

**Programação Orientada a Objetos**

Linguagem de  
Programação JAVA

# 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.

Anotações

2

# Programação Orientada a Objetos

---

## Introdução a Linguagem Java

### Características do Java

- Java é sintática e morfológicamente 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)

### Anotações

---

---

---

---

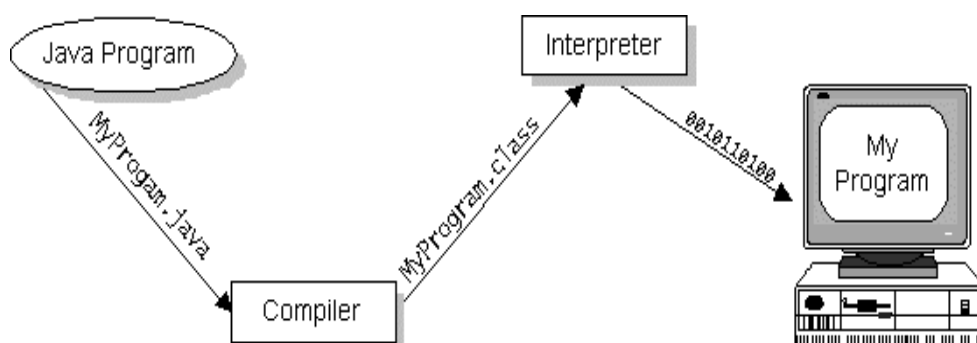
# Programação Orientada a Objetos

---

## 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;



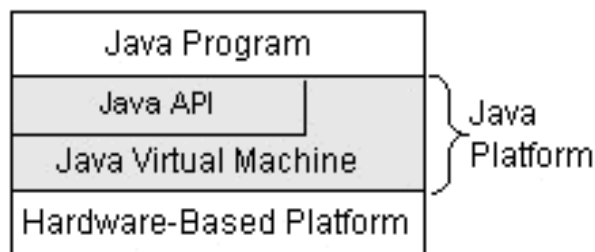
## Plataforma Java

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

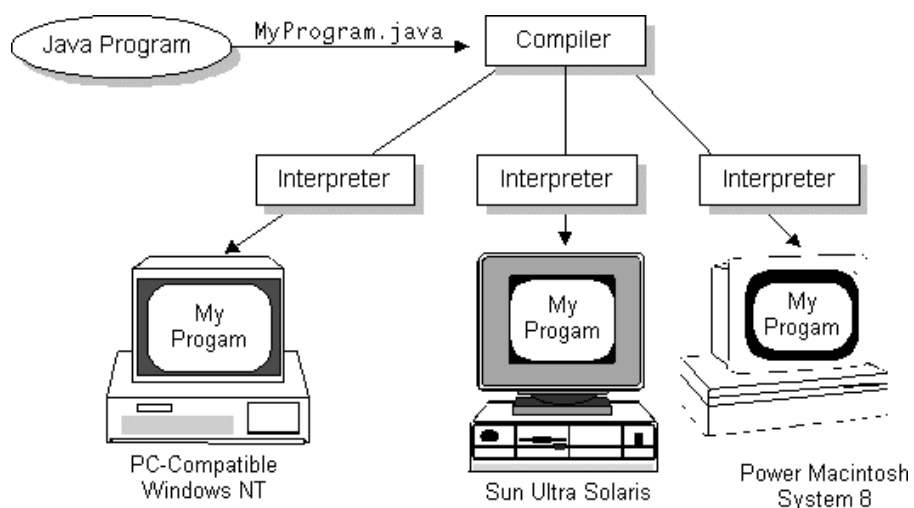
4

# 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:

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))

