

**CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA  
CELSO SUCKOW DA FONSECA**

**Apostila de Desenvolvimento de Sistemas  
Orientados a Objetos com Java**

**Professor:**  
**Rafael Guimarães Rodrigues**  
(Atualizada em 06/08/2024)

## Sumário

1.	Conteúdo Programático da Disciplina .....	4
2.	Metodologia.....	4
3.	A linguagem Java. ....	5
4.	Entendendo a JVM (Java Virtual Machine) .....	6
5.	Pontos quentes da aplicação e compilação dinâmica (JVM, Hotspot e JIT compiler).....	8
6.	O foco da linguagem Java.....	9
7.	Olá Mundo! Compilando o primeiro programa. ....	9
8.	Variáveis e comentários.....	12
9.	Convenções do Java .....	15
10.	Operadores do Java .....	15
11.	Casting e promoção .....	16
12.	IF-Else no Java .....	19
13.	While no Java .....	20
14.	Elementos de Repetição .....	20
15.	For no Java.....	21
16.	Cuidados com o Pós incremento ++.....	21
17.	Interferindo no loop .....	22
18.	Escopo das variáveis .....	23
19.	Outras coisas a saber.....	24
20.	Orientação a Objetos Básica .....	24
21.	Classes x Objetos.....	27
22.	Construindo um Sistema Orientado a Objetos .....	29
23.	Escrevendo uma classe em Java .....	30
24.	Instanciando e usando um Objeto.....	31
25.	Exercícios práticos 1 .....	32
26.	Métodos de uma Classe .....	38
27.	Objetos e Referências.....	41
28.	Classes, métodos, objetos.....	45
29.	Princípio da Alta Coesão.....	45
30.	Controlando o acesso através de modificadores .....	48
31.	O ambiente BlueJ.....	50
32.	Encapsulamento.....	50

33.	Métodos get e set .....	51
34.	Um pouco mais sobre atributos e ... Agregação.....	54
35.	Arrays em Java.....	67
36.	Arrays de referências .....	68
37.	Conhecendo e usando o foreach com arrays: .....	70
38.	Construtores e Java Bean .....	73
39.	Atributo da classe .....	75
40.	Exercícios sobre encapsulamento, construtores e static .....	77
41.	Herança .....	78
42.	Reescrita de Métodos.....	83
43.	Polimorfismo .....	86
44.	Facilidades da IDE Eclipse .....	93
45.	Classes Abstratas .....	110
46.	Interfaces e Sobrecarga .....	130
47.	Tratamento de Exceções .....	130
48.	Organizando suas Classes e Bibliotecas com Pacotes.....	148
49.	Pacote java.io .....	152
50.	Collections Framework.....	161
51.	Programação Concorrente e Threads .....	177
52.	Ferramentas: jar e javadoc .....	187
53.	O pacote Java.lang.....	191
54.	JDBC .....	203

## 1. Conteúdo Programático da Disciplina

### O que vamos aprender?

Neste curso, em um 1º momento, aprenderemos a Linguagem Java e conceitos sobre Orientação a Objetos, além de boas práticas de desenvolvimento orientado a objetos.

Em um 2º momento (disciplina seguinte) a prenderemos a desenvolver aplicações web com acesso a banco de dados, seguindo o modelo MVC, além de alguns padrões de projeto.

### Por que Java?

A escolha pelo Java se justifica pelo constante crescimento de ofertas de emprego para Desenvolvedor Java, por se tratar de uma ferramenta livre e portável, amplamente cobrada em concursos públicos na área de Tecnologia. Atualmente é a 6º linguagem mais utilizada no mundo superando até mesmo o PHP.

## 2. Metodologia

### Qual a melhor maneira de aprender?

Ao ensinar uma nova linguagem ou paradigma, alguns professores mencionam todos os detalhes juntamente com seus princípios básicos. Isso acaba deixando o aluno confuso. Ao receber tanta informação o aluno acaba não conseguindo diferenciar aquilo que é importante aprender naquele momento daquilo que será visto mais adiante quando ele tiver adquirido mais experiência para dominar o assunto.

Quando vamos tratar de um problema em classe, talvez seja necessário introduzir um conceito, como polimorfismo, por exemplo, mas não significa que **naquele momento** seja necessário dizer TUDO sobre polimorfismo. Isso só vai confundir o aluno e desviar o seu foco do problema em questão.

Existe um momento apropriado para falar sobre cada particularidade de um conceito. As informações devem ser passadas aos poucos, sempre dentro do contexto de um problema.

Outro equívoco seria despejar sobre o aluno um monte de conceitos sobre Orientação a Objetos antes de começar a programar.

A Orientação a Objetos é um paradigma novo para vocês, totalmente diferente do paradigma imperativo/procedural, que é o que lhes foi apresentado até aqui. Quebrar o paradigma procedural para entrar no mundo da Orientação a Objetos não é simples. Trata-se de uma mudança radical no modo de pensar sobre desenvolvimento de Sistemas.

Os conceitos de Orientação a Objetos serão introduzidos aos poucos, à medida em que formos aprendendo Java. Os dois assuntos caminharão juntos no decorrer do curso para proporcionar um entendimento gradativo e solidificado sobre ambos.

Algumas informações não são mostradas porque só fariam sentido para um programador experiente em Java e não para quem está começando.

Ao escrever este material (que é resultado de consulta a outros livros, apostilas e sites), procurei evitar ao máximo fazer uso de problemas matemáticos em exemplos e

exercícios. A razão é muito simples: A linguagem de programação e o paradigma orientado a objetos, por si só, já são complexos o suficiente. Não precisamos acrescentar a isso a complexidade dos problemas matemáticos.

Antes que você me pergunte: Sim! É possível aprender Java e o paradigma da Orientação a Objetos sem recorrer a exemplos matemáticos o tempo inteiro. Quero que vocês foquem apenas na linguagem e no paradigma!

**Atenção:** Sei que vocês já devem ter ouvido isso, mas é um fato: Não se aprende a programar sem praticar bastante! Todos os exercícios são extremamente importantes. É essencial estudar em casa e fazer (ou refazer) os exercícios sugeridos.

### 3. A linguagem Java.

**Quais são os maiores problemas enfrentados pelos programadores, especialmente quando se está no paradigma procedural?**

- Custo das ferramentas de desenvolvimento.
- Organização.
- Falta de bibliotecas.
- Ter que reescrever o código caso seu cliente mude de Sistema Operacional.

Alguns destes problemas foram superados há muito tempo, nas primeiras versões do Java.

O Java foi desenvolvido e mantido pela Sun Microsystems, mas essa empresa foi adquirida pela Oracle em 2009. A página principal é: [http://www.java.com/pt\\_BR/](http://www.java.com/pt_BR/)

**Acontecimentos importantes (Um breve histórico sobre a linguagem Java):**

#### →1992:

A ideia era utilizar a linguagem Java para pequenos dispositivos. Não exatamente os celulares (eles nem existiam na época), mas TVs, Vídeos-cassete, liquidificadores etc. Estes dispositivos tinham uma espécie de Sistema Operacional chamado Setup Box.

James Gosling e sua equipe pensaram em uma linguagem capaz de rodar em quaisquer destes aparelhos independentemente do fabricante. Esse conceito, que falaremos mais adiante, é denominado **portabilidade**.

O propósito era criar um interpretador (uma máquina virtual), para que um mesmo código pudesse ser interpretado por diferentes aparelhos eletrônicos e, consequentemente, fosse independente do fabricante.

#### →1994:

Houve várias tentativas de fechar contratos com grandes fabricantes como a Panasonic, por exemplo. Havia muitos conflitos de interesses e o projeto não vingou. Hoje o Java domina amplamente o mercado de aplicações para web e para pequenos dispositivos, mas em 1994 ainda era cedo para isso e **o Java não deu certo**.

#### →1998:

Com o surgimento da web, a Sun percebeu que poderia aproveitar a ideia criada em 1992 para rodar pequenas aplicações nos navegadores web que até então só exibiam

conteúdo estático. Assim, como havia diversos fabricantes de aparelhos eletrônicos, cada um com um setup box diferente, o mesmo acontecia com os navegadores e Sistemas Operacionais na web.

Dessa forma, seria uma grande vantagem poder programar códigos em uma única linguagem de forma que pudessem ser executados em diversos navegadores diferentes. O melhor de tudo é que, em vez de apenas renderizar HTML, os navegadores seriam capazes de realizar operações. Surgiu o Java 1.0 junto com os “Java Applets” que deixariam a web mais dinâmica.

→1998: Java 1.2 (JVM com Hot Spot). Veremos o que é isso adiante.

→1999: Java EE – Começa a rodar do lado do Servidor.

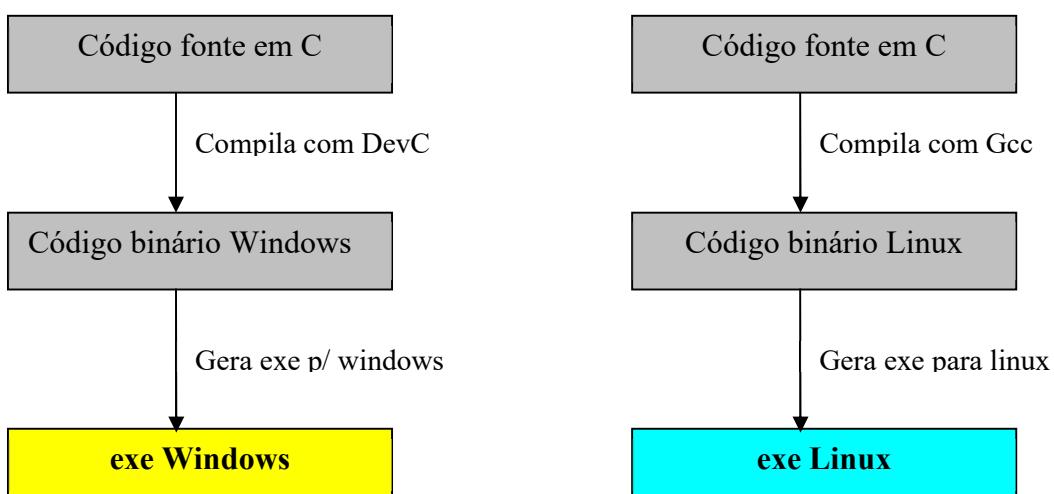
→2001: Java 1.4

→2005: Java 5 (ou Java 1.5) – Generics, Annotations. (Não se preocupe com isso agora)

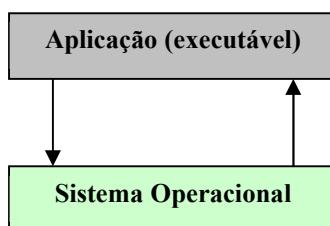
→2006/2007: Java 6. Muitas melhorias na performance da JVM. A partir deste momento a linguagem Java começa a ser amplamente utilizada e começa a dominar o mercado. **A linguagem Java ressurgiu com bastante força!**

#### 4. Entendendo a JVM (Java Virtual Machine)

**O que acontece quando vamos compilar um programa em C ou Pascal, por exemplo?**



O código fonte é compilado para uma plataforma específica, geralmente usando suas APIs. Caso o cliente mude de Sistema Operacional, é preciso reescrever o código e gerar um novo executável. Até o surgimento da linguagem Java, o modelo de comunicação entre aplicação e Sistema Operacional era o seguinte:



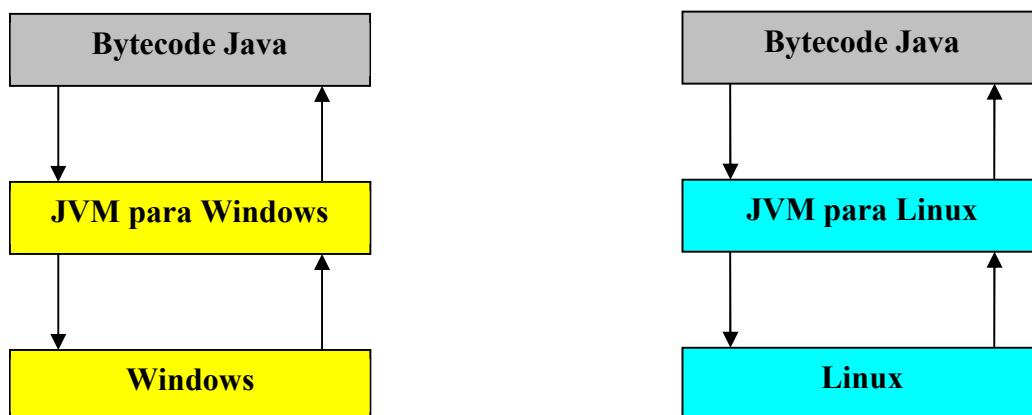
## Como acontece com o Java?

O Java utiliza uma máquina virtual que consiste em uma camada extra entre a aplicação e o Sistema Operacional. Dentre outras coisas a máquina virtual é responsável por traduzir o que sua aplicação deseja fazer para as respectivas chamadas ao Sistema Operacional.

No C ou Pascal, o programador compila para Windows, para Linux ou para outro Sistema Operacional gerando um código binário próprio de cada plataforma. Este código binário é o que vocês conhecem como “executável”.

No Java o programador compila para a máquina virtual, gerando um **bytecode** (**não um executável**) que pode ser interpretado e compilado dinamicamente por uma máquina virtual para Windows e por uma máquina virtual para Linux. Resumindo: caso queira trocar de Sistema Operacional, basta baixar a máquina virtual apropriada. Não é preciso reescrever código.

**Bytecode** é o termo dado ao código binário gerado pelo Java para a JVM. Repare na figura abaixo que sua aplicação roda sem nenhum envolvimento com o Sistema Operacional, sempre conversando somente com a Java Virtual Machine (JVM).



Veja que ganhamos independência do Sistema Operacional!

## A Máquina Virtual é apenas um interpretador?

Não. Na verdade, trata-se de um conceito muito mais amplo. Uma máquina virtual é um computador virtual: tem tudo que um computador tem. A JVM gerencia memória, pilhas de execução, threads etc. Trata-se ainda de uma camada de isolamento.

## Qual a vantagem disso?

Como tudo passa pela JVM, ela pode tirar métricas, decidir quando e onde alocar memória, entre outros. Ela pode otimizar a aplicação. Se uma JVM termina inesperadamente, somente as aplicações que estavam rodando nela caem. O Sistema Operacional continua, bem como outras JVMs que estejam rodando.

**IMPORTANTE:** A JVM não interpreta Java, interpreta bytecode. A ideia da JVM é uma especificação. Portanto, não existe apenas uma JVM (a mantida pela Oracle), existem também a Mac OS Runtime for Java, J9 (IBM), Avian, dentre outras. Todas elas devem seguir as especificações da Oracle.

## 5. Pontos quentes da aplicação e compilação dinâmica (JVM, Hotspot e JIT compiler).

Hoje em dia a JVM utiliza uma tecnologia chamada Hotspot para identificar pontos quentes (código muito executado, geralmente dentro de um ou mais loops) da aplicação. Sempre que isso ocorrer a JVM vai compilar aquele código para instruções nativas do Sistema Operacional, o que certamente vai melhorar o desempenho da sua aplicação.

Essa compilação nativa é feita através do JIT Compiler (Just In Time Compiler). O compilador que aparece “bem na hora” em que você precisa.

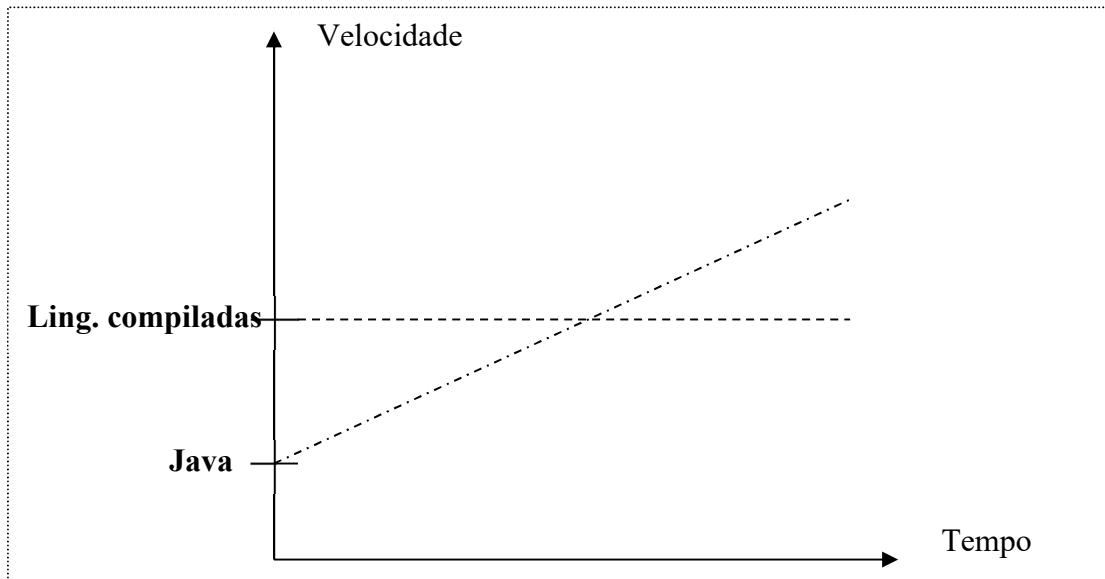
A JVM não compila tudo antes de executar a aplicação porque compilar dinamicamente pode gerar um desempenho melhor. Lembre-se que a JVM conversa com o Sistema Operacional o tempo todo e vai colhendo cada vez mais informações sobre ele.

Um .exe gerado pelo Delphi, pelo C ou pelo Visual Basic é estático. Ele já foi gerado com base em heurísticas (projeções baseadas em métricas e suposições). Isso significa que o compilador pode não ter tomado uma decisão tão boa.

Por outro lado, por estar compilando dinamicamente durante a execução do programa, a JVM pode identificar que determinado trecho de código não está com a performance adequada e resolver otimizá-lo ou mudar sua estratégia de compilação.

É por isso que as JVMs mais recentes como as do Java 6, 7 e 8, por exemplo, podem fazer sua aplicação alcançar um desempenho superior ao de códigos gerados pelo C, Delphi, Visual Basic etc. se estiver rodando durante um certo tempo.

Quanto mais tempo a JVM estiver rodando, mais ela aprende sobre o bytecode gerado e sobre o Sistema Operacional. Com isso ela otimiza a execução o tempo inteiro!



Quanto mais tempo em execução, mais rápida a JVM se torna.

## JVM, JRE, JDK. O que baixar no site do Java?

### Java SE:

- **JVM** → Máquina Virtual Java. Esse download não existe. A JVM vem junto com os pacotes abaixo:
- **JRE (Java Runtime Environment)** → JVM + Bibliotecas. Ambiente de Execução Java. Tudo o que você precisa para executar uma aplicação Java. Serve para instalar no Cliente.
- **JDK (Java Development Kit)** → JRE + Compilador + Bibliotecas. Pacote do Desenvolvedor. Faremos o download do JDK SE (Standard Edition).

### Há ainda:

- **Java EE** → Java Enterprise Edition (adicionais + componentes)
- **Java ME** → Java Micro Edition (adicionais + componentes)
- Entre outros...

## 6. O foco da linguagem Java.

No decorrer do curso você vai perceber que a linguagem que você utilizava é mais simples para criar os pequenos programas que desenvolveremos aqui. Contudo, o intuito do Java não é criar sistemas pequenos onde temos um único desenvolvedor.

O foco são as aplicações de médio e grande porte criadas por um time de desenvolvedores. Devemos pensar que esse time pode vir a sofrer substituições e crescer.

Outra coisa que o desenvolvedor deve ter em mente é que a missão do Java não é escrever um código mais rapidamente do que outras linguagens, mas sim facilitar a manutenção do software. **Esse é o diferencial!**

Um código bem escrito em Java facilita e muito a manutenção. Diferente de outras linguagens, **o Java te obriga a se organizar**.

Outro ponto importante a se destacar é a infinidade de bibliotecas gratuitas para realizar diversos trabalhos (Geração de código de barra, Gráficos, Relatórios, Sistemas de Busca, manipulação XML/JSON, tocadores de vídeo, aplicações para pequenos dispositivos, dentre outros).

Você pode criar aplicações sofisticadas sem ter que comprar nenhum componente. O Java foca em aplicações que possam crescer. Aplicações em que a legibilidade do código é importante e que a manutenção seja facilitada.

## 7. Olá Mundo! Compilando o primeiro programa.

```
public class Olamundo {  
    public static void main(String[] args) {  
        // Início do miolo do programa  
        System.out.println("Olá Mundo!");  
        // Fim do miolo do programa  
    }  
}
```

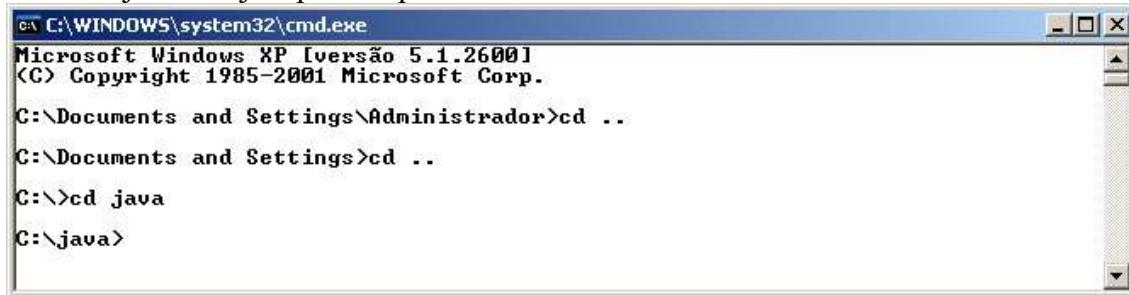
Sobre o código acima: Importante saber que todo programa em Java precisa de um ponto de entrada. Esse ponto é o método main (este será explicado mais adiante). Por

enquanto basta saber que o que vai ser executado é o que está no miolo da aplicação, dentro do método main. O código `System.out.println("Olá Mundo!");` ; faz com que o conteúdo entre aspas seja mostrado na tela.

Vamos criar um diretório padrão chamado Java em c:\ e após digitar o código acima em um bloco de notas, salve-o neste diretório como OlaMundo.java.

Para compilar o programa vamos chamar o console:  
Iniciar → Executar → cmd → ok:

Usando os comandos DOS padrão, vamos voltar diretórios e posicionar no diretório java> Veja o passo a passo abaixo:



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [versão 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrador>cd ..

C:\Documents and Settings>cd ..

C:\>cd java
C:\java>
```

**Para compilar → javac OlaMundo.java**

O comando acima vai gerar o bytecode chamado OlaMundo.class.

Em seguida levantamos a JVM e rodamos o programa → java OlaMundo

Veja a seqüência abaixo:



```
C:\WINDOWS\system32\cmd.exe
C:\java>javac OlaMundo.java
C:\java>java OlaMundo
Olá Mundo!
C:\java>
```

O javac é o compilador Java.

O java é o responsável por levantar a JVM para interpretar e executar seu programa.

**Visualizando o bytecode:**

O bytecode OlaMundo.class, gerado pelo compilador, não é legível por seres humanos. Pelo menos não por seres humanos normais. Ele está escrito em um formato que a JVM possa entender. É como um código Assembly escrito para essa máquina especificamente.

De qualquer modo, caso queiramos ler o bytecode basta digitar o comando abaixo e apreciar:

→ `javap -c OlaMundo.class`

```
C:\java>javap -c OlaMundo
Compiled from "OlaMundo.java"
public class OlaMundo extends java.lang.Object{
public OlaMundo();
  Code:
    0:  aload_0
    1:  invokespecial #1; //Method java/lang/Object."<init>":()V
    4:  return

public static void main(java.lang.String[]);
  Code:
    0:  getstatic     #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    3:  ldc           #3; //String Ola Mundo!
    5:  invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
    8:  return
}
```

Entendeu alguma coisa? Tenho certeza de que não, mas a JVM entende e é isso que importa!

Um bytecode pode ser revertido para um .java original com a perda de algumas informações como comentários e nomes de variáveis locais.

Caso seu software vá virar um produto é necessário passar um ofuscador no seu código para embaralhar classes, métodos etc. Você pode conseguir um em <http://proguard.sf.net>. Contudo, ainda é muito cedo para você se preocupar com isso.

### Exercício:

- 1) Altere seu programa para imprimir uma mensagem diferente.
- 2) Altere seu programa para imprimir duas linhas de texto usando System.out.println.
- 3) Sabendo que os caracteres \n representam uma quebra de linha, imprima duas linhas de texto usando um único System.out.println.
- 4) Sabendo que System.out.print imprime um conteúdo em tela e continua na mesma linha e que System.out.println imprime um conteúdo na tela pulando para a linha de baixo, imprima uma única linha de texto usando dois System.out.

### Possíveis erros:

Vejamos alguns erros que podem ocorrer na hora de compilar o seu código:

```
public class OlaMundo {
public static void main(String[] args) {
    System.out.println("Falta ponto e vírgula")
}
C:\java>javac OlaMundo.java
OlaMundo.java:3: ';' expected
        System.out.println("Falta ponto e vírgula")^
  1 error
C:\java>_
```

Esse erro de compilação é o mais comum. Outros erros que podem ocorrer são: esquecer de abrir e fechar {}, escrever palavras chave (out, println, etc.) em maiúscula.

### Erros de Execução:

Se você declara a classe como OlaMundo, compila e depois tenta executá-la como olaMundo (minúsculo), o Java te avisa:



```
C:\>java olaMundo
Exception in thread "main" java.lang.NoClassDefFoundError: olaMundo <wrong name: OlaMundo>
```

Outro erro de execução: se esquecer de colocar o static ou o argumento String[] args:

```
public class OlaMundo {
    public void main(String[] args) {
        System.out.println("Falta o static");
    }
}
```



```
C:\>java OlaMundo
Exception in thread "main" java.lang.NoSuchMethodError: main
C:\>java _
```

Mais um erro de execução: Não declarar main como public:

```
public class OlaMundo {
    static void main(String[] args) {
        System.out.println("Falta o public");
    }
}
```



```
C:\>javac OlaMundo.java
C:\>java OlaMundo
Main method not public.
C:\>java _
```

### Mais Exercícios

- 1) Experimente salvar o arquivo como OlaMundo.java e definir o nome da classe diferente. O que acontece?

## 8. Variáveis e comentários

Variáveis podem ser declaradas e utilizadas dentro de um bloco. Java é uma linguagem **“fortemente tipada”**. Toda variável tem um tipo que não pode ser mudado.

### Como declarar?

**tipoDaVariavel nomeDaVariavel;**

Exemplo:

Podemos ter uma variável `idadeDoAluno` que vale um número inteiro:

```
int idadeDoAluno;
```

A partir deste momento a variável passa a existir e você pode atribuir valores a ela.

### Comentários de código:

No Java podemos comentar uma linha ou um bloco, conforme mostramos abaixo:

```
// Comentário de uma linha  
/* Comentário  
de  
bloco */
```

```
// idadeDoAluno vai passar a valer 20  
idadeDoAluno=20;
```

Este valor pode ser modificado também. Veja este exemplo de código:

```
// Declara a variável e atribuindo valor  
int idadeDoAluno = 24;  
  
// idadeDoAluno vai passar a valer 25  
idadeDoAluno=25;  
  
// Imprime a idadeDoAluno  
System.out.println(idadeDoAluno);  
//out referencia um objeto do tipo PrintStream da classe System que é uma public final  
//class! Não pode ser estendida. Caso não tenha entendido essa linha, não se preocupe, você  
//entenderá até o final do curso.  
  
//Declara outra variável chamada idadeDoAlunoNoAnoQueVem  
int idadeDoAlunoNoAnoQueVem;  
  
//Faz contas com variáveis  
idadeDoAlunoNoAnoQueVem=idadeDoAluno+1;
```

Também podemos usar os operadores de adição(+), subtração(-), divisão(/) e multiplicação(\*). Outro operador que pode ser utilizado é o módulo(%) que retorna o resto de uma divisão inteira.

Veja exemplos:  
`int quatro = 2+2;`  
`int três = 4-1;`  
`int dez = 5*2;`  
`int cinco = 10/2;`  
`int um = 5%2; // 5 dividido por 2 dá 2 e tem resto 1;`  
`// Lembrando que o operador % pega o resto de uma divisão inteira.`

## Como testar os códigos?

Estes trechos de código devem ser colocados dentro do método main, que vimos anteriormente. Ou seja, devem ficar no miolo do programa. Veja um programa completo abaixo:

```
class TestaIdade {  
    public static void main(String[] args) {  
        // declara e atribui valor a idade  
        int idade = 18;  
        // imprime a idade  
        System.out.println(idade);  
        // gera uma idade no ano seguinte  
        int idadeNoAnoQueVem;  
        idadeNoAnoQueVem = idade + 1;  
        // imprime a idade no ano que vem  
        System.out.println(idadeNoAnoQueVem);  
    }  
}
```

Escreva o programa acima e faça o “chinês” dele.

Outro tipo de variável muito utilizada é o double, que armazena um número com ponto flutuante. O tipo double também pode armazenar um inteiro.

```
double salário = 589.15;  
double x = 3 * 5;
```

Existe também o tipo boolean que pode armazenar apenas dois valores distintos: true ou false;

```
boolean verdadeiro=true;
```

Um boolean também pode ser determinado através de uma expressão boleana (um trecho de código que retorna verdadeiro ou falso como resposta). Exemplo:

```
int idade = 30;  
boolean menorDeIdade = idade < 18;
```

O tipo char representa apenas um caractere e deve estar entre aspas simples.  
**IMPORTANTE:** uma variável char não pode guardar um código como ‘ ’ por exemplo. Vazio não é um caractere!

Exemplo:

```
char letra = 'b';  
System.out.println(letra);
```

Não vemos variáveis do tipo char sendo usadas com frequência. Mais adiante veremos que o uso da variável String é mais frequente. Contudo, ao contrário das variáveis vistas até o momento, String não é um tipo primitivo.

## Tipos primitivos do Java:

As variáveis vistas até então são do tipo primitivo. O valor que elas guardam são o real conteúdo da variável. Quando você utilizar o **operador de atribuição** = o valor será **copiado**.

```
int a = 10; //a recebe uma cópia do valor 10  
int b = a; //b recebe uma cópia do valor de a  
a = 1 + a; //a vira 11.
```

### O que acontece com b??

b continua a valer 10.

Apesar de na 2ª linha haver a atribuição b = a, essas variáveis não têm qualquer relação uma com a outra. O que acontece com uma não reflete na outra. Há apenas a cópia do valor de uma variável para a outra.

Vimos até aqui os tipos primitivos mais usados (int, double, boolean, char). O Java tem outros como o byte, o short, o long e o float. No total são 8 tipos primitivos.

Cada tipo possui características próprias que só vão fazer muita diferença para um programador avançado.

## 9. Convenções do Java

**Nome de classe** → Iniciando com maiúscula. Ex: Pessoa.java, Mundo.java, Bicicleta.java.

**Nome de variável** → Iniciando com minúscula. Ex: idade, sexo, nome, altura.

### Outras convenções:

**Nome composto** → a partir do 2º nome inicia com maiúscula, tanto para classe quanto para variável. Ex:

**Classes** → OlaMundo.java, MinhaNovaBicicleta.java, PessoaFisica.java.

**Variáveis** → idadeDaPessoa, sexoDoAluno, nomeDoFuncionario, nomeDePessoaFisica.

As convenções acima são chamadas **CamelCase** (A provável origem do termo se deve às corcovas do camelo)

## 10. Operadores do Java

+ - / \* → adição, subtração, divisão e multiplicação.

% → módulo (resto de uma divisão inteira).

== → igual.

!= diferente.

> → maior.

>= → maior ou igual.

< → menor.

$\leq$  → menor ou igual.  
! → negação.  
 $++$  → incremento (o mesmo que  $+=$ )  
 $--$  → decremento (o mesmo que  $-=$ )

## 11. Casting e promoção

Nem todas as atribuições são bem-sucedidas. Veja abaixo:

```
double x = 3.14;  
int i = x; //Não compila;
```

O mesmo ocorre no trecho abaixo:

```
int i = 3.14; //Não compila;
```

Até aqui fica fácil entender o porquê. Afinal, um inteiro não pode receber um valor double.

O interessante é que o código abaixo também não compila:

```
double d = 5; //Compila sem problemas: um double pode receber um valor inteiro  
int i = d; //Não compila!!!
```

Apesar de d ter recebido um valor inteiro, para o Java ele continua sendo um double e um double não pode ser atribuído a um inteiro.

### Lembra que Java é fortemente tipada?

O que acontece é que **o compilador Java analisa o tipo da variável** e não seu valor. Para o Java, se d foi declarado como um double, não importa que valor ele receba, vai ser sempre um double. O compilador não sabe qual é o valor contido em d. Então, avalia pelo seu tipo.

O contrário pode acontecer sem problemas.

