a parte do aplicativo que recebe via janela de diálogo o número de um estado e exibe, também via janela de diálogo, a população média deste estado.

Utilizar para isto uma matriz de população de 4 estados com 5 municípios cada, gerada através da classe Populacao.

```
    }  
}
```

Anotações

363

## Programação Orientada a Objetos

---

24) Crie uma aplicação que simule o cálculo do valor final de uma venda, dependendo da forma de pagamento escolhida pelo usuário. O usuário entra com um valor, escolhe a forma de pagamento e o cálculo do preço final é realizado conforme os seguintes critérios: para pagamento em dinheiro, desconto de 5%, para pagamento em cheque, acréscimo de 5%, para pagamento com cartão, acréscimo de 10%. A figura abaixo apresenta a janela de execução deste exercício.

A janela 'Cálculo do preço final' possui um título azul com ícone de aplicativo. O formulário contém:

- Um campo de texto para 'Entre com o valor da venda:'.
- Um menu suspenso para 'Escolha a forma de pagto:' com 'Cartão' selecionado.
- Um campo de texto para 'Preço final a pagar'.
- Dois botões: 'Calcular' e 'Limpar'.

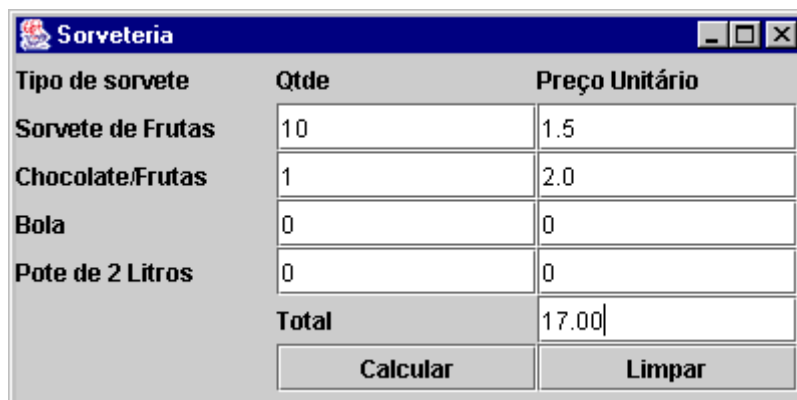
25) Crie uma aplicação que simule o cadastramento de pessoas. O usuário digita o nome e endereço de uma pessoa, escolhe o sexo e o estado civil por meio de componentes do tipo Combo. Ao pressionar o botão mostrar, todos os dados cadastrados são copiados para um componente TextArea, conforme apresenta a figura abaixo.

A janela 'Cadastramento de Pessoas' possui um título azul com ícone de aplicativo. O formulário contém:

- Campos de texto para 'Nome:' (contendo 'Jadir') e 'Endereço:' (contendo 'Rua a').
- Menus suspensos para 'Sexo' (com 'Masculino' selecionado) e 'Estado Civil' (com 'Casado' selecionado).
- Dois botões: 'Mostrar' e 'Limpar'.
- Uma área de texto (TextArea) na parte inferior que exibe os dados cadastrados: 'Nome: Jadir', 'Endereço: Rua a', 'Sexo: Masculino' e 'E.C.: Casado'.

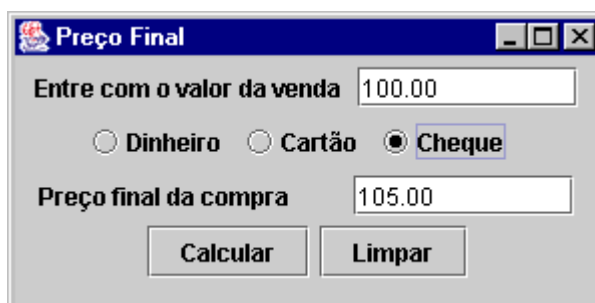
## Programação Orientada a Objetos

26) Crie uma aplicação que simule vendas de sorvete em um sorveteria, de acordo com o apresentado na figura abaixo.



Tipo de sorvete	Qtde	Preço Unitário
Sorvete de Frutas	10	1.5
Chocolate/Frutas	1	2.0
Bola	0	0
Pote de 2 Litros	0	0
<b>Total</b>		17.00
<b>Calcular</b>		<b>Limpar</b>

27) Crie uma aplicação que simule o cálculo do valor final de uma venda, dependendo da forma de pagamento escolhida pelo usuário. O usuário entra com um valor, escolhe a forma de pagamento e o cálculo do preço final é realizado conforme os seguintes critérios: para pagamento em dinheiro, desconto de 5%, para pagamento em cheque, acréscimo de 5%, para pagamento com cartão, acréscimo de 10%. A figura abaixo apresenta a janela de execução deste exercício.