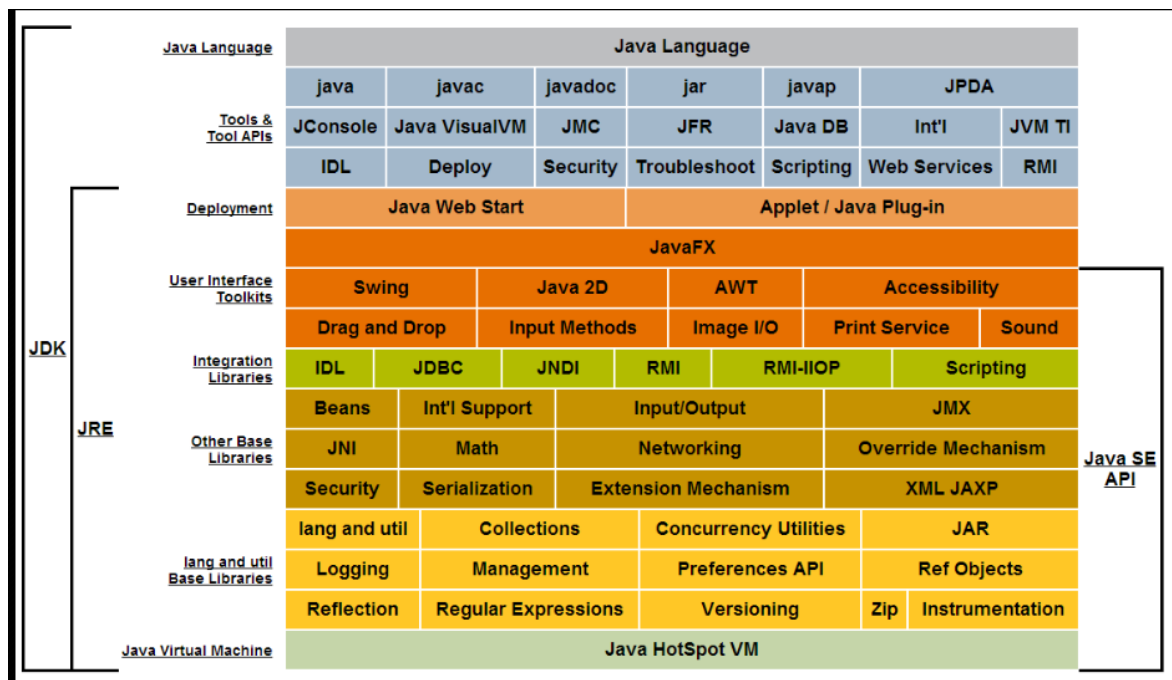
Eclipse([www.eclipse.org](http://www.eclipse.org))

---

Anotações

6

# Programação Orientada a Objetos



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

Anotações

7

# Programação Orientada a Objetos

---

## O Compilador javac

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

Argumento	Descrição
classpath path	Localização das classes. Sobrepõe 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

8



# 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

---

---

---

---

# 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.

Anotações

10

# 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**

12

# 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

---

Anotações

15

# 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**

16





# 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

---

---

---

---

# 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";
```

---

Anotações

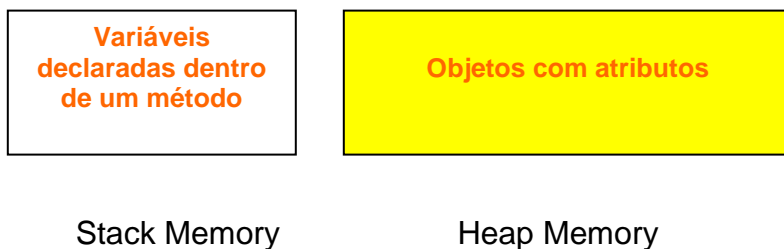
21

# Programação Orientada a Objetos

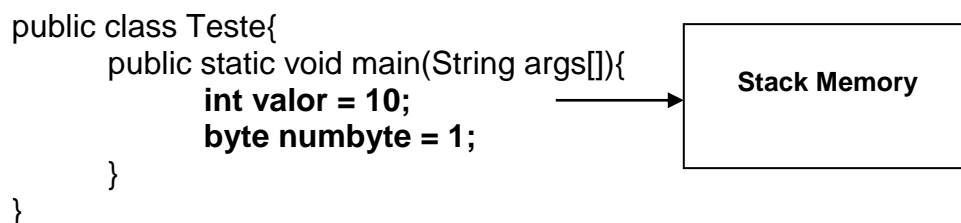
---

## 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.



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,



# Programação Orientada a Objetos

---

## 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);
    }
}
```

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);
    }
}
```

# Programação Orientada a Objetos

---

## 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"

Anotações

---

---

---

---

24



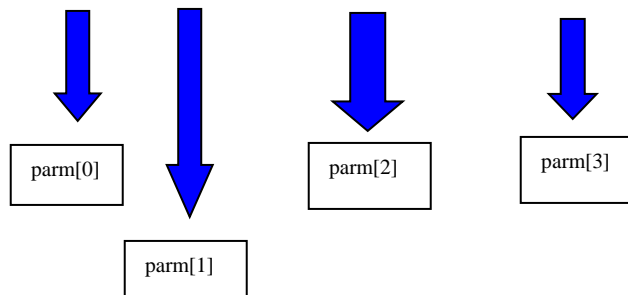
# Programação Orientada a Objetos

## 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

# 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

26

# 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

---

---

---

---

# 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

---

---

---

---

# 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.

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## 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

30

# 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 <u>e</u> 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 <u>pi</u></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

31

# Programação Orientada a Objetos

---

## Exemplos de métodos da classe Math

O método **Math.abs** retorna o valor absoluto de cada número.

Exemplo: `System.out.println( Math.abs(-10));` // Resultado igual a 10

O método **Math.sqrt** retorna a raiz quadrada de um número.

Exemplo: `System.out.println( Math.sqrt(625));` // Resultado igual a 25

O método **Math.pow** retorna a potência de número por outro.

Exemplo: `System.out.println( Math.pow(2,3));` // Resultado igual a 8

O método **Math.random** retorna um número aleatório entre 0.0 e menor que 1.0.

Exemplo: `System.out.println( Math.random() );` // Resultado: qualquer valor entre 0.0 (inclusive) e menor que 1.0

## Classe Integer, Float e Double – Conversão para primitivo

(1)	<code>String str = "35"; Integer x; x = Integer.valueOf(str );  Resultado → x = 35</code>	Método estático que converte em um objeto Integer o conteúdo de um objeto String.
(2)	<code>int nro; String str = "348"; nro = Integer.parseInt(str);  Resultado → nro = 348</code>	Método estático que converte para o formato primitivo <b>int</b> o conteúdo de um objeto String.

### Para Float e Double:

```
x = Float.valueOf(str );  
x = Float.valueOf(str );  
nro = Double.parseDouble(str);  
nro = Double.parseDouble(str);
```

## Anotações

---

---

---

---



# 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><u>bloco if</u></b> ou do <b><u>bloco else</u></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

---

---

---

---

33

# 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; }  Os comandos switch também podem ser aninhados.	<ul style="list-style-type: none"><li>↳ 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.</li><li>↳ Se nenhum dos testes produzir um resultado verdadeiro, são executados os comandos do bloco default, se codificados.</li><li>↳ 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.</li></ul> <p><b>Exemplo:</b></p> <pre>switch (valor) {     case 5:     case 7:     case 8:         printf (•••)         break;     case 10:         ... }</pre>
---	---

Anotações

---

---

---

---

34

# 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;
```

# Programação Orientada a Objetos

---

## 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>...
```

**Anotações**

36

# 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>

Anotações

37

# Programação Orientada a Objetos

---

## Instrução while e do while

Comando while	
<pre>while (condição) {     bloco de comandos;     if (condição-2)         break;     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>do {     bloco de comandos;     if (condição-2)         break;     bloco de comandos; } while (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

---

---

---

---

38

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

## Comandos continue e exit

Comando continue	
<b>continue</b> ;	Comando usado dentro dos laços tipo <u><b>for</b></u> , <u><b>while</b></u> ou <u><b>do ... while</b></u> 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.  <u><b>código de retorno</b></u> → 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

---

---

---

---

39

# 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];
```

## 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));  
    }  
}
```

## Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## 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]);
    }
}
```

### Exercícios

1) Que opção declarará, construirá e inicializará um array de maneira válida? (Selecione uma).

- a. int [ ] myList = { "1", "2", "3"};
- b. int [ ] myList = (1,2,3);
- c. int myList[ ][ ] = {5,6,7,8};
- d. int [ ] myList = {1,7,3};
- e. int myList[ ] = {7;8;10;4};

2) Quais opções causam erro de compilação(Selecione duas)

- a. float[] = new float(3);
- b. float f2[] = new float[];
- c. float[] f1 = new float[3];
- d. float f3[] = new float[3];
- e. float f5[] = { 1.0f, 2.0f, 2.0f };
- f. float f4[] = new float[] { 1.0f. 2.0f. 3.0f};