```
int i = 5;  
double d = i; //Um double pode receber um int.
```

### Ok, mas qual é a solução para os problemas anteriores?

É comum que em determinado ponto do código precisemos que um número quebrado seja arredondado para um inteiro. Nesses casos é preciso determinar que este número seja moldado(casted) para um inteiro. Esse processo recebe o nome de **casting**.

Veja a seguir:

```
double d = 3.14;  
int i = (int) d; //Estamos moldando d para um inteiro. Agora i vai valer 3.
```

O mesmo ocorre para valores int e long. Veja:

```
long x = 10000;  
int i = x; //Não compila, pois pode estar perdendo informação.
```

O jeito é fazer um casting:

```
long x = 10000;  
int i = (int) x;
```

### Outros casos menos comuns de casting e atribuição:

**float** x = 0.0; // Não compila, pois todos os literais com ponto flutuante são considerados double pelo Java. Um float não poderia receber um valor double sem perda de informação.

**float** x = 0.0f; // Compila! A letra f indica que aquele literal deve ser tratado como um float.

Outro caso mais comum:

```
double d = 5;  
float f = 3;
```

**float** x = f + (float) d; // Aqui você precisa fazer o casting porque o Java sempre faz as contas armazenado no maior tipo encontrado na operação, que no caso seria um double.

Até casting de variável do tipo char pode ocorrer.

Exemplo:

```
char letra = 'a';  
int letraInteiro = letra;  
// + é o operador de concatenação  
System.out.println("Letra e: "+letra); // Vai imprimir a  
System.out.println("LetraInteiro e: "+letraInteiro); //Vai imprimir 95 (valor inteiro de a na tabela ASCII)
```

O único tipo primitivo que não pode ser atribuído a nenhum valor é o boolean.

### Tipos que suportam casting, por ordem de tamanho:

#### Literais:

byte → 1 byte;  
short → 2 bytes;  
char → 2 bytes;  
int → 4 bytes;  
long → 8 bytes;

#### Precisão:

float → 4 bytes;  
double → 8 bytes; // Precisão dupla.

boolean → NÃO TEM CASTING!

**Todos os literais acima são tipos primitivos do Java!**  
**Existem dois tipos de casting:**

- **Casting explícito** → de maior para menor. Exemplo: double para int. Pode ser feito com literal ou variável. Exemplo: a ( double ) para b ( int ).
- **Casting implícito** → de menor para maior. Também conhecido como *casting de promoção*. Exemplo: de int para double.

Veja o trecho de código abaixo:

```
class Exemplo {  
    public static void main(String[] args) {  
        double d = 5.4;  
        int i = (int) d; // casting de variável. i passa a valer 5  
  
        float pi = 3.14; Não compila  
        float pi = 3.14f; // Casting de literal. Compila!  
        char letra = 65;  
        System.out.println(letra); // vai imprimir a letra A, cujo  
        // código na // tabela asc é  
        // 65  
        int num = (int) letra;  
        System.out.println(num); // vai imprimir 65  
    }  
}
```

**Exercício:** Escreva o código acima. Em seguida faça testes e modificações para que ele rode.

**Castings possíveis:**

PARA →	Byte	Short	Char	Int	Long	Float	double
DE :	-	Impl.	(char)	Impl.	Impl.	Impl.	Impl.
Byte	-	Impl.	(char)	Impl.	Impl.	Impl.	Impl.
Short	(byte)	-	(char)	Impl.	Impl.	Impl.	Impl.
Char	(byte)	(short)	-	Impl.	Impl.	Impl.	Impl.
Int	(byte)	(short)	(char)	-	Impl.	Impl.	Impl.
Long	(byte)	(short)	(char)	(int)	-	Impl.	Impl.
Float	(byte)	(short)	(char)	(int)	(long)	-	Impl.
double	(byte)	(short)	(char)	(int)	(long)	(float)	-

Impl. = **Casting implícito** (casting de promoção).

**Exercícios:**

- 1) Uma Empresa possui tabelas informando quanto faturou em cada mês. Para saber a venda do primeiro trimestre precisamos saber o valor total vendido. Sabemos que em janeiro a empresa faturou 20000 reais, em fevereiro 35000 reais e em março 16000 reais.

Sendo assim, faça um programa que calcule e imprima o faturamento total do primeiro trimestre, seguindo os passos abaixo:

- a. Crie uma classe chamada TestaFaturamentoTrimestral com um bloco main, como fizemos anteriormente;
- b. Dentro do main (miolo do programa), declare uma variável inteira chamada faturamentoJaneiro e inicialize-a com o valor 20000.
- c. Faça o mesmo para faturamentoFevereiro e faturamentoMarco, cada um com seu respectivo valor;
- d. Crie uma variável chamada faturamentoTrimestral e inicialize-a com a soma das outras 3 variáveis;
- e. Imprima a variável faturamentoTrimestral;
- f. Crie uma variável chamada faturamentoMedioTrimestral que deverá receber a média de vendas do trimestre. Perceba que essa variável não poderá ser do tipo int. Imprima o valor dessa nova variável.

## 12.IF-Else no Java

A sintaxe é a seguinte:

```
if (condiçãoBoleana) {  
    //instruções;  
}
```

A condição boleana deve retornar **true** ou **false**. Para isso, o programador pode usar operadores como: <, >, <=, >=, ==, !=, entre outros. Veja um exemplo:

```
int idade = 16;  
if (idade < 18) {  
    System.out.println("menor de idade");  
} else {  
    System.out.println("maior de idade");  
}
```

Também é possível concatenar expressões boleanas através dos operadores E (&) e OU ()|.

Veja:

```
class OlaMundo {  
    public static void main(String[] args) {  
        int idade = 16;  
        boolean amigoDoDono=true;  
        if (idade < 18 & ! amigoDoDono) {  
            System.out.println("Entrada proibida");  
        } else {  
            System.out.println("Entrada autorizada");  
        }  
    }  
}
```

Obs:

**! amigoDoDono** é o mesmo que:

**NOT amigoDoDono== true** é o mesmo que:

**amigoDoDono== false**.

Lembrando que **!** é um operador de negação.

Também podemos utilizar o **if / else if**.

Ex:

```
class OlaMundo2 {  
    public static void main(String[] args) {  
        int idade = 16;  
        boolean amigoDoDono=true;  
        if (idade < 18) {  
            System.out.println("Entrada proibida");  
        } else if (idade < 65){  
            System.out.println("Entrada autorizada");  
        } else {  
            System.out.println("Entrada gratuita autorizada");  
        }  
    }  
}
```

## 13. While no Java

O While é um laço de repetição utilizado para que um trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int diaDoMes = 1;  
while (diaDoMes < 31) {  
    System.out.println("Hoje é dia "+diaDoMes);  
    diaDoMes = diaDoMes+1; //Incrementa o contador  
}
```

No momento em que a condição for falsa (diaDoMes==31), o trecho não será mais executado.

## 14. Elementos de Repetição

Todo laço de repetição possui elementos essenciais para o seu funcionamento. São eles:

- **CONTADOR** (ou variável de controle) → Ex: int i=0;
- **CONDIÇÃO** → Condição para que se continue ou não a executar o trecho de código. Ex: (i<10)
- **INCREMENTO/DECREMENTO** → Ex: i++; ou i=i+1; // No caso do decremento i--; ou i=i-1;

```
int i = 0; //CONTADOR  
while (i < 10) { //CONDICÃO  
    System.out.println(i);  
    i++; //INCREMENTO  
}  
// Imprime de 0 a 9
```

Outro exemplo:

```
int i = 9; //CONTADOR  
while (i > 0) { //CONDICÃO  
    System.out.println(i);  
    i--; //DECREMENTO  
}
```

```
// Imprime de 9 a 1
```

## 15. For no Java

O For usa a mesma idéia do While, só que reserva espaço numa única linha para determinar: CONTADOR, CONDIÇÃO e INCREMENTO/DECREMENTO.

À primeira vista o código fica mais legível e o número de repetições mais evidente.

```
for(contador; condição; incremento) {  
    trecho de código;  
}
```

Na prática:

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

Veja que o for acima poderia ser trocado por:

```
int i = 0; //CONTADOR  
while (i < 10) { //CONDIÇÃO  
    System.out.println(i);  
    i++; //INCREMENTO  
}
```

### Exercícios para fixar a sintaxe:

Lembrando que o aprendizado de determinada linguagem depende não somente da compreensão em classe, mas também da repetição dos exercícios a fim de exercitar a lógica e fixar a sintaxe. **Programação é lógica e prática, muita prática!**

Seguem alguns exercícios. Cada um deve ser salvo com a extensão .java e com um método main dentro dele. Veja:

```
Class ExercicioX {  
    Public static void main(String[] args) {  
        // Seu exercício vai ser feito aqui!  
    }  
}
```

**Aproveite para praticar!** Digite os códigos, sem copiar e colar, para:

- 1) Imprimir todos os números de 100 a 200.
- 2) Imprima a soma dos números de 1 até 500.
- 3) Imprima todos os múltiplos de 5, de 1 a 3000.

## 16. Cuidados com o Pós incremento ++

Quando não houver outra variável ou atribuição envolvida, `i = i+1` pode ser substituído por `i++`. No entanto quando essa instrução estiver envolvida em uma atribuição, devemos tomar cuidado com algumas situações. Veja:

```
int i = 5;  
int x = i++;
```

Quanto vale `i`? E `x`? Quanto vale?

### O que acontece?

### Como devo fazer?

## 17. Interferindo no loop

Embora tenhamos condições booleanas para que um laço termine. Em alguns casos podemos querer parar o loop antes do esperado, sem que o resto do laço seja executado.

Veja o código abaixo:

```
for(int i=0;i<20;i++){  
    if (i==17){  
        System.out.println("Encontrei o numero 17 e quero sair do loop");  
        break; //interromp o loop abruptamente  
    }  
}
```

No caso acima o contador só chega até 17 e o laço é interrompido.

Também podemos obrigar o loop a executar o próximo passo. Para isso usamos a palavra-chave **continue** em vez de **break**.

Veja:

```
for(int i=0;i<20;i++){  
    if (i>8 && i<15 ){  
        continue;//passa para o laço seguinte e não imprime i  
    }  
    System.out.println(i);  
}
```

Neste caso os números impressos seriam: 0,1,2,3,4,5,6,7,8,15,16,17,18,19.  
Os números 9,10,11,12,13 e 14 não seriam impressos.

### Exercícios:

- 1) Usando os conhecimentos adquiridos acima. Escreva um programa para imprimir todos os números pares de 1 a 200.

2) Agora imprima todos os números ímpares de 1 a 200.

## 18.Escopo das variáveis

As variáveis em Java podem ser declaradas a qualquer momento. No entanto, dependendo de onde ela foi declarada, só vai valer de um ponto a outro do seu código.

//Aqui a variável x não existe.

int x = 10;

//Aqui x já existe.

O **escopo da variável** é o nome dado ao trecho de código onde ela existe e pode ser acessada.

Quando abrimos um novo bloco com as chaves e declaramos uma variável ali dentro, significa que ela só vale até o fim daquele bloco.

//Aqui a variável x não existe.

int x = 10;

//Aqui x já existe.

While (condição){

//O x ainda vale aqui. O y ainda não existe.

int y = 5;

//O y passa a existir.

}

// O y não existe mais. O x continua valendo.

Também é preciso tomar cuidado dentro de blocos if-else, for, etc.

Veja:

```
if (condição){  
    int x = 10;  
}  
else{  
    int x = 15;  
}
```

System.out.println(x); // CUIDADO! x não existe fora do IF-else nesse caso.

**Veja agora:**

```
int x;  
if (condição){  
    x = 10;  
}  
else{  
    x = 15;  
}  
System.out.println(x); // Agora funciona!
```

Usando o for:

```
for(int x=0;x<10;x++){
    System.out.println("Testando...");}
System.out.println(x); // CUIDADO! Neste caso o x não existe fora do for.
```

**Veja agora:**

```
int x;
for(x=0;x<10;x++){
    System.out.println("Testando...");}
System.out.println(x); // Agora funciona!
```

Quando existe um bloco dentro do outro, os blocos de dentro enxergam as variáveis do bloco de fora. O contrário não acontece.

## 19. Outras coisas a saber...

Até agora só vimos os comandos mais utilizados para controle de fluxo. Caso ainda não tenham aprendido em disciplinas anteriores, pesquisar sobre o do..while e sobre o switch.

Responda as perguntas abaixo:

- O que acontece se tentarmos dividir um inteiro por 0? E por 0.0?
- O que as instruções abaixo fazem?
  - `x += 4;`
  - `x -= 1;`

## 20. Orientação a Objetos Básica

### Paradigma procedural x paradigma OO.

A programação Orientada a Objetos é uma forma diferente de programar, mais organizada.

O que você deve saber é muito mais trabalhoso desenvolver um programa orientado a objetos, seja ele de qual porte for. No entanto você desenvolverá um programa mais organizado, legível e sem muitos dos problemas da programação procedural.

A manutenção de um programa orientado a objetos é infinitamente mais simples do que a de um programa procedural, ou seja, perde-se mais tempo no desenvolvimento para que a manutenção seja facilitada!

### Problema clássico: validação de CPF.

Normalmente recebemos o cpf através de um formulário submetemos os caracteres a uma função que irá validá-lo, conforme pseudocódigo abaixo:

```
cpf = formulário → campo_cpf  
valida(cpf)
```

### Quais são as implicações?

#### **Alguém te obriga a validar esse cpf?**

Nada impede que você se esqueça de chamar esse validador por inúmeras vezes, certo? E se você estiver desenvolvendo em equipe, com 5 desenvolvedores por exemplo? Pior: imagine que seu sistema possui mais de 50 telas onde é preciso validar o cpf. Se são 5 programadores trabalhando nesses formulários, quem é o responsável pela validação? Resposta: Todos! Isso é péssimo!

**O que já é ruim pode sempre piorar:** Caso entre um novo desenvolvedor na equipe, devemos avisá-lo de que sempre será necessário validar o cpf de um formulário.

Mas será que em um programa procedural a única coisa que um novo desenvolvedor precisa saber é sobre validar um cpf?

Um programa procedural é cheio de regras de negócio “soltas” que os desenvolvedores devem saber. Aí é que surgem aqueles guias de programação que os novos integrantes da equipe devem ler para saber sobre o projeto. Geralmente trata-se de um documento enorme.

Resumindo: Todo desenvolvedor é obrigado a **assimilar uma quantidade enorme de informações** que geralmente não estão sequer relacionadas à sua parte no projeto. Mas ele **precisa ler tudo isso** e não se esquecer de nada! Complicado, não?

Outro problema da programação procedural fica evidenciado quando precisamos ler o código que foi escrito por outro desenvolvedor e descobrir como ele funciona. Um sistema bem-organizado não deveria gerar essa necessidade.

Em um grande sistema, simplesmente não temos tempo para ler uma parte grande do código!

Sendo muito otimista e considerando que você e sua equipe tenham uma excelente comunicação e que passariam bem por estes problemas, vamos imaginar outro problema: Imagine que a empresa que encomendou o sistema determinou uma nova regra: agora também é preciso validar a idade do cliente. Ele precisa ser maior de 18 anos. Vamos ter que colocar um IF, mas onde? Por todo o seu código!

Mas, como você é inteligente, vai criar uma função para validar a idade!

Agora em vez de escrever um IF em cada uma das 50 telas, basta chamar a função que valida a idade em cada uma delas. Acha pouco? Você vai ter que escrever a chamada desta função em cada uma das 50 telas! E qual é a chance de você esquecer de colocar isso em alguma destas telas? **É enorme!**

Não quero ser chato, mas a equipe de desenvolvedores pode sofrer modificações e o sistema pode aumentar de 50 para 60 telas onde se trata idade e cpf de cliente. Nesse caso surgem algumas perguntas:

- Será que os novos desenvolvedores vão lembrar-se de chamar os métodos?
- Será que vão ler o guia com atenção?

- Será que vão perceber que já existe uma função validacpf? ou..
- Será que vão criar uma outra função e chamá-la nas 10 novas telas? Agora teríamos 2 funções diferentes para validar cpf.
- E se a regra de validação mudasse?

Perceba que **tanto o cpf, quanto a idade são atributos do seu cliente**. A **responsabilidade de validar uma idade ou um cpf ficou espalhada por todo o seu código**. Todas as telas.

Isso acontece porque **na programação procedural, dados e métodos andam separados**. Não há coesão.

Não seria bom se pudéssemos concentrar essa responsabilidade em um único lugar para não haver chance de esquecer-se disso?

Mesmo que a regra de validação mudasse, teríamos que mudar em um único lugar e melhor: os outros programadores nem precisariam saber disso!

Um único programador seria responsável pela validação e os outros apenas escreveriam os formulários sem se preocupar com as regras de idade ou cpf. Sem terem que se preocupar em chamar métodos! **Mundo ideal?** Não. **Isso pode ser feito!** É o paradigma OO facilitando a sua vida!

Não seria ótimo se na declaração de uma variável do tipo cpf pudéssemos determinar que ela automaticamente passaria por um método de validação? Dessa forma todo cpf preenchido seria validado! Os desenvolvedores não precisariam se preocupar com isso! No paradigma OO é fácil fazer esse tipo de amarra através da linguagem.

Tudo isso pode ser feito ao definir classes. Por enquanto vamos dizer que **cpf e idade são atributos de um cliente** e que **teríamos um arquivo com esse nome onde declararíamos as variáveis cpf e idade, bem como os métodos validaCpf e validaIdade**.

Quando a regra mudar, precisaremos modificar parte de um único arquivo e melhor: não teremos dificuldade em identificar qual é o arquivo a modificar!  
Obs: classes são parecidas com os structs em C e registros em Pascal.

### Resultado:

- Agora você não vai correr o risco de haver mais de uma função validaCpf;
- Agora você não vai precisar lembrar-se de chamar essa função em cada uma das 50 ou 60 telas;
- Se a regra mudar, você não vai precisar percorrer o sistema inteiro fazendo alterações em cada uma das telas. Você vai alterar a regra em um único lugar!
- Você vai saber exatamente onde procurar a regra e a modificação vai repercutir por toda a sua aplicação sem causar impacto!

Percebeu quantas vantagens adquirimos falando apenas de uma simples validação de cpf?

### Vantagens do paradigma OO:

- Código organizado;
- Responsabilidades concentradas no ponto certo;

- Maior flexibilidade do código;
- Legibilidade;
- Reutilização de código ( vamos ver adiante);

## 21.Classes x Objetos.

No tópico anterior pudemos perceber que podemos concentrar responsabilidades em um único ponto e promover coesão entre dados e métodos para validá-los. Também dissemos que para concentrar tudo isso em um único lugar, precisaremos escrever classes. Mas o que são classes?

Classes são especificações para a construção de um objeto. Objeto é a coisa concreta.

Fazendo uma analogia:

**A planta de uma casa seria uma classe e a casa seria um objeto.**

A **planta** é a **especificação** de como construir uma casa.

A receita de um bolo seria a classe e o bolo (o caso concreto) seria o objeto.

Analise alguns objetos do mundo real, bem como suas características e seu comportamento.

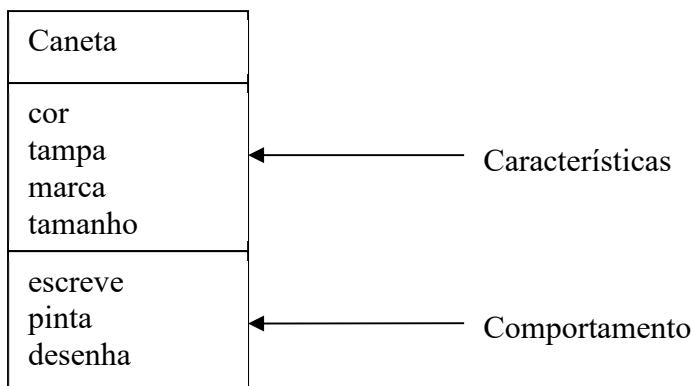
Uma caneta possui que características?

O que uma caneta pode fazer? Como ela se comporta?

**Para ser caneta, basta ter as mesmas características de uma caneta?**

Para ser caneta, não basta ter as mesmas características. Tem que ter o mesmo comportamento!

Vamos definir a especificação de uma caneta, através do desenho UML de uma classe Caneta:



Para ser uma caneta tem que ter a especificação de uma caneta: (características + comportamento).

**CUIDADO:** A **especificação** de uma caneta **não contém** dados (valores) para seus atributos (Ex: sabemos que uma caneta tem cor, mas não sabemos qual é a cor).  
A **especificação** de uma caneta não se comporta como uma caneta.

### Especificação != Objeto

**Eu posso comer a receita de um bolo?** Não! Preciso construir um bolo através de uma receita para depois comê-lo.

Eu posso escrever com a especificação de uma caneta? NÃO! Primeiro preciso construir uma caneta através de uma especificação para depois escrever com ela.

Pode parecer simplório, mas a **dificuldade inicial do paradigma Orientado a Objetos é justamente saber distinguir o que é classe e o que é objeto!**

**É comum o programador iniciante utilizar erradamente essas duas palavras como sinônimos.**

A especificação de uma caneta diz quais são suas características (atributos) e como ela deve se comportar.

### Especificações dão origem a coisas concretas:

Canetas são produzidas a partir de uma especificação!

Então, vamos produzir canetas:

Caneta1	Caneta2	Caneta3
Cor: azul Tamanho: 10 Marca: Pilot Tampa: sim	Cor: azul Tamanho: 10 Marca: Pilot Tampa: sim	Cor: preta Tamanho: 10 Marca: Pilot Tampa: sim

Agora vamos observar as canetas criadas:

- As 3 possuem as mesmas características (por isso são canetas).

Podemos dizer que as canetas 1 e 2 são iguais? Não! Apenas possuem valores iguais para seus atributos.

Todos os objetos de uma mesma classe possuem uma série de atributos e comportamentos em comum, mas não são iguais. Podem variar nos valores destes atributos e no modo como realizam estes comportamentos.

### Simplificando:

Especificação = Classe.

Nome da Especificação = Caneta.

Características = atributos.

Comportamento = métodos.

Logo: Objeto é a coisa concreta, construído a partir de uma classe!

Na representação acima temos 3 objetos: Caneta1, Caneta2 e Caneta3.

Objetos possuem valores para seus atributos e se comportam de determinada maneira.

### **Exercícios com o Professor:**

Escreva a classe Caneta e uma classe de teste para instanciar (criar) ao menos 2 objetos da classe Caneta, atribuir valores a seus atributos e simular seu comportamento.

Divida seu caderno em 3 partes (Classe, Programa e memória) e faça as simulações.

## 22. Construindo um Sistema Orientado a Objetos

Antes de tudo precisamos identificar o domínio do problema (coisas relevantes).

Se desejarmos construir um sistema para uma simulação de tráfego, obviamente você vai precisar modelar uma entidade chamada carro. Mas... o que seria um carro em nosso contexto? Uma classe ou um objeto? Algumas perguntas podem ajudá-lo a perceber a diferença:

- De que cor é o carro?
- Ele é muito velho?
- Onde ele se encontra nesse exato momento?
- Ele está limpo?
- Qual é o modelo?

Você já deve ter chegado à conclusão óbvia de que, para responder a estas perguntas, precisaremos falar de um carro em específico. O motivo é que a palavra carro, nesse contexto, refere-se à classe carro. Nesse caso, estamos falando de carros em geral e não sobre um carro em particular.

Se eu digo “meu carro vermelho e sujo que está estacionado ali fora”, poderemos responder quase todas as perguntas acima:

- O carro é vermelho;
- Não está limpo;
- Está estacionado no pátio do CEFET;

Se eu digo “meu Stepway 2012” em vez de “meu carro..”, automaticamente estaria passando as demais informações:

- O modelo é stepway;
- Ele não é muito velho;

### **Continuando a identificar o domínio do problema:**

Se formos construir um sistema para uma biblioteca, por exemplo, não será muito difícil identificar algumas entidades envolvidas, como por exemplo: livro, tomador (aquele que faz o empréstimo), autor, editora, usuário (aquele que opera o sistema), área de conhecimento (Matemática, Geografia, Informática etc.) etc.

Vamos analisar o domínio de um banco:

### **O que é relevante e tem papel fundamental?**

Não é difícil chegar à conclusão de que uma conta é uma entidade extremamente importante para um banco. Afinal, tudo gira em torno de uma conta!

Vamos analisar uma conta:

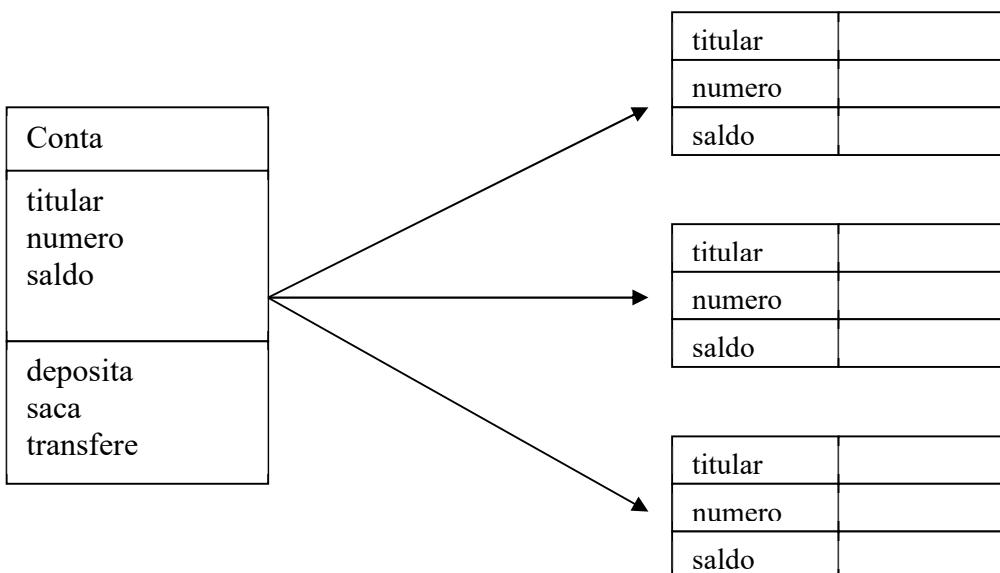
O que toda conta tem e é importante para nós?

- Número
- Nome do Titular
- Saldo

O que se pode fazer com uma conta e que é importante para nós? Ou seja, o que gostaríamos de “pedir para uma conta”?

- Sacar um valor x
- Depositar um valor x
- Mostrar o nome do Titular da conta
- Mostrar o saldo atual
- Transferir um valor x para uma outra conta y

Com isso temos o projeto de uma conta. Podemos pegar esse projeto e acessar seu saldo? Não! O que temos é só a especificação de uma conta. Precisamos criar contas para acessar o que elas têm e pedir que façam algo.



Observe a figura: Apesar de declararmos à esquerda que toda Conta tem um saldo, um número e um titular, nas instâncias de uma Conta (à direita) é que há espaço para armazenar estes valores. À esquerda temos a classe e à direita os objetos (que são instâncias desta classe) que foram criados a partir da classe. E se eles foram criados, provavelmente estão na memória do computador.

## 23. Escrevendo uma classe em Java

Escrever uma classe em Java consiste em traduzir o desenho UML para um trecho de código em Java.

**IMPORTANTE:** Escrever um arquivo para cada classe é uma boa prática de programação!

Traduzindo a classe Conta parcialmente...

```
public class Conta {  
    int numero;  
    String titular;  
    double saldo;  
}
```

Repare que as variáveis foram declaradas dentro do escopo da classe. Quando isso acontece chamamos de variáveis de objeto ou **atributos**.

Por hora declaramos todos os atributos que uma conta deve ter.

## 24. Instanciando e usando um Objeto

Já criamos, com o auxílio da linguagem Java, uma classe para manipular uma conta. A classe foi definida e salva como Conta.java. Mas como fazemos para usá-la? Além da classe temos que criar o Programa.java. A partir dele é que usaremos a classe conta.

Para instanciar (construir, criar) uma Conta, precisamos usar a palavra chave new. Utilizamos os parênteses também (veremos, mais adiante, para que servem).

```
public class TestaConta {  
    public static void main(String[] args) {  
        new Conta();  
    }  
}
```

O código acima cria uma Conta, mas como vamos acessar o objeto que foi criado? Precisamos ter uma forma de referenciar esse objeto. Precisamos de uma variável.

Veja:

```
2 public class TestaConta {  
3     public static void main(String[] args) {  
4         Conta umaConta;  
5         umaConta = new Conta();  
6         //acesso aos atributos  
7         umaConta.numero = 1;  
8         umaConta.titular = "Fulano";  
9         umaConta.saldo = 1000.0;  
10        //Acesso aos atributos para imprimir no console  
11        System.out.println("A conta "+umaConta.numero+" de "+umaConta.titular+" tem saldo "+umaConta.saldo);  
12    }  
13 }
```

É importante frisar que o ponto foi utilizado para acessar algo em umaConta. No caso foi usado para acessar titular e depois saldo.

Agora umaConta tem número 1, pertence a Fulano e tem saldo 1000.0;

Vamos fazer uma analogia entre o Java e uma Construtora:

- Uma Construtora constrói casas, em determinado terreno, a partir de uma planta.
- Java constrói objetos, na memória, a partir de uma classe.

Java → Construtora.

new → Constrói (cria, instancia) o objeto.

Classe → Planta (especificação) para se construir um objeto.

Memória → É o “terreno” onde se constrói objetos.

Não se pode acessar o titular da classe Conta. Podemos acessar o titular de um objeto (caso concreto) da classe Conta. No exemplo acima, umaConta representa um caso concreto de Conta. Veja como acontece:

```
umaConta = new Conta();
```

umaConta agora passa a apontar para um endereço de memória onde foi criado um objeto do tipo Conta.

```
umaConta.titular = "Fulano";
```

umaConta → Me posicionei no endereço apontado por umaConta.

. → entrei no objeto referenciado por umaConta.

