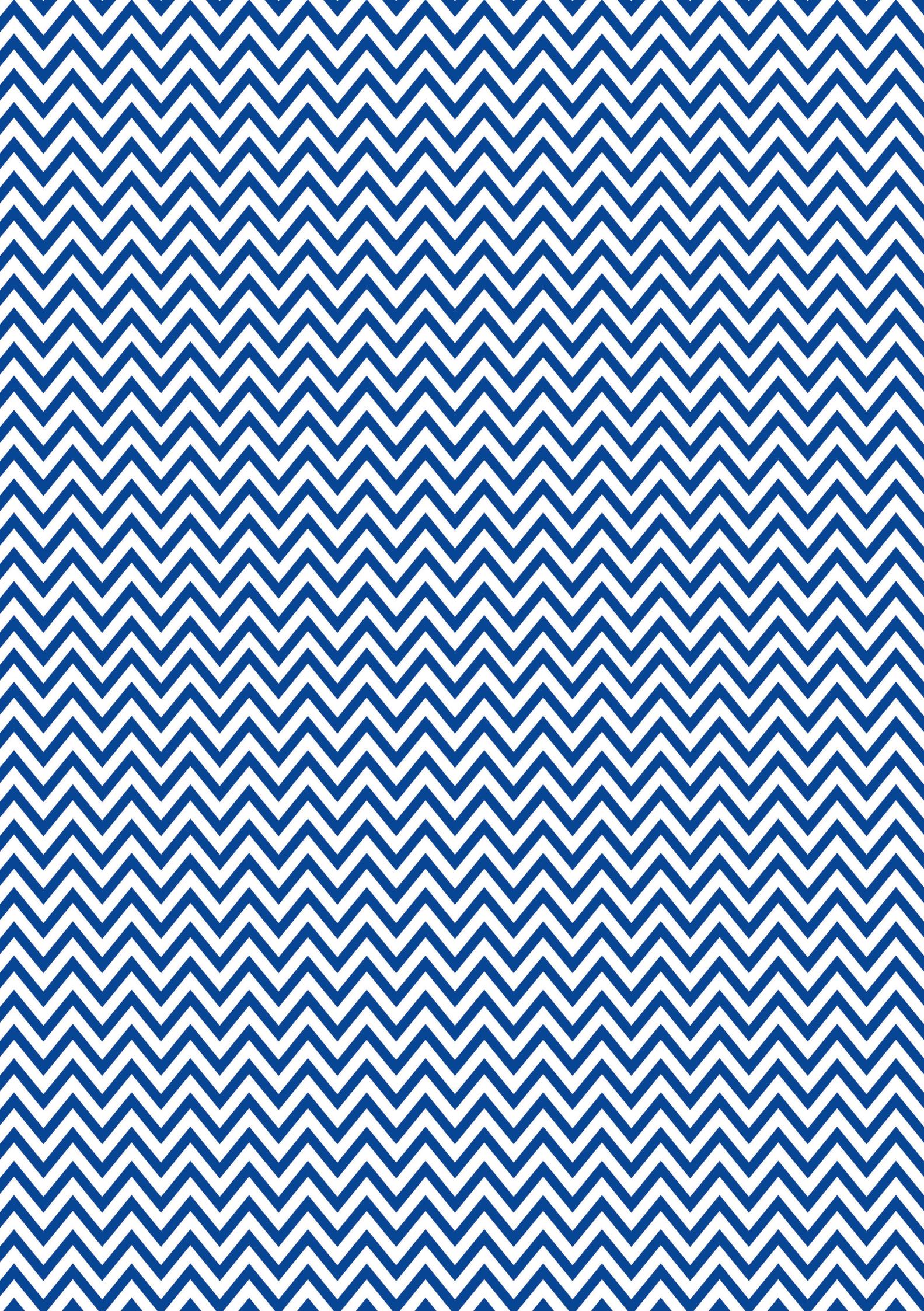




# **INTRODUÇÃO À LINGUAGEM JAVA**







# **INTRODUÇÃO À LINGUAGEM JAVA**

São Paulo  
maio/2017

---

Nome do aluno





## **ADMINISTRAÇÃO REGIONAL DO SENAC NO ESTADO DE SÃO PAULO**

### **Gerência de Desenvolvimento**

Claudio Luiz de Souza Silva

### **Coordenação Técnica**

Ozeas Vieira Santana Filho

### **Apoio Técnico**

Rodrigo Moura Galhardo

### **Elaboração do Recurso Didático**

Fábio Luís Ramon

### **Atualização do Recurso Didático**

Kátia Canto

### **Editoração Eletrônica**

Marcio S. Barreto

### **Coordenação de Preparação e Revisão de Texto**

Luiza Elena Luchini

### **Preparação/Revisão do Texto**

Johannes Bergmann

**© Senac São Paulo, 2017**

Proibida a reprodução sem autorização expressa.  
Todos os direitos reservados ao Senac São Paulo.



# sumário

## **INTRODUÇÃO / 7**

- O que é Java / 7
- Java Machine / 8
- Java e suas edições / 8
- Documentação / 9

## **1. AMBIENTE DE DESENVOLVIMENTO / 11**

- JDK / 11
- Eclipse / 12
- Variáveis de ambiente do Windows / 13

## **2. PRIMEIRO PROGRAMA JAVA / 15**

- Execução do Eclipse / 17
- Projetos / 18

## **3. ESTRUTURA DE UM PROGRAMA JAVA / 21**

- Tipos e classes básicas / 22
- Classes especiais / 22
- Variáveis / 23
- Conversão de tipos / 24
- Operadores e expressões / 24
- Instruções básicas / 26
- Estruturas de controle / 26
- Parâmetros de entrada / 28

# **INTRODUÇÃO À LINGUAGEM JAVA**

## **4. CLASSES E OBJETOS / 31**

- Pacotes / 31
- Programação orientada a objetos / 31
- Conceito de classe / 33
- Atributos, métodos e interfaces / 34
- Métodos especiais / 35
- Definição de uma classe e alocação / 35
- Notação UML / 37

## **5. ARRAY E STRING / 43**

- Arrays / 43
- Alocação de arrays / 44
- Arrays multidimensionais / 45
- String / 45

## **6. HERANÇA E INTERFACES / 47**

- Herança / 47
- Interfaces / 52

## **7. ENCAPSULAMENTO / 57**

## **8. POLIMORFISMO / 61**

- Polimorfismo e herança / 62

## **9. ENTRADA E SAÍDA / 67**

- Entrada – Inputstream / 68
- Saída – Outputstreams / 70

## **10. THREADS / 73**

- Diagrama / 74
- Sincronização de threads / 75

## **11. EXCEÇÕES E ERROS / 77**

- Maneira comum de tratar exceções/erros / 78
- Tratamento em Java / 78

## **12. TÓPICOS AVANÇADOS / 83**

- Coleções / 83
- Networking / 85
- Serversocket e Socket / 85
- Generics / 86

## **REFERÊNCIAS / 89**



# Introdução

O objetivo do curso Introdução à linguagem Java é desenvolver conhecimento básico da linguagem Java, sua estrutura e sintaxe, editando, compilando, depurando e executando aplicações simples orientadas a objetos.

Para isso, será usado o IDE Eclipse, que é uma ferramenta de uso livre e um ambiente completo para desenvolvimento na plataforma Java.

## O QUE É JAVA

Java é uma poderosa plataforma de desenvolvimento que usa a linguagem de mesmo nome para a programação de aplicações. O principal enfoque da plataforma são as aplicações web, mas aplicações de outra natureza também podem utilizar Java.

Originalmente criada pela Sun Microsystems em 1995, a linguagem Java é hoje de propriedade da Oracle e continua sendo aprimorada e estendida para atender o mercado com eficiência. A plataforma é amplamente usada, atingindo um grande número de usuários, sejam desenvolvedores ou consumidores.

*“Java é a base de praticamente todos os tipos de aplicativos em rede e é o padrão global para desenvolvimento e fornecimento de aplicativos incorporados, jogos, conteúdo on-line e software corporativo. Com mais de 9 milhões de desenvolvedores em todo o mundo, o Java permite desenvolver e implementar aplicativos e serviços incríveis com eficiência.”*

Fonte: Oracle <<https://www.oracle.com/br/java/technologies/index.html>>.

A linguagem Java foi criada sob o paradigma da orientação a objetos, incluindo os conceitos de classes, métodos, atributos, encapsulamento, herança, polimorfismo, entre outros.

Java trabalha com uma máquina virtual própria para interpretar e executar aplicações, o que a torna multiplataforma, compatível e portável.

### JAVA MACHINE

A JVM, ou Java Virtual Machine, é a denominação dada ao interpretador Java rodando sob o sistema operacional (SO) de qualquer máquina. Esta combinação (interpretador+SO) tem um comportamento idêntico para qualquer SO suportado por Java. Dessa forma, caracteriza-se como uma máquina virtual que executa seu código binário Java, chamado também de bytecode.

O código fonte é compilado para código binário Java, que é interpretado e executado em diferentes plataformas.



### JAVA E SUAS EDIÇÕES

#### Java SE

A plataforma Java Standard Edition (Java SE) é o conjunto de recursos para desenvolver e implementar programas em desktops, servidores ou ambientes incorporados. Inclui o kit de desenvolvimento (Java Development Kit – JDK) e outros recursos importantes, como o ambiente de execução Java (Java Runtime Environment – JRE), que já inclui as bibliotecas, a JVM e outras ferramentas.

“A Plataforma Java, Standard Edition (Java SE), permite que você desenvolva aplicativos seguros, portáteis e de alto desempenho para a maior variedade possível de plataformas de computação. Disponibilizando aplicativos em ambientes heterogêneos, as empresas podem agilizar a produtividade do usuário final, a comunicação e a colaboração — além de reduzir drasticamente o custo de propriedade de aplicativos tanto de empresas quanto de clientes.”

Fonte: <<https://www.oracle.com/br/java/technologies/index.html>>.

#### Java EE

É a edição com todo suporte para rodar aplicações em servidores web e servidores de aplicação. Oferece um conjunto completo de bibliotecas que implementam as rigorosas necessidades de um ambiente de execução seguro e crítico.

“O Java Platform, Enterprise Edition (Java EE), é o padrão do setor para computação Java empresarial. Com novos recursos que melhoram o suporte a HTML5, aumentam a produtividade do desenvolvedor e aprimoram ainda mais a forma de atender às demandas corporativas, o Java EE 7 permite que os desenvolvedores escrevam menos códigos, tenham suporte melhor para os mais recentes aplicativos da Web e estruturas e tenham acesso à mais capacidade de expansão e funcionalidades mais avançadas.”

Fonte: <<https://www.oracle.com/br/java/technologies/index.html>>.

## **DOCUMENTAÇÃO**

O principal guia de referência para o desenvolvedor Java encontra-se no seguinte endereço:

<<https://docs.oracle.com/javase/8/docs/>>

A documentação Java, da Oracle, é a referência oficial para a atual versão da linguagem, onde encontram-se todos os recursos recomendados, além de informações sobre a obsolescência de recursos de versões anteriores.





# 1. Ambiente de desenvolvimento

Para desenvolver aplicações com a linguagem Java, é necessário montar um ambiente de desenvolvimento com ferramentas e configurações específicas. Isso inclui as ferramentas da Oracle com a estrutura da plataforma e uma ferramenta de edição que reconheça e execute Java.

## JDK

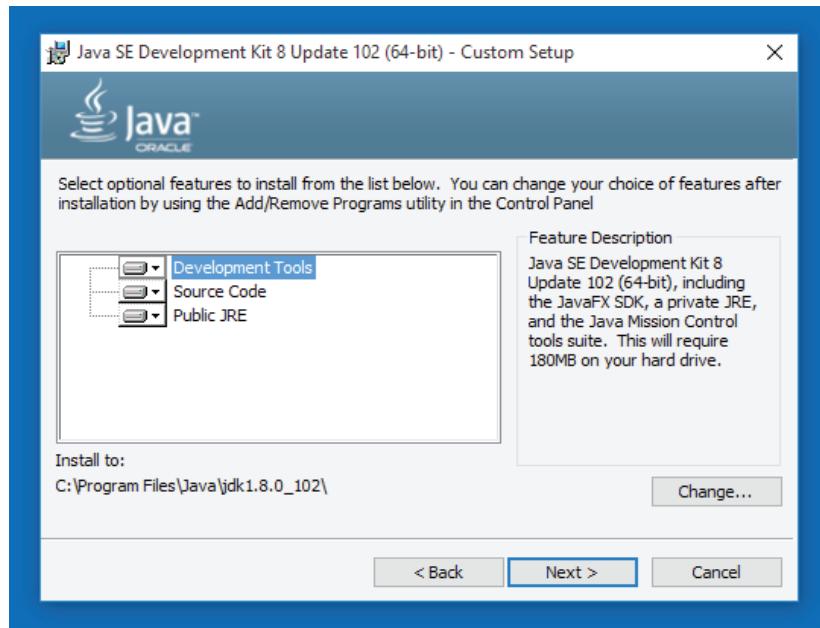
A Oracle disponibiliza um kit de desenvolvimento o JDK (Java Development Kit), que consiste nas bibliotecas Java, a máquina virtual JVM e outros recursos de desenvolvimento.

Baixe o arquivo de instalação para o seu Sistema Operacional em:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html?ssSourceSiteId=otnpt>

Execute o arquivo do JDK como administrador da máquina, instalando todos os recursos disponíveis:

## INTRODUÇÃO À LINGUAGEM JAVA



O processo de instalação do JDK também vai instalar ou atualizar o plugin Java para o consumidor final, necessário para executar os programas desenvolvidos com a plataforma Java.

## ECLIPSE

Também é necessário usar uma ferramenta para editar os códigos e que entenda a linguagem Java e seus requisitos de programação, reconhecendo as bibliotecas, organizando a estrutura de arquivos, informando sobre erros e permitindo configurações diversas. Para a linguagem Java, uma das melhores opções para esse ambiente de desenvolvimento é o Eclipse, um IDE (Integrated Development Environment – ambiente de desenvolvimento integrado) completo.

O Eclipse é uma ferramenta de desenvolvimento de aplicações Java que pode ser utilizada livremente, sem pagamento de licenças. A grande vantagem do Eclipse é a sua arquitetura de plugins, que permite que seus recursos sejam bastante expandidos.

Para baixar o Eclipse, vá até o endereço [eclipse.org](http://eclipse.org), e procure a versão mais recente. Este IDE é constantemente atualizado e a migração para novas versões é recomendada e não gera maiores problemas de compatibilidade.

Execute o instalador do Eclipse e, na lista de ferramentas que podem ser instaladas, escolha a opção básica:



### Eclipse IDE for Java Developers

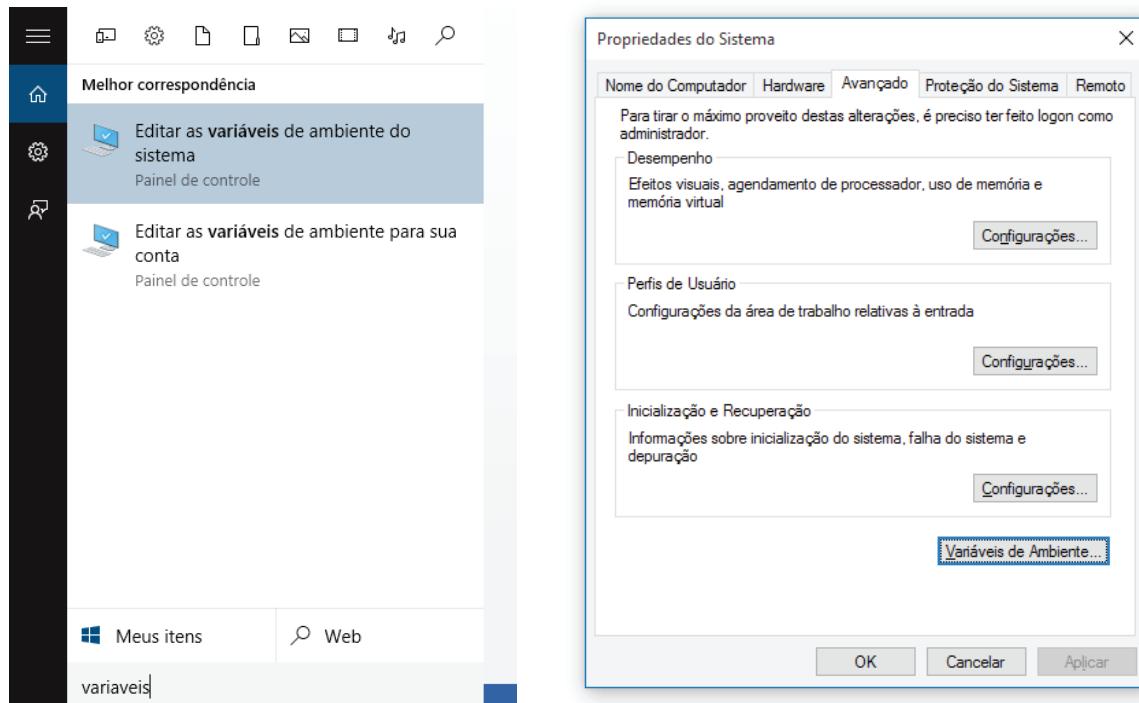
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven and Gradle integration.

Aguarde o processo de instalação finalizar para que o Eclipse esteja pronto para uso.

## VARIÁVEIS DE AMBIENTE DO WINDOWS

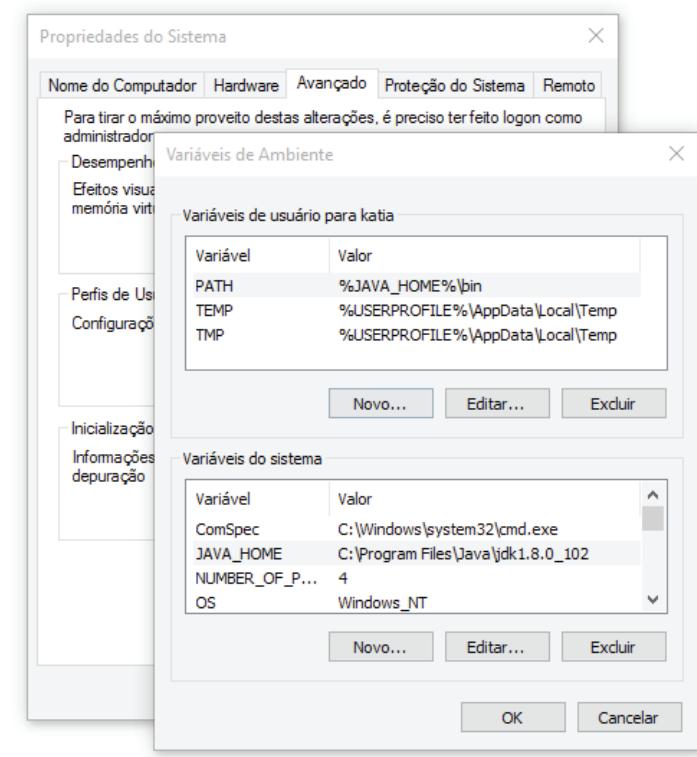
A linguagem Java é um ecossistema bastante grande e complexo. Para que os projetos em Java sejam desenvolvidos com consistência, não é o suficiente ter apenas o IDE. Por exemplo, pode ser necessário usar linhas de comando para compilar um arquivo Java. Para isso, é preciso ajustar algumas variáveis de ambiente do Sistema Operacional, que indicarão o caminho do compilador, entre outras configurações. Abaixo, as indicações dessa configuração no SO Windows.