### Anotações

---

---

---

---

# Programação Orientada a Objetos

---

## 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.

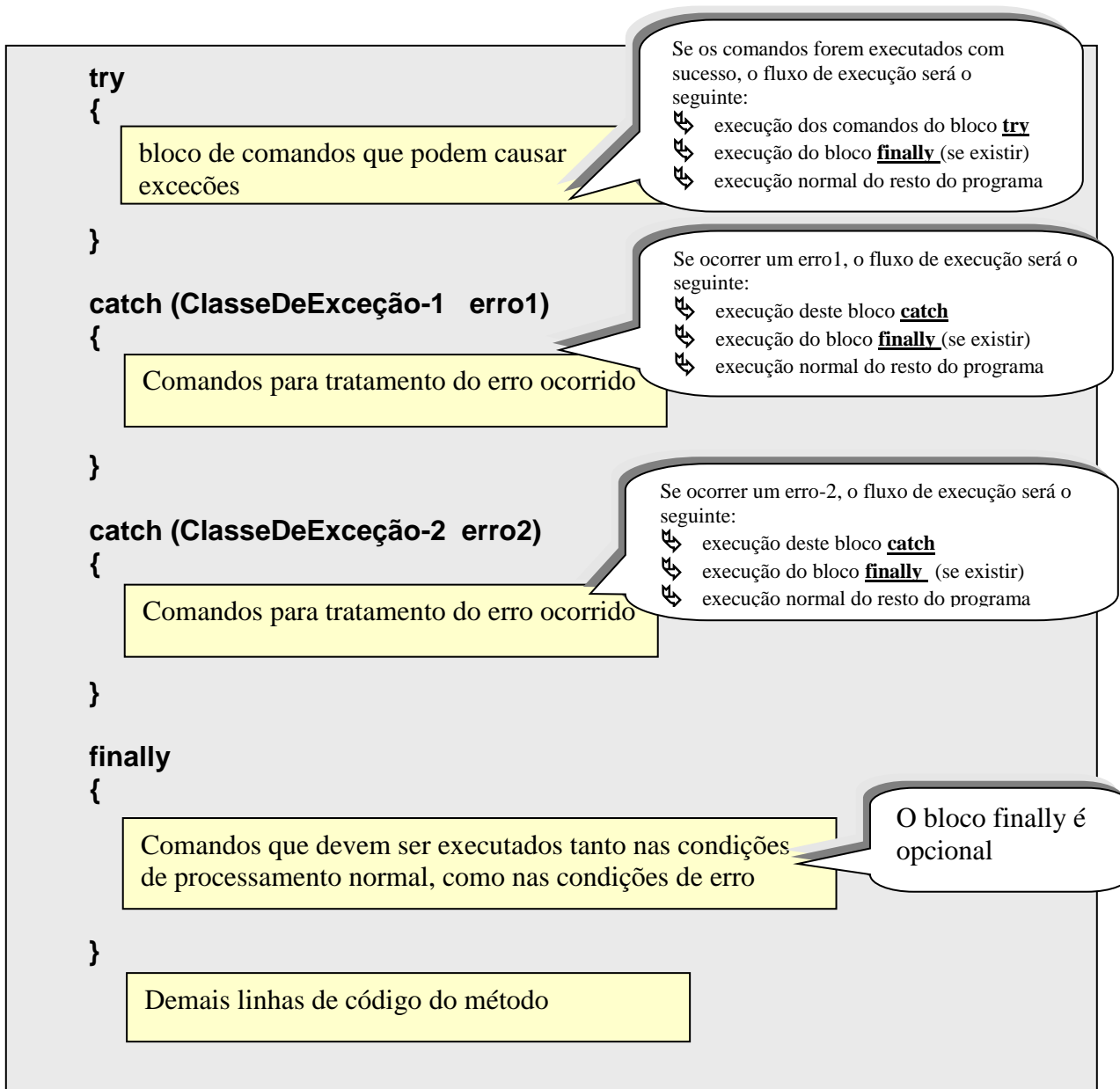
Dentro dos pacotes do Java existem alguns tipos de exceções já previamente definidas por meio de classes específicas. Estas exceções, pré estabelecidas, fazem parte dos pacotes do Java, e todas elas são sub-classes da classe **Throwable**, que pertence ao pacote **java.lang**.

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:

Anotações

43

# 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

44

# 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

45

# 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

---

---

---

---

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

Anotações

---

---

---

---

47

# 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 instanciação 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

---

## Classe Scanner

É uma classe do pacote java.util, com ela podemos receber valores pelo console. Essa classe tem métodos parecidos com os do c++.

```
import java.util.Scanner;
public class Conta{
    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        String nome = "";
        int numero = 0;

        System.out.print("Digite o nome do cliente: ");
        nome = sc.nextLine();

        System.out.print("\nDigite o numero da conta: ");
        numero = sc.nextInt();

        System.out.print("\nNome do Cliente: " + nome +
                        "\nNumero da Conta: " + numero);
    }
}
```

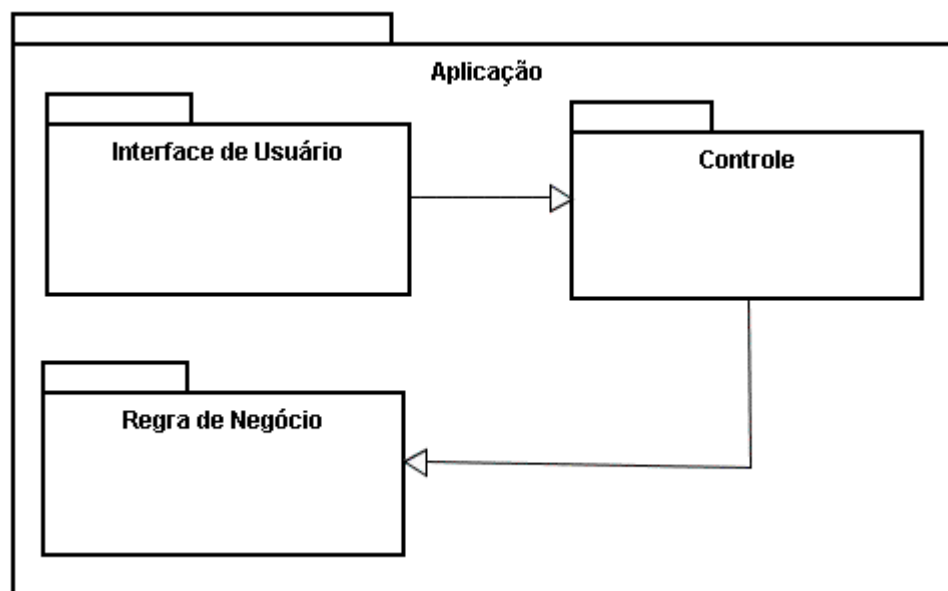
# Programação Orientada a Objetos

---

## Pacotes

A linguagem Java é estruturada em pacotes, estes pacotes contém classes que por sua vez contém 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.