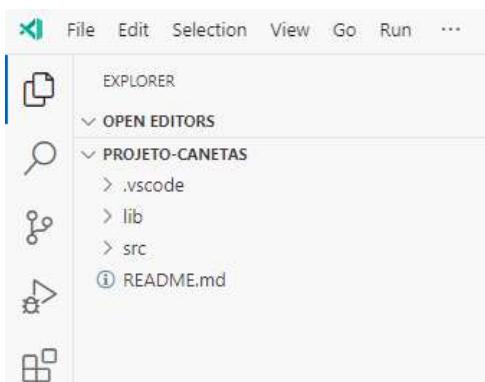
Titular = "Fulano"; → dentro do objeto accesei e modifiquei o atributo titular;

É importante frisar que no momento em que se cria umaConta, seus atributos são inicializados com valores default. Ex:

```
titular = null;  
saldo = 0.0;  
numero = 0;
```

## 25.Exercícios práticos 1

- 1) Abra o Vscode e crie um projeto Java (CTRL+SHIFT+P → Create: Java Project  
→ No build tools → em seguida escolha seu workspace (c:/dev/workjava2024)) chamado projeto-canetas.



- 2) Com o botão direito do mouse sobre a pasta src, crie a classe Caneta (new → Java File → Class ...).

The screenshot shows two Java files in a code editor:

**Caneta.java**

```
src > J Caneta.java > Caneta
1 public class Caneta {
2     String cor;
3     String marca;
4     int tamanho;
5     String formato;
6     boolean tampa;
7 }
```

**TestaCanetas.java**

```
src > J TestaCanetas.java > TestaCanetas > main(String[])
1 public class TestaCanetas {
2     Run | Debug
3     public static void main(String[] args) {
4         //#12345
5         Caneta umaCaneta = new Caneta();
6         //#12345.cor = "azul"
7         umaCaneta.cor = "azul";
8         umaCaneta.tamanho = 12;
9         umaCaneta.marca = "Bic";
10
11        //##4321
12        Caneta outraCaneta = new Caneta();
13        outraCaneta.cor = "preta";
14        outraCaneta.tamanho = 12;
15        outraCaneta.marca = "Bic";
16        outraCaneta.tampa = true;
17
18        System.out.println("umaCaneta tem cor "+umaCaneta.cor+" e é da marca "+umaCaneta.marca+".");
19        System.out.println("outraCaneta tem cor "+outraCaneta.cor+" e é da marca "+outraCaneta.marca+".");
20    }
}
```

The output console shows the printed statements:

```
umaCaneta tem cor azul e é da marca Bic.
outraCaneta tem cor preta e é da marca Bic.
```

- 3) Crie uma classe de testes com um método main para testar a criação e manipulação de canetas (objetos do tipo Caneta) em memória. Esta classe deverá se chamar TestaCanetas.
- 4) Para verificar a saída no console, clique com o botão direito do mouse sobre a classe TestaCanetas e escolha Run code.

- 5) Por fim, escreva outra classe de testes chamada TestaCanetas2 e verifique se canetas preenchidas com valores iguais para seus atributos são, de fato, canetas iguais.

```
src > J TestaCanetas2.java > TestaCanetas2 > main(String[])
1  public class TestaCanetas2 {
2      Run | Debug
3      public static void main(String[] args) {
4          Caneta umaCaneta = new Caneta();
5          umaCaneta.cor = "azul";
6          umaCaneta.tamanho = 12;
7          umaCaneta.marca = "Bic";
8
9          Caneta outraCaneta = new Caneta();
10         outraCaneta.cor = "azul";
11         outraCaneta.tamanho = 12;
12         outraCaneta.marca = "Bic";
13
14         if(umaCaneta == outraCaneta)
15             System.out.println("Iguais");
16         else
17             System.out.println("Diferentes");
18
19         System.out.println(umaCaneta);
20         System.out.println(outraCaneta);
21     }
```

- 6) Abra o Vscode e crie um projeto Java (CTRL+SHIFT+P → Create: Java Project  
→ No build tools → em seguida escolha seu workspace (c:/dev/workjava2024))  
chamado projeto-banco01. Em seguida crie a classe Conta.

```
src > J Conta.java
1  public class Conta {
2      String titular;
3      int numero;
4      double saldo;
5  }
```

- 7) Com o botão direito do mouse sobre a pasta src, crie a classe TestaContas e rode  
a aplicação (Run code).

```
src > J TestaContas.java > 📁 TestaContas > ⚒ main(String[])
1  public class TestaContas {
2      Run | Debug
3      public static void main(String[] args) {
4          Conta conta1 = new Conta();
5          conta1.numero = 123;
6          conta1.titular = "Fulano";
7          conta1.saldo = 3000;
8
9          Conta conta2 = new Conta();
10         conta2.numero = 456;
11         conta2.titular = "Beltrano";
12         conta2.saldo = 4000;
13
14         Conta conta3 = new Conta();
15         conta3.numero = 789;
16         conta3.titular = "Ciclano";
17         conta3.saldo = 2500;
18
19         System.out.println("Conta 1:");
20         System.out.println("Número: "+conta1.numero);
21         System.out.println("Titular: "+conta1.titular);
22         System.out.println("Saldo: R$"+conta1.saldo);
23         System.out.println("Conta 2:");
24         System.out.println("Número: "+conta2.numero);
25         System.out.println("Titular: "+conta2.titular);
26         System.out.println("Saldo: R$"+conta2.saldo);
27         System.out.println("Conta 3:");
28         System.out.println("Número: "+conta3.numero);
29         System.out.println("Titular: "+conta3.titular);
30         System.out.println("Saldo: R$"+conta3.saldo);
31     }
```

- 8) Crie uma nova classe de testes TestaContas2 com especial atenção às linhas 12 e 13. Em seguida, run code. Veja o resultado.

```
src > J TestaContas2.java > TestaContas2 > main(String[])
1  public class TestaContas2 {
2      Run | Debug
3      public static void main(String[] args) {
4          Conta conta1 = new Conta();
5          conta1.numero = 123;
6          conta1.titular = "Fulano";
7          conta1.saldo = 3000;
8
9          Conta conta2 = new Conta();
10         conta2.numero = 456;
11         conta2.titular = "Beltrano";
12         conta2.saldo = 4000;
13
14         Conta conta3 = conta2;
15         conta3.titular = "Jhon Arias";
16
17         System.out.println("Conta 1:");
18         System.out.println("Número: "+conta1.numero);
19         System.out.println("Titular: "+conta1.titular);
20         System.out.println("Saldo: R$"+conta1.saldo);
21         System.out.println("Conta 2:");
22         System.out.println("Número: "+conta2.numero);
23         System.out.println("Titular: "+conta2.titular);
24         System.out.println("Saldo: R$"+conta2.saldo);
25         System.out.println("Conta 3:");
26         System.out.println("Número: "+conta3.numero);
27         System.out.println("Titular: "+conta3.titular);
28         System.out.println("Saldo: R$"+conta3.saldo);
29     }
}
```

- 9) Ainda na classe TestaContas2, adicione a linha 15 abaixo e escolha Run code.

```
src > J TestaContas2.java > 🏃 TestaContas2 > main(String[])
1  public class TestaContas2 {
2      Run | Debug
3      public static void main(String[] args) {
4          Conta conta1 = new Conta();
5          conta1.numero = 123;
6          conta1.titular = "Fulano";
7          conta1.saldo = 3000;
8
9          Conta conta2 = new Conta();
10         conta2.numero = 456;
11         conta2.titular = "Beltrano";
12         conta2.saldo = 4000;
13
14         Conta conta3 = conta2;
15         conta3.titular = "Jhon Arias";
16         conta2.saldo = 9000;
17
18         System.out.println("Conta 1:");
19         System.out.println("Número: "+conta1.numero);
20         System.out.println("Titular: "+conta1.titular);
21         System.out.println("Saldo: R$"+conta1.saldo);
22         System.out.println("Conta 2:");
23         System.out.println("Número: "+conta2.numero);
24         System.out.println("Titular: "+conta2.titular);
25         System.out.println("Saldo: R$"+conta2.saldo);
26         System.out.println("Conta 3:");
27         System.out.println("Número: "+conta3.numero);
28         System.out.println("Titular: "+conta3.titular);
29         System.out.println("Saldo: R$"+conta3.saldo);
30     }
}
```

10) Altere a classe Conta para inicializar sempre com um saldo de 500,00.

```
src > J Conta.java > ...
1  public class Conta {
2      String titular;
3      int numero;
4      double saldo = 500;
5  }
```

11) Crie a classe TestaContas3 e veja o resultado.

```
src > J TestaContas3.java > TestaContas3 > main(String[])
1  public class TestaContas3 {
2      Run | Debug
3      public static void main(String[] args) {
4          Conta conta1 = new Conta();
5          conta1.numero = 123;
6          conta1.titular = "Fulano";
7          conta1.saldo = 3000;
8
9          Conta conta2 = new Conta();
10         conta2.numero = 456;
11         conta2.titular = "Beltrano";
12         conta2.saldo = 4000;
13
14         Conta conta3 = new Conta();
15
16         System.out.println("Conta 1:");
17         System.out.println("Número: "+conta1.numero);
18         System.out.println("Titular: "+conta1.titular);
19         System.out.println("Saldo: R$"+conta1.saldo);
20         System.out.println("Conta 2:");
21         System.out.println("Número: "+conta2.numero);
22         System.out.println("Titular: "+conta2.titular);
23         System.out.println("Saldo: R$"+conta2.saldo);
24         System.out.println("Conta 3:");
25         System.out.println("Número: "+conta3.numero);
26         System.out.println("Titular: "+conta3.titular);
27         System.out.println("Saldo: R$"+conta3.saldo);
28     }
```

## 26. Métodos de uma Classe

Também devemos declarar, dentro da classe Conta, o que cada conta faz e como faz. O **comportamento** de uma classe Conta.

**De que maneira uma conta saca dinheiro?** Isso será declarado através de um método dentro da própria classe Conta e não totalmente desatrelado das informações da própria conta como acontece na programação procedural. É por isso que essas funções são chamadas de métodos ( ou comportamento ). Pois definem a maneira de fazer uma operação com um objeto.

Repare que:

- **Atributos** → descrevem as características de um objeto de determinada classe.
- **Métodos** → descrevem o comportamento de um objeto de determinada classe.

Precisamos criar um método que saca uma determinada quantia e não devolve nenhuma informação para quem acionar esse método:

```
2 public class Conta {  
3     //Atributos  
4     int numero;  
5     String titular;  
6     double saldo;  
7     //Métodos  
8     void saca(double valor) {  
9         double novoValor = this.saldo - valor;  
10        this.saldo = novoValor;  
11    }  
12 }
```

A palavra void diz que quando você pedir para sacar um valor, nenhuma informação será enviada de volta a quem acionou esse método.

Ao chamar o método saca devemos informar o valor a ser sacado. Para isso usamos o argumento (ou parâmetro) valor. Valor é uma variável comum que também pode ser chamada de temporária ou local já que ao final desse método ela vai deixar de existir. Lembre-se do escopo das variáveis!

Também declaramos uma nova variável dentro do método. Essa variável, assim como o argumento valor, vai deixar de existir no fim do método, pois esse é o seu escopo.

Usamos a palavra this antes de saldo para mostrar que este é um atributo da classe e não uma simples variável. A palavra this faz referência ao endereço de memória onde está instanciado o objeto que está sendo acessado no momento. Mais adiante veremos que em determinadas situações isso é até opcional.

Perceba que, por enquanto, nada impede que o saldo seja estourado. Mais adiante trataremos essa situação de forma muito elegante.

Também devemos ter um método para depositar uma quantia:

```
13     void deposita(double valor) {  
14         this.saldo += valor;  
15         //O mesmo que this.saldo = this.saldo + valor;  
16     }
```

Repare que desta vez não usamos uma variável auxiliar.

O código a seguir saca uma quantia e em seguida deposita outra quantia. Veja:

```
2 public class TestaConta {  
3     public static void main(String[] args) {  
4         Conta umaConta; //Declara variável de referência do tipo Conta  
5         umaConta = new Conta();  
6         //acesso aos atributos  
7         umaConta.numero = 1;  
8         umaConta.titular = "Fulano";  
9         umaConta.saldo = 1000.0;  
10  
11         //Invocando métodos  
12         umaConta.saca(300);  
13         umaConta.deposita(200);  
14         //Acesso aos atributos para imprimir no console  
15         System.out.println("A conta "+umaConta.numero+ " de "+umaConta.titular+ " tem saldo "+umaConta.saldo);  
16     }  
17 }
```

Para enviar uma mensagem a um objeto e pedir que ele execute um método é preciso usar o ponto. O nome disso é **invocação de método**. Sempre que chamamos um método de determinado objeto é comum darmos a esta operação o nome de “troca de mensagens”. Sempre que ouvir este termo, lembre-se que significa somente que um método (ou serviço) de determinado objeto está sendo invocado.

Se o saldo inicial é 1000.0 , saquei 300.0 e depositei 200.0, ao imprimirmos o saldo, o que será impresso?

## Métodos com retorno

Um método sempre tem que retornar algo, nem que seja vazio, como fizemos ao usar void. Um método pode retornar um valor para o código que o chamou. No caso do método saca, podemos devolver um booleano informando se a operação foi bem-sucedida ou não.

```
boolean saca(double valor) {
    if (this.saldo < valor) {
        return false;
    } else {
        this.saldo -= valor;
        return true;
    }
}
```

A palavra chave return indica que o método vai terminar ali, retornando agora true ou false. Repare que agora fizemos um tratamento impedindo que seja sacado um valor maior que o saldo.

Veja um exemplo de como usar isso no programa:

```
2 public class TestaConta {
3     public static void main(String[] args) {
4         Conta umaConta; //Declara variável de referência do tipo Conta
5         umaConta = new Conta();
6         //acesso aos atributos
7         umaConta.numero = 1;
8         umaConta.titular = "Fulano";
9         umaConta.saldo = 1000.0;
10
11        //Invocando métodos
12        boolean consegui = umaConta.saca(300);
13        if(consegui)
14            System.out.println("Saque efetuado com sucesso!");
15        else
16            System.out.println("Saldo insuficiente para saque");
17        umaConta.deposita(200);
18        //Acesso aos atributos para imprimir no console
19        System.out.println("A conta "+umaConta.numero+ " de "+umaConta.titular+ " tem saldo "+umaConta.saldo);
20    }
21 }
```

Mais adiante veremos que às vezes é mais interessante lançar uma exceção nesses casos. Falaremos de tratamento de exceções mais adiante. Não se preocupe com isso agora!

Veja que no meu método saca me limitei a informar se a operação foi bem-sucedida ou não.

Em vez de retornar um booleano eu poderia retornar uma mensagem String informando se consegui sacar e porque não consegui sacar, por exemplo. Mas, um método deve ser o mais simples possível para que possa ser reutilizado por vários programas. E tem mais: retornando uma String seria fácil eu identificar se consegui sacar? Uma resposta verdadeiro ou falso é mais fácil de tratar e se eu quiser alguma mensagem a mais, coloco isso no programa e não na classe.

Isso é o que chamamos SRP (Princípio da Responsabilidade Única). O método deve sacar e apenas informar se conseguiu ou não, mais do que isso é inventar e dar ao método uma responsabilidade extra que não é dele. Escreva métodos simples e objetivos!

Os métodos saca e deposita ainda não estavam 100%. Veja:

```
boolean saca(double valor) {
    if(this.saldo < valor)
        return false;
    else {
        this.saldo -= valor;
        return true;
    }
}

boolean deposita(double valor) {
    if(valor <= 0)
        return false;
    else {
        this.saldo += valor;
        //O mesmo que this.saldo = this.saldo + valor;
        return true;
    }
}
```

Não podemos correr o risco de receber um depósito negativo, certo? Da mesma forma, o valor do saque deve ser menor ou igual ao saldo.

Importante salientar que um programa pode manter duas ou mais contas na memória ao mesmo tempo. Veja:

```
public class Programa {
    public static void main(String[] args) {
        Conta umaConta, outraConta; // declarando 2 variáveis do tipo Conta.
        umaConta = new Conta();
        outraConta = new Conta();

        umaConta.titular = "Rafael";
        umaConta.deposita(500);

        outraConta.titular = "Paulo";
        outraConta.deposita(2000);
        outraConta.saca(300);

        System.out.println("Saldo atual de " + umaConta.titular + " :"
            + umaConta.saldo);
        System.out.println("Saldo atual de " + outraConta.titular + " :"
            + outraConta.saldo);
    }
}
```

## 27. Objetos e Referências

Ao declarar uma variável para associar a um objeto, na verdade esta variável não guarda o objeto, mas sim a referência para este objeto, ou seja, uma maneira de acessá-lo.

Lembrem-se: Objetos vivem na memória e precisam ser referenciados! É por isso que precisamos usar a palavra new para criar um novo espaço na memória para um objeto.

O que você deve saber é que a variável `umaConta` por exemplo **não é um objeto** do tipo `Conta`. A variável `umaConta` **referencia** um objeto do tipo `Conta`. Logo, `umaConta` é uma variável de referência.

**Uma variável nunca é um objeto. É uma referência para um objeto!**

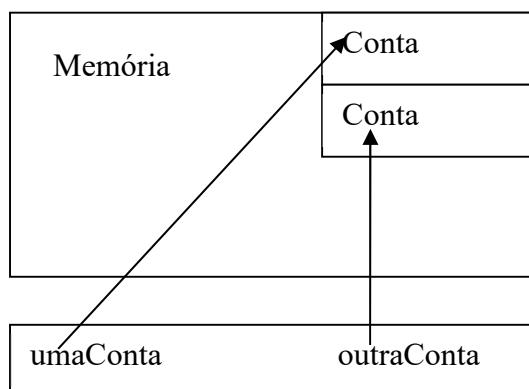
```
public static void main(String[] args) {  
    Conta umaConta; // declarando uma variável de referência  
    para um objeto do tipo Conta.  
    umaConta = new Conta();  
}
```

É comum os programadores dizerem que tem um objeto `umaConta` do tipo `Conta`. Isso é só para encurtar a frase. O correto é dizer **que temos uma referência `umaConta` para um objeto** do tipo `Conta`.

Também é correto dizer que `umaConta` **se refere** a um objeto do tipo `Conta`.

Veja:

```
public static void main(String[] args) {  
    Conta umaConta;  
    Conta outraConta;  
    umaConta = new Conta();  
    outraConta = new Conta();  
}
```



Internamente `umaConta` e `outraConta` vão guardar um endereço que identifica em que posição de memória aquele objeto do tipo `Conta` se encontra.

É assim que ao utilizar o ‘.’ Para navegar, acessamos o objeto do tipo `Conta` que se encontra naquela posição da memória.

É parecido com ponteiro, no entanto não pode ser usado para guardar outras coisas.

Vejamos mais um exemplo interessante:

```
public class TestaReferencia {  
    public static void main(String[] args) {  
        Conta c1;  
        Conta c2;  
  
        c1 = new Conta();  
        c2 = c1; // Linha importante!  
  
        c2.deposita(500);  
  
        System.out.println(c1.saldo);  
        System.out.println(c2.saldo);  
    }  
}
```

O que vai ser impresso na tela? O que aconteceu?

### O que faz o new?

O new faz muitas coisas, mas, por enquanto, basta saber que ele aloca espaço em memória para um objeto e retorna a referência para esse objeto. Ou seja, o endereço de memória onde o objeto está alocado. Quando você atribui esse retorno a uma variável, ela passa a se referir ao mesmo objeto.

Agora analise o código abaixo:

```
public class TestaReferencias {  
    public static void main(String[] args) {  
        Conta c1;  
        Conta c2;  
  
        c1 = new Conta();  
        c2 = new Conta();  
  
        c1.titular = "João";  
        c1.deposita(200);  
  
        c2.titular = "João";  
        c2.deposita(200);  
  
        if (c1 == c2) {  
            System.out.println("Iguais");  
        } else {  
            System.out.println("Diferentes");  
        }  
    }  
}
```

2) E agora? Qual será a resposta para o código acima? O que acontece?

**Lembre-se que** == compara o conteúdo de 2 variáveis nesse caso!

Para saber se dois objetos possuem os mesmos valores para seus atributos, o que devemos fazer?

**Um pouco mais sobre métodos...**

Já definimos dois métodos para nossa classe Conta: saca e deposita. Agora seria interessante ter um método que transfere dinheiro entre duas contas.

Já pensou em como programar esse método?

Pensou em criar um método que receba dois argumentos: conta1 e conta2 do tipo Conta?

**Cuidado!** Você está pensando de maneira procedural ainda!

Quando chamarmos o método transfere, já estaremos chamando-o através de um objeto do tipo Conta (o this). Portanto, o método precisa receber apenas um outro objeto do tipo Conta, que seria contaDestino, mais um argumento que seria o valor a transferir. A contaOrigem nesse caso é o próprio objeto que invocou o método (this).

Veja como ficaria:

```
public void transfere(Conta contaDestino, double valor){  
    //Para facilitar a compreensão por enquanto  
    Conta contaOrigem = this;  
    contaOrigem.saldo-=valor;  
    contaDestino.saldo+=valor;  
}
```

Para deixar o código mais completo e robusto, podemos verificar se a conta possui o valor a ser transferido. Também podemos chamar os métodos deposita e saca para ficar mais interessante. Veja:

```
public boolean transfere(Conta contaDestino, double valor) {  
    // Para facilitar a compreensão por enquanto  
    Conta contaOrigem = this;  
    boolean sacou = contaOrigem.saca(valor);  
    if (sacou) {  
        contaDestino.deposita(valor);  
        return true;  
    } else {  
        // Não foi possível sacar  
        return false;  
    }  
}
```

Perceba que temos dois tipos de passagem de argumento aqui: Uma passagem de argumento por referência e uma passagem de argumento por valor.

No Java quando fazemos atribuição através do “=” estamos copiando um valor para uma variável. A passagem de parâmetro (argumento) funciona da mesma forma. Quando passamos, por exemplo, 200 na posição de valor, significa que o valor 200 foi copiado para a variável valor. Qualquer modificação sofrida por valor dentro do corpo do método transfere, não se reflete fora do método.

No caso de um parâmetro do tipo Conta, estamos copiando uma referência (endereço de memória) para o argumento(variável) contaDestino. Logo, ela vai referenciar o mesmo objeto e, consequentemente, as mudanças ocorridas com contaDestino irão refletir fora do método. Não houve cópia de objetos aqui. Houve cópia de referências.

Em vez de chamarmos esse método de transfere, poderíamos chama-lo de transferePara, isso facilitaria a leitura e compreensão. Veja:

```
conta1.transferePara(conta2,200);
```

A leitura seria a seguinte:

**conta1 transfere para conta2 200 reais. Não ficou melhor?**

**Lembre-se:** Legibilidade é importantíssimo na programação orientada a objetos. Não economize no tamanho dos nomes de classes, métodos ou atributos. O mais importante é escrever códigos legíveis, com nomes que façam sentido logo de cara!

Veja como ficaria essa nova versão:

```
public boolean transferePara(Conta contaDestino, double valor) {  
    boolean sacou = this.saca(valor);  
    if (sacou) {  
        contaDestino.deposita(valor);  
        return true;  
    } else {  
        // Não foi possível sacar  
        return false;  
    }  
}
```

## 28.Classes, métodos, objetos...

No paradigma procedural, métodos e atributos vivem separados, espalhados pelo código. No paradigma OO, métodos e atributos vivem juntos, em uma mesma classe, onde sua relação é evidente.

O que você ainda precisa saber é que:

- Objetos vivem na memória.
- Métodos vivem na Classe, não no objeto.
- Métodos modificam o estado de atributos.

### Exercício:

Divida sua folha de caderno em 3 partes novamente: classe, programa, memória.

Utilizando o último código que fizemos, mostre com setas o caminho percorrido entre programa, classe e memória a cada mudança de estado dos atributos.

## 29.Princípio da Alta Coesão

Até aqui criamos uma classe Conta que possui atributos (titular, numero, saldo) e que possui métodos que modificam o estado de atributos. Para ser mais específico, estamos modificando apenas o estado do atributo saldo, por enquanto. O atributo mais importante! Os métodos definidos foram: saca e deposita.

Perceba que nossos atributos são características de uma Conta de verdade e os métodos refletem operações comuns a contas bancárias.

Outra observação é que todos os nossos métodos alteram o estado de um atributo da classe Conta. Faria sentido algum desses atributos ou métodos estarem em outra classe que não Conta? Claro que não. **Esse é o princípio da alta coesão.** Concentramos na classe Conta, tudo que uma Conta de verdade deve ter e fazer. Métodos e atributos

atrelados e concentrados num lugar específico e que faz sentido. Bem diferente do paradigma procedural.

Outra coisa importante a observar é que cada classe deve ter sua responsabilidade bem definida, assim como seus métodos. O método saca, por exemplo, não poderia estar em outra classe porque a operação sacar é de responsabilidade de uma conta.

**Atribuir a uma classe responsabilidades que não são dela compromete o SRP (Princípio da responsabilidade única) e, consequentemente, a reutilização.**

O mesmo pode ser dito em relação aos métodos. **Um método deve ter uma responsabilidade bem definida e simples.** O método saca deve apenas sacar e informar se a operação foi efetuada ou não.

Outros tratamentos, como mensagens a exibir para o usuário, geralmente ficam a cargo do programa, já que cada programa pode precisar trabalhar a informação de um modo particular.

### Simplificar os métodos é uma boa prática de programação!

Estatísticas sobre as rotinas de um Sistema:

- **45%** das rotinas são usadas **frequentemente**.
- **25%** das rotinas são usadas **ocasionalmente**.
- **30%** das rotinas nunca são **usadas**.

## 30.Exercícios práticos 2

- 1) Abra o Eclipse e crie um projeto chamado projeto-banco02. Em seguida crie a classe Conta.

```
2 public class Conta {  
3     // Atributos  
4     String titular;  
5     double saldo = 500;  
6     int numero;  
7     //Comportamento  
8     public boolean saca(double valor) {  
9         if (valor > this.saldo) {  
10             return false;  
11         } else {  
12             //Se eu invocar saca com conta1, this-->xpto  
13             //Se eu invocar saca com conta2, this-->abc  
14             this.saldo -= valor;  
15             return true;  
16         }  
17     }  
18  
19     public boolean deposita(double valor) {  
20         if (valor<=0)  
21             return false;  
22         else {  
23             //this.saldo += valor; // Faz a mesma coisa que a linha abaixo  
24             this.saldo = this.saldo + valor;  
25             return true;  
26         }  
27     }  
}
```

```
28 public boolean transferePara(Conta contaDestino, double valor) {  
29     //conta1 --> #abc -->this -->#abc  
30     if(this.saca(valor)==true) {  
31         boolean conseguiuDepositar = contaDestino.deposita(valor);  
32         return conseguiuDepositar; // O retorno será true, o mesmo do método deposita  
33     }  
34     return false;  
35 }  
36 }  
37 }
```

- 2) Com o botão direito do mouse sobre a pasta src, crie a classe TestaMetodos e rode a aplicação.

```
2 public class TestaMetodos {  
3     public static void main(String[] args) {  
4         Conta conta1 = new Conta();  
5         conta1.titular = "Fulano";  
6         conta1.numero = 123;conta1.saldo = 3000;  
7         Conta conta2 = new Conta();  
8         conta2.titular = "Beltrano";  
9         conta2.numero = 456;conta2.saldo = 4000;  
10        if(conta1.saca(300)!=  
11            System.out.println("Saldo insuficiente na conta num "+conta1.numero);  
12            conta1.deposita(2000);  
13  
14            conta2.saca(300); //#abc.saca(300);  
15            conta2.deposita(-8000); //#abc.deposita(-8000);  
16            conta1.transferePara(conta2, 2000);  
17  
18            System.out.println("conta1: ");  
19            System.out.println("Número: "+conta1.numero);  
20            System.out.println("Titular: "+conta1.titular);  
21            System.out.println("Saldo: R$"+conta1.saldo);System.out.println("#####");  
22            System.out.println("conta2: ");  
23            System.out.println("Número: "+conta2.numero);  
24            System.out.println("Titular: "+conta2.titular);  
25            System.out.println("Saldo: R$"+conta2.saldo);System.out.println("#####");  
26        }  
27    }
```

- 3) Para ficar mais fácil de trabalhar, crie um método chamado mostraDados na classe conta. Importante salientar que o método em questão serve apenas para facilitar a vida do programador e que deve ser sempre possível exibir os dados separadamente.

```
38     public void mostraDados() {  
39         System.out.println("Número: "+this.numero);  
40         System.out.println("Titular: "+this.titular);  
41         System.out.println("Saldo: R$"+this.saldo);  
42     }
```

- 4) Crie a classe TestaMetodos2 utilizando este novo método.

```
2 public class TestaMetodos2 {  
3     public static void main(String[] args) {  
4         Conta conta1 = new Conta();  
5         conta1.titular = "Fulano";  
6         conta1.numero = 123; conta1.saldo = 3000;  
7         Conta conta2 = new Conta();  
8         conta2.titular = "Beltrano";  
9         conta2.numero = 456; conta2.saldo = 4000;  
10        if(conta1.saca(3001)!=true)  
11            System.out.println("Saldo insuficiente na conta num "+conta1.numero);  
12        conta1.deposita(2000);  
13  
14        conta2.saca(300); //abc.saca(300);  
15        conta2.deposita(-8000); //abc.deposita(-8000);  
16        conta1.transferePara(conta2, 2000);  
17  
18        System.out.println("conta1: ");  
19        conta1.mostraDados();  
20        System.out.println("#####");  
21        System.out.println("conta2: ");  
22        conta2.mostraDados();  
23        System.out.println("#####");  
24    }  
25 }
```

### 31. Controlando o acesso através de modificadores

Um dos problemas que temos na nossa classe conta é que a função saca permite um saque além do limite estipulado. Veja como deixamos a classe Conta:

```
2 public class Conta {}  
3     // Atributos  
4     int numero;  
5     String titular;  
6     double saldo;  
7  
8     // Métodos  
9     boolean saca(double valor) {  
10        if (this.saldo < valor)  
11            return false;  
12        else {  
13            this.saldo -= valor;  
14            return true;  
15        }  
16    }
```

Veja na classe a seguir, como é fácil ultrapassar o limite usando o método saca:

```
2 public class TestaEstouro1 {  
3     public static void main(String[] args) {  
4         Conta minhaConta = new Conta();  
5         minhaConta.saldo = 1000;  
6         minhaConta.saca(5000); // saldo + limite é só 2000!!  
7         System.out.println("Saldo: R$"+minhaConta.saldo);  
8     }  
9 }
```

**Quem garante que o usuário da classe vai sempre usar o método saca() para alterar o saldo da conta?**

Como evitar isso? Uma forma simples de se resolver isso seria testar se não estamos ultrapassando o saldo quando vamos sacar.

```
public class TestaContaEstouro3 {  
    public static void main(String[] args) {  
        Conta minhaConta = new Conta();  
        minhaConta.saldo = 1000;  
        minhaConta.limite = 1000;  
        double novoSaldo = -3000;//  
  
        if (novoSaldo < (minhaConta.saldo + minhaConta.limite)) {  
            System.out.println("Não posso sacar este valor.");  
        } else {  
            minhaConta.saldo = novoSaldo;  
        }  
    }  
}
```

O problema é que além deste código se repetir ao longo de toda nossa aplicação, alguém pode esquecer-se de fazer essa comparação em algum momento, deixando a conta em estado inconsistente. A melhor forma de resolver isso é obrigar o usuário da classe conta a usar o método saca e não permitir que ele acesse diretamente o atributo saldo. É o mesmo caso da validação de CPF. Lembra?

Para isso funcionar no Java, basta declarar que os atributos não podem ser acessados fora da classe. Isso pode ser feito através da utilização da palavra chave private:

```
public class Conta {  
    private int numero;  
    private double saldo;  
    private String titular;  
//...
```

**Private** é um **modificador de acesso** (também conhecido como **modificador de visibilidade**).

Quando marcamos um atributo como privado, dizemos que o mesmo só pode ser acessado dentro da própria classe. Desta forma o código abaixo não compila:

```
public class TestaContaEstouro1 {  
    public static void main(String[] args) {  
        Conta minhaConta = new Conta();  
        minhaConta.saldo = 1000;  
    //Não compila. Não se pode acessar um atributo privado de outra classe.  
    }  
}
```

Com relação à Orientação a Objetos, é prática quase obrigatória proteger seus atributos com private.