Entre com o valor da venda: 100.00

☐ Dinheiro ☐ Cartão ☒ Cheque

Preço final da compra: 105.00

**Calcular** **Limpar**

Anotações

365

# Programação Orientada a Objetos

---

28) Desenvolva uma aplicação que simule uma calculadora conforme mostra a figura abaixo.

**Observação:**

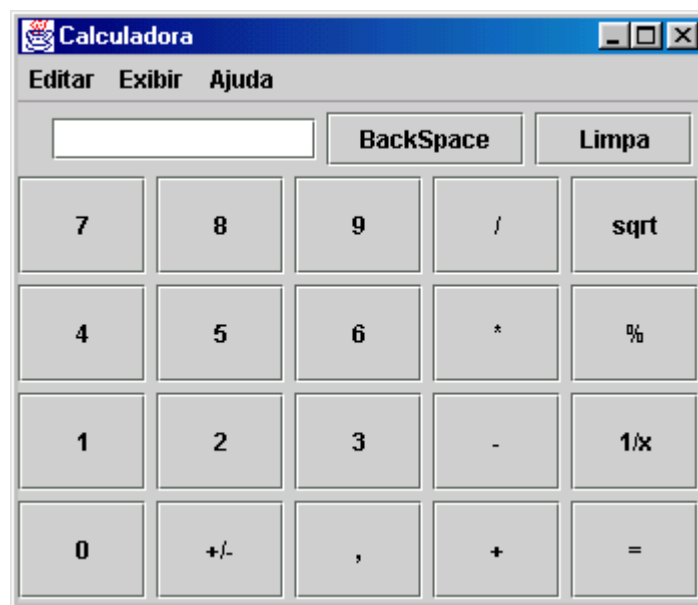
**Menu Editar**

- Copiar
- Colar

**Menu Exibir**

**Menu Ajuda**

- Sobre a Calculadora

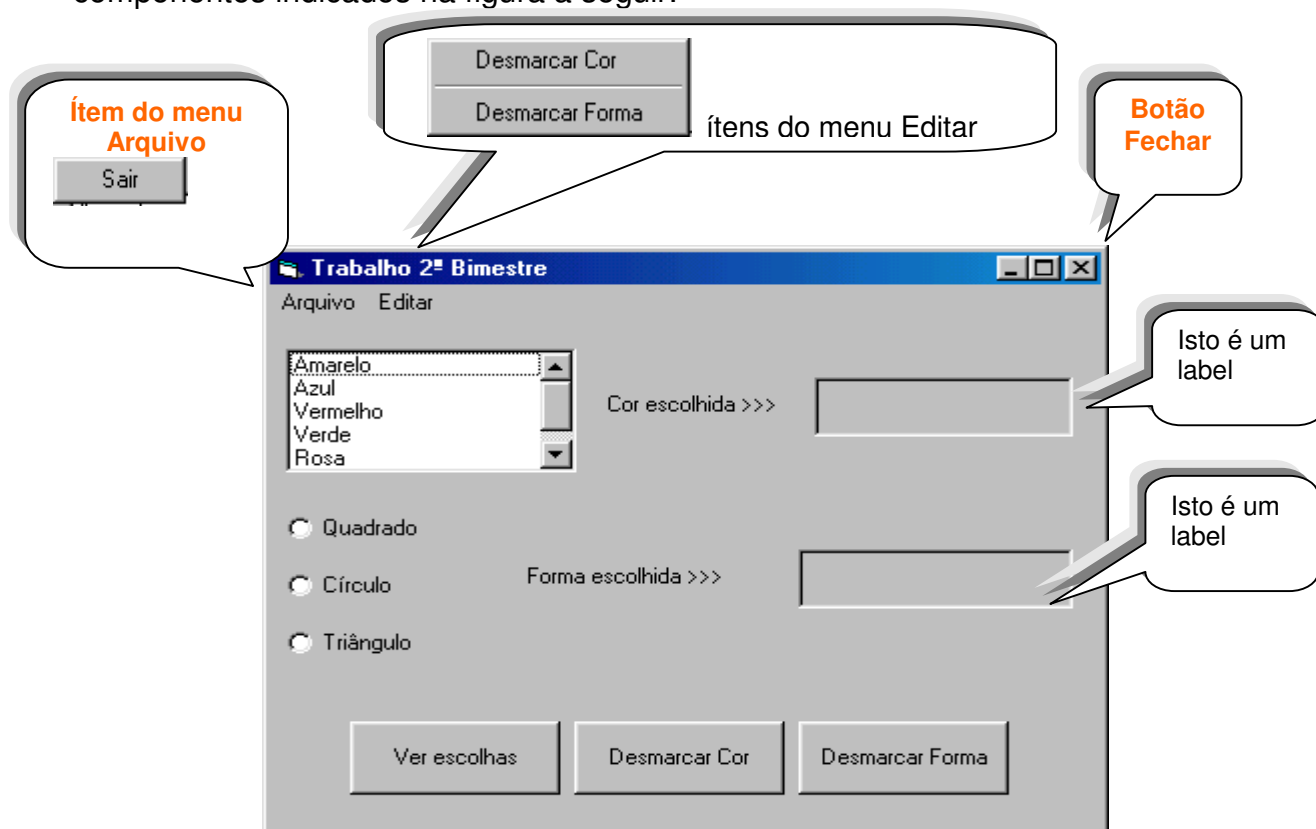


**Anotações**

366

# Programação Orientada a Objetos

29) Codificar uma classe MinhaJanela (extensão de JFrame), que contenha os componentes indicados na figura a seguir:



## Observações a respeito da janela:

- ✎ A lista de cores deve ser carregada dinamicamente com cores que serão fornecidas via linha de comando do aplicativo que utilizar esta janela.

## Funcionalidade dos componentes da janela:

Menu Sair	Deve finalizar o processamento
Menu Desmarcar Cor	Deve desmarcar a cor selecionada dentro da lista de cores, e limpar o label onde está cor está exibida
Menu Desmarcar Forma	Deve desmarcar o RadioButton selecionado, e limpar o label onde a forma escolhida está exibida
Botão Desmarcar Cor	Deve desmarcar a cor selecionada dentro da lista de cores, e limpar o label onde está cor está exibida

Anotações

367

## Programação Orientada a Objetos

Botão Desmarcar Forma	Deve desmarcar o RadioButton selecionado, e limpar o label onde a forma escolhida está exibida
Botão Ver escolhas	Deve exibir uma caixa de mensagem (JOptionPane) mostrando a cor e a forma escolhidas pelo usuário
Lista de cores	Ao clicar em um ítem desta lista, o nome da cor marcada deve ser exibido no label correspondente
Grupo de Radio Buttons	Ao clicar em um ítem deste grupo, o nome da forma marcada deve ser exibido no label correspondente
Botão Fechar da janela	Deve encerrar o processamento

As funções executadas por mais de um componente devem ser codificadas, cada uma, em um módulo específico que será chamado pelo evento do componente.



# Programação Orientada a Objetos

---

30) Leia o que está sendo solicitado nos códigos e seguir e os implementes.

```
/**
 * 1) Adicione os seguintes atributos na classe Agencia:
 * - numero (String)
 * - banco (int)
 */
```

```
public class Agencia {
```

```
}
```

```
/**
 * 1) Adicione os seguintes atributos na classe Cliente:
 * - nome (String)
 * - cpf (String)
 */
```

----- **XX** -----

```
public class Cliente {
}
```

```
/**
 * 1) Adicione os seguintes atributos na classe Conta:
 * - saldo (double)
 * - numero (String)
 * - titular (String)
 * - agencia (int)
 * - banco (int)
 */
```

----- **XX** -----

```
public class Conta {
}
```

```
/**
 * 1) Crie um objeto da classe Agencia
 * 2) Inicialize todos os atributos deste objeto.
 * 3) Imprima os valores dos atributos da classe Agencia de forma a obter o
 * seguinte resultado:
```

**Anotações**

369

# Programação Orientada a Objetos

---

```
* -----  
* AGENCIA: 1   BANCO : 234  
* -----  
* Sugestoes: Utilize '\t' para tab  
* Ex: System.out.println("Texto\tTexto");  
* produziria a seguinte saida:  
* Texto  Texto  
*/
```

----- **XX** -----

```
public class TestaAgencia {
```

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

```
/**
```

```
* 1) Crie um objeto da classe Cliente  
* 2) Inicialize todos os atributos deste objeto.  
* 3) Imprima os valores dos atributos da classe Cliente de forma a obter o  
* seguinte resultado:
```

```
*
```

```
*
```

```
* -----
```

```
* NOME: FULANO CPF : 123154
```

```
* -----
```

```
*
```

```
* Sugestoes: Utilize '\t' para tab
```

```
* Ex: System.out.println("Texto\tTexto");
```

```
* produziria a seguinte saida:
```

```
* Texto  Texto
```

```
*/
```

----- **XX** -----

---

**Anotações**

370

# Programação Orientada a Objetos

---

```
public class TestaCliente {
```

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

```
/**
```

```
 * 1) Crie um objeto da classe Conta  
 * 2) Inicialize todos os atributos deste objeto.  
 * 3) Imprima os valores dos atributos da classe Conta de forma a obter o  
 * seguinte resultado:
```

```
 *
```

```
 * -----
```

```
 * AGENCIA: 1   BANCO : 234
```

```
 * NUMERO : 01945
```

```
 * TITULAR: FASP
```

```
 * SALDO : R$10000.0
```

```
 * -----
```

```
 *
```

```
 * Sugestoes: Utilize '\t' para tab
```

```
 * Ex: System.out.println("Texto\tTexto");
```

```
 * produziria a seguinte saida:
```

```
 * Texto  Texto
```

```
 */
```

```
----- XX -----
```

```
public class TestaConta {
```