### Anotações

50

# 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

---

---

---

---

# Programação Orientada a Objetos

---

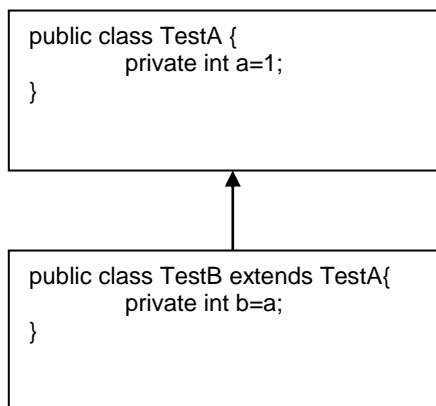
## Acessibilidade

### 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 resolvemos 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).*

*Este 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.

Modificador	Mesma Classe	Mesmo Package	SubClasse	Universo
<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

53

# Programação Orientada a Objetos

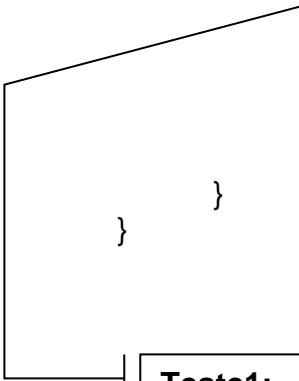
---

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

# 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

55

# Programação Orientada a Objetos

---

## 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.**

Anotações

57

# 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

# Programação Orientada a Objetos

---

## 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.

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.

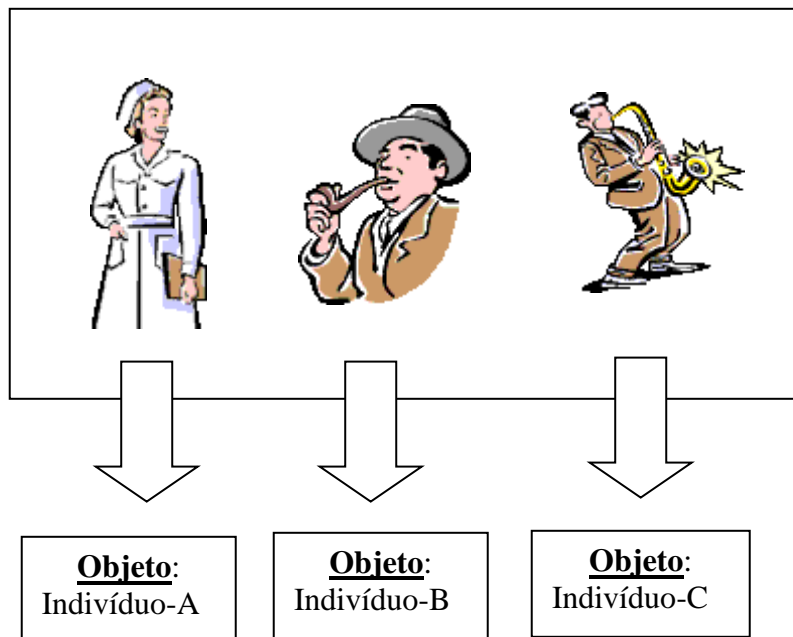
Um objeto é uma entidade real, que possui:

- 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.

Anotações

59

## 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.

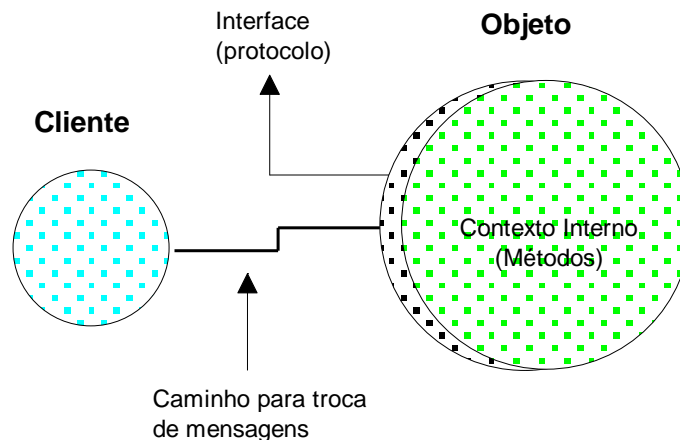
Através do encapsulamento pode ser estabelecida uma interface bem definida para interação do objeto com o mundo externo (interfaces públicas), isolando seus mecanismos de implementação, que ficam confinados ao próprio objeto.

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.