Pesquise no Windows pelo termo “variáveis”. Deverá aparecer algo como “Editar as variáveis do ambiente do sistema”; tecle *Enter* para ver a segunda janela como nas figuras abaixo:



Clique em “Variáveis de Ambiente”, para criar ou editar as variáveis. Crie ou edite a variável de sistema JAVA\_HOME, com o valor contendo o caminho da instalação do JDK. Crie também, caso não exista, a variável de usuário PATH, que deve conter no valor (sem as aspas): “%JAVA\_HOME%\bin”. Se esta variável já existir, edite o valor inserindo o sinal de ponto e vírgula ao final dos valores e completando com o novo valor.

## INTRODUÇÃO À LINGUAGEM JAVA



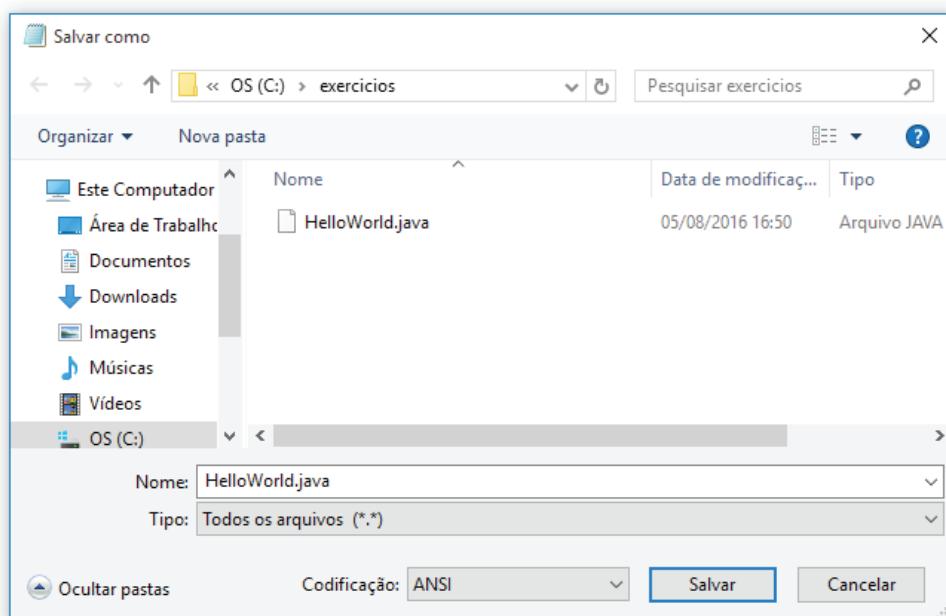
Pronto, o ambiente de trabalho já está configurado.



## 2. Primeiro programa Java

Antes de usar o IDE Eclipse, é importante experimentar o desenvolvimento de uma classe Java, sua compilação e sua execução de modo mais “puro”. Esta experiência facilitará a compreensão sobre como um programa Java funciona.

Abra o Bloco de Notas do seu sistema operacional e salve um arquivo com o nome de HelloWorld.java como abaixo:



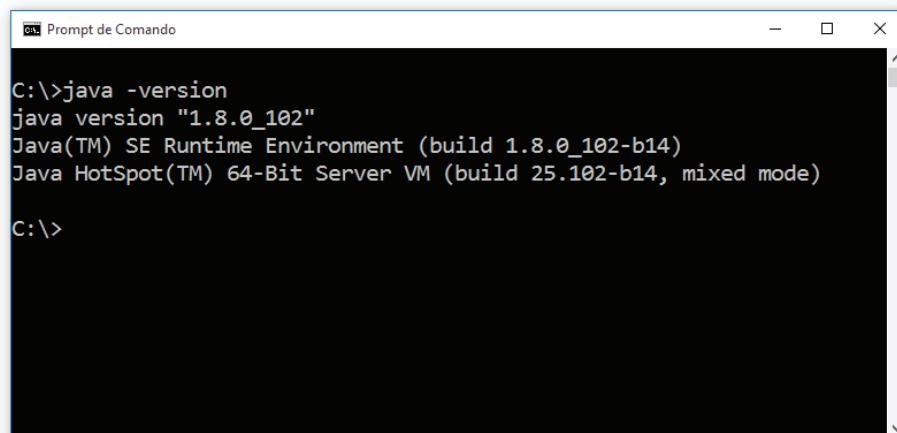
## INTRODUÇÃO À LINGUAGEM JAVA

Nesse arquivo, digite os códigos a seguir exatamente como está apresentado aqui, incluindo a tabulação, para criar a primeira aplicação Java:

```
//pacote  
//importar pacotes para a aplicação  
//definição da classe principal  
public class HelloWorld {  
    //definição do método principal  
    public static void main(String args[ ]) {  
        System.out.println("Hello World!");  
    }  
}
```

Abra o prompt de comando do Windows. Nele, vamos verificar inicialmente se o Java foi instalado e qual a sua versão. Digite e confirme:

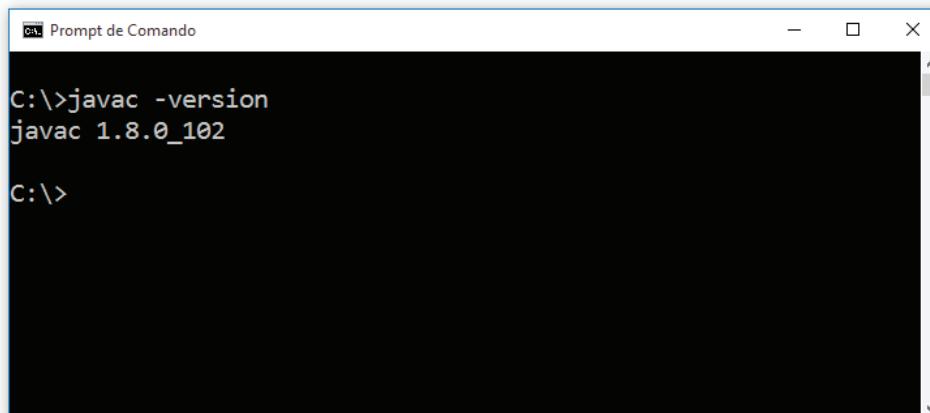
```
java -version
```



The screenshot shows a Windows Command Prompt window titled "Prompt de Comando". The command "java -version" is entered, and the output is:  
C:\>java -version  
java version "1.8.0\_102"  
Java(TM) SE Runtime Environment (build 1.8.0\_102-b14)  
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)  
C:\>

Teste também a versão do compilador javac, digitando e confirmando:

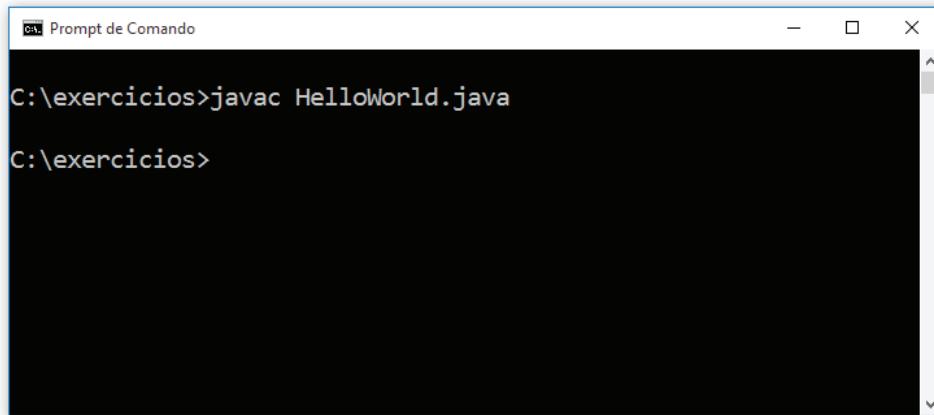
```
javac -version
```



The screenshot shows a Windows Command Prompt window titled "Prompt de Comando". The command "javac -version" is entered, and the output is:  
C:\>javac -version  
javac 1.8.0\_102  
C:\>

A seguir, para compilar o arquivo HelloWorld.java, navegue até a pasta onde você salvou o arquivo, digite e confirme:

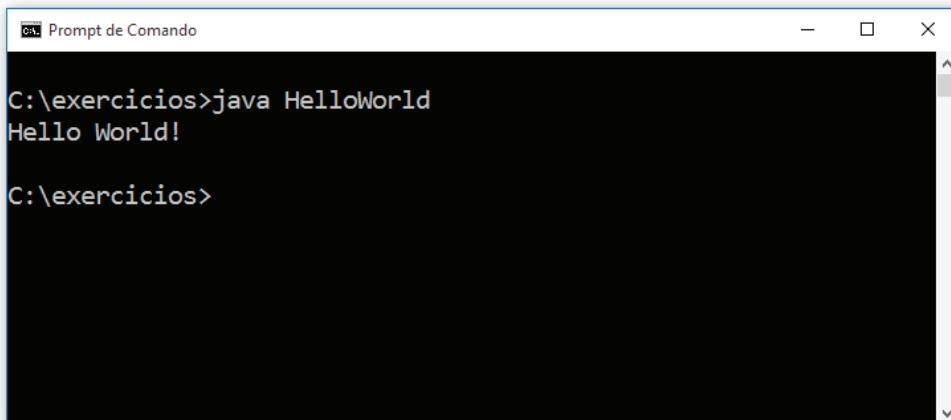
```
javac HelloWorld.java
```



```
C:\exercicios>javac HelloWorld.java
C:\exercicios>
```

E por fim, para interpretar o arquivo e ver seu conteúdo, digite e confirme:

```
java HelloWorld
```



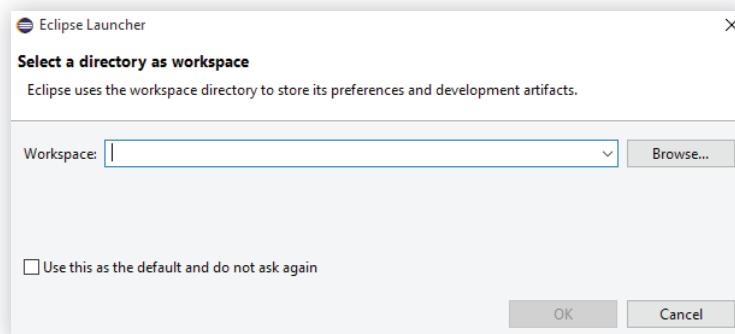
```
C:\exercicios>java HelloWorld
Hello World!
C:\exercicios>
```

## EXECUÇÃO DO ECLIPSE

Vamos repetir a experiência da primeira classe usando, agora, o IDE Eclipse, aproveitando para conhecer a interface do programa, seu uso e seus recursos.

Na pasta onde o Eclipse foi instalado, clique no arquivo que executa o programa.

Sempre que o Eclipse for aberto, ele pedirá que seja indicado um diretório de trabalho (workspace), onde ficarão os arquivos editados nele. Uma workspace pode conter vários projetos java.

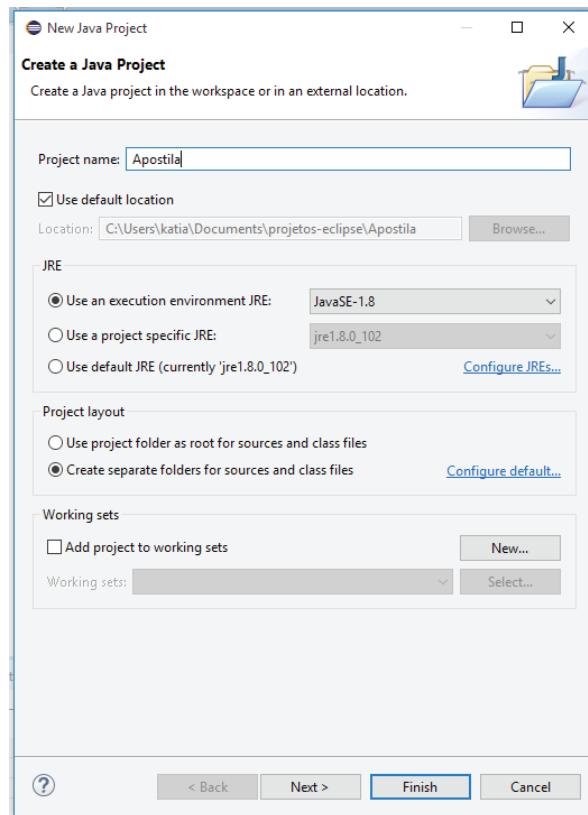


## INTRODUÇÃO À LINGUAGEM JAVA

### PROJETOS

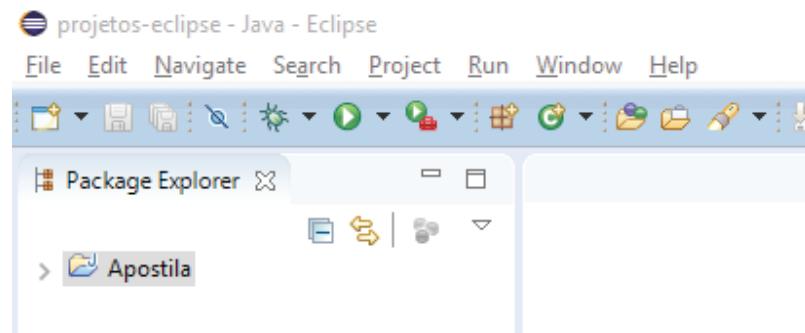
Cada programa Java pode ser composto por várias classes. Essas classes e as suas configurações ficam agrupadas em um projeto gerenciado pelo Eclipse.

Vá em File > New > Java Project e acrescente o nome do projeto:



Ao confirmar a criação do projeto, já se pode ver a área de trabalho do Eclipse. A interface da ferramenta é organizada em áreas distintas que podem ser fechadas e reabertas a qualquer momento (Window > Show View). A área de trabalho do Eclipse pode ser personalizada para atender melhor ao desenvolvedor e suas preferências.

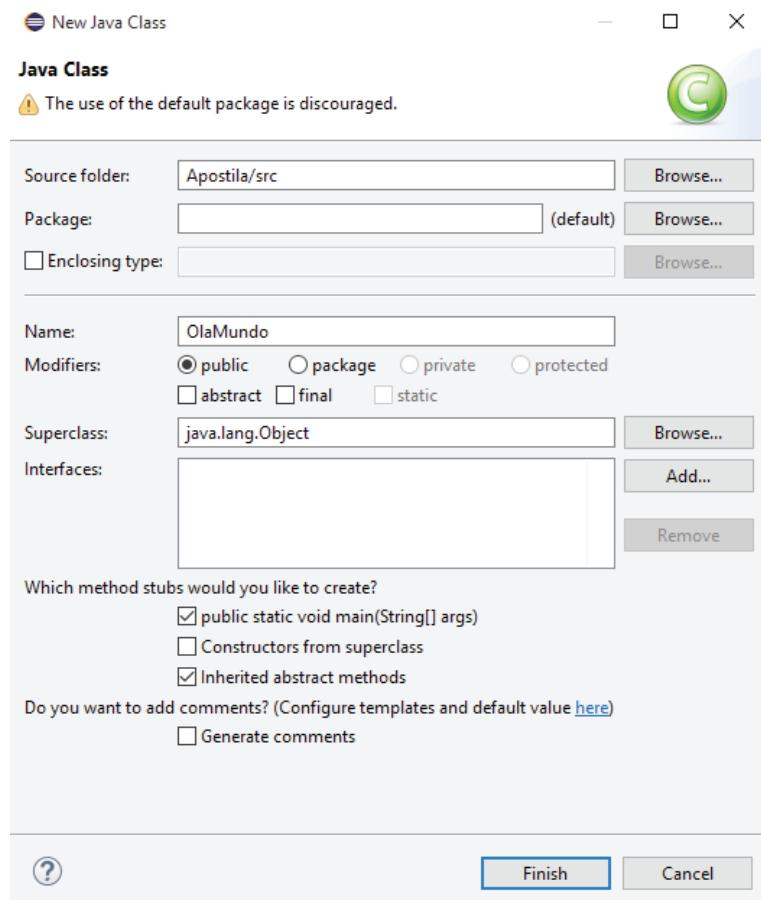
Uma das principais áreas na interface do Eclipse é a aquela onde os arquivos são editados, no centro da janela. Outra área muito importante é o Package Explorer, que mostra a estrutura do projeto Java:



As classes são um dos conceitos de orientação a objetos, que serão vistos em detalhes mais à frente. Por enquanto, pode-se criar a classe para o primeiro programa Java feito no Eclipse.

Para criar uma nova classe e adicionar ao projeto em File > New > Class.