Uma classe é responsável por controlar o estado de seus atributos. Para isso deve fazer uso de regras que não os deixem em estado inconsistente. Se permitirmos acesso direto aos atributos esse controle sobre a consistência dos valores de cada atributo deixa de ser da classe. Lembra do SRP (Princípio da responsabilidade única)?

**A responsabilidade de validar seus atributos deve estar somente na classe.** Desta forma, quando precisarmos mudar alguma regra, o fazemos em um único lugar, facilitando a manutenção.

Perceba que a partir de agora, quem invoca o método saca() nem imagina que existe uma checagem acerca do valor solicitado.

Quem for usar a classe **precisa saber apenas o que cada método faz e não de que forma ele faz**. O que um método faz é sempre mais importante do que como ele faz. Mudar a implementação de um método é simples, mudar a assinatura de um método pode causar impacto em vários pontos de sua aplicação e gerar problemas.

A palavra chave **private** também serve para modificar o acesso a um método. Isso geralmente é usado em métodos auxiliares de outros métodos da própria classe.

Devemos sempre expor o mínimo possível de nossas funcionalidades, promovendo assim o **baixo acoplamento** entre as classes.

Assim como temos o private, temos também o public, que permite a todos acessarem um atributo ou método. Exemplo:

```
public boolean saca(double valor) {  
    if (this.saldo >= quantidade) {  
        this.saldo -= quantidade;  
        return true;  
    } else {  
        return false;  
    }  
}
```

Quando **não** definimos um modificador de acesso, seu método ou atributo, fica num estado intermediário entre o private e o public. Assim que estávamos deixando nossos atributos até agora. Veremos em detalhes mais adiante.

Em geral declaramos os atributos como private e os métodos que modificam os valores destes atributos como public. Desta forma, os objetos se comunicarão por troca de mensagens, ou seja, acessando seus métodos. Algo muito mais educado do que mexer em um atributo que não é seu.

Quando você chama um método, você está pedindo. Quando tenta acessar diretamente um atributo você está “metendo a mão” e pegando ou modificando algo sem permissão.

O melhor de tudo é que se algum dia precisarmos mudar as regras para a realização de um saque, só precisaremos mudar em um único lugar: no método saca().

Imagine que a partir de agora tenhamos que cobrar CPMF sobre cada saque. Onde fariammos essa mudança? O melhor de tudo é que as outras classes que usam o método saca() nem precisarão sofrer mudanças. Elas não sentem o impacto da mudança. Afinal, elas apenas chamam o método e esperam efetuar um saque. Não interessa como esse saque acontecia até agora e continuará não interessando depois dessa modificação.

Resumindo: Você faz uma manutenção em parte do sistema sem causar impacto nele como um todo.

## 32.Encapsulamento

O que começamos a ver com private é a idéia de encapsular, isto é, esconder membros de uma classe, além de esconder como funcionam os métodos (detalhes de sua implementação).

Para que seu sistema seja suscetível a mudanças, encapsular é fundamental! Agora, diferente de como acontecia na programação procedural, não precisamos mudar uma regra de negócios em vários lugares. Elas agora não estão espalhadas como antes.

Modificamos em apenas um único lugar, já que esta regra está encapsulada. Veja o caso do método saca().

O conjunto de métodos públicos de uma classe é também chamado de interface da classe, pois esta é a única maneira pela qual você se comunica com objetos dessa classe. Veja como funciona na figura abaixo:

Classe Conta	
Atributos públicos (se houver) saca deposita transferePara	Atributos privados e corpo dos métodos
<b>Interface</b> (o que faz! Porção visível para o mundo externo).	<b>Implementação</b> (Como faz! Não preciso me preocupar muito, uma vez que posso mudar a qualquer hora!)

## Programe voltado para a interface e não para a implementação!

Pense sempre que para cada usuário o que importa sobre um método de uma classe é “o que ele faz”! Isso dificilmente muda. O método saca sempre vai sacar um dinheiro da conta. O que muda com o tempo é a implementação de cada método o “como faz”. Isso o usuário final não precisa conhecer.

**IMPORTANTE:** Sempre que vamos acessar um objeto utilizamos sua interface.

Quando vamos usar um celular queremos fazer ligações, atender ligações. Existem diversos celulares no mercado e todos fazem e atendem ligações. De que forma cada celular faz isso, certamente varia de modelo para modelo.

De qualquer forma, o que nos importa é **o que ele é capaz de fazer** por nós, certo? Da mesma forma, não importa como um computador acessa a memória, processa dados, acessa a internet. O que importa é que ele seja capaz de fazer estas coisas.

## Métodos get e set

Como pudemos ver nos exemplos anteriores, o modificador private impede que o atributo seja lido ou modificado. Agora temos um grande problema: Como mostraremos o saldo de uma conta, por exemplo, já que não podemos acessá-lo nem para leitura?

Precisamos arranjar um meio de fazer isso. Sempre que precisamos fazer alguma coisa com algum objeto, criamos métodos. Já que o atributo saldo só pode ser lido de dentro da própria classe, criaremos um método público que retorna o valor desse atributo. O método pode se chamar obtemSaldo e deverá retornar um double. Veja:

```
public class Conta {  
    // Atributos  
    private int numero;  
    private String titular;  
    private double saldo;  
  
    //Métodos de acesso  
    public double obtemSaldo() {  
        return this.saldo;  
    }  
    //restante da classe....
```

Agora para acessar o saldo de uma conta podemos fazer da seguinte forma:

```
2 public class TestaObtemSaldo {  
3     public static void main(String[] args) {  
4         Conta c1 = new Conta();  
5         c1.deposita(200);  
6         System.out.println("O saldo atual é: " + c1.obtemSaldo());  
7     }  
8 }
```

Já que definimos que nossos atributos, em geral, serão private. Para permitirmos um acesso controlado a estes atributos criamos um método público que retorna o valor do atributo (leitura) e outro método público para modificar o valor do atributo (escrita).

Usando as convenções do Java determinamos que o método que retorna o valor do atributo se chamará getAtributo e o método que modifica o valor do atributo se chamará setAtributo.

**IMPORTANTE:** Use as convenções do Java sempre! Mais adiante vai perceber que o Java oferece uma série de facilidades para o programador, desde que este esteja seguindo suas convenções. Veja um exemplo:

```
2 public class Conta {  
3     // Atributos  
4     private int numero;  
5     private String titular;  
6     private double saldo;  
7     //Métodos de acesso  
8     public int getNumero() {  
9         return this.numero;  
10    }  
11    public void setNumero(int numero) {  
12        this.numero = numero;  
13    }  
14  
15    public String getTitular() {  
16        return this.titular;  
17    }  
18    public void setTitular(String titular) {  
19        this.titular = titular;  
20    }  
21  
22    public double getSaldo() {  
23        return this.saldo;  
24    }  
25  
26    // Métodos de comportamento |
```

```
27 boolean saca(double valor) {  
28     if (this.saldo < valor)  
29         return false;  
30     else {  
31         this.saldo -= valor;  
32         return true;  
33     }  
34 }  
35 boolean deposita(double valor) {  
36     if (valor <= 0)  
37         return false;  
38     else {  
39         this.saldo += valor;  
40         return true;  
41     }  
42 }  
43 public boolean transferePara(Conta contaDestino, double valor) {  
44     if (this.saca(valor)==true) {  
45         contaDestino.deposita(valor);  
46         return true;  
47     } else  
48         return false;  
49 }  
50 }
```

**Esquecemos de criar o método getSaldo()????**

Agora vamos pensar em uma conta de verdade: existe a possibilidade de o saldo da sua conta mudar sem que seja através de um saque, um depósito ou uma transferência?

Perceba que já tínhamos os métodos necessários para mudar o saldo e sem pensar, poderíamos ter criado um método setSaldo() que permite que o saldo seja modificado sem estar atrelado a uma operação bancária. Este método não deve ser criado!

**Má prática de programação:** criar uma classe e, logo em seguida, criar getters e setters para cada um de seus atributos. Não invente! Crie métodos de acordo com a sua necessidade!

**Outro detalhe importante:** um método getAtributo não precisa retornar necessariamente o valor daquele atributo somente. Imagine que o banco tenha um limite e queira que retornemos como saldo o valor do limite somado ao saldo. Logo nosso método getSaldo retornará limite+saldo. É uma regra que vai estar encapsulada (escondida) dentro do método getSaldo(). Olha o encapsulamento aí!

Bom, parece que agora estará tudo bem com a nossa conta né? Existe a chance de ela ficar com menos dinheiro que o saldo? Graças às regras de integridade e ao encapsulamento dos nossos dados a resposta é não!

Agora temos métodos específicos para mudar e obter o valor de cada atributo e se existem valores inconsistentes que estes atributos não podem receber, tratamos isso dentro de cada método, através do encapsulamento!

**Encapsular** = esconder atributos e detalhes da implementação.

Encapsular diminui o impacto das mudanças para o usuário final da classe.

**Métodos get e set** = métodos acessórios (métodos de acesso).

### 33.Exercícios práticos 3

- 1) Modifique a classe conta para que tenha atributos privados e métodos get e set (quando necessários)

- 2) Crie uma classe de testes e instancie pelo menos duas contas. Preencha valores para seus atributos, faça depósitos e saques válidos.
- 3) Teste essa conta com depósitos, saques e transferências inválidas.
- 4) Faça com que a classe de testes imprima os valores dos atributos das contas criadas.
- 5) Crie um método mostraDados que não devolve nenhum valor, apenas imprime os dados de cada conta. Isso não significa que não precisaremos mais dos métodos get. Afinal, devemos garantir que seja necessário obter os valores de cada atributo individualmente.

### 34.Um pouco mais sobre atributos e ... Agregação.

Os atributos, diferentemente das variáveis locais e temporárias (declaradas dentro de um método), recebem um valor padrão. Vimos, nos tópicos anteriores, quais são os valores padrão para cada um dos tipos de variáveis/atributos mais utilizadas.

Nada impede que você mesmo dê valores default para eles:

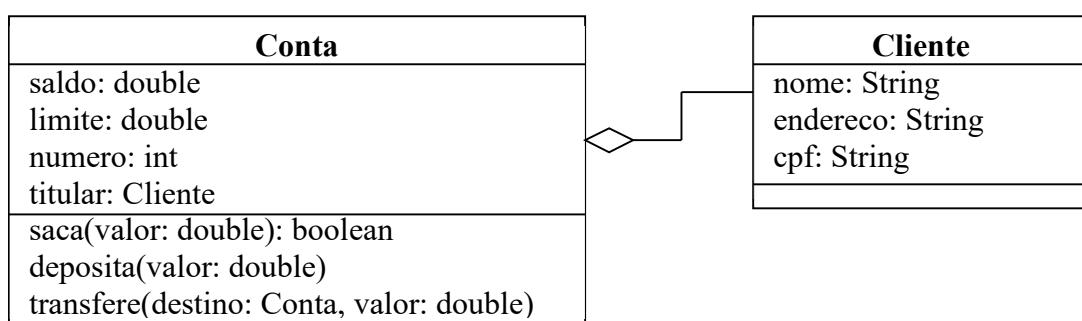
```
public class Conta {  
    int numero = 123;  
    String titular = "Maria";  
    double saldo = 0.0, limite = 100.0;  
  
    // ...  
}
```

No caso acima, quando você criar uma conta, seus atributos já estarão populados. Mas essa não é a melhor opção nesse caso.

Agora vamos imaginar que queiramos adicionar alguns outros atributos à classe conta. Ao invés do atributo titular vamos querer nome, sobrenome, cpf e endereço do cliente titular da conta.

Começaríamos a ter muitos atributos, não acha? E se a gente parar para pensar direito, vamos perceber que uma conta não tem nome, sobrenome, CPF e endereço. Quem tem esses atributos é o cliente. Então, podemos criar uma nova classe Cliente e fazer com que o atributo titular da classe Conta passe a ser do tipo Cliente.

Isso é o que chamamos de **agregação**. Perceba que um atributo de uma classe pode conter uma referência para outra classe. Veja a figura:



Observe que a Classe Conta agora agrupa uma referência para um objeto do tipo Cliente.

Outra perspectiva é observar que Cliente é parte integrante de Conta.

**Agregação** → estabelece uma relação *todo-parte* entre classes, sendo que a parte pode existir sem o todo.

Ex: Carro e Roda. Uma Roda é parte de um Carro, porém a Roda existe por si só fora do Carro. Você pode por exemplo remover a roda de um carro para colocar em outro.

Cliente pode existir sem a Conta. Pode abrir nova Conta até mesmo em outro banco. Um caso de agregação. Mais adiante veremos que existe um tipo de agregação mais forte chamado **composição**.

Vejamos agora como ficaria esse Código:

```
public class Cliente {  
    private String nome;  
    private String endereco;  
    private String cpf;  
    //Métodos get e set  
}  
  
public class Conta {  
    private int numero ;  
    private Cliente titular ;  
    private double saldo, limite ;  
  
    //..  
}  
  
public class Programa {  
    public static void main(String[] args) {  
        Conta umaConta;  
        umaConta = new Conta();  
        Cliente c;  
        c = new Cliente();  
        umaConta.setTitular(c);  
        //passo uma referência de Cliente para o atributo titular,  
        que a partir daí passa a se referir a uma conta.  
        umaConta.getTitular().setNome("João");  
        //através da referência contida em titular, acesso  
        "indiretamente" o atributo nome da classe Cliente.  
        umaConta.deposita(1000.0);  
  
        //...  
    }  
}
```

Um sistema orientado a objetos é um conjunto de classes que vão se comunicar e delegar responsabilidades para quem for mais apto a realizar determinada tarefa. Em um sistema mais completo, a classe Banco usaria a classe Conta que usaria a classe Cliente, que poderia usar uma classe Endereço e daí por diante.

Estes objetos colaboram entre si enviando mensagens e pedindo coisas uns aos outros. Desta forma acabamos tendo um sistema com muitas classes, cada uma com um tamanho geralmente pequeno.

Mas, e se eu me esquecesse de dar **new** em Cliente e quisesse acessar um de seus atributos através do método **getTitular()**, contido em Conta? Simplesmente não funcionaria.

Ocorreria um erro durante a execução (**NullPointerException**, que veremos mais adiante).

O atributo titular é uma referência para um objeto do tipo Cliente.

No momento em que eu crio um objeto do tipo Conta, todos os seus atributos recebem valores default.

Lembrando que o valor default para uma variável de referência , como é o caso de titular, é null.

Nesse caso titular apontaria para null. Ou seja , não vai conter de cara referência para nenhum objeto do tipo Cliente. Uma solução seria declarar Conta de outra forma. Veja:

```
public class Conta {  
    private int numero ;  
    private Cliente titular = new Cliente() ; //mudança  
    private double saldo, limite ;  
  
    //..  
}
```

Repare que agora, toda vez que eu criar uma Conta, automaticamente estarei criando um cliente e referenciando este cliente através do atributo titular.

### Exercitando agregação.

Como novo exemplo, vamos ver como ficaria uma Fábrica de Carros. Criaremos a classe Carro, com seus atributos, que descrevem suas características, e com métodos, que descrevem seu comportamento:

```
public class Carro {  
    private String cor, modelo; // get e set  
    private double velocidadeAtual; // Apenas get  
    private double velocidadeMaxima = 140; // Definido por default + get  
  
    // Método para ligar o carro  
    public void ligar() {  
        System.out.println("O carro está sendo ligado...");  
    }  
  
    // Método que faz o carro acelerar uma determinada velocidade  
    public void acelerar(double velocidade) {  
        this.velocidadeAtual += velocidade;  
    }  
  
    // Método que devolve a marcha em que o carro está  
    public int obtemMarcha() {  
        if (this.velocidadeAtual <= 0)  
            return -1;  
        else if (this.velocidadeAtual <= 20)  
            return 1;  
        else if (this.velocidadeAtual <= 40)  
            return 2;  
        else if (this.velocidadeAtual <= 60)  
            return 3;  
        else if (this.velocidadeAtual <= 80)  
            return 4;  
        else  
            return 5;  
    }  
  
    // Métodos get e set
```

Represente graficamente esta classe.

Escreva o código acima.

**Não se esqueça de escrever os métodos get e set citados!**

Em seguida, vamos testar nosso carro em um programa de testes:

```
public class TestaCarro {  
  
    public static void main(String[] args) {  
        Carro meuCarro = new Carro();  
        meuCarro.setCor("Azul");  
        meuCarro.setModelo("Gol");  
  
        //ligando o carro  
        meuCarro.ligar();  
        //Acelerando o carro  
        meuCarro.acelerar(80);  
        //Obtendo a marcha atual  
        int marchaAtual = meuCarro.obtemMarcha();  
  
        //Imprimindo informações sobre o carro e o estado dos atributos  
        System.out.print("Meu "+meuCarro.getModelo()+" "+meuCarro.getCor()+" ");  
        System.out.print("está andando na marcha "+marchaAtual+" a ");  
        System.out.println(meuCarro.getVelocidadeAtual()+" Km/h!!!");  
  
        //Acelerando o carro novamente  
        meuCarro.acelerar(20);  
        //Obtendo a marcha atual novamente  
        marchaAtual = meuCarro.obtemMarcha();  
  
        //Imprimindo ( novamente) informações sobre o carro e o estado dos atributos  
        System.out.print("Agora, meu "+meuCarro.getModelo()+" "+meuCarro.getCor()+" ");  
        System.out.print("está andando na marcha "+marchaAtual+" a ");  
        System.out.println(meuCarro.getVelocidadeAtual()+" Km/h!!!");  
    }  
}
```

Faça aquele exercício de dividir a folha do seu caderno em 3 partes: programa, memória e classes.

Ótimo! Mas... meu carro também pode ter outros atributos como Tipo de combustível e potência, certo?

Estes atributos são do carro ou do motor? Do motor, é claro! Se você conseguir notar que atributos e métodos da sua classe constituem uma outra classe, você detectou um caso de agregação.

Além dos atributos cor, modelo, velocidade Máxima e velocidade Atual, um objeto da classe Carro também contém um motor, que pode ser considerado uma classe. Afinal, um motor possui características e comportamento próprio, apesar de ser parte integrante de um Carro. Veja:

Veja como ficará a classe Motor:

```
public class Motor {  
    private double potencia; //get e set  
    private String tipoDeCombustivel; //get e set  
    private boolean ligado=false; //is e set  
  
    public boolean ligar(){  
        this.ligado=true;  
        System.out.println("Ligando o motor...");  
        return true;  
    }  
  
    public boolean desligar(){  
        this.ligado=false;  
        System.out.println("Desligando o motor...");  
        return true;  
    }  
  
    public double getPotencia() {  
        return potencia;  
    }  
    public void setPotencia(double potencia) {  
        this.potencia = potencia;  
    }  
    public String getTipoDeCombustivel() {  
        return tipoDeCombustivel;  
    }  
    public void setTipoDeCombustivel(String tipoDeCombustivel) {  
        this.tipoDeCombustivel = tipoDeCombustivel;  
    }  
    public boolean isLigado() {  
        return ligado;  
    }  
    public void setLigado(boolean ligado) {  
        this.ligado = ligado;  
    }  
}
```

Não há como desligar um carro que esteja em movimento, certo?  
Logo, antes de desligar o carro é preciso pará-lo! Veja como ficará a classe carro:

```
public class Carro {  
    private String cor, modelo; // get e set  
    private double velocidadeAtual; // Apenas get  
    private double velocidadeMaxima = 140; // Definido por default + get  
    private Motor motor; //get e set  
  
    // Método para ligar o carro  
    public void ligar() {  
        if(this.motor.ligar())  
            System.out.println("O carro está sendo ligado...");  
    }  
    //Método para parar o carro  
    public void parar(){  
        System.out.println("Parando o carro...");  
        this.velocidadeAtual=0;  
    }  
    // Método para ligar o carro  
    public void desligar() {  
        if(this.velocidadeAtual<=0){  
            if(this.motor.desligar())  
                System.out.println("O carro está sendo desligado...");  
        }  
        else{  
            System.out.println("Não há como desligar um carro em movimento.");  
        }  
    }  
  
    //Métodos get, set, acelerar, obtemMarcha....
```

Veja como ficará a nova versão da classe TestaCarro:

```
public class TestaCarro {  
  
    public static void main(String[] args) {  
        Carro meuCarro = new Carro();  
        meuCarro.setCor("Azul");  
        meuCarro.setModelo("Gol");  
  
        //ligando o carro  
        meuCarro.ligar();  
        //Acelerando o carro  
        meuCarro.acelerar(80);  
        //Obtendo a marcha atual  
        int marchaAtual = meuCarro.obtemMarcha();  
  
        //Imprimindo informações sobre o carro e o estado dos atributos  
        System.out.print("Meu "+meuCarro.getModelo()+" "+meuCarro.getCor()+" ");  
        System.out.print("está andando na marcha "+marchaAtual+" a ");  
        System.out.println(meuCarro.getVelocidadeAtual()+" Km/h!!!!");  
  
        //Parando o carro  
        meuCarro.parar();  
  
        //Desligando o carro  
        meuCarro.desligar();  
    }  
}
```

## Exercícios no BlueJ

- 1) Reescreva as classes acima, compile-as e execute apenas a classe TestaCarro!  
E aí? Tudo funcionou? O que houve?

Assim como fizemos com o exemplo do banco, agora podemos criar carros e mexer com seus atributos e métodos. A diferença é que agora estamos utilizando agregação. Então: Cuidado com o NullPointerException !!!!

Veja a nova versão da classe TestaCarro:

```
public class TestaCarro {  
  
    public static void main(String[] args) {  
        Carro meuCarro = new Carro();  
        meuCarro.setCor("Azul");  
        meuCarro.setModelo("Gol");  
  
        //Criando o motor  
        Motor motor = new Motor();  
        motor.setPotencia(1.6);  
        motor.setTipoDeCombustivel("GASOLINA");  
        //Vinculando o motor ao carro  
        meuCarro.setMotor(motor);  
        //ligando o carro  
        meuCarro.ligar();  
        //Acelerando o carro  
        meuCarro.acelerar(80);  
        //Obtendo a marcha atual  
        int marchaAtual = meuCarro.obtemMarcha();  
        //Imprimindo informações sobre o carro e o estado dos atributos  
        System.out.print("Meu "+meuCarro.getModelo()+" "+meuCarro.getCor()+" ");  
        System.out.print("está andando na marcha "+marchaAtual+" a ");  
        System.out.println(meuCarro.getVelocidadeAtual()+" Km/h!!!!");  
        //Parando o carro  
        meuCarro.parar();  
        //Desligando o carro  
        meuCarro.desligar();  
        //Imprimindo informações sobre o motor e o carro  
        String estadoDoMotor = (meuCarro.getMotor().isLigado())?"LIGADO":"DESLIGADO";  
        System.out.print("Meu "+meuCarro.getModelo()+" "+meuCarro.getMotor().getPotencia());  
        System.out.println(" está com o motor "+estadoDoMotor+".");  
    }  
}
```

Execute-a novamente!

## VER SE ENCAIXA AQUI

Analogias à parte, agora temos conhecimento suficiente para resolver aquele probleminha de validação de CPF, lembra?

```
private class Cliente {  
    String nome, cpf;  
  
    private boolean validaCpf(String cpf) {  
        if (cpf.length() == 11) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

```
public void recebeCpf(String cpf) {  
    if (this.validaCpf(cpf)) {  
        this.cpf = cpf;  
    }  
}
```

Perceba que como o atributo CPF é privado, a única forma de alterar seu valor é através do método `recebeCpf`. Nesse caso concentrarmos nossa regra de negócio neste método. Importante perceber que também criamos um método auxiliar e privado para validar o CPF. Trata-se de um método **encapsulado**. Esse método só pode ser invocado dentro da própria classe.

### Exercícios:

- 1) Ao declarar uma classe, existem regras para dar nome a métodos. Pesquise sobre isso.
- 2) Pesquise sobre as convenções de código, dadas pela Sun para dar nome às variáveis. O termo “Code conventions” pode ser usado na pesquisa.
- 3) Existe um padrão para representar suas classes e operações de abstração (agregação, por exemplo). Esse padrão é a UML. Represente nosso exemplo de carro e motor em diagramas de classe.
- 4) É obrigatório utilizar a palavra `this` quando for acessar um atributo da classe? Se não for obrigatório, por que utilizar?
- 5) No modelo anterior determinamos que a classe carro tivesse, dentre vários atributos, um chamado `velocidadeMaxima`. Vamos definir agora uma regra que determina que um carro não pode andar numa velocidade acima da `velocidadeMaxima`. Implemente essa regra no modelo.
- 6) Agora vamos considerar que um carro possui um tanque de combustível com os atributos: `capacidadeMaxima = 45` e `quantidadeDeCombustivelAtual`. Este carro (e/ou sua(s) classe(s) agregada(s)) também deve(m) ter o método:
  - Abastecer: não podendo exceder a capacidade máxima do tanque;
- 7) No programa `TestaCarro`, fazer com que o carro receba valores para os atributos do motor, bem como para estes outros referentes ao tanque de combustível. Fazer com que o carro ligue, abasteça e desligue.
- 8) Ainda no programa `TestaCarro`, fazer com que seja impressa, antes e depois de andar, uma mensagem informando a quantidade de combustível no tanque.
- 9) Nova regra: Um carro não pode ligar sem combustível. Implemente essa regra.
- 10) Faça aquele exercício de dividir a folha do seu caderno em 3 partes: programa, memória e classes.
- 11) Ainda há outras possibilidades para a classe carro. Use a imaginação e crie novos atributos, métodos, regras.

### Exercícios Obrigatórios:

Agora o objetivo é criar um sistema para gerenciar funcionários do banco. Este modelo será utilizado e modificado em exercícios posteriores. **Não deixe de fazer!**

- 1) Modele (apenas esboce uma classe em um papel) um funcionário que deve ter os seguintes atributos:
  - nome;

- departamento ( onde o funcionário trabalha);
  - salário;
  - ativo ( um atributo que indica se o funcionário ainda trabalha na empresa ou não). Você vai precisar criar alguns métodos. Dentre deles, crie o método aumentarSalario, que deverá aumentar o salário do funcionário de acordo com o percentual passado como argumento. Crie também um método demite, que não recebe parâmetro algum, apenas muda o valor booleano que indica que o funcionário não trabalha mais no banco.
- 2) Abra o projeto chamado projeto-banco, criado anteriormente;
  - 3) Escreva uma classe Java a partir do modelo acima. Teste-a usando a bancada do BlueJ e depois um programa ( outra classe que tenha o método main). Um esboço da classe:

```
public class Funcionario {  
    private String nome, departamento;  
    private double salario;  
    private boolean ativo;  
  
    public void aumentarSalario(double percentual){  
        //Implementar...  
    }  
  
    public void demite(){  
        //Implementar...  
    }  
  
    //Métodos get e set...
```

**Boa prática de programação:** você deve compilar seu arquivo Java sem que tenha terminado de escrever toda a classe. Isso evitara que você siga por um caminho errado ou que encontre dezenas de erros de uma vez só na primeira compilação.

Crie os atributos e compile. Se estiver tudo certo a cada novo método declarado compile novamente.

3.1)Crie uma classe chamada TestaFuncionario:

Um esboço do programa (classe que possui o main):

```
public class TestaFuncionario {  
    public static void main(String[] args) {  
        Funcionario funcionario1 = new Funcionario();  
        Funcionario funcionario2 = new Funcionario();  
  
        funcionario1.setNome("Fulano de Tal");  
        funcionario1.setSalario(3000);  
        System.out.print("O funcionário "+funcionario1.getNome());  
        System.out.println(" tem um salário de "+funcionario1.getSalario());  
        funcionario1.aumentarSalario(15);  
        System.out.print("Depois do aumento, o funcionário "+funcionario1.getNome());  
        System.out.println(" tem um salário de "+funcionario1.getSalario());  
  
        funcionario2.setNome("Rafael");  
        funcionario2.setSalario(1000);  
        System.out.print("O funcionário "+funcionario2.getNome());  
        System.out.println(" tem um salário de "+funcionario2.getSalario());  
    }  
}
```

**LEMBRE-SE:** As classes de teste ( que possuem o método main ) não precisam ser instanciadas. Basta chamar o método estático main usando o botão direito do mouse sobre a classe.

Faça outros testes. Use todos os atributos e métodos. Imprima outros atributos e de forma mais completa.

**IMPORTANTE:** não se esqueça de seguir a convenção Java, isto é: NomeDeClasse, nomeDeAtributo, nomeDeVariavel, nomeDeMetodo, etc...

- 4) Crie um método mostra(), que simplesmente imprime, linha a linha, todos os atributos de um funcionário. Assim, você evita ter que ficar copiando e colando um `System.out.println` a cada mudança de estado de seus atributos. Você apenas vai chamar:

```
funcionario1.mostra();
```

Produza as saídas desejadas.

Veja abaixo:

```
Nome: Paulo  
Salário: 2100.0  
Departamento: Compras  
Está na empresa? Sim
```

```
Nome: Paulo  
Salário: 2500.0  
Departamento: Compras  
Está na empresa? Sim
```

```
Nome: Paulo  
Salário: 2500.0  
Departamento: Compras  
Está na empresa? Não
```

A implementação do método ficaria mais ou menos assim:

```
void mostra() {
    System.out.println("Nome: "+this.nome);
    // Imprimir outros atributos ...
}
```

Mais adiante veremos uma solução muito mais elegante para mostrar um objeto como string através do método `ToString`.

- 5) Escreva um novo programa `TestaFuncionario2`, instanciando dois funcionários através do `new` e comparando-os com `==`. E se eles tiverem os mesmos valores para seus atributos? Veja:

```
public class TestaFuncionario2 {
    public static void main(String[] args) {
        Funcionario funcionario1 = new Funcionario();
        Funcionario funcionario2 = new Funcionario();

        funcionario1.setNome("Fulano de Tal");
        funcionario1.setSalario(3000);

        funcionario2.setNome("Fulano de Tal");
        funcionario2.setSalario(3000);

        if(funcionario1==funcionario2)
            System.out.println("IGUAIS");
        else
            System.out.println("DIFERENTES");
    }
}
```

O que vai ser impresso?

- 6) Desta vez crie duas referências para o mesmo funcionário e compare-os novamente:

```
public class TestaFuncionario2 {
    public static void main(String[] args) {
        Funcionario funcionario1 = new Funcionario();
        Funcionario funcionario2 = new Funcionario();

        funcionario1.setNome("Fulano de Tal");
        funcionario1.setSalario(3000);

        funcionario2 = funcionario1;

        if(funcionario1==funcionario2)
            System.out.println("IGUAIS");
        else
            System.out.println("DIFERENTES");
    }
}
```

E agora? O que vai ser impresso?

O que aconteceu no exercício 5? Quantos objetos foram criados? Quantos objetos vão ficar na memória?

- 7) Digamos que agora um funcionário vai ter mais um atributo chamado `dataDeNascimento`. Em vez de criar um atributo do tipo `String` para representá-lo,

vamos criar uma classe Data que vai conter 3 atributos do tipo String (dia, mês, ano). Nesta mesma classe, crie um método que retorne uma String que representará a data no formato dd/mm/aaaa.

- 8) Faça com que o atributo dataNascimento de Funcionário seja do tipo Data e que na declaração do atributo um objeto do tipo Data já seja criado:

**private Data** dataDeNascimento = **new Data()**;

- 9) Modifique seu programa TestaFuncionario para que seja definida a data de nascimento do funcionário e que através do método mostra() da classe Funcionário seja exibida a data de nascimento do funcionário.

- 10) O que acontece quando você tenta acessar diretamente um atributo da classe?

Como, por exemplo:

```
Funcionario.nome = "Paulo";
```

Isso faz sentido?

E este código, faz sentido?