# Programação Orientada a Objetos

---

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

---

## Anotações

---

---

---

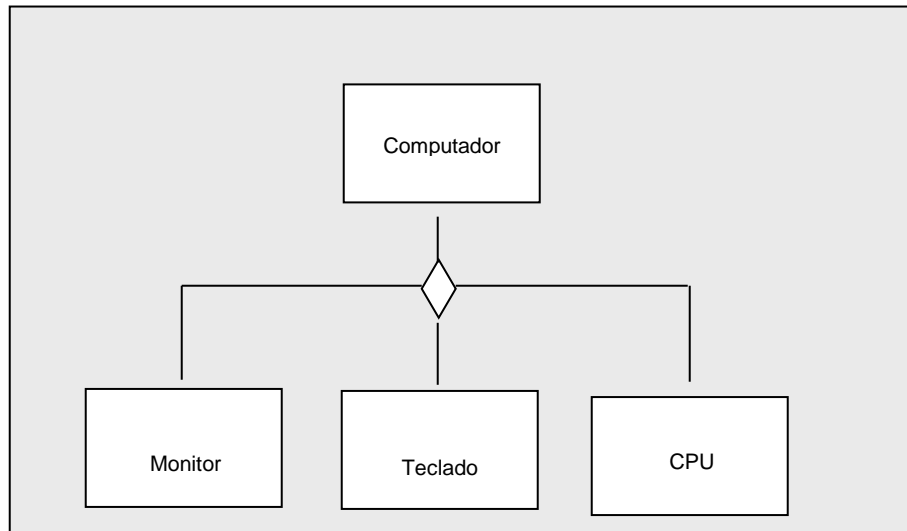
---

# 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.

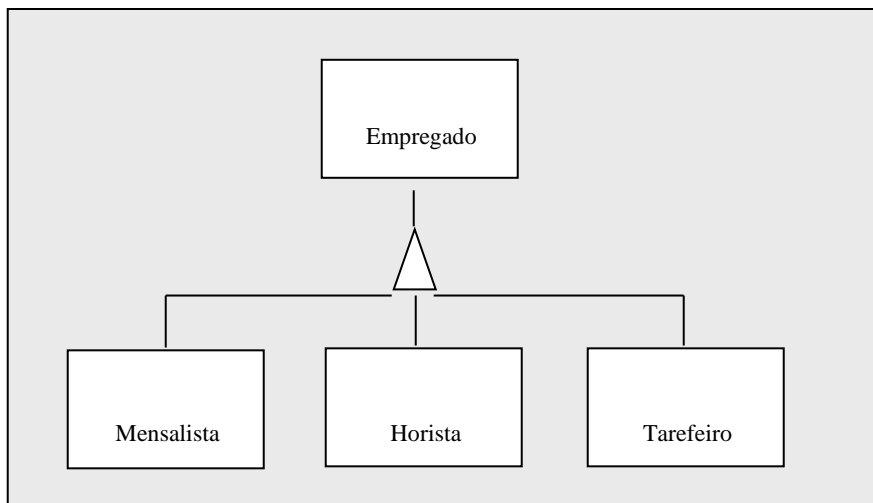
**Anotações**

62

# Programação Orientada a Objetos

---

## 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.

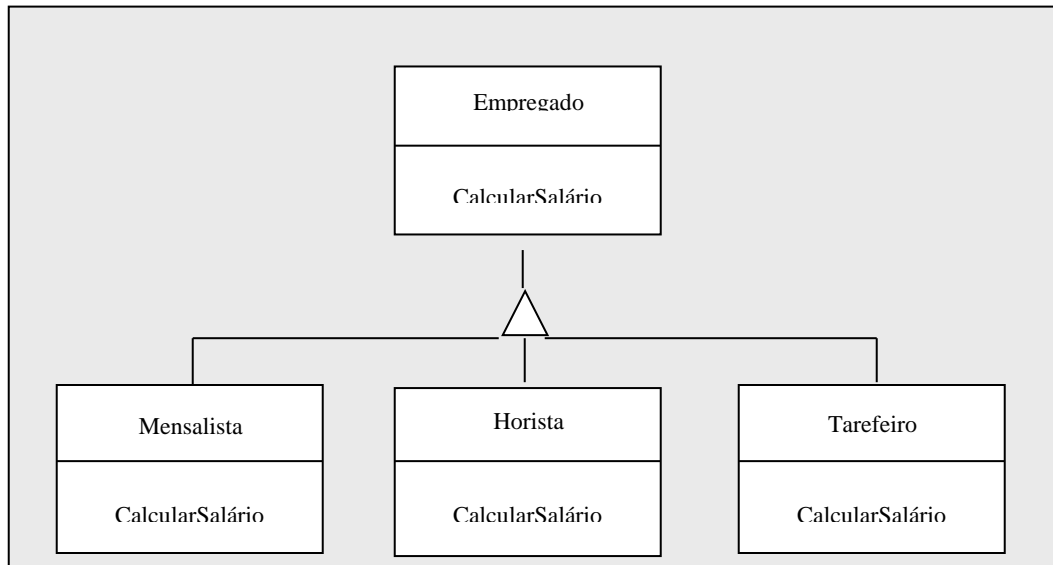
---

**Anotações**

63

# Programação Orientada a Objetos

---



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

**Anotações**

64



# 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>Pacotes</b>	<ul style="list-style-type: none"><li>↳ Conjunto de classes</li></ul>