Na janela, defina o nome da classe e marque para criar o método principal, usado para testar e dar as saídas do programa: public static void main(String[ ] args)



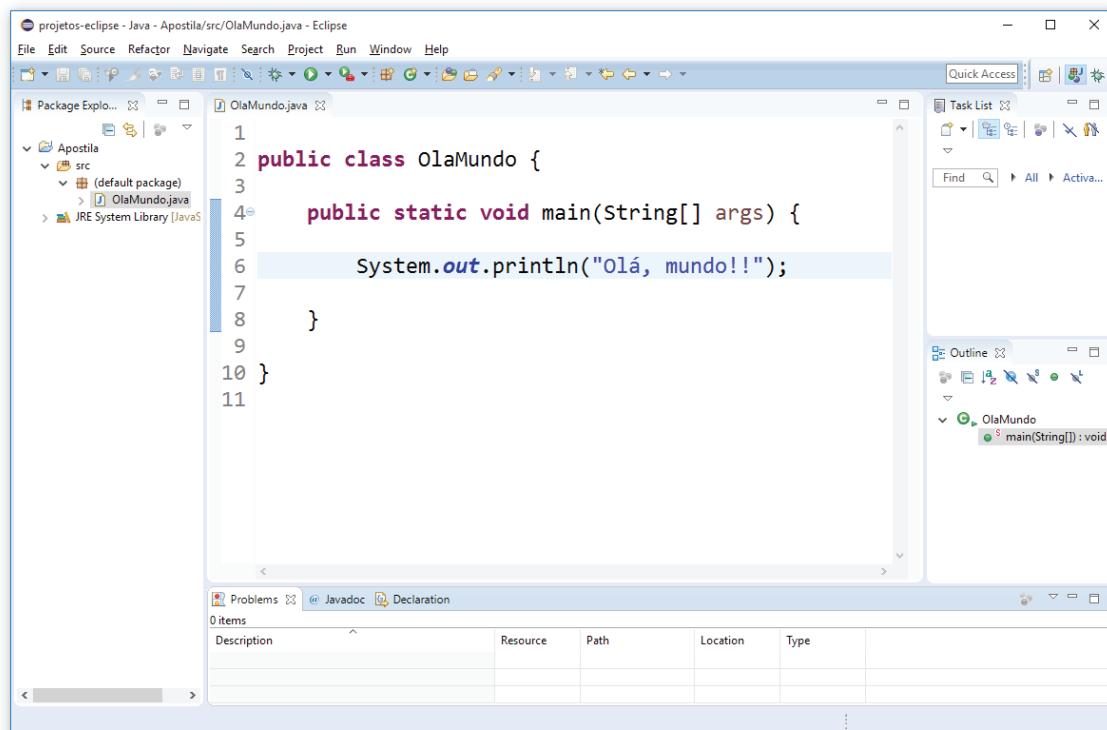
Repare que o nome da classe deve começar com letra maiúscula e, se for um nome composto, a segunda palavra também deve começar com maiúscula. Este é o padrão para nomes de classes em programação Java.

A classe foi criada dentro do projeto ativo e está aberta para edição na área central do Eclipse.

Para editar o arquivo java (classe) e executá-lo em seguida, adicione a seguinte linha de código dentro do método main:

```
System.out.println("Olá, mundo!");
```

## INTRODUÇÃO À LINGUAGEM JAVA

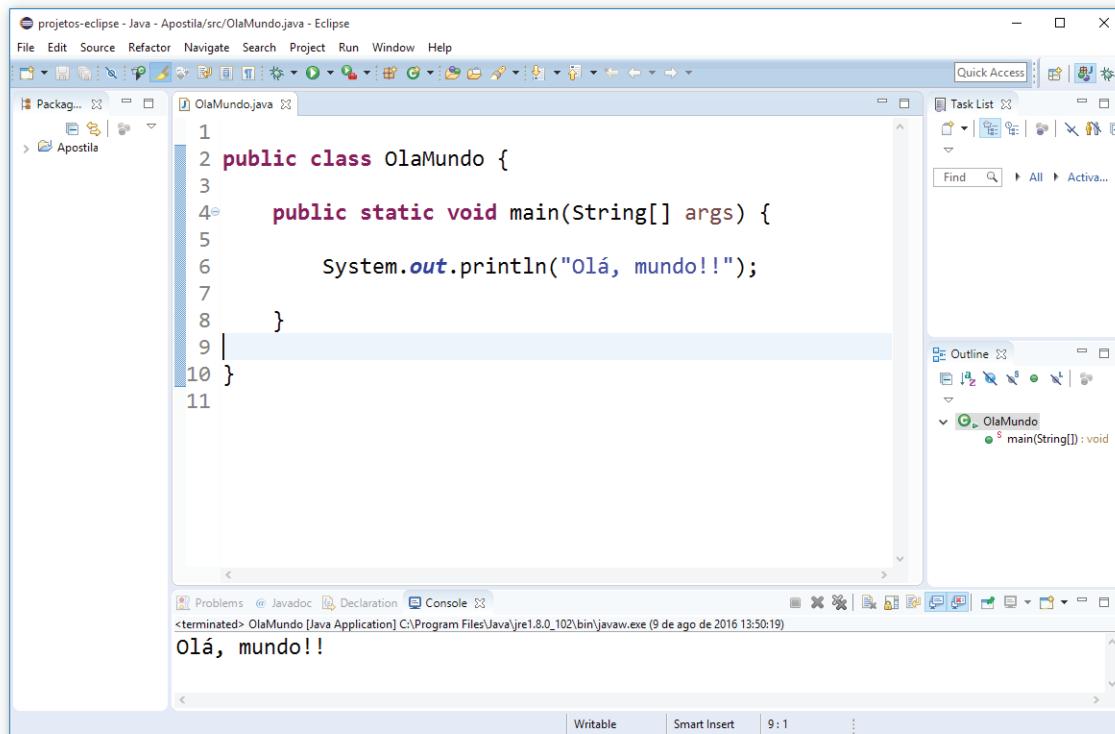


The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** projetos-eclipse - Java - Apostila/src/OlaMundo.java - Eclipse
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Standard Eclipse toolbar with various icons for file operations.
- Package Explorer:** Shows a project named "Apostila" with a "src" folder containing "OlaMundo.java".
- Outline View:** Shows the class "OlaMundo" and its method "main(String[] args)".
- Code Editor:** Displays the Java code for "OlaMundo.java":

```
1 public class OlaMundo {  
2     public static void main(String[] args) {  
3         System.out.println("Olá, mundo!!");  
4     }  
5 }  
6  
7 }  
8 }  
9 }  
10 }  
11 }
```
- Problems View:** Shows 0 items.
- Console View:** Not visible in this screenshot.

Para executar (“rodar”) o programa, salve-o e clique no círculo verde com uma seta na barra de ferramentas do Eclipse ou tecle **Ctrl+F11**. A execução do programa deve mostrar o console logo abaixo da área de códigos, onde será exibido o texto que foi escrito na aplicação.



The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** projetos-eclipse - Java - Apostila/src/OlaMundo.java - Eclipse
- Menu Bar:** File Edit Source Refactor Navigate Search Project Run Window Help
- Toolbar:** Standard Eclipse toolbar with various icons for file operations.
- Package Explorer:** Shows a project named "Apostila" with a "src" folder containing "OlaMundo.java".
- Outline View:** Shows the class "OlaMundo" and its method "main(String[] args)".
- Code Editor:** Displays the Java code for "OlaMundo.java":

```
1 public class OlaMundo {  
2     public static void main(String[] args) {  
3         System.out.println("Olá, mundo!!");  
4     }  
5 }  
6  
7 }  
8 }  
9 }  
10 }  
11 }
```
- Problems View:** Shows 0 items.
- Console View:** Shows the output of the program execution:

```
<terminated> OlaMundo [Java Application] C:\Program Files\Java\jre1.8.0_102\bin\javaw.exe (9 de ago de 2016 13:50:19)  
Olá, mundo!!
```



### 3. Estrutura de um programa Java

Um programa Java é uma coleção de classes que interagem entre si de forma totalmente orientada a objetos. Por padrão, os nomes das classes devem começar com letra maiúscula, não pode incluir espaços em branco e se forem nomes compostos, a segunda palavra também começará com letra maiúscula. A linguagem Java diferencia letras maiúsculas e minúsculas. Há outras regras para a sintaxe Java que serão vistas ao longo da apostila.

Nas classes, encontram-se atributos e métodos, sendo que deve haver uma classe principal, usada para testar a aplicação e seus recursos. Nessa classe principal, há necessariamente o método de partida de um programa Java, chamado de main. Esta classe é a que dá nome ao programa.

Exemplo:

```
//classe principal chamada HelloWorld:  
public class HelloWorld {  
  
    //método principal:  
    public static void main(String args[ ]) {  
  
        /*declaração do método com a instrução  
           para imprimir um texto na janela de comando: */  
        System.out.println("Hello World");  
    }  
}
```

Observe as linhas que começam com barras // e com /\*. Essas linhas são reservadas para os comentários do desenvolvedor, sendo que o texto depois das barras são comentários de linha única e o texto entre /\* e \*/ são comentários de bloco que podem conter várias linhas.

## INTRODUÇÃO À LINGUAGEM JAVA

### TIPOS E CLASSES BÁSICAS

Tipos primitivos: não são objetos e independem da máquina em termos de precisão e tamanho.

Inteiro:

- Byte 8 bits
- Short 16 bits
- Int 32 bits
- Long 64 bits

Ponto-flutuante:

- Float 32 bits
- Double 64 bits
- Caracter (Unicode)
- Char 16 bits

Boleano:

- Boolean, true ou false

Faixa de valores dos tipos primitivos:

- Byte: -128 a 127
- Short: -32.768 a 32.767
- Int: -2.147.483.648 a 2.147.483.647
- Long: -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
- Float: -3,4E-38 a 3,4E\_38
- Double: -1.7E-308 a 1.7E+308

### CLASSES ESPECIAIS

Algumas classes em Java assumem um papel especial ou apresentam um comportamento especial. Entre elas, é interessante citar duas:

String

- Implementa um objeto String, de forma mais precisa e clara do que um array de caracteres.
- Não são arrays de caracteres.
- Dispõe de métodos que facilitam seu uso.

Um objeto da classe String é criado automaticamente quando se declara e inicializa uma variável. Exemplo:

```
String curso="curso de Java";
```

Esses objetos String implementam concatenação (embora a linguagem Java não implemente sobre-carga de operadores). Exemplo:

```
Curso="Meu primeiro"+curso;
```

Uso do operador +=

```
Curso+="Meu primeiro";
```

## System

- Realiza parte da interação do programa com o sistema operacional nativo.
- Contém a maioria dos métodos que interagem com o sistema operacional.  
Exemplo: System.out.println(.....);

## VARIÁVEIS

- Devem ser declaradas antes de serem utilizadas, em qualquer parte do programa.
- Declarar uma variável é dar o tipo e o nome.
- Podem ser inicializadas na declaração.
- Variáveis locais devem ser inicializadas antes de serem utilizadas.
- Podem ser utilizados sufixos para definir o tipo de um literal.

As variáveis em Java podem ser declaradas em praticamente qualquer parte do programa. Lembre que não existem variáveis globais. Uma vez que todo o programa está encapsulado dentro de classes, as variáveis mais “externas” deste programa pertencem a esta classe.

Existem, portanto dois tipos de variáveis:

- De classe ou de instância.
- Locais a funções ou blocos.

As restrições aplicadas às variáveis são:

- Nomes das variáveis não podem começar por número.
- Cuidado com símbolos que representam operadores: \*, + etc.
- Conjunto de caracteres Unicode (existem exceções).

Tipos:

- Primitivos.
- Classe ou interface.
- Array.

Declaração e inicialização de variáveis:

```
int a;
String nome;
float b;
boolean isNumber;
byte c, d, e, f;
long g = 0;
float h, i, j = 1.2;
char primeira_letra = 'a';
String sobrenome = "Silva";
boolean isChar = true;
```

## Escopo de variáveis

Variáveis locais: Podem ser utilizadas dentro do método ou bloco onde foram declaradas.

Variáveis de classe ou de instância: Podem ser utilizadas dentro da classe ou objeto, ou até mesmo por outras classes e objetos, dependendo de sua acessibilidade.

## CONVERSÃO DE TIPOS

Conversão implícita é aquela onde o valor não corre o risco de sofrer alguma perda, porque a capacidade do tipo convertido é maior que a capacidade do tipo original. Para fazer essa conversão basta fazer a atribuição direta dos valores.

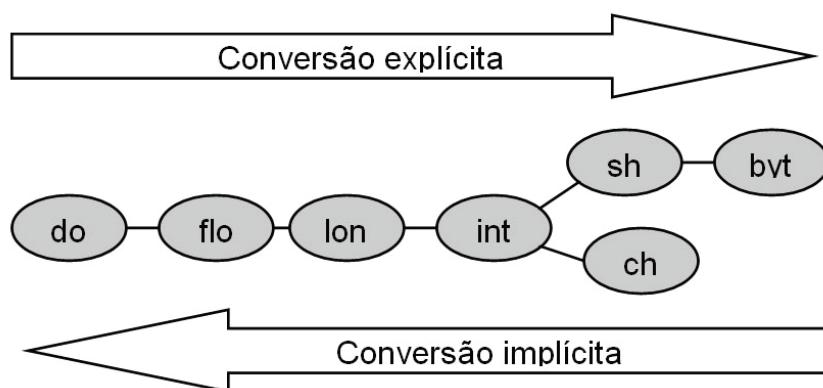
Implícita: de tipos “menores” para tipos “maiores” (em números de bits do tipo).

```
byte b = 5;  
int a = b; // o valor 5 é convertido para inteiro  
float f = 123.756;  
double d = f; // o valor 123.756 é convertido para double
```

Conversão explícita é aquela onde o valor do tipo original corre o risco de ser alterado, porque a capacidade do tipo convertido é menor. Conversões explícitas somente podem ser feitas com o uso do casting, que é definir, junto ao valor original, o tipo para o qual se quer a conversão, entre parênteses.

Explícita: de tipos “maiores” para tipos “menores” (em números de bits do tipo), deve ser feito casting:

```
long l = 12000000;  
int i = (int)l;  
double d = 395E+45;  
float f = (float)d;
```



## OPERADORES E EXPRESSÕES

### Operadores aritméticos

Básicos:

- + (adição)
- (subtração)
- \* (multiplicação)
- / (divisão)
- % (módulo, ou seja, resto da divisão inteira)

Tipo do resultado: O mais preciso entre os tipos dos operandos. Portanto  $26/7 = 3$ , uma vez que o resultado é inteiro.

## Operadores aritméticos conjugados com atribuição de valores

`a+=b <=> a=a+b`

`a-=b <=> a=a-b`

`a*=b <=> a=a*b`

`a/=b <=> a=a/b`

Cuidado com a ordem de avaliação das expressões, pois uma mesma expressão lógica pode afetar de forma diferente o resultado esperado.

## Operadores de comparação

`== igual`

`!= diferente`

`< menor`

`> maior`

`<= menor ou igual`

`>= maior ou igual`

## Operadores de incremento e decremento

`a++ ou ++a <=> a=a+1`

`a-- ou --a <=> a=a-1`

Cuidado: `b=a++` pode não ter o mesmo efeito que `b=++a !!!`

O operador de incremento, quando colocado antes da variável, faz com que ela seja incrementada antes de ser utilizada. O operador de incremento, quando colocado depois, faz com que o valor da variável seja utilizado antes do incremento.

```
int a = 5;
int b = a++; // resulta em b com valor 5
int b = ++a; // resulta em b com valor 6
```

## Operadores lógicos

`& ou && <=> A-D`

`| ou || <=> OR`

`! <=> -OR`

Existem diferenças entre `&` e `&&`; e entre `|` e `|| !!!`

## Operadores bit a bit

`& <=> A-D`

`| <=> OR`

`^ <=> XOR`

`<< <=> Deslocamento à esquerda`

`>> <=> Deslocamento à direita`

`~ <=> Complemento`

`>>> <=> Deslocamento à direita, preenchimento com zero`

## INTRODUÇÃO À LINGUAGEM JAVA

Os operadores && e || somente podem ser aplicados a tipos booleanos. Os operadores & e | podem ser aplicados a qualquer tipo primitivo, sendo que nesse caso ele realiza a operação bit a bit entre os bits dos operandos.

## INSTRUÇÕES BÁSICAS

### Comentários

Comentários de uma linha: // linha comentada

Comentários de múltiplas linhas: /\* Linhas comentadas  
linhas comentadas  
\*/

Comentários que aparecem na geração automática de código (javadoc):

```
/**Linhas comentadas linhas comentadas  
*/
```

## ESTRUTURAS DE CONTROLE

### Estrutura if else

A condição sempre deve ser um literal, variável ou expressão booleana (incluem-se aí também os retornos de métodos).

Quando o bloco de código possuir apenas uma linha, não são necessárias as chaves.

O uso do else é opcional.

```
if (condição) {  
    // bloco de código  
} else {  
    // bloco de código  
}
```

### Estrutura switch case

A expressão deve ser uma variável ou expressão que resulte em um número inteiro ou em um número que possa ser convertido para inteiro (byte, short e char).

O valor deve ser sempre um literal inteiro.

O break não é obrigatório. Nesse caso a execução do programa prossegue para o próximo bloco de código.

O default não é obrigatório. No caso de nenhum case ser igual a expressão, nenhum código é executado.

```
switch(expressao) {  
    case valor:  
        // bloco de código  
        break;  
  
        // outros cases  
  
    default:  
        // bloco de código  
}
```

## Laços: estruturas while e do while

A condição sempre deve ser um literal, variável ou expressão booleana (incluem-se aí também os retornos de métodos).

Quando o bloco de código possuir apenas uma linha, não são necessárias as chaves.

```
while(condição) {
    // bloco de código
}
```

```
do {
    // bloco de código
} while (condição);
```

Exemplos:

```
while(i<2) {
    ...
    i++;
}

do {
    ...
    i++;
} while (i<10);
```

## Laços: estrutura for

A inicialização pode declarar uma ou mais variáveis cujo escopo é o bloco de código.

A condição sempre deve ser um literal, variável ou expressão booleana (incluem-se aí também os retornos de métodos).

A inicialização, a condição e o comando não são obrigatórios.

```
for(inicialização; condição; comando) {
    // bloco de código
}

for(iteração sobre coleção) {
    // bloco de código
}
```

Exemplos:

```
for(int i=0; i<10; i++) {
    ...
}

int [ ] arrayValores = new int[5];
...
for(int a : arrayValores) {
    soma += a;
}
```

### Outras instruções importantes

Importar para o arquivo fonte as classes ou pacotes indicados:

```
import <classe ou pacote>;
```

Definir o pacote ao qual a classe pertence sendo definida por:

```
package <nome do pacote>;
```

A importação de arquivos ou pacotes deve ser feita sempre no início do arquivo Java.

A definição do pacote deve sempre ser feita antes da definição de qualquer classe ou interface no arquivo Java.

### PARÂMETROS DE ENTRADA

Um programa de linha de comando pode receber parâmetros pela própria linha de comando.