```
Funcionario.demite();
```

Faz sentido pedir para que a classe ( especificação de um objeto) demita?

### Exercícios de fixação:

Dada a estrutura de uma classe, fazer seu esboço em UML, traduzí-la para a linguagem Java e usá-la, instanciando objeto(s) em um programa simples (classe com o método main).

- 1) Classe: Pessoa → Atributos: nome, idade. → Método: void fazAniversario();
  - Criar uma pessoa;
  - Fazer com que ela receba nome e idade;
  - Imprimir seus atributos;
  - Fazer com que ela faça aniversário algumas vezes e imprimir novamente seus atributos.
- 2) Classe: Porta. → Atributos: aberta, cor, largura, altura. → Métodos: void abre(), void fecha(), void pinta(String cor), boolean estaAberta();
  - Crie uma porta;
  - Abra e feche a porta;
  - Determine suas dimensões;
  - Pinte-a algumas vezes;
  - Use o método estaAberta() para saber sobre o estado da porta;
  - Imprima seus atributos.
- 3) Classe: Casa. → Atributos: cor, porta1, porta2, proprietário. → Métodos: void pinta(String cor), int quantasPortasEstaoAbertas().
  - Crie uma casa;
  - Defina seu proprietário;
  - Pinte-a;
  - Crie 2 portas e coloque-as na casa;

- Abra e feche cada porta algumas vezes;
  - Imprima quantas portas estão abertas.
  - Imprima atributos da casa e de seus objetos integrantes.
- 4) Crie os objetos usando a bancada de objetos do BlueJ e teste seus métodos!

## 35.Arrays em Java

Podemos declarar algumas variáveis dentro de um bloco e usá-las.

```
int idade1;  
int idade2;  
int idade3;  
int idade4;
```

Não fica muito prático, certo?

Podemos então criar um **array (vetor)** de inteiros:

```
int [] idades; //Array de inteiros chamado idades
```

O **int []** é um tipo e a variável **idades** é um objeto (uma referência para um array de inteiros). Precisamos criar um objeto do tipo **int[] idades** para ter acesso a esse **array**.

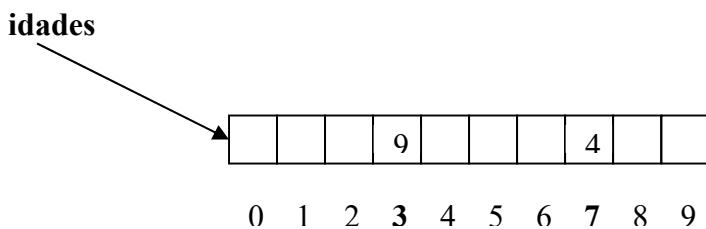
Como criar o **objeto-array**?

```
idades = new int[10]; //Cria um array de inteiros com 10 posições  
referenciado por idades.
```

Acabamos de criar um **array** de inteiros com 10 posições que podemos referenciar através da variável de referência **idades**.

Agora podemos acessar alguma posição do array, lembrando que se são 10 posições, a primeira posição é 0 e a última é 9:

```
idades[3] = 9;  
idades[7] = 4;
```



No caso acima, você acessou a 4<sup>a</sup> e 8<sup>a</sup> posições do array. Se você tentar acessar alguma posição fora dos limites do array (0 a 9), vai acontecer um erro durante a execução:

```
idades[11] = 4;  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:  
11  
at Teste.main(Teste.java:14)
```

A mensagem acima informa que houve um erro na linha 14 da classe Teste.java e que o erro de execução é `ArrayIndexOutOfBoundsException: 11`. Ou seja, você tentou acessar a posição 11 que está fora dos limites do array.

## 36.Arrays de referências

Sempre que criarmos um array de alguma classe, na verdade estaremos criando um array com referências. O objeto, como sempre, fica na memória principal e, no seu array, só ficam guardadas as referências para objetos. Na verdade, como objetos já são referências, a grosso modo podemos dizer que um array de objetos é uma referência para referências.

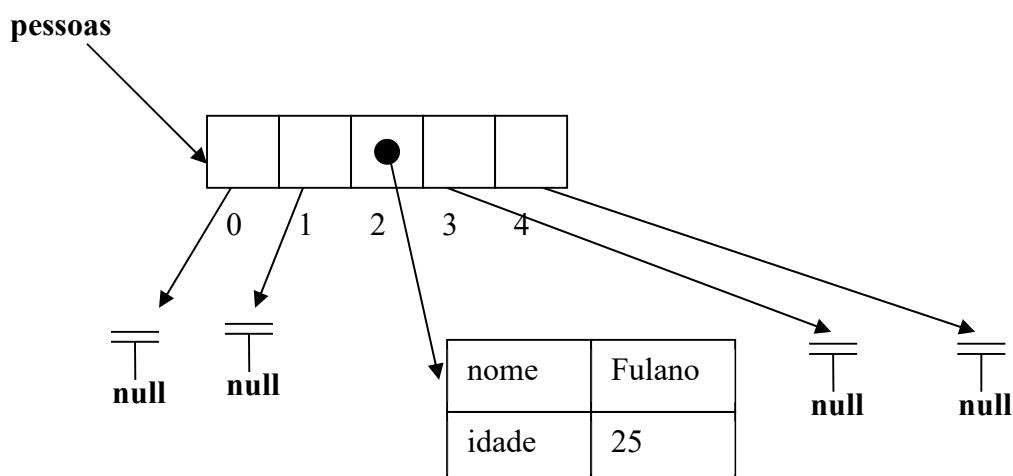
Veja:

```
Pessoa [] pessoas;  
pessoas = new Pessoas[5];
```

Quantas pessoas criamos aqui? Na verdade nenhuma. Criamos 5 espaços que podemos utilizar para guardar referências para objetos do tipo Pessoa. Por enquanto eles referenciam `null`.

```
pessoas[2] = new Pessoa;  
pessoas[2].nome = “Fulano”;  
pessoas[2].idade = 25;
```

Agora a posição 2 (3<sup>a</sup> do array) referencia um objeto do tipo Pessoa que recebe valores para seus atributos. Veja:



Perceba que as demais posições do array possuem referências nulas, já que ainda não referenciam nenhum objeto.

Ao criar o código abaixo estamos referenciando diretamente uma Pessoa:

```
pessoas[2] = new Pessoa();
pessoas[2].setNome ("Fulano");
pessoas[2].setIdade (25);
```

Também podemos fazer de forma indireta. Veja:

```
Pessoa umaPessoa = new Pessoa();
umaPessoa.setNome ("Fulano");
umaPessoa.setIdade(25);
```

```
pessoas[2] = umaPessoa;
```

O resultado é o mesmo.

## O que você precisa saber?

- Um array de tipos primitivos guarda valores;
- Um array de objetos guarda referências;

## Percorrendo um array:

Quando nós é que criamos o array, percorrê-lo é muito simples. Afinal, sabemos seus limites. Veja:

```
Int[] idades = new int[5];
for (int i = 0; i < 5 ; i++) {
    System.out.println(arrayIdade[i]);
}
```

E se tivéssemos que criar um método que recebe um array e percorre este array imprimindo seus valores? Não da pra saber quantas posições tem o array que vai ser passado como argumento, concorda? Uma hora pode ser um array de 5 posições, outra hora um de 15 posições. Veja:

```
static void imprimeArray(int[] arrayIdade) {
    for (int i = 0; i < ??? ; i++) {
        System.out.println(arrayIdade[i]);
    }
}
```

Até onde o for deve percorrer?

**Como resolver isso?** Veja:

```
static void imprimeArray(int[] arrayIdade) {
    for (int i = 0; i < arrayIdade.length; i++) {
        System.out.println(arrayIdade[i]);
    }
}
```

}

Todo array em Java tem um atributo chamado length que você pode usar para saber o tamanho do array que você está referenciando no momento.

**Importante saber que:** a partir do momento em que um array é criado ele não muda de tamanho. Se precisar de mais espaço, deve criar outro array com mais espaço e antes de se referir a ele, copiar todo o conteúdo do array velho.

### 37. Conhecendo e usando o foreach com arrays:

A partir da versão 5 do Java surgiu uma nova sintaxe mais “econômica” para percorrermos arrays e coleções, que veremos mais adiante. No caso não é necessário manter uma variável que guarda o nome do índice e nem indicar até que ponto percorrer. Ele percorre todo o array.

Este tipo de for mais econômico é conhecido como enhanced-for ou **foreach**.

Veja:

```
for (int i : idades) {  
    System.out.println(i);  
}
```

Isso vale também para arrays de referências. Ex:

```
Pessoa[] pessoas;  
pessoas = new Pessoa[5];  
  
for (Pessoa p : pessoas) {  
    System.out.println(p.getNome());  
}
```

O **foreach**, nada mais é, do que um truque de compilação para facilitar a vida do programador e tornar o código mais simples e legível.

### Exercícios obrigatórios com arrays:

- 1) No mesmo projeto que contém a classe funcionário, crie uma classe Empresa que terá nome, cnpj e uma referência a um array de Funcionário.

```
public class Empresa {  
    private String nome, cnpj;  
    private Funcionario[] empregados;  
  
    //...  
}
```

- 2) A Empresa deve ter um método adiciona que recebe uma referência a um Funcionário como argumento e guarda esse funcionário no array. Ficaria mais ou menos assim:

```
public class Empresa {  
    private String nome, cnpj;  
    private Funcionario[] empregados;
```

```
public void adiciona(Funcionario func) {  
    this.empregados[???] = func;  
}  
}
```

Em que posição guardaríamos esse funcionário? Precisamos guardá-lo em uma posição do array que esteja livre. Existem algumas maneiras para se fazer isso. Uma delas seria percorrer o array em busca de uma posição vazia, que aponte para null. Outra forma mais elegante é guardar um contador que vai ter sempre a posição livre e a cada Funcionário adicionado este contador seria incrementado.

É importante salientar que o método *adiciona* não recebe nome, departamento, salário, etc. A referência de um funcionário aponta para um objeto da memória que já possui estes atributos. Então, passando essa referência, você já está passando o Funcionário completo, com todos os seus atributos. Veja como ficaria:

```
public class Empresa {  
    private String nome, cnpj;  
    private Funcionario[] empregados;  
    private int posicaoLivre;  
  
    public void adiciona(Funcionario f){  
        this.empregados[this.posicaoLivre]=f;  
        this.posicaoLivre++;  
    }  
    //métodos get e set...  
}
```

- 3) Crie uma um programa TestaEmpresa, que obviamente deverá possuir um método main. Crie uma empresa e várias instâncias de Funcionário, passando para a Empresa através do método adiciona. Antes você vai precisar criar o array já que empregados por enquanto referencia null (lugar nenhum).

```
public class TestaEmpresa {  
    public static void main(String[] args) {  
        Empresa empresa = new Empresa();  
        empresa.setEmpregados(new Funcionario[5]);  
        //...  
    }  
}
```

Outra forma de se fazer a mesma coisa é na criação da Empresa já criar um array para empregados. Veja:

```
public class Empresa {  
    private String nome, cnpj;  
    private Funcionario[] empregados = new Funcionario[5];  
    private int posicaoLivre;  
  
    public void adiciona(Funcionario f){  
        this.empregados[this.posicaoLivre]=f;  
        this.posicaoLivre++;  
    }  
    //métodos get e set...  
}
```

No programa TestaEmpresa, crie alguns funcionários, preencha seus atributos e adicione à Empresa através do método adiciona.

```
public class TestaEmpresa {
    public static void main(String[] args) {
        Empresa empresa = new Empresa();

        // Funcionário 1
        Funcionario f1 = new Funcionario();
        f1.setNome("Rafael");
        f1.setDepartamento("INFORMÁTICA");
        f1.setSalario(2000);
        f1.aumentarSalario(10);
        f1.getDataDeNascimento().setDia("06");
        f1.getDataDeNascimento().setMes("11");
        f1.getDataDeNascimento().setAno("1974");

        // Funcionário 2
        Funcionario f2 = new Funcionario();
        f2.setNome("Maria");
        f2.setDepartamento("BIBLIOTECA");
        f2.setSalario(3000);
        f2.aumentarSalario(5);
        Data data = new Data();
        data.setDia("10");
        data.setMes("04");
        data.setAno("1995");
        f2.setDataDeNascimento(data);

        // adicionando os funcionários à Empresa
        empresa.adiciona(f1);
        empresa.adiciona(f2);
    }
}
```

- 4) Usando o foreach, percorra o atributo empregados de sua Empresa e imprima todas as informações a respeito de cada funcionário. Para fazer isso crie um método mostraEmpregados dentro de sua classe Empresa. Cuidado, alguns índices do seu array podem não conter uma referência para um Funcionário construído, ainda se referindo a null. Trate isso.
- 5) Lembra que na classe Funcionário você havia criado um método mostra()? No método mostraEmpregados() da Empresa, ao invés de imprimir atributo a atributo, use o método mostra() de Funcionário.
- 6) Na classe Empresa, crie um método para verificar se um determinado funcionário faz parte da Empresa ( ou seja: verificar se ele existe no array empregados). Se você tem um array de 100 posições e o funcionário que vc procura foi encontrado na posição 8, você precisa percorrer o restante do array? Pense nisso antes de escrever este método!
- 7) E se na hora de adicionar um novo funcionário o array já estiver cheio? Um array não muda de tamanho, lembra? Caso isso aconteça, criar um novo array, realocando os valores do array antigo para o novo array. Que tal criar um novo método encapsulado chamado aumentarArray?
- 8) Para que nosso sistema continue funcionando normalmente, ao remover um funcionário o array precisa ser reorganizado. Escreva um método para remover um funcionário do array. Escreva também outro método encapsulado para

- reorganizar o array e saiba como chama-lo toda vez que um Funcionário for removido. Que tal chamá-lo de reorganiza?
- 9) Faça testes na bancada de objetos do BlueJ.
- 10) **Dica:** Existe uma classe utilitária em Java chamada Arrays. Pesquise sobre ela. Com o uso da classe Arrays você poderia, por exemplo, utilizar este código para redimensionar o array:

```
this.empregados = Arrays.copyOf(this.empregados, this.empregados.length+1);
```

O método copyOf é o que chamamos de método estático. Observe que pudemos chama-lo diretamente, sem precisar instanciar um objeto da classe Array. Na classe Array existem métodos estáticos para ordenar os elementos do array, entre outros.

### Exercício para quem ainda está com dificuldade:

- 1) faça o esboço UML, escreva as classes em Java ou use as que já estão prontas, crie um programa para manipular as classes.
  - a. Classe: Casa → Atributos: cor, totalDePortas, portas (Array para Porta)  
→ Métodos: void pinta(String cor), int quantasPortasEstaoAbertas(), void adicionaPorta(Porta p), int totalDePortas().
  - b. Crie uma casa com 3 portas, pinte-a, coloque as portas na casa através do método adicionaPorta, abra e feche as portas. Imprima o número total de portas e quantas estão abertas.

## 38.Construtores e Java Bean

Sempre que usamos a palavra chave new, estamos construindo um objeto. Isso acontece porque quando o new é chamado, ele executa o construtor da classe. O construtor é um bloco declarado com o mesmo nome da classe. Ex:

```
public class Conta {  
    private double saldo;  
    private double limite;  
    //Construtor  
    public Conta(){  
        System.out.println("Construindo uma conta...");  
    }  
    //outros métodos
```

Desta forma, quando fizermos:

```
Conta c1 = new Conta();
```

A mensagem “Construindo uma conta...” aparecerá.

Um construtor não é um método. É uma rotina de inicialização que é chamada sempre que um novo objeto é criado.

Até agora não escrevemos o construtor em nenhuma de nossas classes. Então, como era possível dar new se todo new chama o construtor obrigatoriamente? Simples. Quando

você não declara o construtor em sua classe o Java cria um para você. É o chamado construtor default que não recebe nenhum argumento e seu corpo (implementação) é vazio. A partir do momento em que você declara um construtor, o construtor default não é mais fornecido.

Outro aspecto importante é que um construtor também pode receber argumentos, inicializando alguns atributos, por exemplo. Veja:

```
public class Conta {  
    private double saldo;  
    private double limite;  
    private Cliente titular;  
    int numero;  
  
    public Conta(Cliente titular){  
        this.titular=titular;  
    }  
    //O construtor recebe um argumento do tipo Cliente que vai inicializar  
    o atributo titular.
```

Isso significa que quando criarmos uma conta, ela já terá um titular criado. Veja:

```
Cliente fulano = new Cliente();  
fulano.setNome("fulano");  
  
Conta c = new Conta(fulano);  
System.out.println(c.getTitular().getNome());
```

### Por que usar um construtor?

Se toda conta precisa de um titular, criar um construtor que recebe esse valor é uma forma de obrigar que todos os objetos que forem criados tenham um valor deste tipo.

Um construtor pode obrigar o usuário a inicializar determinados atributos ou dar essa possibilidade a ele.

**IMPORTANTE:** Você pode ter mais de um construtor em sua classe. No momento de dar new, o construtor apropriado será escolhido.

Para ter mais de um construtor em sua classe, suas assinaturas devem ser diferentes.

Isso se chama sobrecarga (overload) de construtores. Existe também sobrecarga de métodos, que segue o mesmo padrão. Nome do método igual para assinaturas diferentes. Lembrando que construtores não são métodos.

Um construtor também serve para evitar a chamada de vários métodos set por exemplo, obrigando o usuário a passar as informações como argumento.

No exemplo do cliente, podemos obrigar o usuário a passar o CPF através do construtor. Assim teremos a garantia de que todo cliente terá um CPF.

### Um pouco mais sobre sobrecarga ( overload ):

Veja um exemplo com várias assinaturas para o método soma:

```
public class OperacoesMatematicas {  
  
    public int soma( int num1,int num2){  
        return num1+num2;  
    }  
  
    //método soma sobreescarregado  
    public int soma( int num1,int num2, int num3){  
        return num1+num2+num3;  
    }  
  
    //método soma sobreescarregado  
    public double soma( double num1,double num2){  
        return num1+num2;  
    }  
  
}
```

Perceba que quando falamos em assinaturas diferentes, falamos em diferença quanto ao número de argumentos ou até mesmo sobre o tipo de dado de cada argumento e/ou do retorno do método.

## Java Bean

Quando criamos uma classe com todos os atributos privados, métodos get e set para seus atributos e um construtor vazio, na verdade estamos criando um Java Bean. Não confunda com o EJB que é o Enterprise Java Bean. Veremos mais adiante.

## 39. Atributo da classe

Imagine que nosso banco também quer controlar quantas contas existem no sistema. A princípio podemos pensar em fazer assim:

```
Conta c = new Conta();  
totalDeContas = totalDeContas + 1;
```

Voltamos aqui a um problema enfrentado na validação de CPF. Estamos espalhando código por toda aplicação, ou seja, em cada parte da aplicação onde haja a criação de uma nova conta totalDeContas deverá ser incrementado. E quem garante que vamos lembrar de incrementar essa variável a cada nova conta criada?

Uma outra forma seria:

```
public class Conta {  
    private double saldo;  
    private double limite;  
    private Cliente titular = new Cliente();  
    int numero;  
  
    private int totalDeContas;  
  
    public Conta() {  
        this.totalDeContas = this.totalDeContas + 1;  
    }
```

//...

Isso também não resolve o problema porque cada conta criada é um objeto diferente e os atributos que conhecemos até agora são atributos do objeto. Desta forma, ao criar duas contas o valor de totalDeContas para cada uma das contas será 1, pois cada uma tem essa variável. **O atributo é de cada objeto criado.**

O que queremos é ter uma variável única, compartilhada por todos os objetos da classe. Assim, quando mudasse através de um novo objeto o mesmo valor seria enxergado por ambos. Para fazer isso em Java, declaramos a variável como static. Veja:

```
public class Conta {  
    private double saldo;  
    private double limite;  
    private Cliente titular = new Cliente();  
    int numero;  
  
    private static int totalDeContas;  
  
    public Conta() {  
        Conta.totalDeContas = Conta.totalDeContas + 1;  
        //A variável estática deve ser incrementada no construtor  
    }  
//...
```

Quando declaramos um atributo como static ele passa a não ser mais um atributo do objeto e sim um atributo da classe. A informação fica guardada pela classe, não é mais individual para cada objeto. Perceba acima que não usamos a palavra this. Para acessar um atributo estático, não usamos this. Usamos o nome da classe.

Como esse atributo é privado, como podemos acessar essa informação a partir de outra classe? Simples: criamos um método get para ele!

```
public int getTotalDeContas(){  
    return Conta.totalDeContas;  
}
```

Mas como chamaremos esse método?

```
Conta c = new Conta();  
int total = c.getTotalDeContas();
```

Teremos que criar uma Conta para saber quantas contas temos? Isso não faz muito sentido. Como resolver isso?

A idéia é a mesma: transformar este método que todo objeto Conta tem em um método de toda a classe Conta. Usamos a palavra static de novo, mudando a assinatura do método:

```
public static int getTotalDeContas(){  
    return Conta.totalDeContas;  
}
```

Agora para acessar esse método não precisamos de uma referência para um objeto criado da classe Conta. Usamos o nome da classe! Veja:

```
int total = Conta.getTotalDeContas();
```

**IMPORTANTE:** Métodos estáticos só podem acessar métodos e atributos estáticos dentro da própria classe, mesmo porque ele não pode fazer uso da referência this. Um método estático é chamado através da própria classe e não de uma referência dela.

*Static cheira a programação procedural. Por isso, evite o excesso! Use static somente quando for indispensável.*

**Um pouco mais sobre get e set:** O padrão dos métodos get e set não vale para atributos do tipo boolean. Esses atributos são acessados via is e set. Por exemplo, para saber se um funcionário está ativo usariam o método isAtivo() e para mudar seu valor setAtivo ou, como no nosso exemplo, criariam um método demite.

## 40.Exercícios sobre encapsulamento, construtores e static

- 1) Tente criar um Funcionário no main e modificar ou ler um de seus atributos privados, sem que seja através dos métodos. O que acontece? Como resolver?
- 2) Faça com que sua classe Funcionário possa receber, opcionalmente, o nome do Funcionário durante a criação do objeto. Utilize construtores para isso.
- 3) Você deve ter notado que precisamos criar um construtor sem argumentos no exercício anterior para que a passagem do argumento fosse opcional. Por que precisamos fazer isso?
- 4) Adicione à classe Funcionário um atributo chamado identificador. Esse atributo deve possuir um valor único que identifica cada Funcionário instanciado. O primeiro Funcionário instanciado vai ter identificador valendo 1, o segundo valendo 2 e daí por diante. Crie um get para o identificador. Devemos ter um set também?
- 5) Crie os métodos get e set da sua classe Empresa e coloque os atributos como private. Lembre-se que nem todo atributo precisa necessariamente de um get e um set. Por exemplo: seria interessante ter um get e um set para seu array de funcionários? Não seria mais interessante ter um método como este?

```
public Funcionario getFuncionario(int posicao) {
    return this.empregados[posicao];
}
```
- 6) Na classe Empresa, ao invés de criar um array de tamanho fixo, receba como parâmetro no construtor o tamanho do array de Funcionários.
- 7) Crie, na classe Funcionario, o atributo cpf com a garantia de que não haverá nenhum funcionário com CPF inválido. Você não precisa criar um validador de CPF de verdade, basta submeter a um método valida(String cpf), que deve ser um método encapsulado.
- 8) Por que o código abaixo não compila?

```
public class Teste {
    int x = 50;

    public static void main(String[] args) {
        System.out.println(x);
    }
}
```

## 41.Herança

Assim como acontece com a Empresa, nosso banco também tem funcionários. Vamos modelar as classes Funcionário e Gerente por exemplo:

```
public class Funcionario {  
    String nome, cpf;  
    double salario;  
    //métodos ...  
}  
  
public class Gerente {  
    String nome, cpf;  
    double salario;  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public boolean autentica(int senha){  
        if(this.senha==senha){  
            System.out.println("Acesso Permitido!");  
            return true;  
        }else{  
            return false;  
        }  
    }  
    // Outros métodos  
}
```

No nosso banco, além de um funcionário comum, há também outros cargos, como gerente por exemplo. O gerente possui os mesmos atributos de um funcionário comum somados a outros atributos próprios de um gerente, além de ter funcionalidades um pouco diferentes. No nosso banco um gerente possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia.

**Mas será que precisamos mesmo de outra classe? Vejamos algumas possíveis soluções:**

### Solução 1:

Poderíamos colocar todos os atributos na classe Funcionário, deixando-a mais genérica. Se o funcionário não fosse um Gerente, deixaríamos os atributos senha e numero de funcionários gerenciados vazios.

**Problema da solução 1:** O problema é que aí começamos a ter muitos atributos opcionais e a classe ficaria estranha, descaracterizada. Sem falar dos métodos. O método autentica é próprio de um Gerente e não cairia bem a um Funcionário. Daria a falsa impressão de que ele pode se autenticar no sistema.

**Se tivéssemos outro funcionário com poucas características diferentes das de um funcionário comum precisaríamos mesmo criar outra classe e copiar todo o código novamente?**

**E se um dia precisarmos adicionar um novo atributo aos funcionários do banco. Precisaremos criá-lo em cada uma dessas classes??**

O problema acontece novamente por não centralizarmos as informações principais de um funcionário em um único lugar!

Em Java existe um jeito de fazermos com que uma classe se relacione com outra herdando tudo o que ela tem. Trata-se de uma relação de classe mãe e classe filha.

Vejamos nosso caso:

Gostaríamos de fazer com que um Gerente tivesse tudo que Funcionário tem. Gostaríamos que ela fosse uma extensão de Funcionário. Fazemos isso através da palavra-chave **extends**.

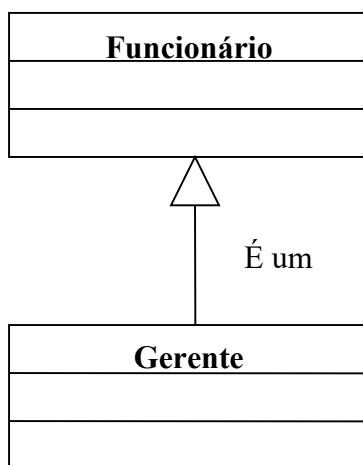
```
public class Gerente extends Funcionario{
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha){
        if(this.senha==senha){
            System.out.println("Acesso Permitido!");
            return true;
        }else{
            System.out.println("Acesso Negado.");
            return false;
        }
    }

    //métodos get e set...
```

Desta forma, toda vez que criarmos um objeto da classe Gerente, este também possuirá os atributos e métodos definidos na classe Funcionário, pois agora, todo Gerente é **um**(extends) Funcionário.

Veja a representação gráfica:



Gerente → é um → Funcionário

Dizemos que Gerente herda todos os atributos e métodos da classe mãe (Funcionário). Para ser bem exato, ele também herda os atributos e métodos privados, mas não consegue acessá-los diretamente.

Veja uma classe que manipula um Gerente:

```
public class TestaGerente {  
    public static void main(String[] args) {  
        Gerente g = new Gerente();  
        g.setNome("Rafael");  
        g.setSenha(123);  
        System.out.println(g.getNome());  
        g.autentica(123);  
    }  
}
```

### Superclasse e subclasse:

Dizer que Funcionário é a **classe mãe** e Gerente é a **classe filha** é o mesmo que dizer que Funcionário é a **superclasse** e Gerente é a **subclasse**. Ambas as nomenclaturas são utilizadas.

### Acessando atributos herdados através do modificador de acesso **protected**:

**E se precisarmos acessar os atributos que herdamos?** Com o modificador de acesso **private** isso não é possível. Com o modificador **public** deixaríamos nossa classe mãe muito exposta e qualquer um poderia acessar e modificar seus atributos.

Existe um outro modificador de acesso que fica entre o **private** e o **public**, chamado **protected**. Um atributo **protected** só pode ser acessado (visível) pela própria classe e por suas subclasses (e mais algumas outras classes. Veremos mais adiante).