Anotações

65

# 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

66

# Programação Orientada a Objetos

---

## 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  
}
```

### Anotações

67

# 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

68

# Programação Orientada a Objetos

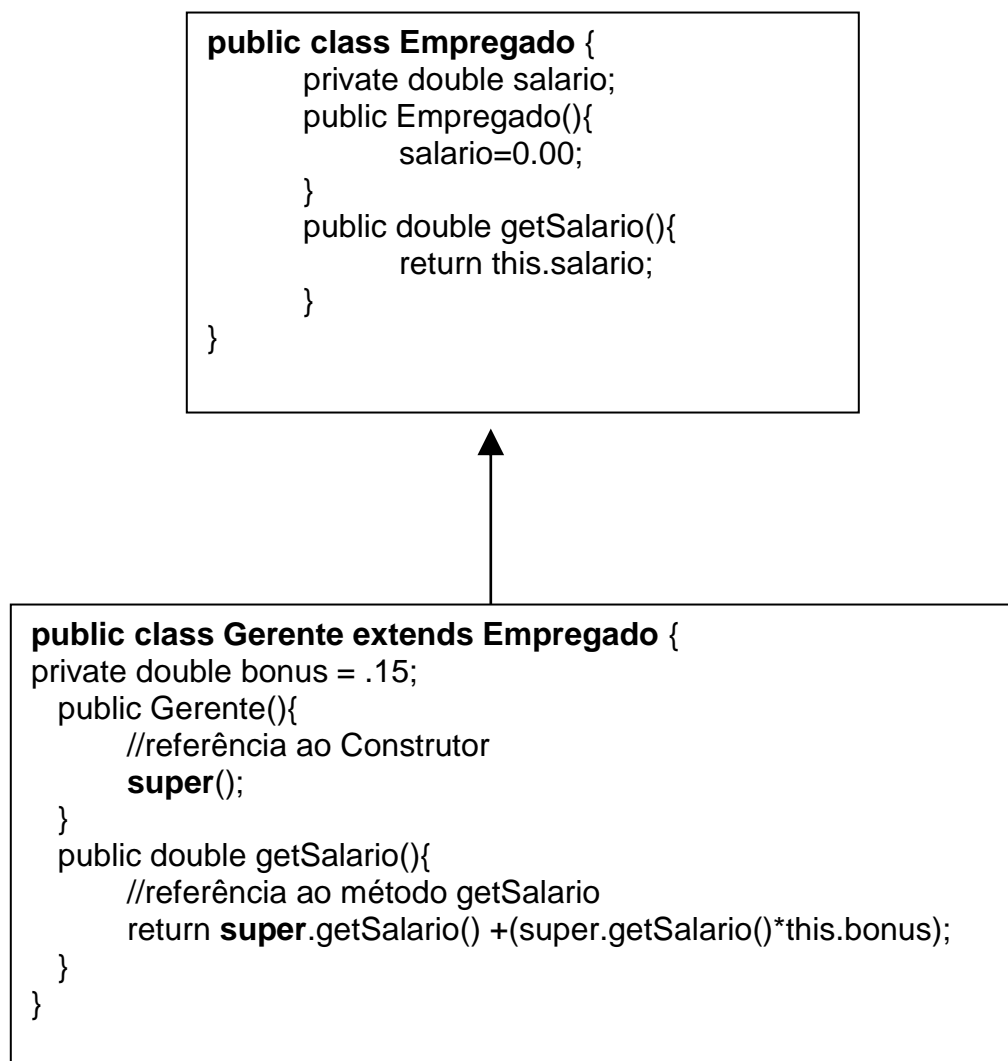
---

## 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



# Programação Orientada a Objetos

---

## 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**

70

# 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.

# Programação Orientada a Objetos

---

## 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;  
    // ...  
}
```

# Programação Orientada a Objetos

---

## Interface Gráfica - Classes do pacote Swing

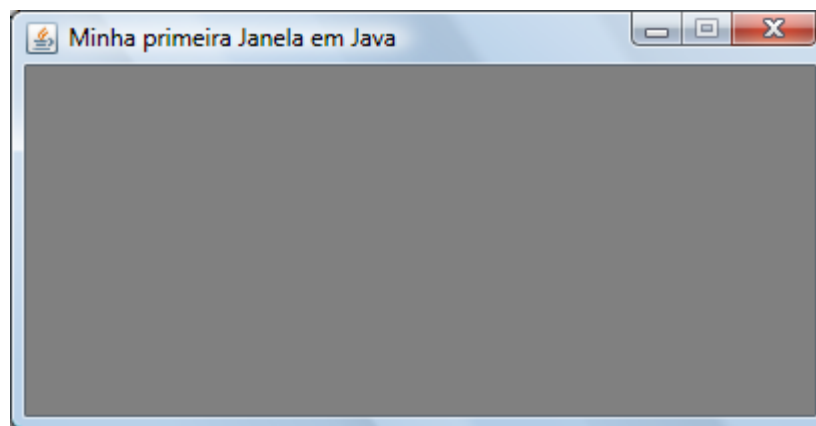
```
package br.com.codemanager;
import java.awt.*; // importa as classes do pacote awt
import java.awt.event.*;
import javax.swing.*;

public class Exemplo01 extends JFrame {
    Exemplo01(){
        setTitle("Minha primeira Janela em Java"); // titulo da Janela
        setSize(400, 200); // dimensoes da janela (Largura e Comprimento)
        setLocation(150, 150); // canto esquerdo e topo da tela
        setResizable(false); // a janela nao pode ser redimensionada
        getContentPane().setBackground(Color.gray); // cor de fundo da janela
    }

    public static void main(String args[]) {
        Exemplo01 janela = new Exemplo01(); // criação do objeto janela
        janela.setVisible(true);

        // libera o controle para o sistema operacional
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

**Resultado:**



**Anotações**

74

# Programação Orientada a Objetos

---