Programa para testar parâmetros de entrada:

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



## EXERCÍCIOS:

1. Faça uma aplicação Java que imprime na saída padrão o número de argumentos e os argumentos passados para o programa.

Exemplo:

```
>java Argumentos abcde 2 "Mais um Teste" 3,1416
```

Número de argumentos: 4 abcde

2

Mais um Teste 3,1416

2. Faça uma aplicação que calcule o fatorial de um número passado como argumento.
3. Faça uma aplicação que calcule o fatorial de um número passado como argumento e que teste a integridade do mesmo – verifica se o número é inteiro e se tem um valor suficientemente pequeno para que o resultado seja coerente – e imprima mensagens para o usuário alertando para possíveis erros no argumento (inclusive avisando, caso o usuário não tenha passado nenhum argumento).
4. Faça uma aplicação que converta números decimais em números hexadecimais. Exemplo:

```
>java Conversor 13
```

Decimal: 13

Hexadecimal: D

Java Conversor 64

Decimal: 64

Hexadecimal: 40

5. Faça o conversor inverso também.

### Para pensar

O que acontece se eu tentar compilar o código abaixo?

```
Short f = 1;
```

```
Short g = 2;short h;
```

```
H = f + g;
```





## 4. Classes e objetos

### PACOTES

- Define o nome completo da classe (“fully qualified name class”).
- Uso da palavra reservada package.
- Devem ser declarados nas primeiras linhas do arquivo Java.
- Um outro arquivo fonte pode referenciar uma classe pelo seu nome completo ou com o uso de import.
- O import também pode importar atributos estáticos de classes.

Exemplos de pacotes:

```
import java.lang.Math.PI;  
import java.util;
```

O nome completo de uma classe compreende o nome do pacote ao qual ela pertence junto com o nome com o qual ela foi declarada. A classe Date, do Java, está no pacote java.util, portanto seu nome completo é java.util.Date. Num arquivo fonte qualquer, para se utilizar a classe Date, ou importamos o pacote no início do fonte ou a referenciamos pelo seu nome completo.

```
import java.util.Date;  
...  
Date dt = new Date();
```

Ou

## INTRODUÇÃO À LINGUAGEM JAVA

```
import java.util.*; // importa todas as classes do pacote  
...  
Date dt = new Date();
```

Ou

```
java.util.Date dt = new java.util.Date();
```

A classe String, do Java, por exemplo, está no pacote java.lang, portanto o nome completo dessa classe é java.lang.String. No entanto, o pacote java.lang é automaticamente importado por todos os arquivos fontes.

A definição do pacote ao qual pertence uma classe é feita usando:

```
package br.com.empres.nomepacote  
public class nomeclasse {...}
```

Essa classe, tem o nome completo br.com.empres.nomepacote.nomeclasse

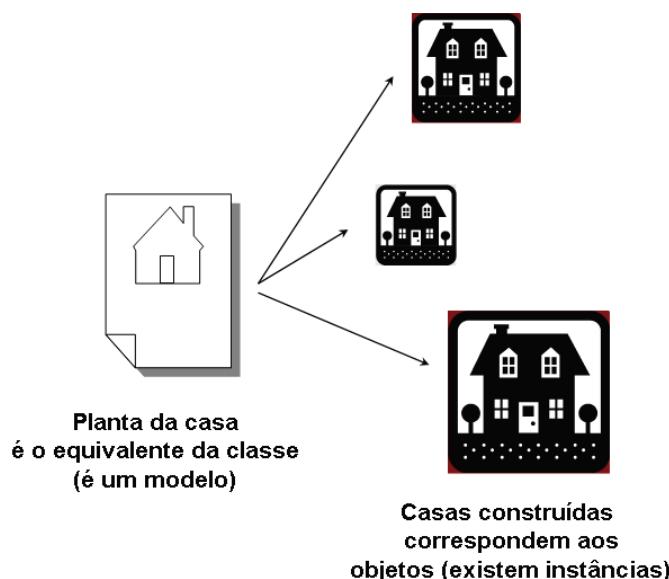
## PROGRAMAÇÃO ORIENTADA A OBJETOS

O que é?

- Unidades de código que interagem são os objetos.
- Classes.
  - Atributos e comportamento – variáveis e métodos.
  - Variáveis e métodos internos podem não ser visíveis externamente.
  - Classes podem ser definidas por herança.

Para que serve?

- Metodologia eficiente.
  - Programas grandes e desenvolvidos por equipes.
  - Grande índice de reaproveitamento do código.
- Objetos são entes.  
– Classes são definições.



Objetos são modelados como os objetos do mundo real, ou seja, possuem existência “física” dentro de um programa, ocupando espaço de memória e armazenando as informações pertinentes.

Classes são definições genéricas de objetos (moldes de objetos). Podem existir, portanto, vários objetos de uma única classe.

Classes são definidas para que a partir delas possam ser gerados os objetos que assumem aquelas propriedades e executam aquela funcionalidade definida na classe.

Um conceito simples que pode ser atribuído à classe, suficiente para sua compreensão de forma clara, é associá-la a um novo tipo. Portanto, as classes seriam os tipos de “variáveis” definidos pelo programador e os objetos seriam as próprias “variáveis” declaradas daquele tipo. Na realidade, um objeto é muito mais que uma variável. Veremos isso nas próximas páginas.

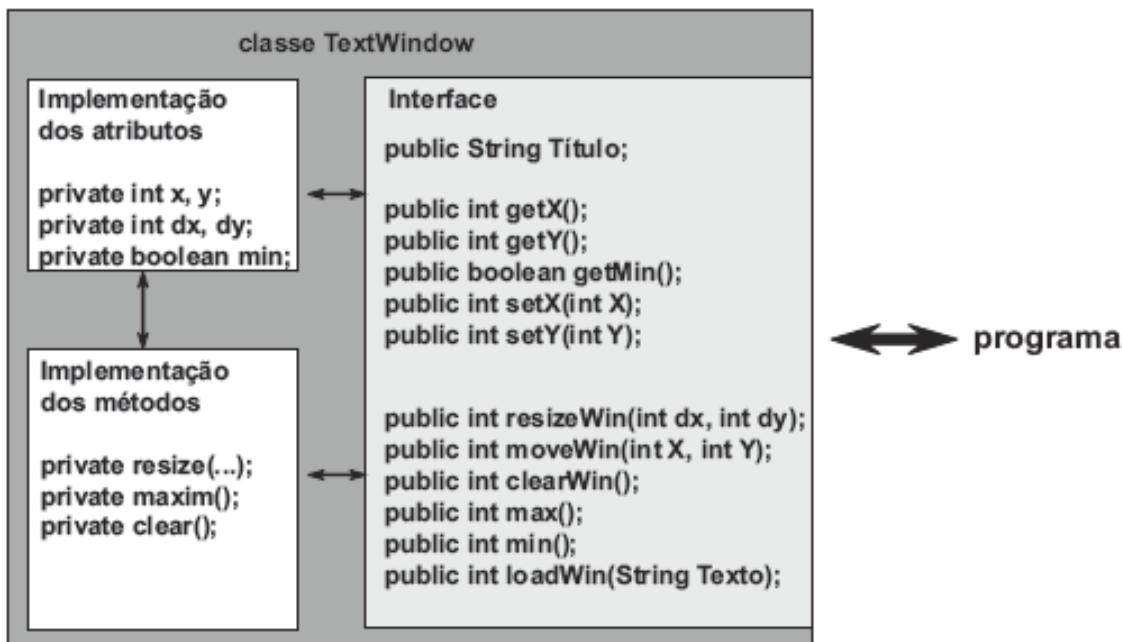
## IMPORTANTE

- Defina cuidadosamente uma classe levando-se em conta aspectos de funcionalidade, generalidade e acessibilidade.
- Nomenclatura: Classes definidas e objetos instanciados (ou seja, um objeto é uma instância de uma dada classe).
  - Defina funcionalmente as classes para tornar mais lógico o encapsulamento.
  - Defina claramente os elementos que compõem a interface permitem que outros usem seus objetos de forma simples e correta.
  - Inicie de classes mais genéricas para facilitar o reaproveitamento do código ao mesmo tempo que define o alcance da classe.
  - Tenha em mente que as classes são definidas e os objetos declarados. Portanto, todo objeto criado ocupa espaço no seu programa ou na máquina executando o programa, e deve-se evitar criar objetos não utilizados.
  - Considere aspectos de segurança na definição da acessibilidade da classe.
  - Nomenclatura de linguagem: classes são definidas e objetos são instanciados (ou seja, um objeto é uma instância de uma determinada classe).

## CONCEITO DE CLASSE

Uma classe contém uma definição de um conjunto de variáveis e funções que são encapsuladas conjuntamente e compartilham o mesmo escopo de definições. Assim, uma classe é definida por seus atributos (variáveis) e comportamento (métodos ou funções). Estas características permitem que a classe assuma um papel dentro de um programa unicamente em termos de suas propriedades, ou seja, fatores externos à classe não podem afetá-la, além do permitido por ela. Este encapsulamento é utilizado com uma série de benefícios em relação ao conceito de programação não orientada a objetos.

Toda classe é definida em termos de sua interface e sua implementação.



## ATRIBUTOS, MÉTODOS E INTERFACES

- Atributos: Variáveis ou propriedades de um objeto.
- Métodos: Comportamento do objeto.

```
<Modificadores> <tipo de retorno> <nome do método>(<lista de parâmetros>
{
    //blocos de comandos
}
```

- Interface: Métodos ou variáveis visíveis ao mundo exterior.
- Variáveis: São locais ao objeto e implementam o conceito de atributos ou propriedades de um objeto.
- Métodos: São locais ao objeto e implementam o conceito de comportamento do objeto, ou como suas propriedades se relacionam, tanto entre si, como com o mundo exterior.
- Interface: São métodos ou variáveis visíveis ao mundo exterior ao objeto. É a parte do objeto que interage com os outros objetos de um programa.

### Regras gerais

Atributos: Clareza

Classes devem ser claras e significativas com respeito às propriedades que representam.  
Podem ser outras classes.

Métodos: Abrangência

Deve abranger, por definição, “todo” o comportamento desejado da classe, inclusive aqueles ainda não implementados, ou que devem ser implementados por subclasses.

Interfaces: Segurança

Mais segurança se implementada apenas com métodos, verificação de parâmetros de entrada e mínima, mas funcional. Deve-se buscar o compromisso ideal entre estas características.

## MÉTODOS ESPECIAIS

### Método construtor

Método que inicializa o objeto. É o primeiro método chamado quando o objeto é criado (instanciado) e somente pode ser chamado nesta situação.

Usado para inicializar atributos do objeto e executar os métodos de inicialização (abrir um arquivo, por exemplo).

Deve ter o mesmo nome da classe e não possui valor de retorno. Pode existir em qualquer número para uma mesma classe (difere pelo número de argumentos).

### Método abstrato

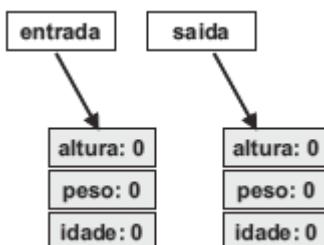
Método declarado, mas não definido, usado para forçar a implementação de um comportamento desejado nas subclasses que herdarem este método.

## DEFINIÇÃO DE UMA CLASSE E ALOCAÇÃO

### Declaração da classe

A classe Biométrica possui três atributos públicos:

```
public class Biometrica {  
    public int altura, peso, idade;  
}
```



### Declaração do objeto

A declaração do objeto nada mais é do que declarar as variáveis que apontarão para as instâncias. Na declaração, os objetos não estão ainda instanciados, apenas nomeados.

```
Biometrica entrada, saída;
```

### Instanciação do objeto

Os objetos sempre são instanciados pelo operador new, que invoca um construtor da classe. Nesse exemplo, o construtor default foi utilizado. O construtor default é definido implicitamente a todas as classes que não possuem construtor definido. Ele é um construtor que não recebe argumento.

```
entrada = new Biometrica ();  
saída = new Biometrica ();
```

## Construtores e instanciação

Construtores podem ser definidos em qualquer número, desde que difiram entre si na lista de argumentos que recebem.

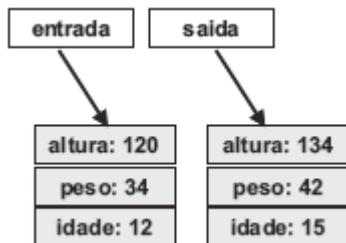
A palavra reservada this, dentro de uma classe, refere-se à instância corrente dessa classe. Assim, this.altura refere-se ao atributo altura da classe, enquanto a variável local altura não é referenciada por ela.

- Definir um construtor mais útil.

```
public class Biometrica {  
    public int altura, peso, idade;  
    public Biometrica (int altura, int peso, int idade) {  
        this.altura = altura; this.peso = peso;  
        this.idade = idade;  
    }  
}
```

- Instanciação de objetos.

```
entrada = new Biometrica (120, 34, 12);  
saída = new Biometrica (134, 42, 15);
```

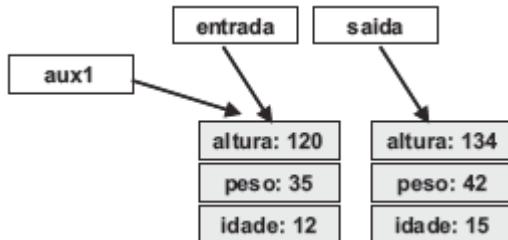


## Referências a objetos

Declaração de objetos que recebem outras instâncias.

```
entrada = new Biometrica (120, 34, 12);  
saída = new Biometrica (134, 42, 15);
```

```
Biometrica aux1 = entrada;  
aux1.peso = 35;
```



Objetos declarados (mas não instanciados) podem ser atribuídos a objetos instanciados. Nesse caso, tem-se duas referências para o mesmo objeto. Qualquer alteração feita por meio de qualquer referência afeta o objeto em questão.

## Declaração de métodos

Métodos são funções internas à classe.

```
public class Biometrica {
    public int altura, peso, idade;
    public Biometrica (int altura, int peso, int idade) {
        this.altura = altura;
        this.peso = peso;
        this.idade = idade;
    }
    public float obterRelacaoAlturaPeso( ) {
        return this.altura/this.peso;
    }
}
```

Um método é uma função definida dentro de uma classe e que implementa um comportamento desta classe. Quando não possuir tipo de retorno deve ser utilizada a palavra chave void.

```
public void incrementaridade( ) {
    this.idade++;
}
```

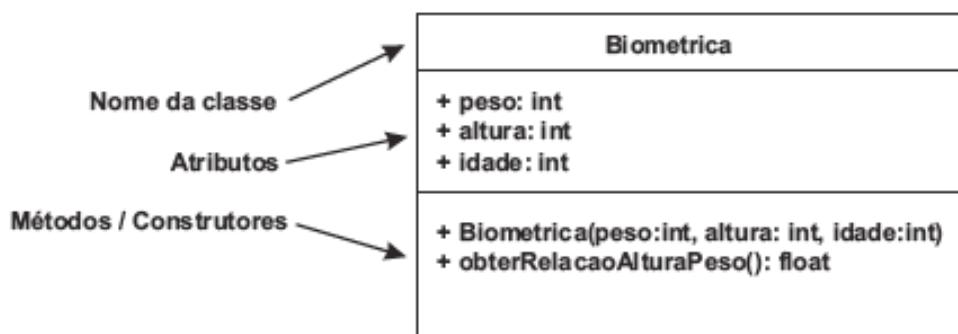
Os argumentos de um método devem ser definidos pelos seus tipos.

```
public void acrescentaraltura(int aumento) {
    this.altura += aumento;
}
```

## NOTAÇÃO UML

A UML (Unified Modeling Language – Linguagem de Modelagem Unificada) é o padrão para especificação de sistemas orientado a objetos.

Classe:



A notação UML é um padrão utilizado amplamente para especificar sistemas orientados a objetos. O diagrama de classes é um dos mais utilizados.

A notação para especificar uma classe é um retângulo que possui três regiões: uma com o nome da classe, outra com os seus atributos e uma terceira com os seus métodos ou construtores.

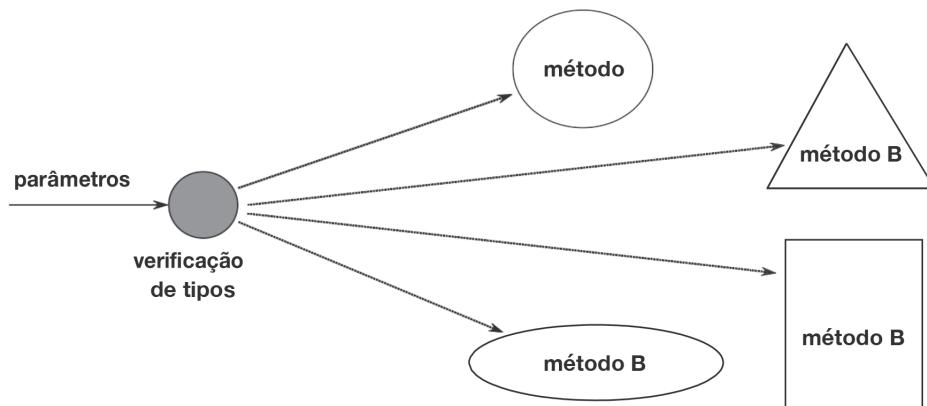
Os atributos são especificados com seus tipos e sua acessibilidade:

- + para acesso público e
- para atributos privados

Os métodos são especificados com seus tipos de retorno e lista de argumentos, além da acessibilidade, como os atributos.

## Sobrecarga de métodos

Definição de múltiplos métodos com o mesmo nome, mas tipos de parâmetros diferentes.



Polimorfismo de métodos ou sobrecarga de métodos permite que métodos com o mesmo nome sejam implementados de forma diferente. Isto se torna muito útil quando empregado para prover o “mesmo” comportamento para um conjunto de parâmetros diferentes. Por exemplo: um conjunto de métodos chamados soma pode ser definido para executar a soma entre inteiros, floats e strings, e o emprego do mesmo nome facilita a programação e a leitura do código.

- Métodos com mesmo nome, mas com lista de argumentos diferentes:

```
public void setpeso(int peso) {  
    this.peso = peso;  
}  
public void setpeso(int peso, int redutor) {  
    this.setpeso(peso - redutor);  
}
```

- Invocação dos diferentes métodos:

```
entrada.setpeso(55);  
entrada.setpeso(48, 5);
```

- Métodos com mesma assinatura causam erro de compilação:

```
void resize(int x, int y, int dx, int dy) {  
    this.x=x;  
    this.y=y;  
    this.dx=dx;  
    this.dy=dy;  
}
```

```
void resize(int x1, int y1, int x2, int y2) {  
    x=x1;  
    y=y1;  
    dx=x2-x1;  
    dy=y2-y1;  
}
```

- Tipos de retorno não diferenciam métodos:

```
int calc(int a, int b, int c, int d) {...}  
float calc(int a, int b, int c, int d) {...}
```

**CUIDADO:**

Os nomes dos argumentos não diferenciam um método do outro, portanto dois métodos com lista de argumentos em igual número e tipos, mas com nomes diferentes, não são sobrecarregados e geram um erro de compilação.