```
public class Funcionario {  
  
    protected String nome, departamento;  
    protected String cpf;  
    protected double salario;  
    protected boolean ativo;  
    protected Data dataDeNascimento = new Data();  
    protected static int identificador;  
  
    //....
```

### Devemos sempre usar **protected**?

Com tempo e experiência você vai começar a perceber que nem sempre é uma boa idéia deixar que a classe filha accesse os atributos da classe mãe, pois isso quebra um pouco a idéia de que só aquela classe deveria manipular seus atributos. Teremos essa discussão mais adiante.

Você também vai descobrir que além das subclasses outras classes que estejam no **mesmo pacote** conseguem acessar os atributos **protected**.

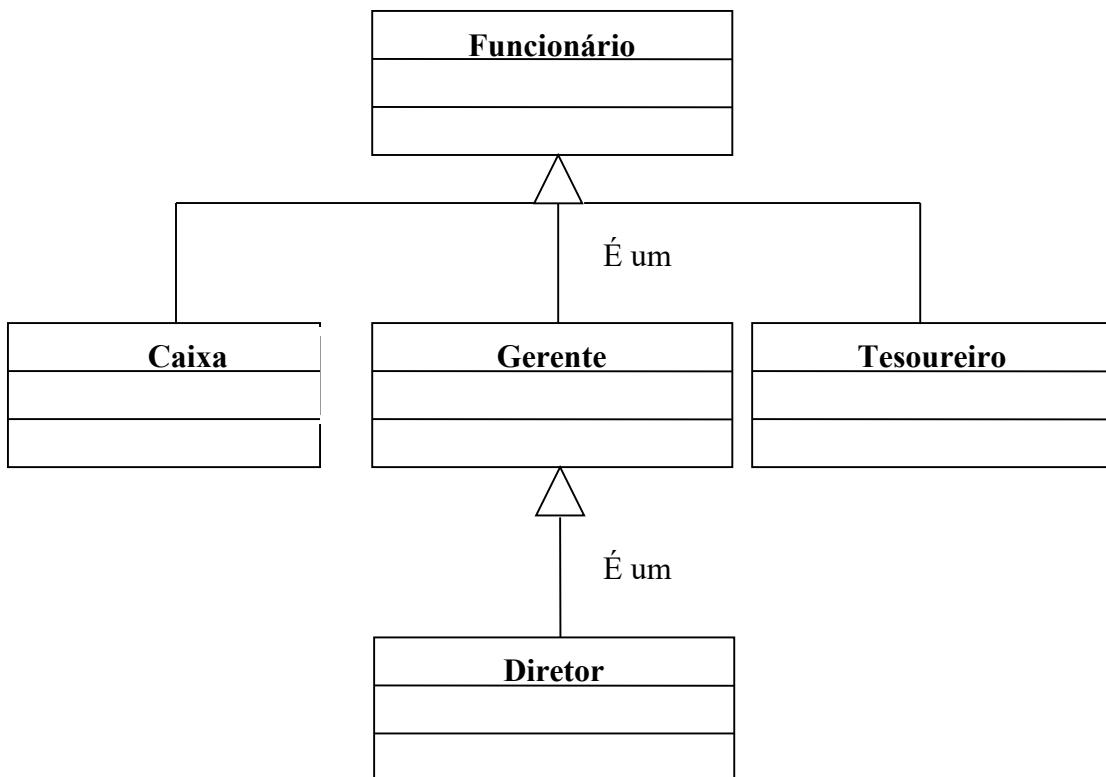
Perceba que há outras possibilidades para o nosso banco. Podemos ter:

- Uma classe Diretor que estenda Gerente;
- Uma classe Caixa que estenda Funcionário;
- Uma classe Tesoureiro que estenda Funcionário;

Que fique claro que essa é uma decisão de negócio. Se diretor vai estender Gerente ou não, vai depender se, “para você”, Diretor é um Gerente.

Veja na representação gráfica e observe que:

- ***Uma classe pode ter várias filhas, mas uma filha só pode ter uma única mãe.***  
Essa é a chamada herança simples do Java.



### IMPORTANTE:

- Construtores não são herdados, mas na subclasse deve haver um construtor compatível!

### Exercícios sobre Herança:

- 1) Crie a classe Pessoa com os atributos nome, cpf, telefone.
- 2) Crie a classe Aluno que estende Pessoa e também tem um atributo matricula.
- 3) Crie a Classe Professor que estende Pessoa e tem os atributos horasDeAulaMes, valorHoraAula e salario, além do método encapsulado calcularSalário. O salário de um professor deve ser o resultado das horasDeAulaMes X valorHoraAula.
- 4) Crie a Classe ProfessorMestre que estende Professor e tem o atributo temaDaDissertacao.

## Um pouco mais sobre construtores e herança:

Imagine que a classe Funcionário tenha apenas um único construtor, que recebe o nome do funcionário como argumento. Veja:

```
public class Funcionario {  
  
    protected String nome, departamento;  
    protected String cpf;  
    protected double salario;  
    protected boolean ativo;  
    protected Data dataDeNascimento = new Data();  
    protected static int identificador;  
  
    public Funcionario(String nome) {  
        this.nome=nome;  
        identificador++;  
    }  
    //....
```

## Como ficaria a classe Gerente?

Antes de responder, observe 3 coisas:

- 1) Um Funcionário só pode ser instanciado fornecendo ao seu construtor um argumento do tipo String para o atributo nome;
- 2) Gerente é um Funcionário. Logo, quando você está instanciando um Gerente, está instanciando também um Funcionário.
- 3) Se para criar um Gerente, implicitamente, você também precisa criar um funcionário, os construtores precisam ser compatíveis.

Veja como poderia ficar o construtor da classe Gerente:

```
public class Gerente extends Funcionario{  
    private int senha;  
    private int numeroDeFuncionariosGerenciados;  
  
    public Gerente(String nome) {  
        super(nome);  
    }  
  
    //...
```

## Observações importantes:

- 1) Se a superclasse tiver apenas um construtor, diferente do construtor padrão ( sem argumentos ), você obrigatoriamente tem que criar um construtor igual na subclasse e invocar o construtor da superclasse dentro dele, utilizando a palavra `super`.
- 2) Se na superclasse você possui um construtor padrão ( sem argumentos ) e outro construtor com argumentos, na subclasse este construtor com argumentos é opcional.

## 42. Reescrita de Métodos

Ao final de cada ano os Funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do seu salário, enquanto que os gerentes recebem 15%. Sendo assim, a classe Funcionário ficaria da seguinte forma:

```
public class Funcionario {  
    protected String nome, cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.10;  
    }  
    // métodos ...
```

Se deixarmos Gerente como ela já está, ela vai herdar o método getBonificação. Veja o código abaixo:

```
Gerente gerente = new Gerente();  
gerente.setSalario(10000.0);  
System.out.println(gerente.getBonificacao());
```

O resultado impresso sera 1000. Mas deveria ser 1500. Afinal, a bonificação de um Gerente é de 15% sobre o salário.

### Como resolver isso?

Uma forma de consertar isso seria criar um novo método na classe Gerente, chamado, por exemplo, de getBonificacaoDoGerente.

**Problema:** O grande problema é que teríamos dois métodos em Gerente, confundindo bastante quem fosse usar esta classe. Além de cada método dar uma resposta diferente para a mesma pergunta: Qual é a bonificação? Isso seria uma má prática de programação e estaríamos ferindo o SRP (princípio da responsabilidade única)! A responsabilidade de informar a bonificação de qualquer tipo de Funcionário é do método getBonificação. Criar outro método para fazer basicamente a mesma coisa, seria desnecessário!

No Java, quando herdamos um método podemos alterar seu comportamento na subclasse. Podemos reescrever(sobrescrever) este método:

```
public class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
    // ...
```

No Delphi e em outras linguagens, precisaríamos usar a palavra chave override para indicar que o método está sendo reescrito na subclasse. No Java não precisa, está implícito!

Agora sim está correto! Refaça o teste e verá que o resultado impresso será 1500!

## Invocando o método reescrito:

Depois de fazer a reescrita, não podemos mais chamar o método antigo que foi herdado da classe mãe: Realmente alteramos seu comportamento. Mas, caso estejamos dentro da subclasse, podemos invocá-lo.

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo da bonificação de um Funcionário somando mais 1000. Poderíamos fazer assim:

```
public class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.10 + 1000;  
    }  
    //...
```

**Fazendo desta forma, teríamos um problema:** No dia em que getBonificacao do Funcionário mudar, precisamos mudar o método do Gerente para acompanhar a nova bonificação. Para evitar isso, o getBonificacao do Gerente pode invocar o getBonificacao do Funcionário utilizando-se da palavra super.

```
public class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return super.getBonificacao() + 1000;  
    }  
    //...
```

Essa invocação vai procurar o método getBonificacao de uma classe que esteja imediatamente acima de Gerente. No caso ele vai encontrar esse método na classe Funcionário.

Se pensarmos que um método reescrito geralmente faz algo a mais que o método da classe mãe, faz todo sentido que esta seja uma prática comum. Chamar o método de cima ou não, depende da sua necessidade.

**ATENÇÃO:** Métodos com assinatura igual e implementação diferente são **reescritos** (ou sobrescritos). Métodos com assinatura diferente e nome igual são **sobre carregados**. Lembre-se dos construtores, por exemplo.  
Sobrecarga e reescrita são coisas completamente diferentes.

## Perguntas:

Existe reescrita sem herança?

Existe sobre carga sem herança?

A reescrita acontece em classes diferentes? E a sobre carga? Pode acontecer em classes diferentes?

### Exercícios sobre Herança e reescrita:

- 1) Baseado no exercício anterior, na classe ProfessorMestre, reescreva o método calcularSalário adicionando mais 20% ao valorHora antes de multiplicar pelas HorasDeAulaMes.
- 2) Escreva a classe Cliente e a classe Conta, que possua um saldo e os métodos para obter o saldo, saca, deposita e transferePara.

```
public class Cliente {  
    private String nome;  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public String getNome() {  
        return this.nome;  
    }  
}  
  
public class Conta {  
    private double saldo;  
    private double limite;  
    private Cliente cliente = new Cliente();  
  
    public boolean deposita(double valor)  
    {  
        if(valor>0){  
            this.saldo+=valor;  
            return true;  
        }  
        return false;  
    }  
  
    public boolean saca(double valor){  
        if(valor>0 && valor<=(this.saldo+this.limite)){  
            this.saldo-=valor;  
            return true;  
        }else{  
            return false;  
        }  
    }  
  
    public boolean transferePara(Conta contaDestino,double valor){  
        Conta contaOrigem = this;  
        if(contaOrigem.saca(valor)){  
            return contaDestino.deposita(valor);  
        }  
        return false;  
    }  
    //Métodos get e set....
```

- 3) Adicione um método na classe Conta que atualiza o saldo dessa conta de acordo com uma taxa percentual fornecida.

```
public void atualiza(double taxa){  
    this.saldo-=taxa;  
}
```

- 4) Crie duas subclasses da Classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar-se com o dobro da taxa e a ContaPoupanca deve atualizar-se com triplo da taxa.
- 5) A ContaCorrente deve reescrever o método deposita retirando uma taxa bancária de 10 centavos a cada depósito.
- 6) Crie uma classe com método main e instancie essas classes, atualize-as e veja o resultado. A classe deve ficar assim:

```
public class TestaContas {  
    public static void main(String[] args) {  
        Conta c = new Conta();  
        ContaCorrente cc = new ContaCorrente();  
        ContaPoupanca cp = new ContaPoupanca();  
  
        c.deposita(1000);  
        cc.deposita(1000);  
        cp.deposita(1000);  
  
        System.out.println("Saldo das contas antes de atualizar:");  
        System.out.println("Conta c: "+c.getSaldo());  
        System.out.println("ContaCorrente cc: "+cc.getSaldo());  
        System.out.println("ContaPoupanca cp: "+cp.getSaldo());  
  
        c.atualiza(0.01);  
        cc.atualiza(0.01);  
        cp.atualiza(0.01);  
  
        //System.err faz imprimir em vermelho  
        System.err.println("Saldo das contas depois de atualizar:");  
        System.err.println("Conta c: "+c.getSaldo());  
        System.err.println("ContaCorrente cc: "+cc.getSaldo());  
        System.err.println("ContaPoupanca cp: "+cp.getSaldo());  
    }  
}
```

Após imprimir o getSaldo() de cada uma delas, o que acontece?

## 43. Polimorfismo

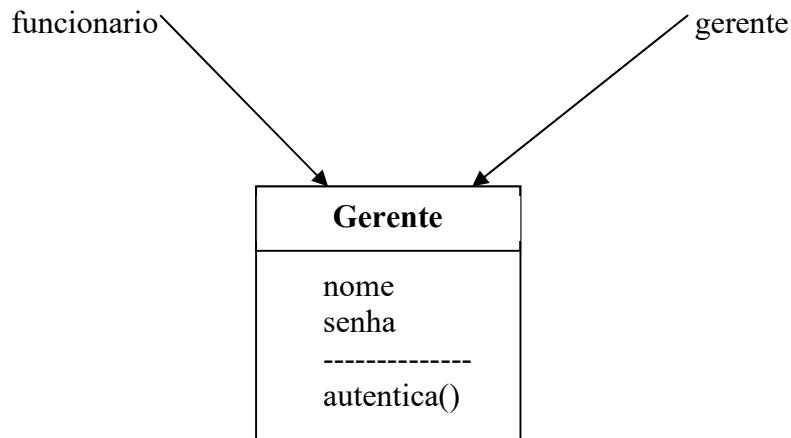
Em uma variável do tipo Funcionário você não guarda um objeto do tipo Funcionário. Você guarda, na verdade, **uma referência** para um Funcionário.

Falando sobre herança percebemos que todo Gerente é um Funcionário. Afinal, Gerente é uma extensão de Funcionário.

Se alguém entrar no banco e gritar: “Ei funcionário!”, o Gerente também vai olhar, certo? Afinal, ele, além de Gerente, é um Funcionário. Se alguém precisa falar com um Funcionário do banco, pode falar com um Gerente, pois como vimos em herança, **Gerente é um Funcionário**.

Veja o código a seguir:

```
Gerente gerente = new Gerente();
Funcionário funcionario = gerente;
funcionario.setSalario(10000.0);
```



**Polimorfismo** é a capacidade de um objeto **poder ser referenciado** de várias formas.

**CUIDADO:** Polimorfismo não significa que um objeto pode ficar se transformando. Muito pelo contrário. Lembre-se que Java é fortemente tipado: **Se um objeto nasce de um tipo, vai morrer daquele tipo**. O que pode mudar é a maneira como nos referimos a ele.

Até agora tudo bem, mas se eu tentar:

```
System.out.println(funcionario.getBonificacao());
```

O que vai ser impresso? 100 ou 150?

No Java, a invocação de método sempre vai ser decidida em tempo de execução. O Java vai procurar o objeto na memória, verificar qual é o seu tipo, e, aí sim, decidir qual método deve ser chamado. Sempre considerando qual é a sua classe de verdade e não a maneira pela qual estamos referenciando ele.

Apesar de estarmos nos referenciando a esse Gerente como um Funcionário, o método executado é o do Gerente. A resposta é 150.

Mas, qual a vantagem de criar um Gerente e referenciá-lo apenas como um Funcionário?

Na verdade a necessidade que costuma aparecer é termos um método que recebe um argumento do tipo mais Genérico. No caso Funcionário. Veja:

```
public class ControleDeBonificacoes {  
    private double totalDeBonificacoes = 0;  
  
    public double getTotalDeBonificacoes() {  
        return totalDeBonificacoes;  
    }  
  
    public void registra(Funcionario funcionario) {  
        this.totalDeBonificacoes += funcionario.getBonificacao();  
    }  
}
```

E, no caso, em algum lugar da minha aplicação (ou no main de uma Classe de teste):

```
public class TestaBonificacoes {  
    public static void main(String[] args) {  
        ControleDeBonificacoes controle = new  
ControleDeBonificacoes();  
  
        Gerente2 f1 = new Gerente2();  
        f1.setSalario(5000);  
        controle.registra(f1);  
  
        Funcionario2 f2 = new Funcionario2();  
        f2.setSalario(2000);  
        controle.registra(f2);  
  
        System.out.println(controle.getTotalDeBonificacoes());  
    }  
}
```

Pense numa porta na agência bancária com o seguinte aviso: “Permitida a entrada somente de funcionários”. Um gerente pode entrar por essa porta? Sim! Afinal, Gerente é um Funcionário.

Foi mais ou menos o que fizemos com o método acima se queremos que ele, assim como a porta do banco, possa receber Gerente e Funcionário, não faz sentido o argumento ser do tipo Gerente. Se um argumento é do tipo Funcionário ele pode receber Funcionário e Gerente. Foi o que aconteceu!

Não importa que o argumento seja do tipo Funcionário. Quando for passado ali um objeto que realmente é um Gerente, seu método reescrito será invocado e retornará a bonificação de um Gerente.

**Não importa como um objeto é referenciado, o método que será invocado é sempre o que é dele!**

### O que acontece na criação do objeto:

Funcionário g = new Gerente();

→ A parte do comando que está sublinhada e em azul é executada pelo compilador. Compila sem problemas.

→ A parte do comando que vem depois do sinal = é executada pela JVM que não encontra problemas. Afinal Gerente é um Funcionário.

Outra vantagem do polimorfismo é evitar impacto em modificações futuras. No dia em que criarmos uma classe Caixa, que estende Funcionário, o método que escrevemos aceitará também um objeto do tipo Caixa sem que tenhamos que fazer qualquer alteração nele.

Esse é o grande poder do polimorfismo e da reescrita de métodos: **diminuir o acoplamento entre as classes para evitar que novos códigos resultem em modificações em inúmeras partes do sistema.**

Repare que quando escrevemos o método registra em ControleDeBonificacoes, jamais imaginamos que haveria uma classe Caixa. Quando há herança e queremos referenciar um grande numero de objetos distintos, nos referimos a ele da forma mais genérica possível.

## Herança X Acoplamento

Cuidado com o uso exagerado de herança. Não programe herança apenas por preguiça de escrever. Tenha critérios. Herança aumenta o acoplamento entre as classes ( o quanto uma classe depende de outra). Às vezes isso atrapalha a fazer mudanças pontuais no sistema.

Por exemplo, imagine se quisermos mudar algo na nossa classe Funcionário, mas não quisermos que todos os funcionários sofressem tal mudança. Teríamos que olhar com atenção cada subclasse ( e podem ser muitas ) e ver se ela comporta tal mudança ou se teremos que reescrever o método modificado. **Esse problema é da herança**, e não do polimorfismo, que resolveremos mais adiante com o uso de interfaces.

## Mais um exemplo:

Vamos modelar um sistema para o CEFET. Este sistema deve controlar as despesas com funcionários e professores. A classe FuncionarioDoCefet, que vai ser uma extensão da classe Funcionário, segue abaixo:

```
public class FuncionarioDoCefet extends Funcionario{
    private int matriculaSiape;

    public double getGastos(){
        return this.getSalario();
    }

    public String getInfo(){
        return "Nome: "+this.getNome()+", Salário: "+this.getSalario();
    }

    //Métodos get e set...
    public int getMatriculaSiape() {
        return matriculaSiape;
    }

    public void setMatriculaSiape(int matriculaSiape) {
        this.matriculaSiape = matriculaSiape;
    }
}
```

A classe acima serve para um funcionário comum do CEFET, mas o gasto que se tem com um professor não é apenas o seu salário. Temos que somar um bônus de R\$400,00 para gastos com alimentação. O que fazer então? Estender FuncionarioDoCefet e reescrever o método getGastos. Aliás, não só o getGastos. O getInfo também deve mostrar que 400,00 são referentes aos gastos com auxílio alimentação.

```
public class ProfessorDoCefet extends FuncionarioDoCefet {
    private double auxilioAlimentacao = 400.00;

    @Override
    public double getGastos() {
        return this.auxilioAlimentacao + super.getGastos();
    }

    @Override
    public String getInfo() {
        // TODO Auto-generated method stub
        return super.getInfo() + " ( " + this.auxilioAlimentacao + " referente ao auxílio alimentação )";
    }
}
```

A novidade aqui foi a palavra super. Apesar de termos reescrito o método, gostaríamos de invocar o método da classe mãe para não ter que copiar e colar a parte que nos interessava. Depois bastou concatenar com a informação nova e retornar.

Como tiramos proveito do Polimorfismo?  
Imagine que temos uma classe relatório:

```
public class GeradorDeRelatorio {
    FuncionarioDoCefet[] empregados;

    public GeradorDeRelatorio(FuncionarioDoCefet[] arrayDeFuncionarios) {
        this.empregados = arrayDeFuncionarios;
    }
    public void imprime() {
        for (FuncionarioDoCefet funcionarioDoCefet : empregados) {
            System.out.println(funcionarioDoCefet.getInfo());
            System.out.println(funcionarioDoCefet.getGastos());
            System.out.println("*****");
        }
    }
}
```

Perceba que, através do construtor da classe, podemos receber um array contendo elementos que podem conter qualquer funcionário do CEFET! Vai funcionar tanto para um FuncionárioDoCefet comum quanto para um ProfessorDoCefet.

Veja como utilizariam os estas classes:

```
public class TestaGeradorDeRelatorios {  
    public static void main(String[] args) {  
  
        ProfessorDoCefet p1 = new ProfessorDoCefet();  
        p1.setNome("Rafael");  
        p1.setSalario(3000);  
  
        FuncionarioDoCefet f1 = new FuncionarioDoCefet();  
        f1.setNome("João");  
        f1.setSalario(2000);  
  
        FuncionarioDoCefet f2 = new FuncionarioDoCefet();  
        f2.setNome("Maria");  
        f2.setSalario(2500);  
  
        // Este array pode receber FuncionárioDoCefet ou ProfessorDoCefet  
        // Olha o polimorfismo aí!!!  
        FuncionarioDoCefet[] funcionarios = new FuncionarioDoCefet[3];  
        funcionarios[0] = f2;  
        funcionarios[1] = p1;  
        funcionarios[2] = f1;  
  
        GeradorDeRelatorio gerador = new GeradorDeRelatorio(funcionarios);  
        // O método imprime vai tratar todos como FuncionarioDoCefet (polimorfismo)  
        gerador.imprime();  
    }  
}
```

Digamos que, no futuro, vamos aumentar nosso sistema e criar uma nova classe chamada CoordenadorDoCefet, que estende ProfessorDoCefet. Mesmo estendendo ProfessorDoCefet ele vai ser um FuncionárioDoCefet também.

Será que vamos precisar alterar alguma coisa na nossa classe GeradorDeRelatórios? Não. Essa é a grande vantagem! Quem programou a classe GeradorDeRelatório nem imaginou que surgiria uma classe CoordenadorDoCefet no futuro e mesmo assim o sistema funciona!

### Mais um pouco sobre herança, reescrita e polimorfismo...

Hoje em dia discute-se muito sobre o abuso no uso da herança. Procure evitar a herança por preguiça. Algumas pessoas usam a herança apenas para reaproveitar o código, quando poderiam ter feito uma **composição**. Pesquise sobre Herança X Composição.

No blog da Caelum há um artigo bem interessante sobre isso em:  
<HTTP://blog.caelum.com.br/2006/10/14/como-nao-aprender-orientacao-a-objetos-heranca/>

Leia também sobre “princípios de Coad”.

Existem testes para saber se é caso de herança. No caso do nosso banco, devemos perguntar:

Gerente é um **tipo especial de funcionário** ou um **papel assumido por** um funcionário?

No nosso caso é um **tipo especial de** funcionário. Logo é um caso de Herança.

Se fosse um papel assumido por um funcionário, talvez fosse um caso de usar interface (fazer uma composição). Essa discussão é mais avançada. Afinal, ainda não falamos sobre interface, que é um poderoso recurso do paradigma orientado a objetos.

Resumindo: **O Polimorfismo é o maior benefício da herança!**

Eu iria além e diria para vocês: Se não for para obter polimorfismo, não use herança!

## Exercícios sobre Herança e Polimorfismo

- 1) Nos exercícios anteriores escrevemos a classe TestaConta. Rode a classe novamente substituindo:

```
Conta c = new Conta();
ContaCorrente cc = new ContaCorrente();
ContaPoupanca cp = new ContaPoupanca();
```

Por:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual a utilidade disso?

Realmente essa não é a forma mais útil de polimorfismo. No próximo exercício veremos seu real poder. Contudo, como já vimos no array de FuncionarioDoCefet, existe uma vantagem em declarar uma variável (por exemplo: o array do tipo FuncionarioDoCefet[]) de um tipo menos específico do que o objeto realmente é.

É muito importante perceber que não importa como nos referimos a um objeto, o método invocado será sempre o mesmo. O do seu tipo real (quando ele foi criado). Em tempo de execução a JVM vai descobrir qual é o verdadeiro tipo do objeto e invocar o método daquele tipo e não do tipo que nos referimos a ele.

- 2) Vamos criar uma classe que seja responsável pela atualização de todas as Contas e gerar um relatório com o saldo anterior e o saldo novo de cada uma das contas.

```
public class AtualizadorDeContas {
    private double saldoTotal = 0;
    private double taxaPadrao;

    public AtualizadorDeContas(double taxaPadrao) {
        this.taxaPadrao = taxaPadrao;
    }

    public void roda(Conta c) {
        // aqui vc imprime o saldo anterior,
        // atualiza a conta
        // e depois imprime o saldo final
        // lembrando de somar o saldo final ao atributo saldoTotal
    }

    // outros métodos, colocar o getter para saldoTotal
}
```

- 3) Vamos criar a classe TestaAtualizadorDeContas, criar algumas Contas e rodá-las:

```
public class TestaAtualizadorDeContas {
    public static void main(String[] args) {
        Conta c = new Conta();
        Conta cc = new ContaCorrente();
        Conta cp = new ContaPoupanca();

        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);

        AtualizadorDeContas adc = new AtualizadorDeContas(0.01);

        adc.roda(c);
        adc.roda(cc);
        adc.roda(cp);