```
package br.com.codemanager;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class Exemplo02 extends JFrame{
    JLabel label1, label2, label3, label4;
    ImageIcon icone = new ImageIcon("C:/temp/java/warn16_1.gif");

    Exemplo02(){
        setTitle("Inserindo Labels e Imagens na janela");
        setSize(350,200);
        setLocation(50,50);
        getContentPane().setBackground(new Color(220,220,220));

        label1 = new JLabel("  Aprendendo",JLabel.LEFT);
        label1.setForeground(Color.red);
        label2 = new JLabel(icone);
        label3 = new JLabel("Inserir  ",JLabel.RIGHT);
        label3.setForeground(Color.blue);
        label4 = new JLabel("Labels e Imagens",icone,JLabel.CENTER);
        label4.setFont(new Font("Serif",Font.BOLD,20));
        label4.setForeground(Color.black);
        getContentPane().setLayout(new GridLayout(4,1));
        getContentPane().add(label1);
        getContentPane().add(label2);
        getContentPane().add(label3);
        getContentPane().add(label4);
    }
}
```

**Resultado:**



**Anotações**

75

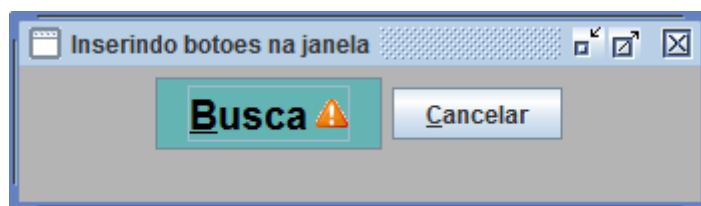
# Programação Orientada a Objetos

---

```
public class Exemplo03 extends JFrame implements ActionListener{
    JButton b1,b2;
    ImageIcon icone = new ImageIcon("C:/temp/java/warn16_1.gif");
    public Exemplo03() {
        setTitle("Inserindo botoes na janela");
        setSize(350,100);
        setLocation(50,50);
        b1 = new JButton("Busca",icone);
        b1.addActionListener(this);
        b2 = new JButton("Cancelar");
        b2.addActionListener(this);
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(b1);
        getContentPane().add(b2);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==b1) {
            JOptionPane.showMessageDialog(null, "Botão 1 pressionado!");
        }
        if (e.getSource()==b2) {
            JOptionPane.showMessageDialog(null, "Botão 2 pressionado!");
        }
    }
}
```

**Resultado:**



**Anotações**

---

---

---

---

76

# Programação Orientada a Objetos

---

## JDBC

A biblioteca da JDBC provê um conjunto de interfaces de acesso ao BD. Uma implementação em particular dessas interfaces é chamada de driver. Os próprios fabricantes dos bancos de dados (ou terceiros) são quem implementam os drivers JDBC para cada BD, pois são eles que conhecem detalhes dos BDs.

Cada BD possui um Driver JDBC específico (que é usado de forma padrão - JDBC). A API padrão do Java já vem com o driver JDBC-ODBC, que é uma ponte entre a aplicação Java e o banco através da configuração de um recurso ODBC na máquina. Os drivers de outros fornecedores devem ser adicionados ao CLASSPATH da aplicação para poderem ser usados.

Desta maneira, pode-se mudar o driver e a aplicação não muda.

As principais classes e interfaces do pacote **java.sql** são:

**DriverManager** - gerencia o driver e cria uma conexão com o banco.

**Connection** - é a classe que representa a conexão com o banco de dados.

**Statement** - controla e executa uma instrução SQL.

**PreparedStatement** - controla e executa uma instrução SQL. É melhor que Statement.

**ResultSet** - contém o conjunto de dados retornado por uma consulta SQL.

A interface Connection possui os métodos para criar um Statement, fazer o commit ou rollback de uma transação, verificar se o auto commit está ligado e poder (des)ligá-lo, etc.

As interfaces Statement e PreparedStatement possuem métodos para executar comandos SQL.

ResultSet possui método para recuperar os dados resultantes de uma consulta, além de retornar os metadados da consulta.