**Sobrecarga de construtores**

- Construtores com lista de argumentos diferentes:

```
public Biometrica(int altura, int peso, int idade) {
    this.peso = peso;
    this.altura = altura;
    this.idade = idade;
}
public Biometrica(int altura, int idade) {
    this(altura, altura - 110, idade);
}
```

- Invocação dos diferentes construtores:

```
entrada = new Biometrica(155, 49, 17);
saída = new Biometrica(180, 21);
```

Da mesma forma que métodos, os construtores também podem ser sobrecarregados. A diferenciação ocorre da mesma forma, pela lista de argumentos.

A invocação de um construtor dentro de outro utiliza a notação `this(...)`, onde, pela lista de argumentos passados é identificado o construtor invocado.

A identificação de qual construtor foi invocado é feita pelos argumentos. Construtores sobrecarregados devem possuir listas de argumentos diferentes em quantidade e/ou tipos.

**CUIDADO:**

Os nomes dos argumentos não diferenciam um construtor do outro, portanto dois construtores com lista de argumentos em igual número e tipos, mas com nomes diferentes, não são sobrecarregados e geram um erro de compilação.

**Classes e métodos abstratos**

Métodos abstratos são aqueles que não possuem implementação.

Uma classe com apenas um método abstrato não pode ser instanciada e é também chamada de classe abstrata.

Pertence à interface da classe, impondo um contrato para as subclasses (Estudaremos isso no próximo capítulo).

```
abstract public class Biometrica {
    ...
    abstract public int calcularPesoideal();
```

Classes abstratas não podem ser instanciadas, uma vez que “falta” parte de sua implementação. Uma classe abstrata serve para estabelecer uma interface cuja subclasses, se precisarem ser instanciadas, necessariamente terão que implementar.

Nos capítulos seguintes, quando for tratado o tema Polimorfismo, ficará clara a vantagem do uso de classes abstratas.

### Atributos e métodos estáticos

- Atributos e métodos de classe.
- Alocação na classe.
- Compartilhamento entre todos os objetos da classe:

```
public class Biometrica {  
    public static int pesoMinimo = 25;  
    public int altura, peso, idade;  
    public Biometrica (int altura, int peso, int idade) {  
        this.altura = altura;  
        this.peso = peso;  
        this.idade = idade;  
    }  
    public static int obterPesoMinimo( ) {  
        return pesoMinimo  
    }  
}
```

Atributos e métodos estáticos são também chamados de atributos ou métodos de classe, em contraposição aos atributos e métodos de instância, que assumem um valor para cada instância (objeto).

São alocados uma única vez em memória, e permanecem alocados enquanto a classe estiver carregada.

Um atributo estático é compartilhado entre todos os objetos da classe. Se o atributo for público, outras classes e objetos também utilizam a sua única alocação em memória.

- Referência a variáveis/métodos estáticos:

```
Biometrica.pesoMinimo = 20;  
int p = Biometrica.obterPesoMinimo( );  
  
entrada.pesoMinimo = 28  
int q = saida.obterPesoMinimo( );
```

As variáveis estáticas são alocadas nas classes e não nos objetos. Por exemplo, a mesma variável pode ser alocada em qualquer uma das formas a seguir:

```
Biometrica.pesominimo  
entrada.pesominimo  
saida.pesominimo
```



## **EXERCÍCIOS:**

1. Construa uma classe – chamada Ponto – que defina um ponto no plano.

Um ponto é caracterizado por duas coordenadas (x e y, por exemplo). O nome do arquivo deve ser Ponto.java.

Faça um método main para essa classe.

Neste método, defina dois objetos pontos p1 e p2. Atribua ao ponto p1 as coordenadas (10,10).

Mostre os valores dessas coordenadas na saída padrão. Atribua ao ponto p2 as coordenadas (50,50).

Mostre os valores dessas coordenadas na saída padrão. Declare uma referência a Ponto com nome p3.

Atribua p1 a p3.

Atribua ao ponto p3 as coordenadas (30,30).

Mostre as coordenadas de todos os três pontos.

2. Faça um método para Ponto que calcule a distância entre o ponto e um outro passado como argumento. A definição do método pode ser:

```
double distancia(Ponto p) {  
    ...  
}
```





# 5. Array e string

## ARRAYS

- Declaração:

```
String palavras[ ];      ou      String[ ] palavras;  
Data[ ] vencimentos;    ou      Data vencimentos[ ];  
int arrVal[ ];          ou      int[ ] arrVal;
```

- Instanciação do Array:

```
arrVal=new int[200];  
vencimentos =new Data[10];  
String[ ] palavras = ("Estas", "palavras", "formam", "uma", "frase");
```

- Acesso:

```
ArrVal[0]=23;  
Vencimento[8] = new Data(23, 4, 2005); String s = palavras[2];  
Tamanho  
Int l = arrval.length;
```

Um array é muito semelhante a um objeto em Java. Ele precisa ser declarado e instanciado, o que pode ser feito numa mesma linha:

```
Data[ ] vencimentos =new Data[10];
```

## INTRODUÇÃO À LINGUAGEM JAVA

Esse “objeto” array tem um único atributo já pré-definido chamado length, que determina o tamanho ao array.

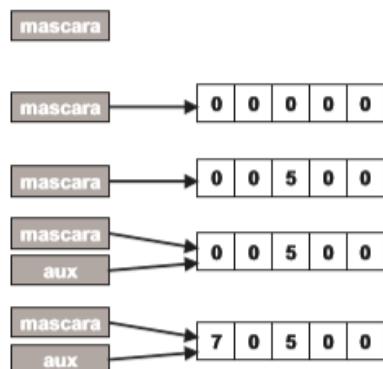
Em linguagem Java, um array é delimitado na sua instanciação. Assim, os casos de referência a índices não existentes não podem ocorrer. Isso garante uma forma segura, direta e simples de uso de arrays. Os índices dos arrays sempre se iniciam em 0. Assim, um array de n elementos, pode ser indexado de 0 a n-1.

```
Int[] intervalos = {2, 5, 6};  
Int x = intervalos[0]; // x recebe 2  
int y = intervalos[3]; // ERRO!  
//Array de 3 elementos tem  
//índices de 0 a 2
```

Arrays podem ser de tipos primitivos ou de objetos. No caso de array de objetos convém notar que a instanciação do array não instancia os objetos que ele pode conter.

### ALOCAÇÃO DE ARRAYS

```
Array de tipos primitivos  
byte[] mascara;  
Mascara = new byte[5];  
mascara[2] = 5;  
byte[] aux = mascara;  
aux[0] = 7;
```

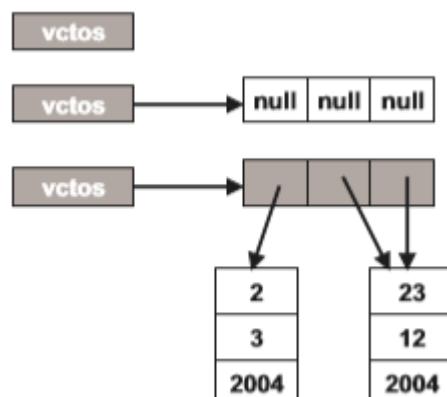


Um array, ao ser declarado, apenas reserva uma referência que pode apontar para um “objeto” array. No exemplo, “mascara” é apenas uma referência. Mais de uma referência podem apontar para o mesmo array, assim como ocorrem para referências de objetos. No exemplo, “mascara” e “aux” apontam para o mesmo array.

Um array de tipos primitivos, ao ser instanciado, tem o seu espaço alocado em memória, e todos valores alocados são inicializados com 0.

- Arrays de objetos:

```
Data[] vctos;  
vctos = new Data[3];  
vctos[0] = new Data(2, 3, 2004);  
vctos[2] = new Data(10, 12, 2003);  
vctos[1] = vctos[2];  
vctos[1].setdia(23);
```



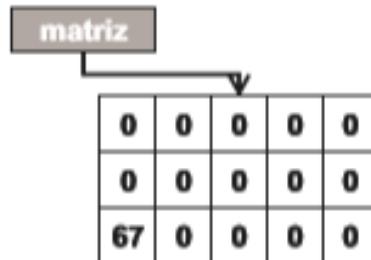
Um array de objetos, ao ser instanciado, tem o seu espaço alocado em memória, e todos os valores alocados são inicializados com null.

Cada elemento do array precisa ser instanciado individualmente e comporta-se como uma referência para o objeto alocado para ele. Expressões do tipo vctos[1].setdia(23) servem para acessar os objetos de cada posição do array.

## ARRAYS MULTIDIMENSIONAIS

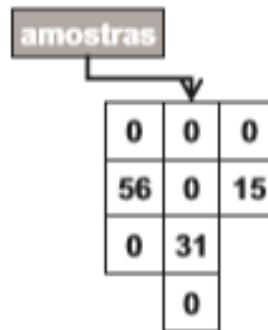
- Arrays de dimensões homogêneas:

```
int[ ][ ] matriz = new int[5][3];
matriz[0][2] = 67;
Ponto[ ][ ][ ] cubo = new Ponto[3][6][2];
cubo[2][5][1] = new Ponto(34, 65);
```



- Arrays de dimensões heterogêneas:

```
int[ ][ ] amostras = new int[3][ ];
amostras[0] = new int[3];
amostras[1] = new int[4];
amostras[2] = new int[2];
amostras[2][1] = 15;
amostras[0][1] = 56;
amostras[1][2] = 31;
```



Arrays multidimensionais com dimensões homogêneas podem ser instanciados numa única linha de comando, passando-se as duas dimensões:

```
int[ ][ ] matriz = new int[5][3];
```

As dimensões podem ser obtidas separadamente: matriz.length resulta em 5 e matriz[0].Length resulta em 3.

Arrays com dimensões heterogêneas precisam ter cada dimensão alocada separadamente. Na instanciação da primeira dimensão, as demais são deixadas sem valor.

```
int[ ][ ] amostras = new int[3][ ];
```

E depois, para cada posição da primeira dimensão devem ser alocadas as demais dimensões.

```
amostras[0] = new int[3];
amostras[1] = new int[4];
amostras[2] = new int[2];
```

## STRING

- String é uma classe, não um tipo primitivo.
- String tem alguns comportamentos especiais, típicos de tipos primitivos:
  - Instanciação é semelhante a inicialização.
  - String s = "teste";
  - Concatenação.
  - S = "um outro" + " teste";
- Strings são sempre constantes.
- Strings devem ser comparadas como objetos (usar método Equals da classe String):
 

```
str1.Equals(str2)
```

## INTRODUÇÃO À LINGUAGEM JAVA

String em Java é uma classe que tem alguns comportamentos diferenciados das demais classes. Uma string pode ser instanciada da mesma forma que um tipo primitivo é inicializado. Repare que continua sendo uma instanciação, ou seja, um objeto é criado em memória. Strings podem ser concatenadas com o operador +. Essa operação de concatenação gera uma outra string.

Strings devem ser comparadas através do método Equals, da própria classe. Esse método compara, caractere a caractere, duas strings. O operador == ao ser aplicado sobre strings compara as referências e não os conteúdos. Muitas vezes esse operador tem resultado semelhante ao método Equals, isso porque a Máquina Virtual Java procura compartilhar as strings iguais, evitando a múltipla instanciação de strings iguais. Isso só é possível porque as strings são constantes.



### EXERCÍCIOS:

1. Fazer uma array método que transforme um array de strings em uma string concatenada.
2. Monte um array método que faça a multiplicação de matrizes usando arrays. Lembrar que:

$$\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \times \begin{array}{cc} X & \\ & \end{array} \begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} = \begin{array}{cc} a_{11}*b_{11} + a_{12}*b_{21} & a_{11}*b_{12} + a_{12}*b_{22} \\ a_{21}*b_{11} + a_{22}*b_{21} & a_{21}*b_{12} + a_{22}*b_{22} \end{array}$$

Considere que as dimensões das matrizes podem ser NXM onde N e M >=1.

### Para pensar

Como fazer cópia de arrays?



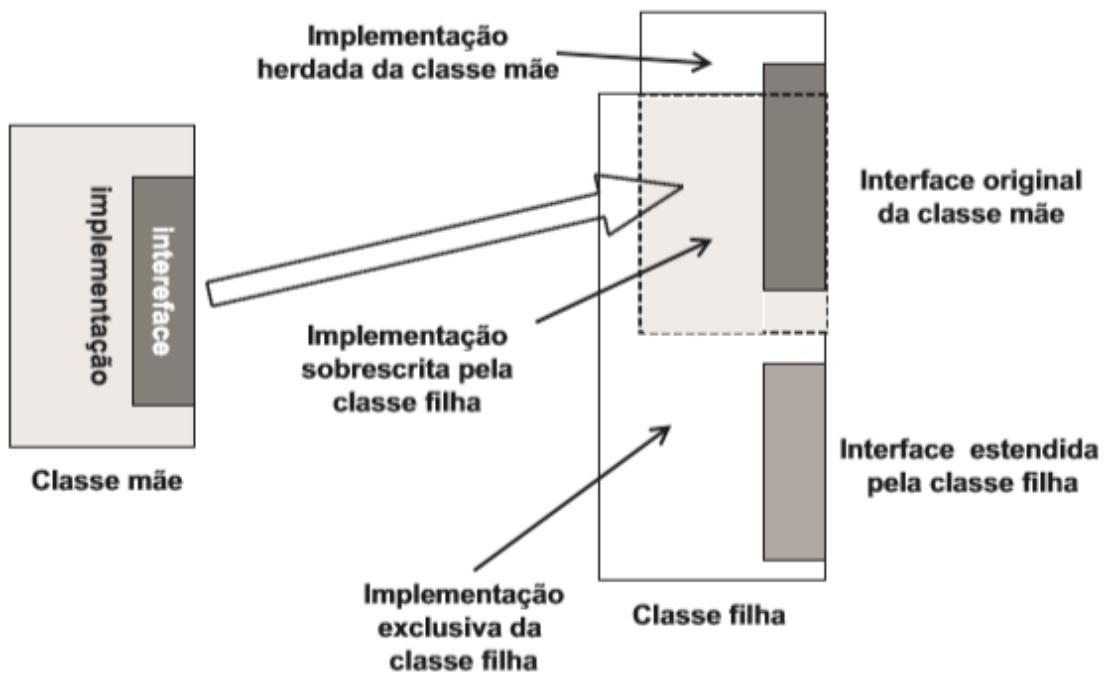
# 6. Herança e interfaces

## HERANÇA

Usando herança podemos definir uma nova classe a partir de outra, preservando sua interface e acrescentando ou alterando sua implementação:

- É uma forma pela qual novas classes são definidas de forma incremental.
- Definição de uma nova classe especificando uma classe mãe e um conjunto adicional de atributos e/ou métodos.
- A classe definida dessa forma herda todos os atributos e métodos da classe mãe.
- A acessibilidade da interface da classe mãe não pode ser redefinida na classe herdada.
- A linguagem Java não permite herança múltipla com classes, mas apenas com interfaces.

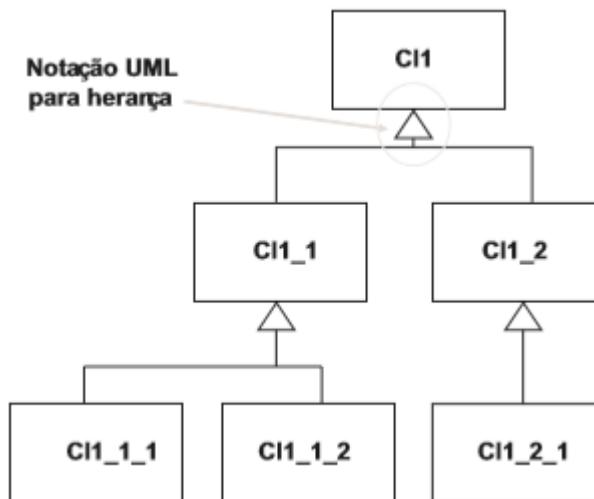
## Anatomia da herança



A partir de uma classe mãe com sua implementação e sua interface, vemos que uma classe filha pode ser definida com sua interface e sua implementação baseada na classe mãe com as seguintes características:

- A interface original da classe mãe é mantida. A herança nunca “perde” interface.
- A classe filha pode estender a interface.
- A implementação da classe mãe é totalmente herdada pela filha, no entanto parte dessa implementação pode ser sobreescrita pela classe filha. Nesse caso, sob a interface herdada da classe mãe podem estar implementações completamente diferentes.
- A classe filha pode estender a implementação.

## Herança e hierarquia de classes



- I1 é a superclasse de Cl1\_1 e Cl1\_2.
- Cl1\_1 e Cl1\_2 são subclasses de I1.
- Cada classe pode possuir uma única superclasse.
- Cada classe pode possuir qualquer número de subclasses.
- Cl1\_1\_1 e Cl1\_1\_2 são classes distintas, bem como Cl1\_2\_1.

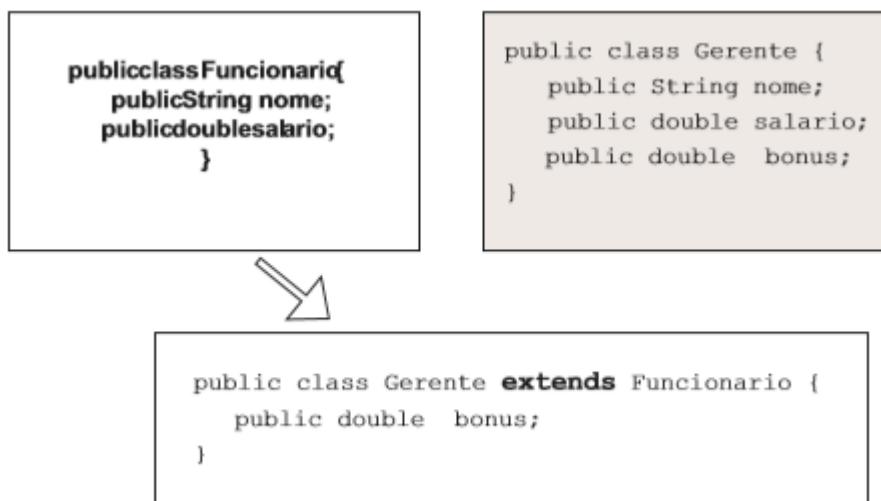
A aplicação de herança cria uma hierarquia de classes, onde as classes primárias são mais genéricas e as classes finais são mais específicas.

O esquema hierárquico de classes permite estabelecer relações especiais entre determinadas classes, ou seja, uma subclass tem um relacionamento diferenciado com sua superclasse e esta, por sua vez, engloba várias características comuns a todas as suas subclasses. A utilização destas relações especiais dá um grande poder a um programa orientado a objetos.