        System.out.println("Saldo Total: " + adc.getSaldoTotal());
    }
}
```

- 4) Crie uma classe Banco que possui um array de Conta. Perceba que num array de Conta você pode colocar tanto uma ContaCorrente quanto uma ContaPoupanca. A classe Banco deve receber, obrigatoriamente, a quantidade de contas que o array deve conter.
- 5) Crie um método void adiciona(Conta c), um método Conta getConta(int posicao) e outro int getTotalDeContas(), muito similar à relação anterior de Empresa-Funcionário.
- 6) Escreva um programa para testar a classe acima, criando várias contas e inserindo-as no Banco. Depois, com um for, percorra todas as Contas do Banco para passá-las como argumento para o método roda de um AtualizadorDeContas.
- 7) Suponha que você agora precisa criar uma classe ContaInvestimento, cujo método atualiza é complicadíssimo. Nesse caso você precisaria alterar as classes Banco e AtualizadorDeContas?
- 8) Existe Polimorfismo sem Herança?
- 9) É realmente necessário colocar os atributos como protected em herança? Preciso realmente afrouxar o encapsulamento do atributo por causa da herança? Como posso fazer para deixar os atributos como private na superclasse e as subclasses conseguirem de alguma forma trabalhar com eles?

## 44. Facilidades da IDE Eclipse

Diferente de uma ferramnta RAD, onde o objetivo é desenvolver o mais rápido possível, fazendo uso do “arrastar e soltar”, o Eclipse é uma IDE (*Integrated Development Enviroment*). Com uma ferramenta RAD, montanhas de código são geradas em *background*, independente da sua vontade. Já uma IDE te auxilia no desenvolvimento, sem fazer muita mágica e evitando intrometer-se.

O Eclipse é a IDE líder de mercado, por isso foi escolhida para o nosso curso. Formada por um consórcio liderado pela IBM, possui seu código livre. Atualmente estamos na versão 4.5. Precisamos do Eclipse 3.1 ou superior, pois a partir dessa versão é que a plataforma dá suporte ao Java 5.0. Atualmente estamos no Java 8.0 Você precisa ter apenas a JRE instalada. No nosso caso, já temos o JDK.

O foco do Eclipse é produtividade! Você vai perceber que ele evita ao máximo te atrapalhar e gera apenas trechos de código óbvios e ao seu comando. Existem também uma série de plugins gratuitos para gerar Diagramas UML, suporte a Servidores de Aplicação, visualizadores de Banco de dados, entre outros.

Baixe o Eclipse no site oficial ([www.eclipse.org](http://www.eclipse.org)). No nosso caso, por enquanto, precisaríamos apenas da versão Java SE, mas, como trabalharemos com a parte web posteriormente, mais adiante, acho mais prático já baixarmos a versão para Java EE. Você pode baixa-la no link: <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/mars1> Apesar de ser escrito em Java, a biblioteca gráfica usada no Eclipse chamada SWT, usa componentes nativos do Sistema Operacional. Por isso você deve baixar a versão correspondente ao seu Sistema Operacional. Descompacte o arquivo e pronto: basta rodar o executável. Neste ano trabalharemos com a IDE Eclipse Mars. Existem várias outras ( Galileo, Ganymed, Luna, Indigo, Juno, Kepler etc).

### Outras IDEs:

Uma outra IDE open source famosa é o NetBeans, da Sun. ([www.netbeans.org](http://www.netbeans.org)).

Nota: Hoje em dia a Sun pertence a Oracle.

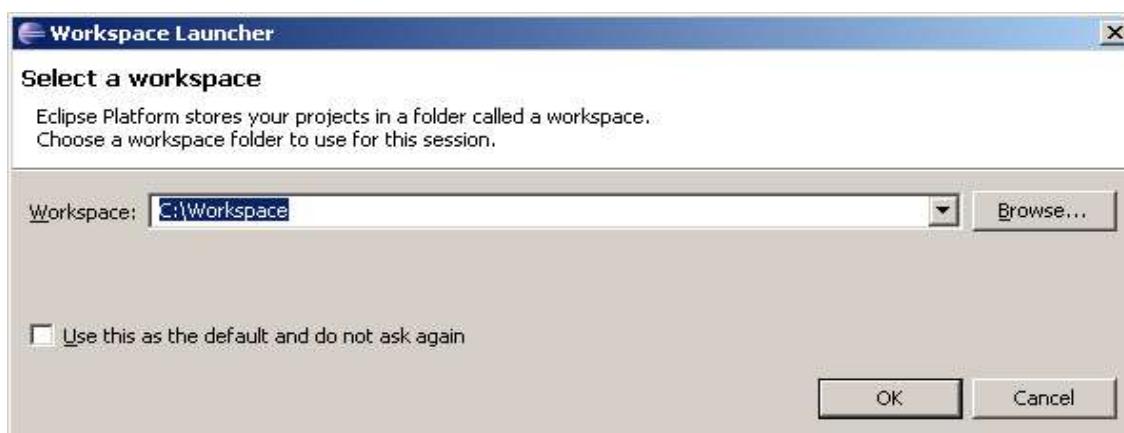
Particularmente eu prefiro o Eclipse. O Netbeans tem o “arrastar e soltar” que acaba gerando trechos de código por conta própria. O Eclipse nos deixa mais donos do nosso próprio código. Por isso é líder de mercado!

Oracle, Borland e IBM possuem IDEs comerciais e algumas versões mais restritas de uso livre.

Uma IDE paga que tenha ganho muitos adeptos ultimamente é o IntelliJ da empresa JetBrains.

### Apresentando o Eclipse:

Clique no ícone Eclipse.exe (dentro da pasta Eclipse). A primeira pergunta que você deve responder é que Workspace vai usar. Workspace é o diretório onde suas configurações pessoais e seus projetos serão gravados.



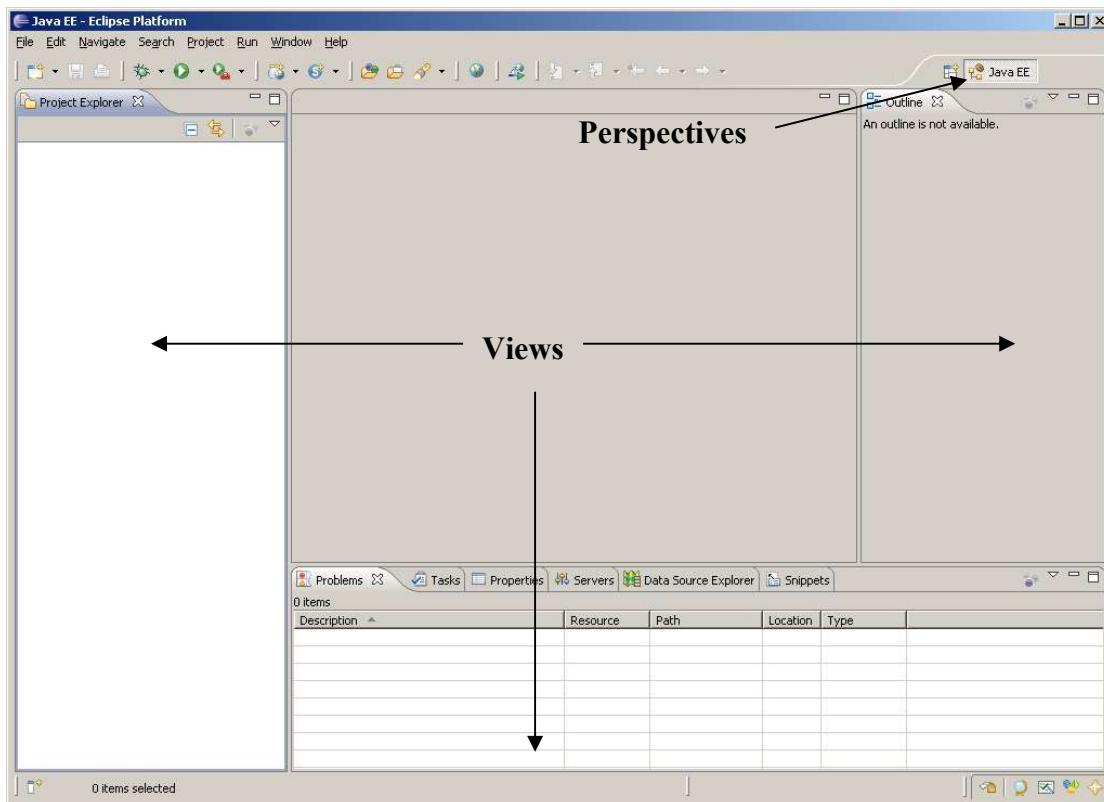
Você tem a opção de deixar o diretório pré-definido. No nosso caso, podemos criar um diretório WorkspaceEnsinoMedio dentro de c:\dev.

Logo em seguida, uma tela de Welcome será aberta, onde você tem diversos links para tutoriais e ajuda. Clique em WorkBench.



### Views e Perspective:

Na tela a seguir destacamos as views e as Perspectives do Eclipse.



Mude para a perspectiva Resource, clicando no ícone ao lado da Perspectiva Java, selecionando Other e depois Resource. Neste momento trabalharemos com essa perspectiva por possuir um conjunto de Views mais simples.

A view Navigator(à esquerda) mostra a estrutura do diretório, assim como está no sistema de arquivos. A view Outline(à direita) mostra um resumo das classes, interfaces e enumerações declaradas no arquivo java atualmente editado (também serve para outros tipos de arquivo).

No menu **Window → Show View → Other**, você pode conhecer as dezenas de views que já vem embutidas no Eclipse.

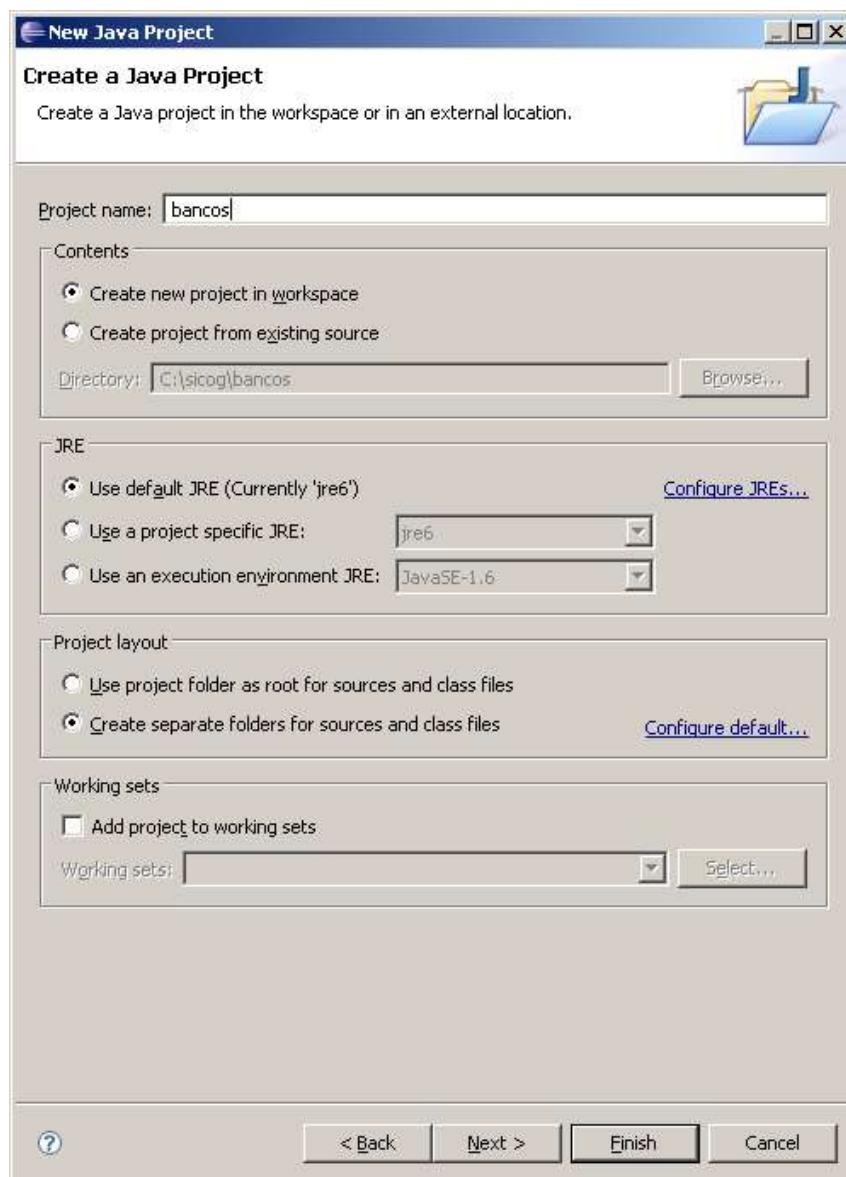
### Criando um novo Projeto:

Vá em **File → New → Project**. Selecione (ou digite) Java Project e clique em Next.



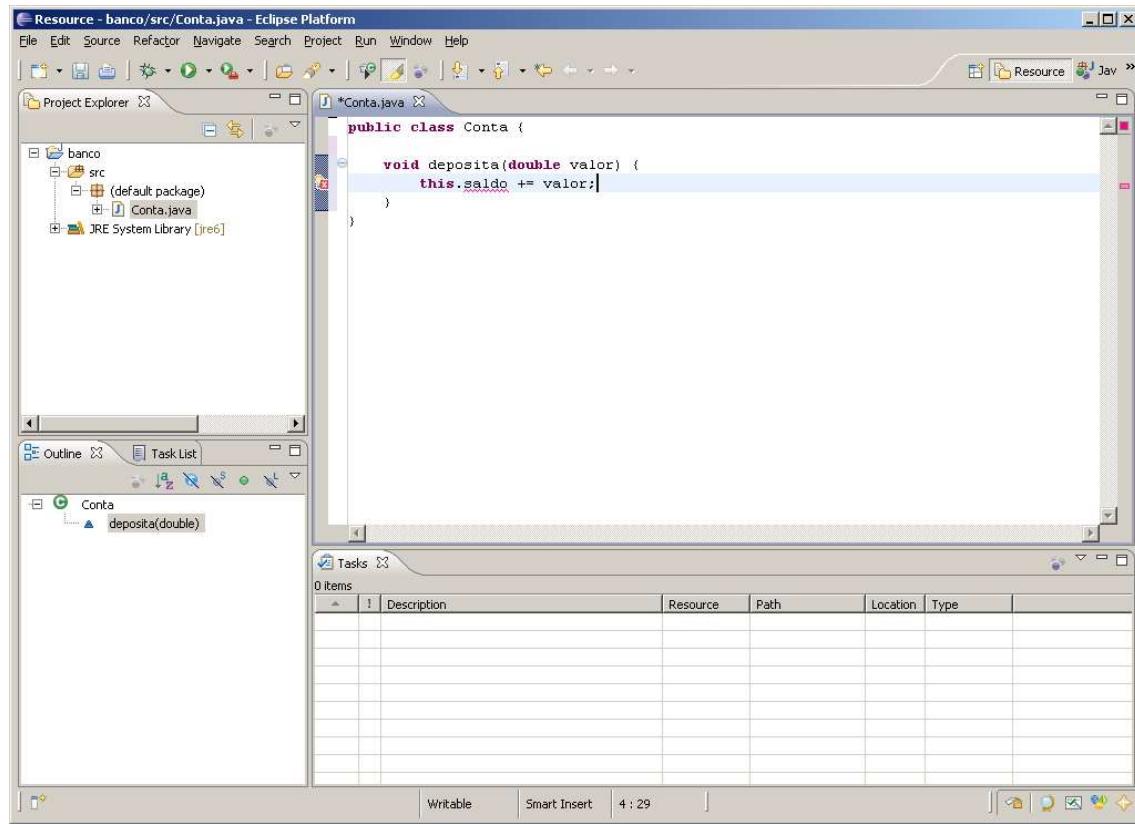
Crie um projeto chamado banco.

Você pode chegar nessa mesma tela clicando com o botão da direita do mouse no espaço da View Navigator e seguindo o mesmo menu. Nesta tela, configure seu projeto conforme tela abaixo:

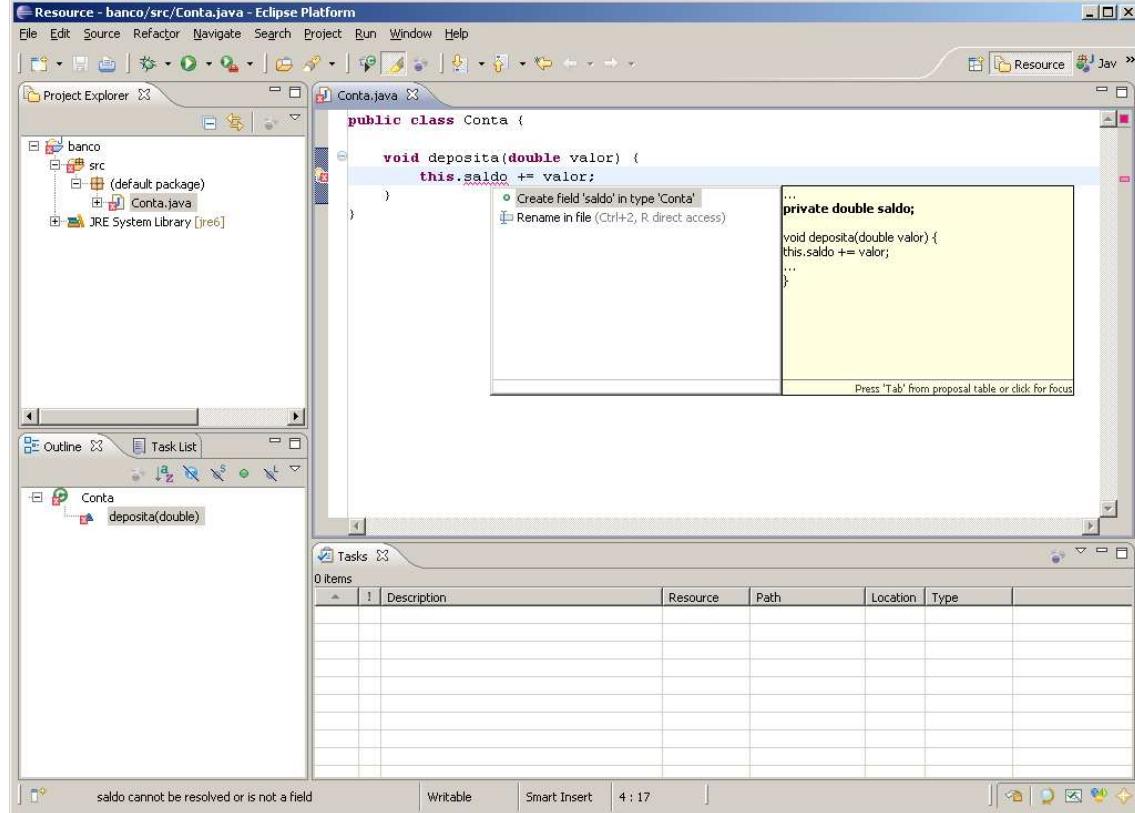


Clique em Finish. O Eclipse possui diversos wizards, mas usaremos o mínimo deles. O que interessa é usar o **code assist** e **quick fixes** que a ferramenta possui.

No menu à esquerda clique com o botão direito do mouse sobre banco, escolha **File → New → Class** e crie a classe Conta. Escreva o método deposita, conforme mostrado abaixo e perceba que o Eclipse reclama de erro em *this.saldo* pois este atributo não existe.



Vamos usar o recurso do Eclipse de quick fix. Coloque o cursor em cima do erro e aperte Ctrl+1.



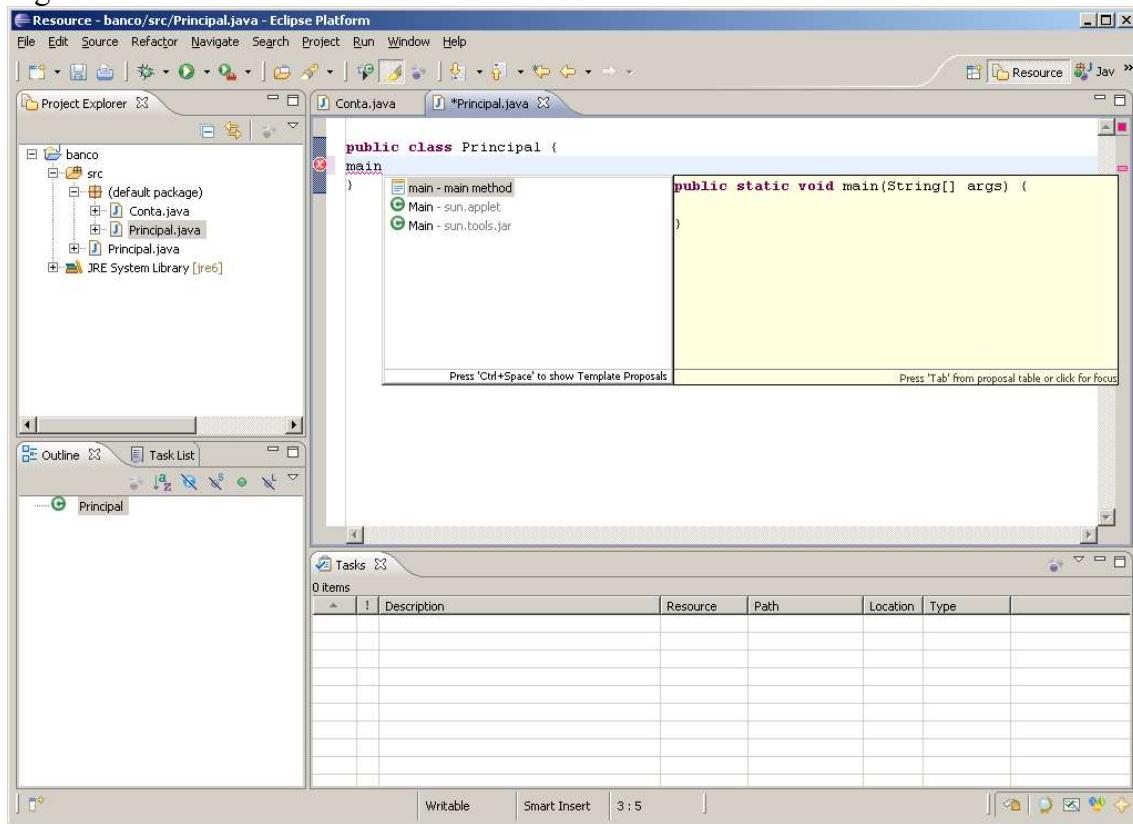
Perceba que ele vai sugerir possíveis formas de consertar o erro. Uma delas é, justamente, criar o campo saldo na classe Conta, que é nosso objetivo. Escolha esta opção e ele vai criar o campo saldo pra você.

Este recurso de quick fixes, acessível pelo Ctrl+1, é uma das facilidades do Eclipse e é extremamente poderoso. Através dele é possível corrigir boa parte dos erros na hora de programar e, como fizemos, economizar a digitação de certos códigos repetitivos. Veja que no nosso exemplo, não precisamos criar o campo antes. O Eclipse faz isso pra você. Ele até acerta o tipo de dado, já que estamos somando a ele um double. O private é colocado pelos motivos que já vimos anteriormente.

No menu, clique em **File → Save** para gravar. Ctrl+S tem o mesmo efeito.

### Criando o main:

Crie uma nova Classe chamada Principal. Vamos colocar um método main para testar nossa Conta. Só que ao invés de escrever todo o método, digite apenas main. Agora vamos usar o **code assist** do Eclipse. Escreva apenas main e Ctrl+Espaço logo em seguida.



O Eclipse vai sugerir a criação do método main completo. Escolha essa opção e ele vai escrever todo o método pra você! O Ctrl+Espaço aciona o **code assist** que é tão importante quanto os quick fixes.

Agora, dentro do método main, comece a digitar o seguinte código:

```
Conta conta = new Conta();
conta.deposita(100.0);
```

Observe que na hora de invocar o método, assim que você digita o ponto, o Eclipse sugere os métodos possíveis. Esse recurso é muito útil, principalmente quando formos programar classes que não fomos nós que criamos, como as da API do Java. Caso o Eclipse não se manifeste, esse recurso pode ser acionado com Ctrl+Espaço.

Agora, vamos imprimir o saldo com o System.out.println, mas, mesmo nesse código, o Eclipse nos ajuda. Digite apenas syso e em seguida Ctrl+Espaço e o Eclipse escreverá o método pra você! Para imprimir, chame conta.getSaldo().

```
Conta conta = new Conta();  
conta.deposita(100.0);  
System.out.println(conta.getSaldo());
```

Só para testar o poder do Eclipse. Entre na classe Conta e crie o atributo telefoneCliente conforme mostrado abaixo:

```
private String telefoneCliente;
```

Agora no main da Classe Principal digite o seguinte:

```
conta.setTelefoneCliente("2522-2900");
```

Clique sobre o erro e pressione Ctrl+1 e o Eclipse vai sugerir que você crie o método setTelefoneCliente na classe Conta. Aceite a sugestão e implemente o método. O Eclipse não vai fazer todo o trabalho pra você. De volta ao método main digite:

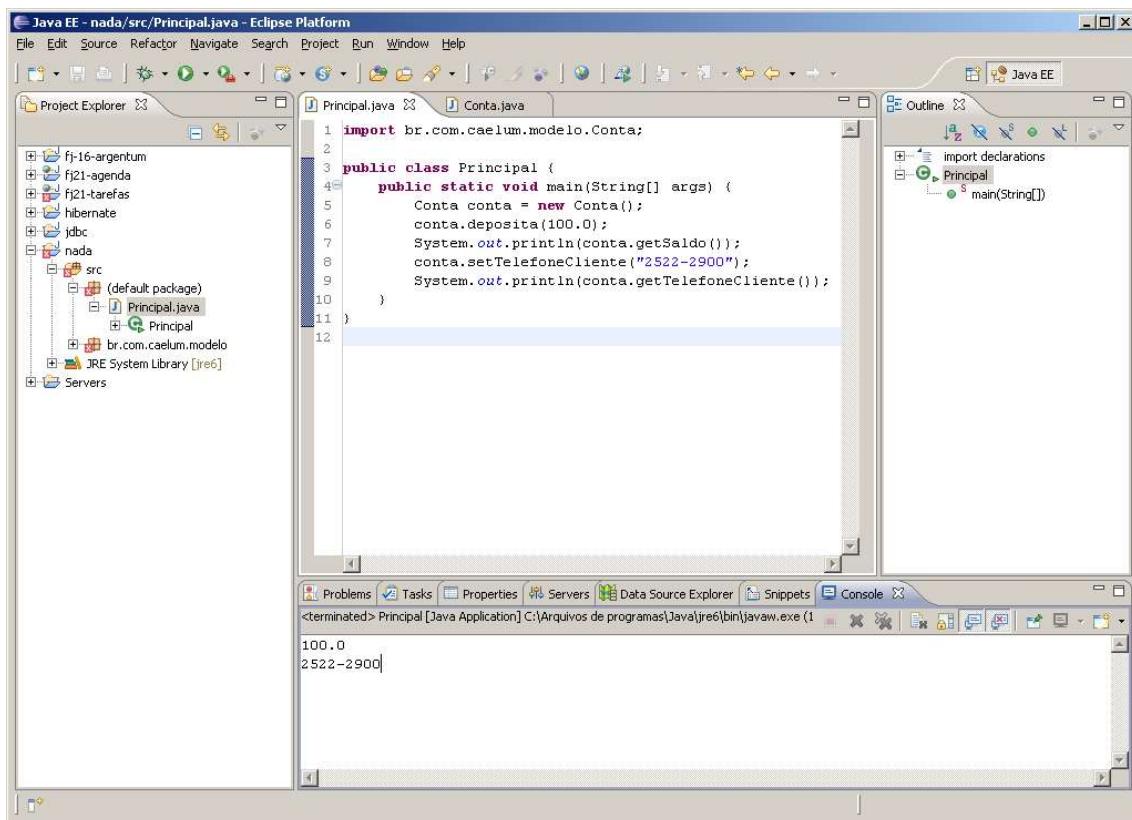
```
System.out.println(conta.getTelefoneCliente());
```

Use o Ctrl+1 para criar o método novamente.

Perceba o quanto isso é útil. Você pode programar sem se preocupar com os métodos que ainda não existem, já que em qualquer momento ele pode gerar o esqueleto( a parte da assinatura do método) pra você.

### Rodando o main:

Para rodar o método main da nossa classe, no Eclipse, clique com o botão direito do mouse sobre a classe Principal e em seguida escolha Run as... → Java Application. Na parte inferior da tela o Eclipse abrirá uma view chamada Console, onde será apresentada a saída do seu programa:



Quando você precisar rodar de novo, basta clicar no ícone verde de play na toolbar, que roda o programa anterior. Ao lado desse ícone há uma setinha onde são listados os últimos 10 executados.

### Outros atalhos do Eclipse:

O Eclipse possui muitos atalhos úteis ao programador. Veja alguns:

- **Ctrl + 1** → Aciona o quick fixes para correção de erros.
- **Ctrl + Espaço** → Completa códigos.
- **Ctrl + 3** → Aciona o modo de descoberta de menu. Experimente digitar Ctrl+3 e depois digitar ggas e enter. Ou então Ctrl+3 e digite new class.
- **Ctrl + Shift + F** → Formata o código segundo as convenções do Java. Faz a identação.
- **Ctrl + Shift + O** → Importa as classes citadas em nosso código.

Outros menos utilizados:

- **Ctrl + Shift + F11** → Roda novamente a última classe que você rodou. Equivale a clicar no botão de play (ícone verde).
- **Ctrl + O** → Exibe um Outline para rápida navegação.
- **Ctrl + Shift + L** → Exibe todos os atalhos possíveis.
- **Ctrl + M** → Expande a View atual para a tela toda. O mesmo que dar dois cliques no título da view.

Mais adiante, quando trabalharmos com pacotes, veremos outros atalhos.

## Exercícios com o Eclipse:

- 1) Dentro do projeto banco, crie as classes Cliente, Conta, ContaCorrente, ContaPoupanca e TestaContas. Na classe Conta crie os métodos atualiza, saca, deposita e transferePara como fizemos anteriormente. Desta vez, tente abusar do Ctrl+Espaço e do Ctrl+1.

Por exemplo:

ContaCorr<Ctrl+Espaço> <Ctrl+Espaço> = new <Ctrl+Espaço>();

Repare que até nome de variável ele cria pra você!

Digite:

New ContaCorrente();

Vá nessa linha e dê Ctrl+1. Ele vai sugerir e declarar a variável para você.

Segue uma “colinha” das Classes:

```
//Classe Cliente
public class Cliente {
    private String nome;

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return this.nome;
    }
}

//Classe Conta
public class Conta {
    protected double saldo;
    protected double limite;
    protected Cliente titular = new Cliente();

    public boolean deposita(double valor)
    {
        if(valor>0){
            this.saldo+=valor;
            return true;
        }
        return false;
    }

    public boolean saca(double valor){
        if(valor>0 && valor<=(this.saldo+this.limite)){
            this.saldo-=valor;
            return true;
        }else{
            return false;
        }
    }

    public boolean transferePara(Conta contaDestino,double valor){
        Conta contaOrigem = this;
        if(contaOrigem.saca(valor)){
            return contaDestino.deposita(valor);
        }
    }
}
```

```
        return false;
    }

    public void atualiza(double taxa) {
        this.saldo-=taxa;
    }

    //Métodos get e set....
    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }

    public double getSaldo() {
        return saldo;
    }

    public Conta() {
        this.limite = 100;
    }

    public Cliente getCliente(){
        return cliente;
    }

    public void setCliente(Cliente cliente){
        this.cliente = cliente;
    }

    public void exibirSaldo()
    {
        System.out.println("O saldo atual é: "+(saldo+limite));
    }
}

//Classe ContaCorrente
public class ContaCorrente extends Conta {
    @Override
    public void atualiza(double taxa) {
        this.saldo -= (2 * taxa);
    }

    @Override
    public boolean deposita(double valor) {
        if(valor>0){
            this.saldo+=valor;
            this.saldo-=0.10;
            return true;
        }
        return false;
    }
}

//Classe ContaPoupanca
public class ContaPoupanca extends Conta{
    @Override
    public void atualiza(double taxa) {
```

```
        this.saldo -= (3 * taxa);
    }

//Classe TestaContas
public class TestaContas {
    public static void main(String[] args) {
        Conta c = new Conta();
        ContaCorrente cc = new ContaCorrente();
        ContaPoupanca cp = new ContaPoupanca();

        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);

        System.out.println("Saldo das contas antes de atualizar:");
        System.out.println("Conta c: "+c.getSaldo());
        System.out.println("ContaCorrente cc: "+cc.getSaldo());
        System.out.println("ContaPoupanca cp: "+cp.getSaldo());

        c.atualiza(0.01);
        cc.atualiza(0.01);
        cp.atualiza(0.01);

        //System.err faz imprimir em vermelho
        System.err.println("Saldo das contas depois de
atualizar:");
        System.err.println("Conta c: "+c.getSaldo());
        System.err.println("ContaCorrente cc: "+cc.getSaldo());
        System.err.println("ContaPoupanca cp: "+cp.getSaldo());
    }
}
```

- 2) Imagine que queiramos criar um setter do saldo para a classe Conta. Dentro da classe Conta digite setSa<Ctrl+Espaço>  
O mesmo vale para reescrever um método. Dentro de ContaPoupanca digite depo<Ctrl+Espaço>

Agora apague os dois métodos que acabamos de criar. Foi só para praticar o uso do Eclipse.

- 3) Vá na classe TestaContas, que tem o método main, e segure o Ctrl apertado enquanto você passa o mouse sobre o seu código. Repare que tudo virou hyperlink. Clique em um dos métodos e veja o que acontece.
- 4) Use o Ctrl+Shift+F para formatar e indentar o seu código.
- 5) Crie no seu projeto as classes AtualizadorDeContas, Banco e TestaAtualizadorDeContas:

```
//Classe AtualizadorDeContas
public class AtualizadorDeContas {
    private double saldoTotal = 0;
    private double selic;

    public AtualizadorDeContas(double selic) {
        this.selic = selic;
    }
```

```
public void roda(Conta c) {
    // aqui vc imprime o saldo anterior,
    System.out.println("Saldo anterior: "+c.getSaldo());
    // atualiza a conta
    c.atualiza(this.selic);
    // e depois imprime o saldo final
    System.out.println("Saldo depois do atualiza:
"+c.getSaldo());
    // lembrando de somar o saldo final ao atributo saldoTotal
    this.saldoTotal+=c.getSaldo();
}

// outros métodos, colocar o getter para saldoTotal
public double getSaldoTotal() {
    return saldoTotal;
}

//Classe Banco
import java.util.Arrays;

public class Banco {
    private Conta[] contas;
    private int posicaoLivre = 0;

    public Banco(int numeroDeContas) {
        this.contas = new Conta[numeroDeContas];
    }

    public void adiciona(Conta c){
        if(! (this.contas.length<posicaoLivre) )
            this.redimensiona();
        contas[this.posicaoLivre]=c;
        this.posicaoLivre++;
    }

    private void redimensiona() {
        this.contas = Arrays.copyOf(this.contas,
this.contas.length+1);
    }

    public Conta pegaConta(int posicao){
        if(contas[posicao]!=null)
            return contas[posicao];
        return null;
    }

    public int getTotalDeContas(){
        return this.contas.length;
    }

    public Conta[] getContas() {
        return contas;
    }

    public void setContas(Conta[] contas) {
        this.contas = contas;
    }
}

//Classe TestaAtualizadorDeContas
```

```
public class TestaAtualizadorDeContas {
    public static void main(String[] args) {
        Conta c = new Conta();
        Conta cc = new ContaCorrente();
        Conta cp = new ContaPoupanca();

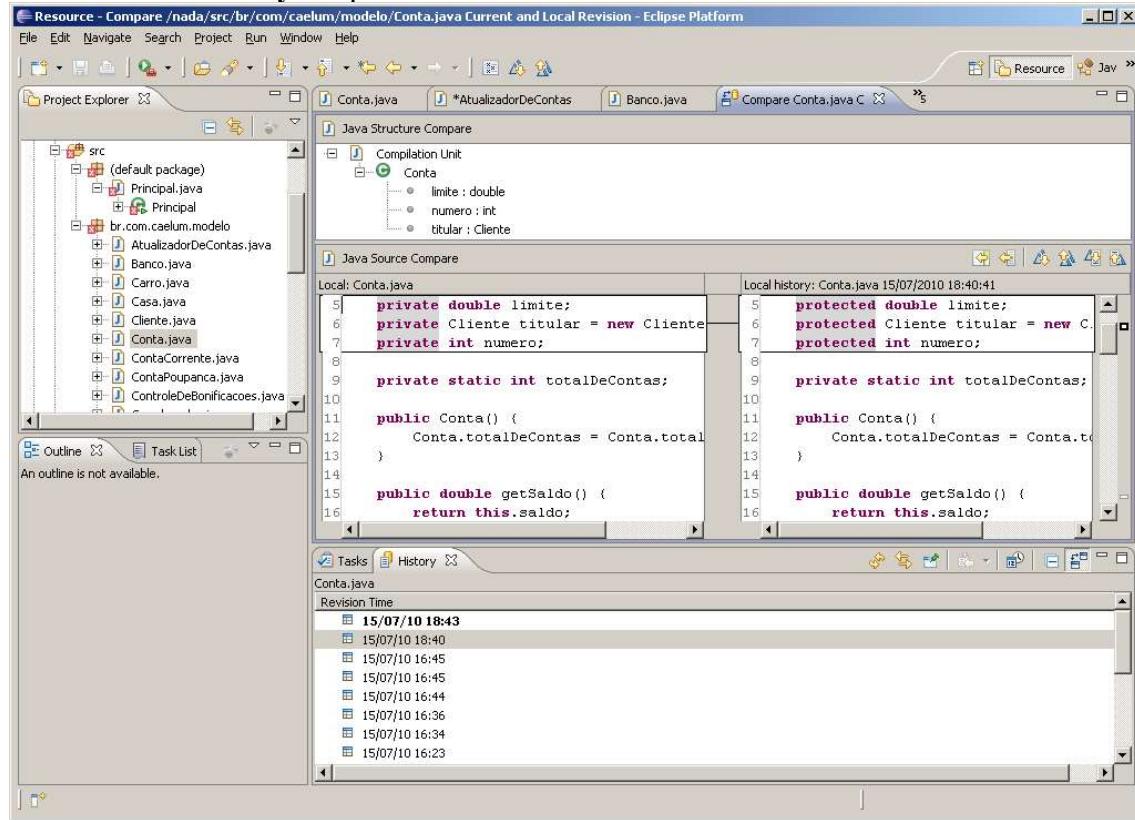
        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);

        AtualizadorDeContas adc = new AtualizadorDeContas(0.01);

        adc.roda(c);
        adc.roda(cc);
        adc.roda(cp);

        System.out.println("Saldo Total: " + adc.getSaldoTotal());
    }
}
```

- 6) Dê um clique da direita em um arquivo no navigator. Escolha **Compare With → Local History**. O que é está tela?



- 7) Clique com o botão direito do mouse sobre o projeto, em seguida escolha propriedades. É uma das telas mais importantes do Eclipse. Aqui você pode configurar uma série de funcionalidades para o seu projeto.
- 8) Pesquise sobre refactoring (refatoração).

### Mais exercícios:

- 1) Crie o projeto projeto-funcionarios com as seguintes classes:

```
//Data
public class Data {
    private String dia, mes, ano;

    public String getData() {
        return this.dia + "/" + this.mes + "/" + this.ano;
    }

    public String getDia() {
        return dia;
    }

    public void setDia(String dia) {
        this.dia = dia;
    }

    public String getMes() {
        return mes;
    }

    public void setMes(String mes) {
        this.mes = mes;
    }

    public String getAno() {
        return ano;
    }

    public void setAno(String ano) {
        this.ano = ano;
    }
}

//Funcionario
public class Funcionario {

    protected String nome, departamento;
    protected String cpf;
    protected double salario;
    protected boolean ativo;
    protected Data dataDeNascimento = new Data();
    protected static int identificador;

    public Funcionario() {
        //..
    }

    public Funcionario(String nome) {
        this.nome=nome;
        identificador++;
    }
    //.....