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

```
    }
```

```
 }
```

```
 /**
```

```
 * O metodo inicializaConta e util para evitar a necessidade de inicializacao
```

```
 * atributo a atributo, desta forma podemos chamar um unico metodo e passar  
 todos
```

```
 * os parametros em uma unica chamada.
```

```
 *
```

**Anotações**

371

# Programação Orientada a Objetos

---

\* 1) Implemente os metodos que nao foram implementados na classe Conta de acordo com a  
\* especificacao nos metodos.  
\*/

**public class Conta {**

double saldo;  
String numero;  
String titular;  
int agencia;  
int banco;

/\*\*

\* @param saldoInicial Saldo Inicial da conta  
\* @param num Numero da conta  
\* @param tit Titular da conta  
\* @param ag Agencia a qual a conta pertence  
\* @param bc Banco a qual a agencia pertence  
\*/

void inicializaConta(double saldoInicial, String num, String tit, int ag, int bc) {  
 System.out.println("Inicializando uma conta com os seguintes dados:");  
 saldo = saldoInicial;  
 numero = num;  
 titular = tit;  
 agencia = ag;  
 banco = bc;  
}

/\*\*

\* @param valor: valor a ser sacado da conta  
\*/

/\*

\* 1. Verificar se o valor do saque e positivo.  
\* 2. Verificar se ha saldo suficiente para efetuar o saque  
\* 2.1. Se o saldo for suficiente, efetuar o saque  
\* 2.2. Se o saldo for insuficiente imprimir na tela que o saldo e Insuficiente  
\*/

void saque(double valor) {  
}

**Anotações**

372

# Programação Orientada a Objetos

---

```
/**
 * @param valor Valor a ser depositado da conta
 */
/*
 * Verificar se o valor do deposito e positivo.
 */
void deposito(double valor) {
}

/**
 * Metodo para impressao de todos os dados da classe
 */
void imprimeDados() {
    System.out.println("\n-----");
    System.out.println("AGENCIA:\t"+agencia+"\t BANCO:\t"+banco);
    System.out.println("NUMERO: \t"+numero);
    System.out.println("TITULAR: \t"+titular);
    System.out.println("SALDO: \t"+saldo);
    System.out.println("-----\n");
}

/**
 * @return saldo da conta
 */
double getSaldo() {
    return saldo;
}
}

/*
 *
 * Analise o codigo
 *
 */
```

----- **XX** -----

**Anotações**

373

# Programação Orientada a Objetos

---

```
public class Cliente {
```

```
    String nome;  
    String cpf;
```

```
    /**  
     * @param n nome do cliente  
     * @param c cpf do cliente  
     */
```

```
    public void inicializaCliente(String n, String c) {  
        cpf = c;  
        nome = n;  
    }
```

```
    /**  
     * Metodo para impressao de todos os dados da classe  
     */
```

```
    void imprimeDados() {  
        System.out.println("-----");  
        System.out.println("Nome do cliente : " + nome);  
        System.out.println("CPF: " + cpf);  
        System.out.println("-----");  
    }  
}
```

----- **XX** -----

---

**Anotações**

374

---

---

---

---

# Programação Orientada a Objetos

---

Leia o que está sendo solicitado nos códigos e seguir e os implementes.

```
/*
 *
 * 1) Seguindo o modelo da classe Conta, crie o metodo inicializaAgencia() da
classe Agencia
 *
 */
public class Agencia {
    String numero;
    int banco;
    /**
     * Metodo para impressao de todos os dados da classe
     */
    void imprimeDados() {
        System.out.println("-----");
        System.out.println("Agencia no. " + numero);
        System.out.println("Banco no." + banco);
        System.out.println("-----");
    }
}

public class TestaConta {

    public static void main(String[] args) {
        // Criacao da conta
        // Inicializacao da conta
        // Impressao dos dados da conta
        // Saque da conta
        // Impressao dos dados da conta
        // Deposito em conta
        // Impressao dos dados da conta
        // Impressao do saldo da conta, utilizando o metodo getSaldo();
    }
}
```

----- XX -----

**Anotações**

375

# Programação Orientada a Objetos

---

```
public class TestaCliente {  
  
    public static void main(String[] args) {  
        // Criacao do cliente  
        // Inicializacao do cliente  
        // Impressao dos dados do cliente  
    }  
}
```

----- **XX** -----

```
public class TestaCliente {  
  
    public static void main(String[] args) {  
        // Criacao do cliente  
        // Inicializacao do cliente  
        // Impressao dos dados do cliente  
    }  
}
```

----- **XX** -----

**Anotações**

376

---

---

---

---