No esquema apresentado pode-se ver como é a notação UML para indicar herança.

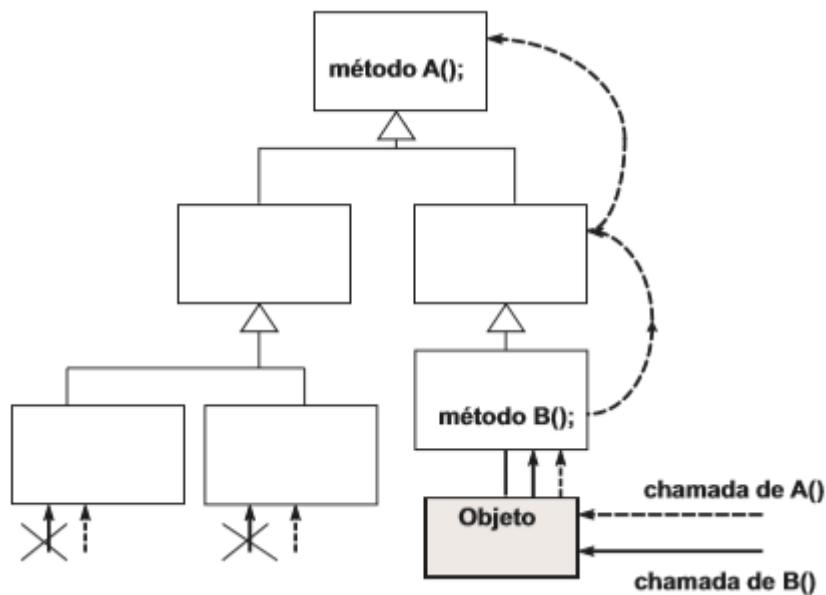
### Definição de subclasses

- Uso da palavra reservada extends:



A palavra-chave **extends** é usada para estabelecer uma relação de herança entre duas classes. Gerente é subclass de Funcionario, e Funcionario, por sua vez, é superclasse de Gerente. Note que a relação estabelece que todas as propriedades de Funcionario são herdadas por Gerente, que também pode definir propriedades adicionais.

### Chamada de métodos na herança



Uma das relações especiais entre superclasse e subclasse é o compartilhamento de métodos. O esquema de herança é responsável pela determinação do local exato da definição de um método, não definido na subclasse, mas em um de seus ancestrais.

### Sobrescrita de métodos

Classe Funcionario com método para calcular custo mensal total do Funcionario:

```
public class Funcionario {  
    public String nome;  
    public double salario;  
    public static double encargo = 1.8;  
    public double obterCusto() {  
        return this.salario * encargo;  
    }  
}
```

Classe Gerente com o mesmo método:

```
public class Gerente extends Funcionario {  
    public double bonus;  
    static public double encargoBonus = 2.1;  
    public double obterCusto() {  
        return this.salario * encargo + this.bonus *  
encargoBonus;  
    }  
}
```

Na classe Gerente a implementação do método `obterCusto()` deve ser diferente, pois deve levar em conta o bônus que somente o gerente recebe. Quando uma subclasse reimplementa um método da superclasse, diz-se que esse método foi sobreescrito na subclasse. Nesse caso, a implementação que veio por herança de Funcionario foi descartada.

Gerente com sobreescrita de métodos, mas utilizando o método da superclasse:

```
public class Gerente extends Funcionario {
    public double bonus;
    static public double encargoBonus = 2.1;
    public double obterCusto() {
        return super.obterCusto() + this.bonus *
encargoBonus;
    }
}
```

Ao sobreescrivendo um método, uma subclasse deve reaproveitar o método original, para não reescrever regras. A sobrecarga de métodos deve preferencialmente prover um comportamento adicional à nova classe. Sendo assim, o comportamento original, herdado da superclasse, deve colaborar para compor também o novo comportamento.

A palavra reservada `super` tem a função de referenciar membros definidos na superclasse. Então, `super.obterCusto()` referencia o método definido na superclasse `Funcionario`.

## **Herança e construtores**

Construtores não são herdados.

- Construtor de `Funcionario`:

```
public Funcionario(String nome, double salario) {
    this.nome = nome;
    this.salario = salario;
}
```

- Construtor de `Gerente`

```
public Gerente(String nome, double salario, double bonus) {
    super(nome, salario);
    this.bonus = bonus;
}
```

Os construtores de uma classe não são herdados pelas suas subclasses. No entanto, as regras definidas nos construtores das superclasses podem e devem ser reutilizadas nos construtores das subclasses. Na classe `Gerente`, o uso de `super(...)` invoca o construtor da superclasse e portanto utiliza o comportamento já definido em `Funcionario`.



### EXERCÍCIOS:

1. Faça uma classe que represente uma conta bancária.
2. Faça uma subclasse de conta em que o método debita; leve em conta o limite.
3. Faça uma classe Banco que contenha um grupo de contas (que devem ser alocados em um array de Contas). Faça métodos que efetuem cadastramento de contas, movimentações financeiras (débitos e créditos em contas especificadas pelo seu número). Faça um método que informe o ativo do banco (soma de todos os saldos). Faça um método que debite um valor fixo (taxa de administração de R\$ 3,50) de todas as contas.

## INTERFACES

- Classes e interfaces são equivalentes, exceto pelo fato de que interfaces não podem ser instanciadas.
- Interface é uma classe abstrata com todos os métodos abstratos.
- Herança múltipla em Java: cada classe pode ter apenas UMA superclasse e implementar VÁRIAS interfaces.

Interfaces são o conjunto de atributos estáticos (variável única compartilhada por todos os objetos de uma mesma classe) e/ou métodos abstratos (métodos com assinatura definida, mas corpo não definido) que estabelecem uma série de comportamentos que se espera da classe que a implementa.

Ausência de herança múltipla na linguagem Java é uma limitação amenizada pelo emprego de interfaces. A linguagem permite que uma classe implemente uma ou mais interfaces.

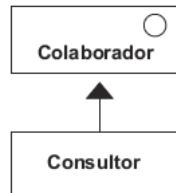
Vantagem:

- Duas classes distintas que implementam a mesma interface respondem a um conjunto idêntico de funções (embora não necessariamente da mesma forma).

Classes e interfaces são equivalentes, exceto pelo fato de que interfaces não podem ser instanciadas. Interface é uma classe abstrata com todos os métodos abstratos.

- Declaração de interfaces

```
public interface Colaborador { public double obtercusto();  
}
```



- Implementação de uma interface

```
public class Consultor implements Colaborador {  
    public double valorhora;  
    public int totalhoras;  
    public double obterCusto() {  
        return this.valorhora * this.totalhoras;  
    }  
}
```

Uma interface é declarada com o uso da palavra reservada `interface`, que elimina a necessidade de declarar os métodos como abstratos. Numa interface, todos os métodos são necessariamente abstratos.

Uma classe pode implementar uma ou mais interfaces. Para a classe ser concreta (não abstrata) é necessário que todos os métodos das interfaces que ela implementa sejam implementados.

Na notação UML uma interface é representada como uma classe, mas marcada com o estereótipo de interface (um círculo no canto superior direito do retângulo da classe).

## Exemplo do uso de interfaces

Vamos supor um programa usado por um país fictício que cobra um imposto único anual sobre todas as propriedades e/ou bens de seus cidadãos. Vamos limitar os bens a imóveis e automóveis e modelar as classes que comporiam este programa.

Como se trata de um país moderno, as unidades de registro são os bens e não as pessoas (direito à privacidade).

Como existem vários tipos de imóveis e automóveis, convém criar as classes básicas a partir das quais são herdados os vários registros (classes) diferentes.

Vamos simplificar nosso exemplo!

Definição das classes bases:

```
class Imovel {
    String proprietario;
    String localizacao;
    int valor;

    void mudaProp(String p) {
        //verifica e efetua;
    }

    void mudaValor(int v) {
        //verifica e efetua;
    }
}
```

```
class Auto {
    String proprietario;
    String endereco;
    String placa;
    int valor;

    boolean licenciamento() {
        //efetua;
    }

    int verificaMultas() {
        //verifica e efetua;
    }
}
```

Uma interface é utilizada para se definir um contrato que deve ser obedecido por todas as classes que a implementam.

No exemplo dado, são definidas duas hierarquias de classes, uma relacionada a bens móveis e outra a bens imóveis. Como são hierarquias distintas, não existe uma superclasse comum a todas essas classes que forma um contrato para todas ao mesmo tempo. A solução, então, é definir uma interface que força a implementação de um método que se deseja que todas as classes implementem.

## INTRODUÇÃO À LINGUAGEM JAVA

Exemplo. Definição das classes derivadas:

```
class Casa extends Imovel {  
    int areaConstruida;  
    int numPavimentos;  
    boolean usoComercial;  
    .  
}  
  
class Predio extends Imovel {  
    int areaTerreno;  
    int numApto;  
    .  
    boolean vistoria() {...}  
}  
  
class Terreno extends Imovel {  
    int area;  
    boolean murado;  
    .  
}
```

```
class Carro extends Auto {  
    String modelo;  
    String cor;  
    int numPassageiros;  
    .  
}  
  
class Caminhao extends Auto {  
    String modelo; int taraMax;  
    .  
    void autorizaCarga() {...}  
}  
  
class Moto extends Auto {  
    String modelo;  
    int cc;  
    .  
}
```

Exemplo. Definição da interface:

Um imposto único se aplica a todos os bens, portanto a todas as classes anteriores.

O cálculo pode ser distinto, mas existe um procedimento único na cobrança, no cálculo e no registro, portanto pode ser implementada uma interface para isso.

```
public interface IUSB { //Imposto Unico Sobre Bens  
    static final float aliquota=0.01; // 1% - aliquota unica  
    static final Data venc = new Data(30, 9, 2006);  
  
    abstract double calculoImposto();  
    abstract boolean verificaVencimento(Data data);  
}
```

O modificador final, quando aplicado a variáveis, as tornam constantes, ou seja, uma vez atribuídas, não podem ser alteradas.

Exemplo. Redefinição das classes bases:

```
abstract class Imovel
    implements IUSB{
    String proprietario;
    String localizacao;
    int valor;

    void mudaProp(String p) {
        //verifica e efetua;
    }

    void mudaValor(int v) {
        //verifica e efetua;
    }
}
```

```
abstract class Auto
    implements IUSB{
    String proprietario;
    String endereco;
    String placa;
    int valor;

    boolean licenciamento() {
        //efetua;
    }

    int verificaMultas() {
        //verifica e efetua;
    }
}
```

Ao definir que as classes bases implementam a interface IUSB, força-se para que todas as suas subclasses devam implementar os métodos definidos na interface.

Como as classes bases não implementam (nesse exemplo) os métodos abstratos da interface, elas devem ser declaradas abstratas (pois possuem agora métodos abstratos, herdados da interface).

Exemplo. Redefinição da classe:

```
class Casa extends Imovel {
    int areaConstruida;
    int numPavimentos;
    boolean usoComercial;

    int calculoImposto() {
        return valor*indiceArea(areaConstruida)*aliquota;
    }

    boolean verificaVencimento(Data data) { . . . }
}
```

```
class Caminhao extends Auto {
    String modelo;
    int taraMax;

    int calculoImposto() {
        return valor*aliquota-desconto(taraMax);
    }

    boolean verificaVencimento(Data data) { . . . }
}
```

As subclasses de Imovel e de Auto que forem concretas (podem ser instanciadas) devem agora implementar os métodos definidos pela interface. Note que essas subclasses não são abstratas, porque não possuem mais métodos abstratos.



### EXERCÍCIO:

1. Codificar e testar o exemplo de uso de interfaces.

### Modificador final

O modificador final, quando aplicado a variáveis, as transformam em constantes. Uma vez atribuídas, não podem mais ser alteradas.

O modificador final, quando aplicado a métodos, impedem que eles sejam sobreescritos.

O modificador final, quando aplicado a classes, impede que elas sejam utilizadas como superclasses. Ou seja, uma classe final não pode ser herdada por nenhuma outra classe.



## 7. Encapsulamento

- É a característica da orientação a objetos que permite que uma classe defina a acessibilidade de seus membros.
- Outras classes ou objetos de outras classes têm acesso somente aos membros cuja classe permitiu acesso.
- Nem mesmo uma subclasse pode ter acesso a membros que a superclasse não deu acessibilidade.

O encapsulamento é uma característica que confere à classe o poder de determinar como será o acesso a seus membros. Essa característica possibilita que atributos sejam corretamente protegidos de acessos indevidos.

Com o uso correto do encapsulamento evita-se que objetos assumam valores de propriedades inconsistentes. Evita-se também que comportamentos não públicos de uma classe sejam utilizadas externamente a ela.

Os modificadores de acesso são os instrumentos da linguagem Java para definir o encapsulamento.

### MODIFICADORES DE ACESSO

Palavras reservadas que determinam o acesso permitido a classes e seus membros:

public  
protected  
private

## INTRODUÇÃO À LINGUAGEM JAVA

Notação UML ( + , - e #)

- + public
- private
- # protected

Na linguagem Java, existem quatro modificadores de acesso: os três indicados pelas palavras-chaves public, protected e private, mais o modificador dado pela ausência de palavra-chave. Ou seja, a ausência de modificador de acesso é uma definição de acessibilidade também.

A notação UML emprega os símbolos + para indicar que um membro é público, - para privado e # para protegido.

### **Public**

Quando aplicado a classes e interfaces determinam que a classe pode ser utilizada livremente por qualquer outra classe ou interface.

E quando é aplicado a membros determinam que o membro pode ser utilizado livremente por qualquer outra classe.

### **Protected**

Não se aplica a classes e interfaces.

Aplicado a membros determinam que o membro pode ser invocado ou utilizado apenas por classes do mesmo pacote ou por subclasses.

### **Sem modificador**

Ao aplicar para classes e interfaces determinam que a classe pode ser utilizada apenas no mesmo pacote.

E aplicado a membros determinam que o membro pode ser utilizado apenas no mesmo pacote.

### **Private**

Não se aplica a classes e interfaces.

Aplicado a membros determinam que o membro pode ser invocado ou utilizado apenas pela própria classe.

Exemplo:

```
public class Data {

    private int dia, mes, ano;
    public void setDia(int dia) {
        if(dia<0) this.dia=1;
        else if(dia>28 && this.mes==2 && !this.isBissexto()) this.dia=28;
        else if(dia>29 && this.mes==2 && this.isBissexto()) this.dia=29;
        else if(dia>30 && (this.mes==4 || this.mes==6 || this.mes==9 || this.mes==11)) this.dia = 30;
        else if(dia>31) this.dia = 31;
        else this.dia = dia;
    }
    public void setMes(int mes) {
        if(mes<0) this.mes = 1;
        else if(mes>12) this.mes = 12;
        else this.mes = mes;
    }
    public void setAno(int ano) {
        this.ano = ano;
    }
    final public boolean isBissexto() {
        return (this.ano%4==0 && (this.ano%100!=0 || this.ano%400==0));
    }
}
```

Todos os atributos de Data são privados, o que impede que eles recebam diretamente valores vindos externamente à classe. Assim, data1.dia = 35, não é permitido, evitando datas inconsistentes.

Para atribuir um valor a um atributo de Data deve-se utilizar um método público. Na implementação desses métodos é garantido a consistência dos seus atributos. Assim, a cada mês é permitido um valor máximo para o dia. Se o usuário de um objeto da classe Data passar como argumento algum valor incompatível, o método impede que esse valor chegue ao atributo.

Exemplo: Construtor de Data

```
public class Data {
    ...
    public Data(int dia, int mes, int ano) {
        this.setAno(ano);
        this.setMes(mes);
        this.setDia(dia);
    }
}
```

O construtor deve garantir também o encapsulamento.



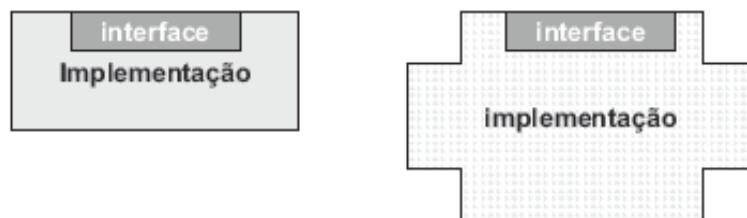
### EXERCÍCIOS:

1. Digitar e testar a classe Data deste capítulo.
2. Encapsular corretamente as classes Conta e ContaEspecial do capítulo anterior.



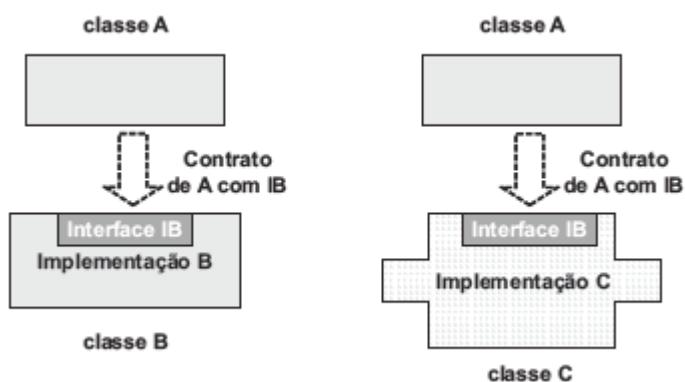
## 8. Polimorfismo

- Característica que permite que uma mesma interface assuma implementações distintas.
- Classes com interfaces semelhantes, mas implementações diferentes, são polimórficas.



Polimorfismo é a característica da orientação a objetos que determina que uma interface e sua implementação são elementos distintos de uma classe. Assim, classes diferentes podem implementar a mesma interface. São classes polimórficas.

### Exemplo conceitual

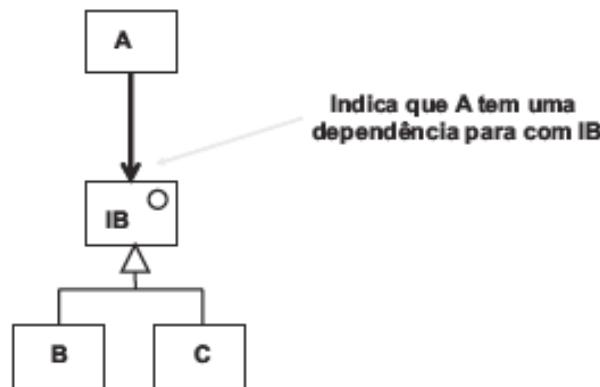


## INTRODUÇÃO À LINGUAGEM JAVA

As classes polimórficas dão uma grande flexibilidade para o desenvolvimento de aplicações onde os contratos estabelecidos entre os objetos são muito bem definidos (esses contratos são as interfaces através das quais os objetos interagem entre si). Um objeto (da classe A), construído para trabalhar com um segundo objeto (da classe B), não conhece a implementação desse segundo, mas apenas a sua interface (IB). Um outro objeto (da classe C, mas com a mesma interface de B) pode substituir esse segundo objeto, sem necessidade de alterações na classe A.

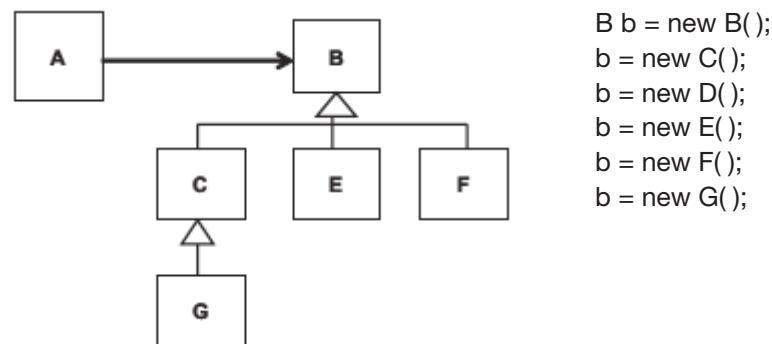
```
public interface IB { ... }
public class B implements IB { ... }
public class C implements IB { ... }
public class A {
    private IB obj;
    public void setObj(IB obj) {
        this.obj = obj;
    }
}
A a = new A();
a.setObj(new B());
a.setObj(new C());
```

### Notação UML



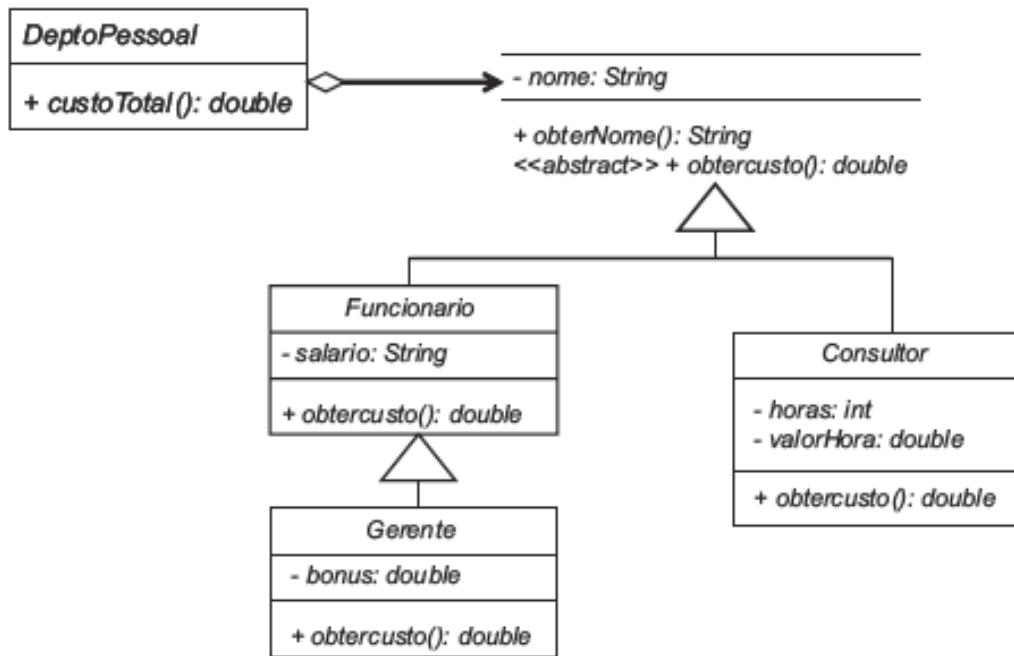
### POLIMORFISMO E HERANÇA

- Uma subclasse implementa a interface da superclasse.
- Classes numa mesma hierarquia podem ser polimórficas pela interface mais genérica.



Se a classe A é dependente de B, então qualquer instância de B, C, D, E, F ou G pode ser utilizada nesta dependência.

Exemplo completo:



Comentários e código:

- No diagrama UML estão indicados apenas os principais atributos e métodos. O código está completo.
- O estereótipo <><> indica que a classe ou o método são abstratos.
- A classe DeptoPessoal não está completa.

Colaborador:

- Classe abstrata.
- Um construtor que recebe o nome.
- Um método abstratoA, que deverá ser implementado por todas as subclasses.

```

abstract public class Colaborador {
    private String nome;
    abstract public double obterCusto();
    public String obterNome() {
        return this.nome;
    }
    public Colaborador(String nome) {
        This.nome = nome;
    }
}

```

Consultor:

- Subclasse de Colaborador.
- Classe concreta: implementa obterCusto().
- O construtor utiliza o construtor da superclasse (super).

## INTRODUÇÃO À LINGUAGEM JAVA

```
public class Consultor extends Colaborador {  
    private int horas;  
    private double valorHora;  
    public Consultor(String nome, int horas, double valorHora){  
        super(nome);  
        this.horas = horas;  
        this.valorHora = valorHora;  
    }  
    public double obterCusto() {  
        return this.valorhora * this.horas  
    }  
}
```

Funcionario:

- Subclasse de Colaborador.
- Classe concreta: implementa obterCusto().
- O construtor utiliza o construtor da superclasse (super).
- Um atributo estático não está no diagrama UML.

```
public class Funcionario extends Colaborador {  
    public double salario;  
    public static double encargo = 1.8;  
    public double obterCusto() {  
        return this.salario * encargo;  
    }  
    public Funcionario(String nome, double salario) {  
        super(nome);  
        this.salario = salario;  
    }  
}
```

Gerente:

- Subclasse de Funcionario.
- Sobrescreve obtercusto() porque sua implementação é diferente da subclasse Funcionario. Utiliza a implementação da superclasse na sobrescrita.
- Um construtor utiliza o construtor da superclasse (super).
- O outro construtor é sobre carregado e utiliza o primeiro construtor (this).
- Um atributo estático não está no diagrama UML.

```
public class Gerente extends Funcionario {  
    public double bonus;  
    static public double encargoBonus = 2.1;  
    public double obterCusto() {  
        return super.obterCusto() + this.bonus * encargoBonus;  
    }  
    public Gerente(String nome, double salario, double bonus) {  
        super(nome, salario);  
        this.bonus = bonus;  
    }  
    public Gerente(String nome, double salario) {  
        this(nome, salario, 0D);  
    }  
}
```

Departamento Pessoal:

- Depende apenas de Colaborador.
- Utiliza obtercusto() através da interface de Colaborador.
- O laço pode iterar sobre qualquer objeto da hierarquia de Colaborador, invocando a implementação de obtercusto() correspondente.
- A relação de dependência da UML (uma agregação) é implementada com um atributo do tipo array.

```
public class DeptoPessoal {  
    private Colaborador[ ] colaboradores;  
    public deptopessoal( ) {  
        super( );  
    }  
    public double custoTotal( ) {  
        double total = 0D;  
        for(int i=0; i<colaboradores.length; i++){  
            total += colaboradores[i].obtercusto( );  
        }  
        return total;  
    }  
}
```



### **EXERCÍCIO:**

1. Digitar e testar o exemplo.





## 9. Entrada e saída

- Conceito de stream.
- Classes básicas:
  - Inputstream;
  - Outputstream;
  - São “simétricas”.

Podem ser aninhadas (flexibilidade).

Stream é um conceito que representa qualquer caminho de comunicação entre uma fonte de informação e seu destino.

Este conceito pode ser empregado para qualquer fonte ou destino de informação, e torna-se útil porque padroniza essa comunicação independentemente destes. Assim, a classe stream implementa os mesmos métodos, seja o stream originário (ou destinatário), um dispositivo de I/O (entrada/saída), a memória, outra aplicação ou outro thread da mesma aplicação.

Os streams que implementam I/O são os mais empregados e o escopo maior desta seção.

Os streams de entrada e saída são “simétricos” em muitos aspectos. Assim, apenas um dos casos será estudado detalhadamente.

A possibilidade de aninhar classes de stream na linguagem Java torna seu uso muito flexível e prático.

### ENTRADA – INPUTSTREAM

- `InputStream` – métodos:
  - `read()`
  - `skip()`
  - `available()`
  - `mark()`
  - `reset()`
  - `close()`

A classe básica que implementa streams de entrada é chamada de `InputStream`. Essa classe incorpora uma série de métodos. Os principais estão descritos abaixo:

- `read()`: realiza a leitura de bytes do stream;
- `skip()`: ignora um número de bytes do stream, passado como seu parâmetro;
- `available()`: retorna o número de bytes disponíveis no stream;
- `mark()`: marca uma posição sobre o stream;
- `reset()`: retorna à posição marcada por `mark()`;
- `close()`: fecha o stream (não é obrigatório).

`InputStream` – Exemplo:

```
InputStream s=getUmStream(...);
byte[ ]    buffer= new byte[256];
byte b;

b=s.read();
s.read(buffer);
s.read(buffer, 300, 256);

s.skip(2048);

if (s.available()==2048) {
    s.mark(1024);
    ...
    s.reset();
}
s.close();
```

No exemplo, a primeira chamada a `read()` realiza a leitura de um único byte. Na segunda chamada, a variável `buffer` é preenchida com bytes do stream (desde que existam). E na terceira chamada, a leitura é realizada a partir da posição 300 do stream.

O método `mark()` exige como parâmetro o número de bytes do stream onde se estende a sua validade. Assim, se um `reset()` for chamado após 1024 posições já percorridas, um erro será gerado. Esse mecanismo força o programador a alocar recursos bem definidos para o uso do `mark()`.

## Classes herdadas de InputStream

- ByteArrayInputStream e StringBufferInputStream:
  - Cria um stream a partir de um buffer ou de uma string.
- FileInputStream:
  - Tipo mais comum de stream;
  - Pode abrir e acessar arquivos do sistema;
  - Pode tratar como stream arquivos já abertos.

Exemplo:

```
InputStream filein= new FileInputStream("/usr/fulano/arquivo");
```

As demais classes de streams de entrada são herdadas a partir de InputStream.

ByteArrayStream e StringBufferStream criam um stream a partir de buffer de bytes ou a partir de uma string, respectivamente. Portanto, agem de forma inversa ao mostrado no exemplo anterior.

FileInputStream permite a abertura e a manipulação de streams originários de arquivos do sistema. Juntamente com FileOutputStream fornecem a base de acesso a arquivos em Java.

- FilterInputStream
  - Fornece base para classes de stream que possam ser aninhadas.
- Subclasses de FilterInputStream
  - BufferedInputStream: stream com cache;
  - DataInputStream: stream de tipos distintos (e não somente de bytes, como vimos até agora);
  - PushBackInputStream: implementa o método unread( ), que coloca de volta no stream um byte já lido.

FilterInputStream é a classe básica que implementa a base para classes de stream que possam ser aninhadas. O aninhamento de classes torna bem flexível o uso dos streams, como pode ser visto nos exemplos a seguir.

Algumas classes são geradas diretamente de FilterInputStream e implementam algumas facilidades básicas no uso de streams, tais como cache e interpretação de tipos.

Um exemplo de aninhamento de stream:

```
InputStream sin= new BufferedInputStream(new FileInputStream ("arquivo"));
```

Essa declaração cria um stream originário de um arquivo do sistema e ao mesmo tempo utiliza cache. Podem ser feitos quantos aninhamentos se desejar, desde que as classes aninhadas sejam subclasses de FileInputStream.

- PipedInputStream:
  - Stream que estabelece comunicação entre dois threads.
- SequenceInputStream:
  - Executa a concatenação de streams.

Exemplo:

```
InputStream sin1 = new FileInputStream("metade1");
InputStream sin2 = new FileInputStream("metade2");
```

```
InputStream sin=new SequenceInputStream(sn1, sn2);
```

Existem outras classes não mencionadas aqui pois são de uso muito restrito. Entretanto, na literatura avançada podem ser encontradas algumas descrições.

### SAÍDA – OUTPUTSTREAMS

- Simétrico a InputStream:
  - Implementa write( ) ao invés de read( ), em todas as suas subclasses herdeiras.
  - flush().
  - close().
- PrintStream:
  - Não tem InputStream correspondente;
  - System.out.print( );
  - System.out.println( ).

Não convém repetir todo o texto para as classes OutputStream e suas subclasses. É trivial compreender que o seu comportamento é simétrico em relação ao fluxo do stream, com respeito às classes de InputStream.

Evidentemente, o método read( ) e seus equivalentes é substituído pelo método write( ) (e seus equivalentes). Além desse, existe o método flush( ) que força o stream a escrever na saída os dados que eventualmente estejam em cache ou temporariamente em algum lugar.

A classe PrintStream não encontra correspondente nos streams de entrada. Ela já foi usada neste curso, uma vez que a variável de classe System.out contém uma instância desta classe.

### READERS E WRITERS

- As classes de manipulação de streams trabalham com bytes.
- Para se trabalhar com caracteres utilizam-se os Readers e Writers, que são equivalentes respectivamente aos streams de entrada e saída.

Superclasse: Reader

- BufferedReader, FilterReader, StringReader, InputStreamReader (ponte entre streams e readers).

Superclasse: Writer

- BufferedWriter, FilterWriter, StringWriter, OutputStreamReader (ponte entre streams e writers).

Para a manipulação de arquivos, o uso de Readers e Writers é mais prático.

```
FileInputStream fis = new FileInputStream("C:\\arquivo.txt");
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader in = new BufferedReader (isr);
System.out.println(in.readLine());
```

Observar que o arquivo foi aberto usando um stream e manipulado usando um reader. Para isso foi usada a ponte entre eles.

### A classe File

- Manipula arquivos e diretórios (e não seu conteúdo).

Teste dos atributos do arquivo:

- canRead, canWrite, isHidden, isFile, isDirectory.

Manipulação física:

- getAbsolutePath, getName, createNewFile, renameTo, length.

Manipulação de diretórios:

- listFiles, mkdir.

O mesmo exemplo anterior agora utilizando a classe File.

```
File file = new File("C:\\arquivo.txt");
FileInputStream fis = new FileInputStream(file);
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader in = new BufferedReader(isr);
System.out.println(in.readLine());
```



## **EXERCÍCIOS:**

1. Faça uma aplicação que mostre na tela o conteúdo de um arquivo qualquer.
2. Faça uma aplicação que copie o conteúdo de um arquivo de caracteres para outro, substituindo todas as letras pela sua letra seguinte (a por b, b por c etc.). Faça o processo inverso também. Utilize Readers e Writers.
3. Faça uma aplicação que fique esperando strings na entrada padrão e as mostre na saída padrão. A aplicação espera a entrada de uma linha e após o ENTER, mostra esta linha.





# 10. Threads

- Possibilidade de executar simultaneamente várias threads de um único programa.
- A execução do programa não segue um padrão linear, mais derivativo e paralelo.
- Usar multithread é uma boa prática, no entanto a maioria dos servidores Web e de aplicação realizam isso implicitamente.
- Situações necessárias:
  - Loops infinitos (animação, sockets agents);
  - Longos trechos de código sem interação com terceiros;
  - Múltiplas instâncias de um mesmo trecho de programa.

Multithread é a capacidade de um programa de rodar diferentes trechos (ou diferentes instâncias do mesmo trecho) de forma independente. Permite, por exemplo, que o programa realize tarefas em background enquanto mantém “viva” a interface com o usuário.

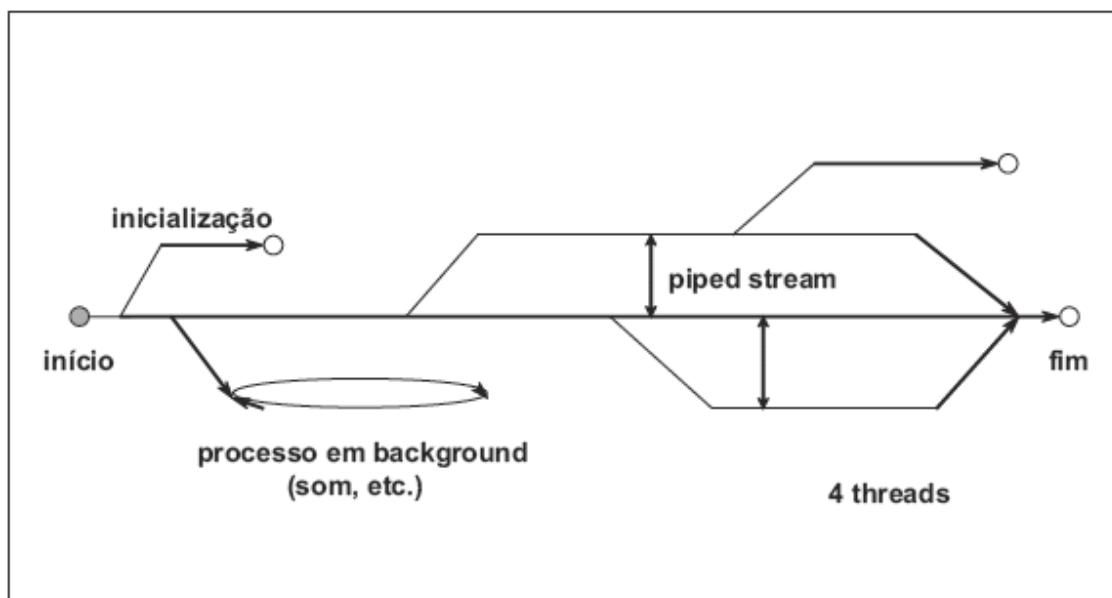
O uso de multithread em Java é fácil e implementado diretamente pela linguagem, ou seja, o controle dos diferentes threads é feito automaticamente.

Um exemplo de uso de multithread são os próprios browsers internet. Enquanto um thread realiza a busca dos dados na rede, o que muitas vezes ocorre de forma lenta, outros threads permitem que a página sendo carregada já seja exposta na tela, com rolagem e links ativos.

Num ambiente de janelas fica evidente que a programação em threads é fundamental.

### DIAGRAMA

Fluxo de um programa multithread:



O diagrama acima dá uma ideia do fluxo de execução de um programa multithread. Todas as linhas mostradas são implementadas com threads, inclusive a linha “principal”. Esta é uma boa política de programação.

Verifica-se que alguns threads podem ter curta duração (inicialização) e outros podem ficar rodando indefinidamente em loop infinito, tais como processos permanentes em background, terminando apenas quando a aplicação finaliza.

Um bom procedimento é finalizar todos os threads iniciados antes de terminar a execução do programa. Isto não é obrigatório, mas não seguir esta regra pode comprometer recursos do sistema de forma indesejável.

### Implementação

- Classe deve implementar interface Runnable.
- Classe deve possuir uma variável que contenha um tipo Thread.
- Método start( ) deve ser invocado para disparar um novo thread.
- O método run( ) deve conter o código que roda dentro do thread.

A criação de uma classe para trabalhar com multithreads é muito simples. Basicamente deve-se declarar a classe com a interface Runnable, portanto com a assinatura:

```
public class relogioMultiThread implements Runnable { .... }
```

Esta interface força a implementação um método run() que agora deve passar a conter todo o código a ser executado dentro do thread.

Uma aplicação com multithread é mostrada nas páginas seguintes. É o exemplo de relógio rodando dentro de um thread, enquanto a aplicação principal continua seu fluxo normal de execução.

## Exemplo

```

import java.util.*;
public class RelogioMultiThread implements Runnable {
    Date data;
    public void run() {
        while (true) {
            data = new Date();
            System.out.println(data);
            try { Thread.sleep(1000);
            }
            catch (InterruptedException e) { }
        }
    }
}
public class Programa {
    public static void main(String[] args) {
        RelogioMultiThread relogio = new RelogioMultiThread();
        Thread t = new Thread(relogio);
        t.start();
    }
}

```

## SÍNCRONIZAÇÃO DE THREADS

Mecanismo para evitar que um mesmo trecho de código seja executado por dois threads simultaneamente.

Manipulação de recursos únicos (arquivos, entrada e saída, variáveis) que não podem ser acessados simultaneamente.

```

public synchronized void metodo() {
    ... // acesso a recursos de forma única
}

// dentro de um método qualquer..
synchronized(this) {
    ... // acesso a recursos de forma única
}

```

A sincronização de acesso a recursos que são compartilhados entre threads é muito útil para evitar erros de acesso simultâneos, em recursos que não o suportam, ou não podem ter acessos simultâneos, como por exemplo, arquivos abertos, variáveis de controle, etc.

A sincronização pode ser feita pelo método usando o modificador `synchronized`, onde todo o corpo do método pode ser executado somente por um thread de cada vez; ou pelo bloco `synchronized (objeto) {...}` Onde o bloco é executado por um único thread de cada vez.

Quando mais de um thread tenta usar os métodos/blocos sincronizados, o interpretador enfileira-os, suspendendo aqueles que estão esperando sua vez para executar.



### EXERCÍCIOS:

1. Faça uma aplicação que dispare três threads. Cada thread deve mostrar na saída padrão, em um loop infinito, o instante que ele roda (com System.out.println( ... + new java.util.Date() )).
2. Coloque um contador em cada thread e verifique quantas vezes cada um deles executa o System.out.println.
3. Crie uma pilha de dados cujos acessos (push e pop) sejam sincronizados.



# 11. Exceções e erros

- Tratamento de situações eventuais, mas que comprometem a correta execução do programa.
- Maneira simplificada de extensivamente capturar situações de exceções e erros.
- Situações previsíveis, mas genéricas.
- A linguagem Java força o tratamento em muitos casos, nem que seja passar a exceção/erro para o sistema.

O tratamento de exceções e erros em Java é uma forma de tornar o código mais previsível e controlável. Por outro lado, é também uma forma de facilitar a programação de rotinas necessárias a esse tratamento.

Normalmente as situações de exceções e erros, como o próprio nome diz, são eventuais. No entanto seu tratamento é necessário para um bom código. Uma sistemática de tratamento destas condições pode abranger grandes porções de código, de forma complexa e extensa.

A linguagem Java provê uma maneira de sistematizar o tratamento dessas situações, para facilitar a programação, ao mesmo tempo que permite a extensão desta sistemática para grande parte do código de forma simplificada.

Em muitos métodos dos pacotes básicos de Java necessitam que o usuário defina um tratamento de erro específico. Isto pode ser tão simples ou tão complexo quanto queira o programador.

Por fim, é importante salientar que o tratamento de exceções/erros não é um código mágico que vai tornar seu programa livre de erros. As situações tratadas podem ser bem genéricas, mas devem ser previsíveis.

### MANEIRA COMUM DE TRATAR EXCEÇÕES/ERROS

- Repetitiva e extensa.
- Dificulta o código.
- Alta relação custo benefício, mas necessário:

```
int valret=metodo_que_preve_exc();
switch (valret) {
    case ...
    ...
}
```

A maneira convencional de tratar exceções e erros tende a ser cansativa, repetitiva e extensa. Exige alterações na funcionalidade propriamente dita do código. Pode também exigir o uso de variáveis globais na passagem de indicadores de erros.

Esta abordagem, além de pouco prática, passa a ser irritante quando necessita ser utilizada em grande parte do código.

Tem-se uma forma de programação de alto custo, com benefícios pouco utilizados (já que as ocorrências são raras), mas no entanto necessária.

### TRATAMENTO EM JAVA

- Exceções e erros são modelados por objetos (classe básica chama-se `Throwable`).
- Atitudes do programador:
  - Avisar usuário do método que uma exceção pode ser gerada;
  - Tratamento “in loco” da exceção;
  - Passagem da exceção adiante.

As exceções e erros são modelados por objetos herdados da classe básica `Throwable`. Este modelamento em objetos permite estabelecer uma hierarquia de erros, muito útil na definição de tratamentos específicos e/ou genéricos.

Diante de um método que pode gerar alguma condição de exceção ou erro, o programador pode tomar três atitudes:

- Não capturar, declarar a passagem da exceção/erro para uma classe específica e não implementar o seu tratamento;
- Capturar e tratar a condição “in loco”;
- Capturar e passar a condição para o nível superior de seu código.

Cada situação em particular pode exigir uma atitude específica do programador. No entanto, a alternativa de tratar a condição localmente é a mais esperada, por motivos óbvios.

### Modificador throws

Uso da palavra reservada `throws` indica que método pode gerar uma condição de exceção/erro.

Normalmente deixa o tratamento a cargo de quem utilizar o método.

Não é muito prático para projetos em equipes.

```

public void metodoQuePreveExc() throws umaExcecao {
    ...
    // chamada a métodos que podem gerar uma exceção
    // do tipo umaExcecao
    ...
}

```

A palavra reservada throws aplicada a um método indica que aquele método pode gerar uma exceção/erro do tipo definido pela cláusula throws. Esta declaração apenas diz, ao usuário daquele método, que tipo de exceções/erros pode-se esperar de sua execução. Fornece uma informação muito útil, mas não provê uma forma de tratar essa condição.

Este é o esquema básico de indicação de condição de exceções/erros em Java e que muitos métodos das bibliotecas básicas de interfaces implementam. Isso acaba forçando o usuário destas interfaces a capturar e tratar os erros passados.

### Tratamento “in loco”

Maneira mais “educada” – o autor do método especifica um tratamento para o erro que ele espera.

Uso de try{...} catch (Erro e) {...}

```

public void metodoQuePreveExc() {
    ...
    try {
        // algum método que pode gerar uma exceção
        // do tipo umaExcecao
    } catch (umaExcecao e) {
        // tratamento da exceção
    }
    ...
}

```

O tratamento local de exceções/erros exige um pouco mais de código, mas representa a forma mais clara e completa de programar esse tipo de tratamento.

Neste caso, o usuário usa a cláusula try aplicada a um bloco para especificar que espera capturar uma condição de exceção/erro naquele bloco. Ao fim do bloco try, a cláusula catch captura a exceção/erro específica (indicada entre parênteses) e determina um tratamento. Em outras palavras: o programa tenta rodar o código dentro do primeiro bloco; caso alguma condição de exceção/erro surja, tratadores específicos são chamados nos blocos seguintes.

Esta forma direta permite que condições de exceções/erros gerem apenas um desvio do código para um tratador específico. Este desvio é completamente definido e delimitado pela cláusula catch e, portanto, não desestrutura o código, como alguns poderiam supor.

Definir um tratamento passa a ser a parte complexa do código e em geral exige experiência do programador. Basicamente devemos nos preocupar com as exceções mais comuns e fornecer um tratamento curto e rápido. Como o próprio nome diz, estas são condições especiais e o bom senso diz o quanto de esforço devemos colocar no tratamento de condições que podem comprometer o resultado do nosso código.

### Passando o tratamento adiante

Capturar a exceção mas passar o seu tratamento adiante.

Utilizar com exceções básicas quando se sabe que o nível externo tem condições de tratá-la.

```
public void metodoQuePreveExc() throws MinhaExcecao{
    try {
        // algum método que pode gerar uma exceção
        // do tipo MinhaExcecao
    } catch (MinhaExcecao e) {
        throw e; // ...para o nível externo tratar...
    }
    ...
}
```

Uma outra alternativa para tratamento de exceção/erro é capturá-lo e simplesmente passá-lo adiante. Este erro é então passado para o nível imediatamente externo à classe em questão. Se a classe mais externa não tratar este erro, então este é passado novamente para a próxima classe e assim sucessivamente até chegar ao sistema.

É importante observar que as exceções lançadas com o comando `throw`, precisam ser declaradas no método com a palavra reservada `throws`.

A adoção desta atitude é mais aconselhável para erros básicos do sistema, onde sabemos que o sistema tem condições de tratá-los.

### Tratamento múltiplo

```
public void metodo()
{ try {
    // algum código que pode gerar vários tipos de exceções
} catch (NullPointerException n) {
    // ... tratamento de ponteiro nulo
} catch (RunTimeException r) {
    // ... tratamento de exceções run-time (exceto anterior)
} catch (IOException i) {
    // ... tratamento de exceções de I/O
} catch (MinhaExcecao u) {
    // ... tratamento de minha exceção
} catch (Exception e) {
    // ... tratamento exceções diferentes dos casos acima
} catch (Throwable t) {
    // ... tratamento de não-exceções, p. ex. erros
}
...
}
```

Um bloco try pode capturar tantos erros quanto se queira. A única ressalva a ser feita é quanto ao ordenamento dos tratadores nas cláusulas catch. É necessário que os erros mais específicos sejam declarados primeiro. Assim, como no exemplo acima, apenas uma condição muito particular e pouco provável iria chamar um tratamento de baixo nível de um objeto da classe básica Throwable.

### A cláusula finally

Além de múltiplos catches, uma cláusula finally pode ser acrescentada ao tratamento de exceções. O bloco finally sempre é executado, independente da captura da exceção.

```
try {
    // algum código que pode gerar exceções
} catch (Exception e) {
    // ... tratamento
} finally {
    // ... sempre é executado
}
```

## DEFININDO EXCEÇÕES

Erros e exceções podem ser definidas pelo desenvolvedor.

Qualquer subclasse de Throwable pode ser definida como um erro ou exceção.

Para a definição de exceções normalmente se utiliza como superclasse a classe Exception.

```
public class MinhaExcecao extends Exception {
    ...
}
...
public void metodo() throws MinhaExcecao {
    ...
    throw new MinhaExcecao();
}
```

O mecanismo de captura e lançamento de exceções pode ser utilizado para trabalhar com exceções definidas pelo desenvolvedor, dando uma grande flexibilidade de programação. É muito comum o uso desse mecanismo para definir desvios no fluxo de programação, com exceções associadas a regras de execução do programa e não apenas a falhas.



### EXERCÍCIOS:

1. Faça uma classe que represente uma data (dia, mês e ano). Defina exceções e lance-as, no construtor, para datas inconsistentes.
2. Faça um programa que leia uma data do teclado, e devolva para um usuário uma mensagem amigável caso a data seja inconsistente.



# 12. Tópicos avançados

## COLEÇÕES

- A linguagem Java oferece uma série de classes que estruturam objetos de uma forma mais prática do que arrays:
  - Listas.
  - Listas ligadas.
  - Hashtables.

### Características

- Sincronizadas para acessos em múltiplos threads;
- Métodos de ordenação;
- Estruturas com uso de tipos específicos (generics).

No pacote `java.util` existe uma série de classes que implementam a interface `Collection`. Todas as classes concretas que implementam essa interface são estruturas para armazenar um conjunto de objetos com as mais variadas finalidades. As principais classes desse pacote são:

- `ArrayList` e `Vector`: Arrays de tamanho ajustável aos elementos armazenados;
- `Hashtable`: Armazena pares chave-valores com métodos que facilitam a busca dos valores pelas suas chaves;
- `LinkedList`: Lista duplamente ligada;
- `Stack`: pilha.

## INTRODUÇÃO À LINGUAGEM JAVA

Observe, na documentação da API, quais dessas coleções permitem acesso sincronizado a seus elementos.

A classe Arrays possui uma série de métodos de ordenação de arrays.

A maioria das classes que manipulam coleções são parametrizadas por tipos.

```
import java.util.*;
public class Listas {
    public static void main(String[ ] args) {
        arraylist<String> arr = new arraylist<String>();
        arr.add("um");
        arr.add("dois");
        for(String s: arr){
            System.out.println(s);
        }
    }
}
```

## Iteradores

As coleções podem ser “percorridas” com iteradores, que podem ser obtidos diretamente delas:

```
Iterator it = colecao.iterator();
hasNext() //testa se existem elemento ainda
next() //retorna próximo elemento
```

Exemplo do uso de iteradores:

```
import java.util.*; public class Listas {
    public static void main(String[ ] args) {
        arraylist<String> arr = new arraylist<String>();
        arr.add("um");
        arr.add("dois");
        iterator it = arr.iterator();
        while(it.hasNext( )) {
            System.out.println(it.next());
        }
    }
}
```



## EXERCÍCIOS:

1. Faça uma classe que funcione como uma fábrica de Datas. A cada data que o programa precisar ele pede para um método estático dessa classe. Esse método verifica se existe uma Data previamente criada (e armazenada num Hashtable), retornando-a. Se a data não existir, então ela é criada e armazenada no Hashtable. Dessa forma, uma única instância de cada data existe em todo o programa.
2. Faça um método nessa classe que coloque num array, em ordem cronológica, todas as datas armazenadas por ela.
3. Liste todas as chaves do Hashtable usando um iterador.

## NETWORKING

### URLConnection

Efetua uma conexão com um servidor Web. Permite o envio de dados e recebimento de arquivos de um servidor Web (páginas HTML, imagens, etc.).

A classe URLConnection permite uma maneira fácil e simples de abrir uma conexão com qualquer servidor na internet.

```
import java.net.*;
import java.io.*;

public class Conexao {
    public static void main(String[ ] args) throws Exception{
        URL url = new URL("http", "127.0.0.1", 8080, "/index.html");
        BufferedReader reader = new BufferedReader(new
        InputStreamReader(url.openStream( )));
        String linha = null;
        while((linha = reader.readLine( )) !=null) {
            System.out.println(linha);
        }
    }
}
```



### EXERCÍCIOS:

1. Escrever um programa que lê uma página da internet cuja URL foi digitada pelo usuário na linha de comando.

## SERVERSOCKET E SOCKET

Servidor e cliente TCP/IP, para envio de pacotes TCP/IP pela rede.

O servidor fica “escutando” (listening, em inglês) uma porta TCP/IP, esperando por conexões:

```
import java.net.*;
import java.io.*;
public class Servidor {
    public static void main(String[ ] args) throws Exception{
        Serversocket servidor = new ServerSocket(10000);
        while(true) {
            Socket socket = servidor.accept( );
            OutputStream saida = socket.getOutputStream( );
            saida.write("Conectou!".getBytes());
            saida.close( ); socket.close( );
        }
    }
}
```

## INTRODUÇÃO À LINGUAGEM JAVA

Cliente se conecta na porta do servidor:

```
import java.net.*;
import java.io.*;
public class Cliente {
    public static void main(String[ ] args) throws Exception{
        Socket socket = new Socket("127.0.0.1", 10000);
        InputStream entrada = socket.getInputStream();
        byte[ ] bytes = new byte[100];
        entrada.read(bytes);
        for(int i=0; i<100; i++) {
            if(bytes[i]==0) break;
            System.out.print((char)bytes[i]);
        }
    }
}
```



### EXERCÍCIO:

1. Escrever um cliente socket que envia dados para um servidor socket.

## GENERICOS

Declaração de classes e métodos com tipos parametrizados.

Os tipos são resolvidos em tempo de compilação.

Eliminação de casting.

Exemplo que utiliza uma pilha de uma classe definida pelo desenvolvedor. Note que a pilha manipula apenas o tipo parametrizado (Dupla):

```
import java.util.*;
public class Generic {
    public static void main(String[ ] args) {
        Stack<Dupla> pilha = new Stack<Dupla>();
        pilha.add(new Dupla(1,34));
        pilha.add(new Dupla(2,67));
        for(Dupla d : pilha) {
            System.out.println(d.tostring());
        }
    }
}

class Dupla {
    Dupla(int a, int b) {
        this.a = a;
        this.b = b;
    }
    int a, b;
    public String tostring() {
        return "(" + this.a + "," + this.b + ")";
    }
}
```

Agora a classe Dupla foi parametrizada para servir como dupla de dois tipos parametrizados quaisquer. Observe que na instanciação dos objetos os tipos são passados como parâmetros.

```
public class Generic {
    public static void main(String[ ] args) {
        Stack<Dupla> pilha = new Stack<Dupla>();
        pilha.add(new Dupla<Integer, String>(1,"trinta e quatro"));
        pilha.add(new Dupla<Integer, String>(2,"sesseta e sete"));
        pilha.add(new Dupla<String, String>("tres", "quarenta"));
        for(Dupla d : pilha) {
            System.out.println(d.tostring( ));
        }
    }
}

class Dupla<E, T> {
    E e;
    T t;
    Dupla(E e, T t) {
        this.e = e;
        this.t = t;
    }
    public String toString() {
        return "(" + e.toString() + "," + t.toString() + ")";
    }
}
```

Por fim, usando o wildcard de tipos <?>.

```
public class Generic {
    public static void main(String[ ] args) {
        Stack<Dupla> pilha = new Stack<Dupla>();
        pilha.add(new Dupla<Integer, Integer>(1,34));
        pilha.add(new Dupla<Integer, String>(2,"sesseta e sete"));
        pilha.add(new Dupla<String, String>("tres", "quarenta"));
        for(Dupla d : pilha) {
            println(d);
        }
    }

    public static void println(Dupla<?,?> dupla) {
        System.out.println(dupla.toString());
    }
}
```



# Referências

SCHILD, Herbert. **Java para Iniciantes**. São Paulo: Alta Books, 2015.

SCHILD, Herbert. **Java – A Referência Completa**. São Paulo: Alta Books, 2014.

**ORACLE – DOCUMENTAÇÃO DA LIGUAGEM JAVA**. Disponível em <<http://docs.oracle.com/javase/8/>>. Acesso em 21 de mar. 201





## **COLOFÓN**

Fonte Família Helvetica Neue

Papel Offset 115 g/m

Acabamento em espiral