    public double getBonificacao(){
        return this.salario * 0.10;
    }
}
```

```
public static int getIdentificador() {
    return Funcionario.identificador;
}

public String getCpf() {
    return cpf;
}

public void setCpf(String cpf) {
    if(this.valida(cpf))
        this.cpf = cpf;
}

private boolean valida(String cpf) {
    return (cpf.length()==11)?true:false;
}

public void mostra(){
    System.out.println("NOME: "+this.nome);
    System.out.println("SALÁRIO: "+this.salario);
    System.out.println("NASCIMENTO:
"+this.dataDeNascimento.getData());
    String situacao = (ativo==true)?"SIM":"NÃO";
    System.out.println("ESTÁ NA EMPRESA? "+situacao);
}

public void aumentarSalario(double percentual){
    if(percentual>0)
        this.salario+=((this.salario*percentual)/100);
}

public void demite(){
    this.ativo=false;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getDepartamento() {
    return departamento;
}

public void setDepartamento(String departamento) {
    this.departamento = departamento;
}

public double getSalario() {
    return salario;
}

public void setSalario(double salario) {
    this.salario = salario;
}

public boolean isAtivo() {
    return ativo;
```

```
    }

    public Data getDataDeNascimento() {
        return dataDeNascimento;
    }

    public void setDataDeNascimento(Data dataDeNascimento) {
        this.dataDeNascimento = dataDeNascimento;
    }
}

//Gerente
public class Gerente extends Funcionario {
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public Gerente(String nome) {
        super(nome);
    }

    public Gerente() {
    }

    public double getBonificacao() {
        return this.salario * 0.15;
    }

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println(this.cpf);
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado.");
            return false;
        }
    }

    // métodos get e set...

    public int getSenha() {
        return senha;
    }

    public void setSenha(int senha) {
        this.senha = senha;
    }

    public int getNumeroDeFuncionariosGerenciados() {
        return numeroDeFuncionariosGerenciados;
    }

    public void setNumeroDeFuncionariosGerenciados(int
numeroDeFuncionariosGerenciados) {
        this.numeroDeFuncionariosGerenciados =
numeroDeFuncionariosGerenciados;
    }
}
```

```
//Caixa
public class Caixa extends Funcionario {
    private int numeroDoGuiche;

    @Override
    public double getBonificacao() {
        return this.salario * 0.12;
    }

    public int getNumeroDoGuiche() {
        return numeroDoGuiche;
    }

    public void setNumeroDoGuiche(int numeroDoGuiche) {
        this.numeroDoGuiche = numeroDoGuiche;
    }
}

//ControleDeBonificacoes
public class ControleDeBonificacoes {

    private double totalDeBonificacoes;

    public void registra(Funcionario f){
        this.totalDeBonificacoes+=f.getBonificacao();
    }

    public double getTotalDeBonificacoes() {
        return totalDeBonificacoes;
    }
}

//TestaControleDeBonificacoes
public class TestaControleDeBonificacoes {
    public static void main(String[] args) {
        ControleDeBonificacoes controleDeBonificacoes = new
        ControleDeBonificacoes();

        Funcionario funcionario = new Funcionario();
        Gerente gerente = new Gerente();

        funcionario.setSalario(2000);
        gerente.setSalario(3000);

        controleDeBonificacoes.registra(funcionario);
        controleDeBonificacoes.registra(gerente);

        System.out.println(controleDeBonificacoes.getTotalDeBonificacoes
());
    }
}
```

## 45.Classes Abstratas

Volte algumas páginas e relembre como ficaram as classes Funcionario, Gerente, Caixa e ControleDeBonificações.

## O que é realmente mais importante, herança ou polimorfismo?

Repare no método contabiliza. Ele recebe como argumento qualquer referência do tipo Funcionário, ou seja, pode receber tanto Funcionário quanto qualquer uma de suas subclasses que já existem ou possam vir a ser criadas sem prévio conhecimento de quem escreveu ControleDeBonificacoes.

No caso acima, estamos utilizando a classe Funcionário para polimorfismo. Se ela não existisse teríamos um prejuízo enorme: seria necessário criar um método contabiliza para cada um dos tipos de Funcionário, um para Gerente, um para Caixa, um para Diretor e assim por diante. Perceba que perder esse poder é muito pior do que perder a pequena vantagem que a herança traz em herdar código.

No entanto, em alguns Sistemas, inclusive no nosso caso, usamos uma classe apenas para isso: **economizar código e ganhar polimorfismo para criar métodos mais genéricos que se encaixem a diversos objetos**.

Faz sentido ter uma classe Funcionário para se beneficiar com polimorfismo, mas, faz sentido instanciar (criar) um objeto do tipo Funcionário?

No mundo real, não existe uma pessoa que é simplesmente um Funcionário do banco. Ela é um Funcionário de algum tipo: ou é Gerente, ou Caixa, ou Diretor, enfim.

Referenciando Funcionário temos o polimorfismo de referência, já que podemos receber qualquer coisa que seja um Funcionário. Porém, dar new em Funcionário não faz muito sentido, certo? Não queremos receber uma referência que seja um Funcionário. Queremos que essa referência seja um Caixa, Diretor, um Gerente, algo mais **concreto** do que um Funcionário.

Veja o código abaixo:

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();
Funcionario f = new Funcionario();
controle.contabiliza(f); //Isso faz sentido?????
```

### Outros exemplos:

Imagine a classe Pessoa e duas filhas: PessoaFisica e PessoaJuridica. Existe uma pessoa que não seja física ou Jurídica? Nesse caso, a classe Pessoa estaria sendo utilizada apenas para ganhar o polimorfismo e um pouco de código que vai ser herdado por suas filhas. Não faz sentido permitir instanciá-la. Para resolver esse problema é que existem as **classes abstratas**.

### Classes abstratas:

A classe Funcionário é o que exatamente? Nossa banco possui apenas Gerentes, Diretores, Caixas, Tesoureiros etc. Funcionário é uma classe que apenas idealiza um tipo, define apenas um rascunho. Na verdade, a classe Funcionário é **apenas um conceito** do que deve ser um Funcionário.

No nosso sistema é inadmissível que um objeto seja apenas do tipo Funcionário. Queremos objetos que especifiquem qual é o tipo de Funcionário. Funcionário é só um conceito (uma abstração) e é muito genérico.

Então como impedir que um Funcionário seja instanciado (criado)? No Java usamos a palavra-chave **abstract** para impedir que uma classe possa ser instanciada. Veja:

```
abstract class Funcionario {
    protected String nome, cpf;
```

```
protected double salario;
boolean estaNaEmpresa = true;
//Métodos ....
}
```

Agora, o código abaixo nem compila:

```
Funcionario f = new Funcionario();
```

O problema é instanciar a classe. Criar referência você pode. Se ela não pode ser instanciada, para que serve? Simples, serve para podermos usar o polimorfismo e herdar atributos e métodos! Serve para colocarmos aquela “plquinha” na porta. Lembra?

Uma coisa que precisa ficar bem clara é que a decisão de transformar a classe Funcionário foi tomada com base nas nossas regras de negócio. Pode haver outro Sistema onde faça sentido instanciar um objeto do tipo Funcionário.

### Métodos abstratos:

Se na classe Gerente, por exemplo, o método getBonificação não fosse reescrito, seria herdado da classe mãe, fazendo com que devolvesse 10% do salário.

Se considerarmos que cada subtipo de Funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado, faz algum sentido ter esse método **implementado** na classe Funcionário? Será que existe uma bonificação padrão para todo tipo de Funcionário? Não. De acordo com o sistema cada subtipo calcula a bonificação de uma forma diferente. Se resolvemos escrever uma nova subclasse de Funcionário, precisaremos reescrever o método getBonificação de acordo com suas regras.

Poderíamos então excluir esse método na classe Funcionário? Para ajudar na resposta vamos relembrar abaixo o trecho de código que escrevemos na classe ControleDeBonificacoes:

```
public void contabiliza(Funcionario funcionario) {
    this.totalDeBonificacoes += funcionario.getBonificacao();
}
```

A resposta é não! Se apagarmos o método, matamos nosso polimorfismo. Afinal, com uma referência do tipo Funcionário, não poderíamos mais chamar **getBonificação**. E tem mais: se o método não existe na classe mãe, quem garante que vai existir nas filhas?

Então como fazer?

Calma, **nem tudo está perdido!** No Java existe um recurso em que dizemos que determinado método será sempre escrito (**implementado**) pelas classes filhas sem que tenhamos que programá-lo na classe mãe. Trata-se de um **método abstrato**.

Ele indica que todas as classes filhas deverão reescrever esse método ou não compilão. Elas vão herdar a responsabilidade de ter aquele método implementado ou herdado.

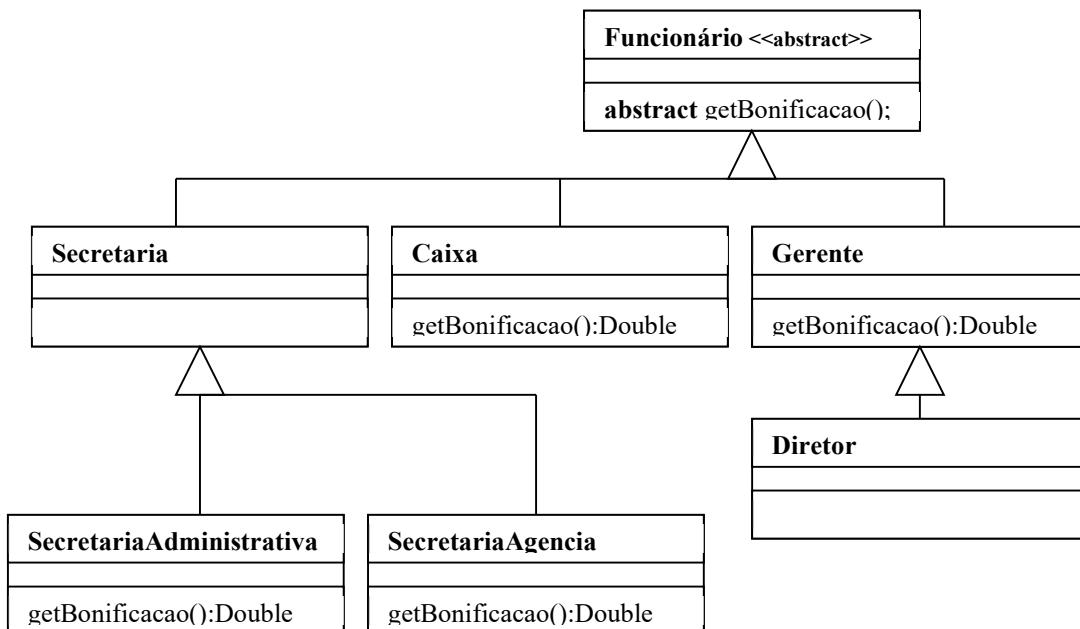
Mas então eu continuo tendo que escrever um método getBonificação na classe Funcionário? Não necessariamente. Basta escrever a palavra chave **abstract** antes da assinatura do método e colocar um ponto e vírgula em vez de abre e fecha chaves. Você só escreve a assinatura. **A implementação do método fica a cargo das subclasses!**

```
abstract double getBonificacao();
```

Como esse método nunca vai ser invocado por um objeto do tipo Funcionário, não há necessidade de que, na classe Funcionário, ele tenha um corpo.

**Como posso chamar o método getBonificação se ele não existe na classe Funcionário?** A resposta é simples. A classe Funcionário não lhe dá o método getBonificação, mas lhe dá a garantia de que suas filhas terão esse método e isso basta!

### Aumentando o exemplo:



Vamos analisar o diagrama acima: Temos uma subclasse Diretor que estende Gerente que por sua vez estende Funcionário. A subclasse diretor não possui o método `getBonificacao()`. Temos também a subclasse Secretaria (abstrata) que não tem `getBonificação` e tem as filhas SecretariaAdministrativa e SecretariaAgencia. Ambas tem o método `getBonificação`. Analisando todo o diagrama eu pergunto: Vai compilar? Vai rodar? A resposta é sim!

Nesse diagrama hierárquico acima podemos perceber que Secretaria é uma classe abstrata e, portanto, está delegando a obrigação de implementar o método `getBonificação` para suas filhas. Diretor não tem `getBonificação`, mas herda de Gerente. Isso quer dizer que a bonificação para um Diretor segue os mesmos padrões de um Gerente. Se a regra fosse diferente, bastaria reescrever o método na classe Diretor.

### Mais coisas a saber:

No exemplo acima vimos que uma classe abstrata pode estender outra classe abstrata e nesse caso ficar isenta de implementar um método abstrato da classe mãe, delegando esse trabalho para suas filhas.

O que você precisa saber é que uma classe que estende uma classe normal também pode ser abstrata. Ela não poderá ser instanciada, mas sua classe mãe e suas filhas sim!

Uma classe abstrata **não precisa** ter necessariamente um método abstrato.

- 1) A partir do projeto-bancov-07, crie o projeto-bancov08.
- 2) Como já vimos, não faz sentido instanciar um objeto do tipo Funcionario. Cuide para que isso não seja mais possível.
- 3) Já que nunca teremos um objeto do tipo Funcionario em memória, faz sentido termos um método getBonificacao() nessa classe? Podemos retirar esse método? A resposta é não.
- 4) Você não precisa ter o método acima implementado, mas precisa garantir que todas as filhas de Funcionario terão uma implementação para esse método. Resolva isso!

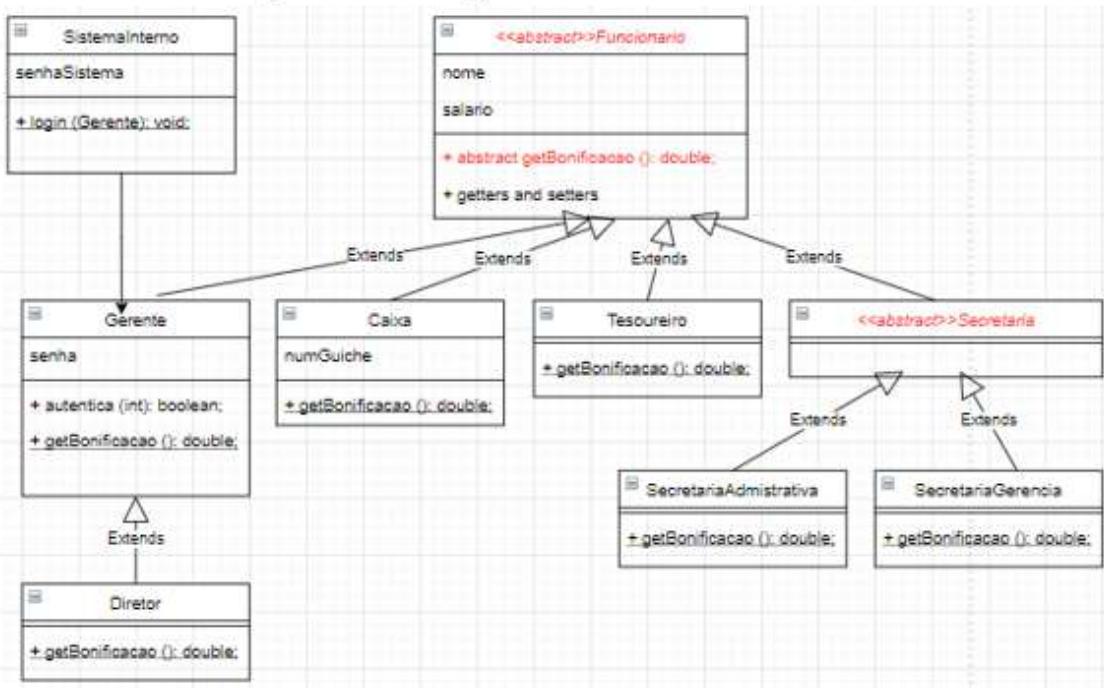
### Exercícios sobre classes abstratas:

- 1) A partir do projeto-bancov08, crie o projeto-bancov09.
- 2) Repare na nossa classe Conta. É uma excelente candidata a classe abstrata. Por quê? Que métodos seriam candidatos a método abstrato? Transforme a classe Conta em abstrata, repare o que acontece no seu main da classe TestaContas.
- 3) Para que o código do main volte a compilar troque o new Conta() por new ContaCorrente(). Se agora não podemos dar new em Conta, qual é a utilidade de ter um método que recebe uma referência a uma Conta como argumento? Aliás, posso ter isso?
- 4) Para entender melhor o abstract vamos fazer algumas modificações em nossas classes.  
Remova o método atualiza() da classe ContaPoupanca, dessa forma ela herdará o método de Conta.  
Transforme o método atualiza() da classe Conta em abstrato. Repare que, ao colocar a palavra chave abstract ao lado do método, o Eclipse rapidamente vai sugerir que você remova o corpo (body) do método com um quick fix.  
Qual é o problema com a classe ContaPoupanca agora?

- 5) Reescreva o método atualiza na classe ContaPoupanca para que a classe possa compilar normalmente. O Eclipse também sugere isso com um quick fix.
- 6) Existe outra maneira da classe ContaCorrente compilar se você não escrever o método abstrato?
- 7) Para que ter o método atualiza na classe Conta se ele não faz nada? O que acontece se simplesmente apagarmos esse método da classe Conta e o deixarmos em suas filhas?
- 8) Posso chamar um método abstrato de dentro de outro método da própria classe abstrata? Exemplo: O mostra() de Funcionário pode chamar o this.getBonificação?
- 9) Não podemos dar new em Conta, mas porque então, podemos dar new em Conta[10], por exemplo?

## 47. Herança e classes abstratas. Um pouco mais...

Vamos imaginar que um Sistema de Controle do Banco pode ser acessado pelo Gerente e agora também pelo Diretor do Banco. Vamos considerar o sistema hierárquico abaixo.



Exercícios:

- 1) Utilize o projeto-banco-v10 fornecido pelo professor e observe que as classes Gerente e Diretor já estão prontas.

```
3 public class Gerente extends Funcionario {  
4     private int senha;  
5  
6     public Gerente(String nome, int senha) {  
7         super.setNome(nome);  
8         this.setSenha(senha);  
9     }  
10  
11    public void setSenha(int senha) {  
12        this.senha = senha;  
13    }  
14    public int getSenha() {  
15        return senha;  
16    }  
17  
18    public boolean autentica(int senha) {  
19        if(this.senha == senha)  
20            return true;  
21        return false;  
22    }  
23 //.....  
//.....  
3 public class Diretor extends Gerente {  
4  
5     public Diretor(String nome, int senha) {  
6         super(nome, senha); //Compatibilidade de Construtores  
7     }  
8     @Override  
9     public double getBonificacao() {  
10        return this.salario * 0.5;  
11    }  
12 }
```

- 2) Crie a classe SistemaInterno e seu controle. Precisamos receber um Gerente ou um Diretor como argumento, verificar se ele se autentica e permitir seu acesso ao Sistema.

```
3 public final class SistemaInterno {  
4     private int senhaDoSistema;  
5  
6     public SistemaInterno(int senhaDoSistema) {  
7         this.senhaDoSistema = senhaDoSistema;  
8     }  
9  
10    public void login(Gerente g) {  
11        if(g.autentica(this.senhaDoSistema))  
12            System.out.println("Acesso liberado");  
13        else  
14            System.out.println("Acesso negado");  
15    }  
16 }
```

O SistemaBancario aceita apenas o tipo de Gerente. Como todo Gerente tem o método autentica, tudo vai funcionar corretamente.

- 3) Crie a classe TestaSistemaInterno. Instancie dois gerentes e um diretor e verifique se tudo funciona normalmente.

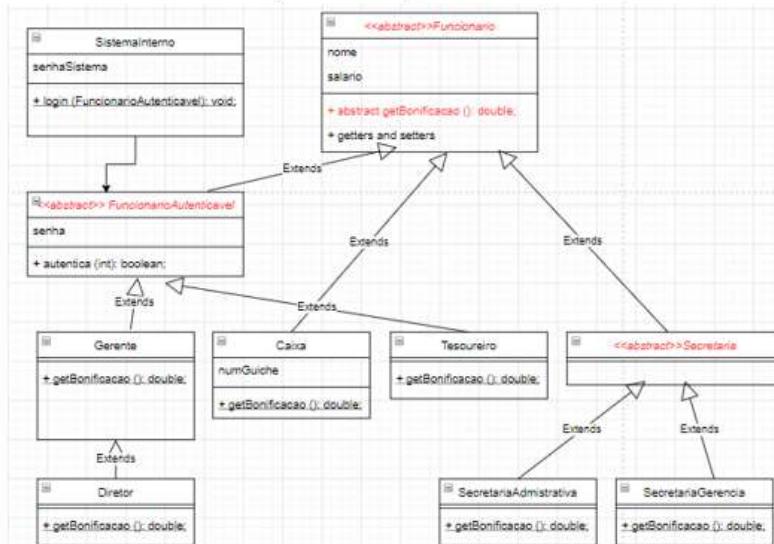
```

7 public class TestaSistemaInterno {
8
9     public static void main(String[] args) {
10         SistemaInterno si = new SistemaInterno(123);
11         Gerente g1 = new Gerente("Fulano", 123);
12         Gerente g2 = new Gerente("Maria", 124);
13         Diretor d1 = new Diretor("Paula", 123);
14
15         si.login(g1);
16         si.login(g2);
17         si.login(d1);
18     }
19 }
```

- Novo problema: Agora precisamos que um Tesoureiro também possa se autenticar em nosso Sistema Interno. Como resolver esse problema? Podemos até colocar um atributo senha e um método autentica em tesoureiro, mas... ainda assim, um Tesoureiro NÃO É um Gerente e logo, não pode ser aceito em nosso método login de SistemaInterno.
- O que fazer então?

Para resolver este problema, podemos fazer uso dos conhecimentos adquiridos anteriormente em relação a classes abstratas e polimorfismo. Veja uma possível solução no diagrama de classes abaixo.

FuncionarioAutenticavel pode se logar no SistemaInterno



Perceba que criamos uma camada extra entre Gerente e Funcionário e entre Tesoureiro e Funcionário. Pegamos toda a lógica de autenticação e jogamos para a

classe FuncionarioAutenticavel. Como nunca precisaremos instanciar um objeto desse tipo, obviamente trata-se de uma classe abstrata. Ganhamos polimorfismo!

A partir de agora, o método login de SistemaInterno espera receber um FuncionarioAutenticavel e não um Gerente.

Perceba que não forçamos a barra em relação à herança. Afinal, Gerente é um Gerente, um FuncionarioAutenticavel e um Funcionario, ou seja, um Funcionário que pode se autenticar no Sistema. O mesmo acontece com Tesoureiro. Diretor é um Gerente. Logo também é um FuncionarioAutenticavel e um Funcionario. Todo FuncionárioAutenticavel não deixa de ser um Funcionario.

**IMPORTANTE:** Como a lógica de autenticação é a mesma para Gerente, Diretor e Tesoureiro, podemos evitar o espalhamento de regras concentrando o atributo senha (com seus métodos get e set) e a implementação do método autentica em FuncionarioAutenticavel. Tesoureiro, Gerente e Diretor herdarão tudo isso e a regra estará em um único lugar.

**RESUMINDO:** Resolvemos nosso problema sem grandes traumas utilizando polimorfismo e classe abstrata!

## 48.Exercícios práticos 10

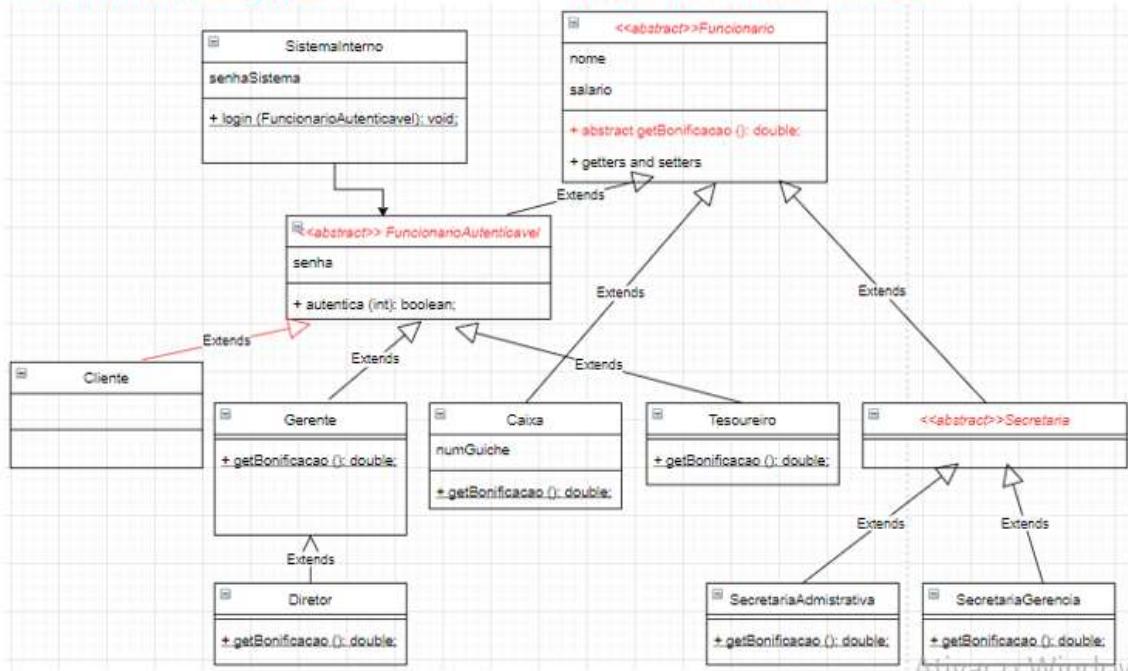
- 1) A partir do projeto-bancov10, crie o projeto-bancov11 e faça as modificações propostas acima.
- 2) Não se esqueça de Instanciar um Tesoureiro na classe TestaSistemaInterno e submetê-lo ao método login de SistemaInterno.

**Vamos imaginar uma situação mais complexa:** Agora precisamos que todos os clientes também tenham acesso ao SistemaBancario. E agora? O que faremos? Uma hipótese seria criar outro método login dentro de SistemaBancario, mas isso nós já descartamos anteriormente.

Uma outra hipótese que é comum entre os programadores inexperientes é fazer uma herança sem sentido para resolver o problema. Por exemplo, escrever Cliente extends FuncionarioAutenticavel. Realmente resolve o problema, mas trará diversos outros. Cliente definitivamente não é sequer um Funcionário, muito menos um FuncionarioAutenticavel. Se você fizer essa “gambiarrinha”, o Cliente terá um método getBonificacao, um atributo salário e outros atributos e métodos que não fazem o menor sentido para esta classe!

**Regra importante:** Não faça herança quando a relação não for “é um”.

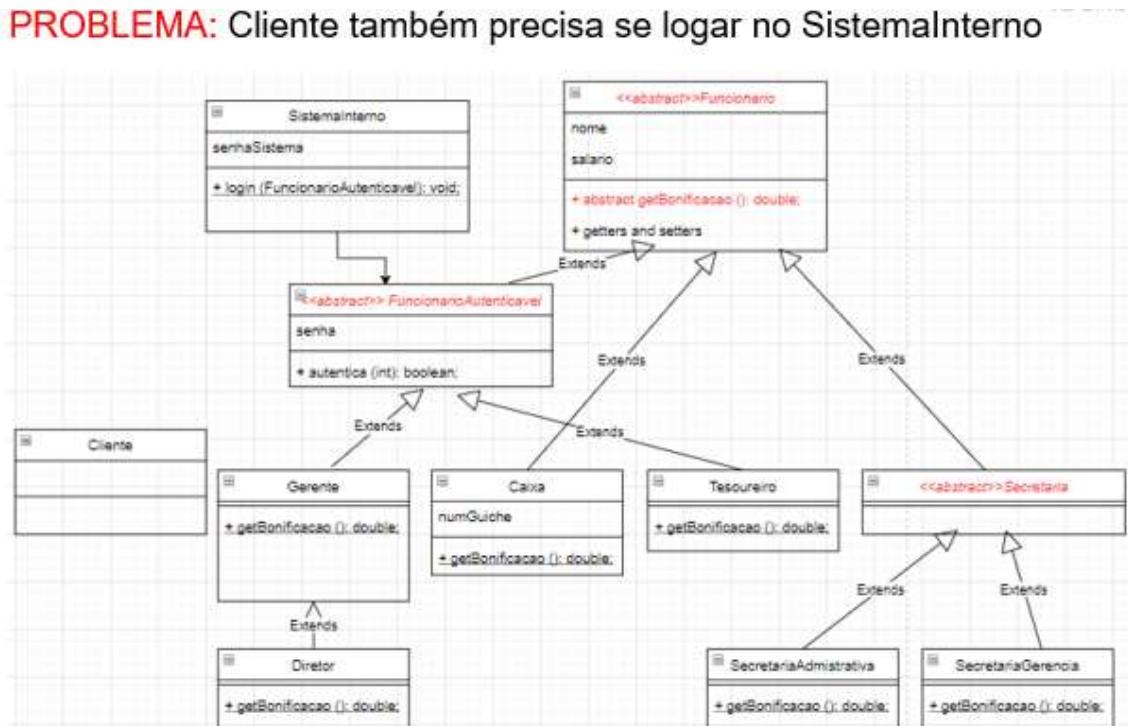
## Possível abordagem: Cliente é um FuncionarioAutenticavel ????



**Como resolvemos a situação então?** Veja que conhecer a sintaxe da linguagem não é o suficiente, precisamos estruturar / desenhar bem nossa estrutura de classes. O diagrama de classes é essencial para termos uma bela visão do todo e ver se faz sentido.

## 49. Interfaces

**Antes de tudo, voltemos ao design anterior, mas com o mesmo problema:**



Para resolver o problema precisamos encontrar uma forma de referenciar Diretor, Gerente e Cliente de uma mesma maneira, isto é, achar um fator comum.

Se existisse uma forma na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema.

Toda classe define 2 itens:

- O que faz (as assinaturas dos métodos);
- Como faz (o corpo dos métodos e os atributos privados).

Podemos criar um “contrato” que define tudo que uma classe deve fazer se quiser ter um determinado “status”. Imagine:

**Contrato Autenticável:**

Quem quiser ter o “status” Autenticável precisa saber fazer:

- 1) Dada uma senha, autenticar devolvendo um booleano.

Quem quiser pode “assinar” esse contrato, sendo assim obrigado a explicar como será feita essa autenticação. A vantagem é que, se um Gerente assinar esse Contrato, podemos nos referenciar a um Gerente como um Autenticável. Afinal, ele ganhou esse “status” ao assinar o contrato!

O melhor é que: **Podemos criar esse contrato em Java!** O nome disso é **interface**!

## Exercícios sobre interface

- 1) No projeto-bancov08 fornecido pelo professor, crie a interface Autenticavel.

```
public interface Autenticavel {  
    boolean autentica(int senha);  
}
```

**Interface** é a maneira pela qual poderemos conversar com um Autenticável. **Interface** é a maneira através da qual conversamos com um objeto.

Lemos a interface da seguinte maneira: “*quem desejar ser Autenticável precisa saber autenticar, recebendo um inteiro e retornando um booleano*”. É um contrato onde quem assina se responsabiliza por implementar seu(s) método(s) (cumprir o contrato).

Uma interface pode definir uma série de métodos, mas nunca (isso mudou nas novas versões) a implementação deles. Uma interface só expõe **o que um método deve fazer**, e não **como ele faz**, nem **o que ele tem**. **Como ele faz** vai ser definido em uma **implementação** dessa interface, ou seja, em uma classe que assina o contrato (uma classe que implementa Autenticável).

**RETIFICANDO:** A partir das novas versões do Java, uma interface pode ter métodos públicos e implementados e atributos públicos. Pesquise sobre o uso da palavra default.

O Gerente pode “assinar” o contrato, ou seja, implementar a interface. Ao implementar essa interface ele precisa escrever o(s) método(s) exigido(s) pela interface. Isso é muito parecido com o efeito de herdar métodos abstratos. Aliás, métodos de uma

interface são (eram) públicos e abstratos, sempre\*\*! Para implementar, usamos a palavra chave implements na classe.

- 2) Faça com que a classe Gerente implemente a interface Autenticável e, consequentemente, implemente seu método autentica.

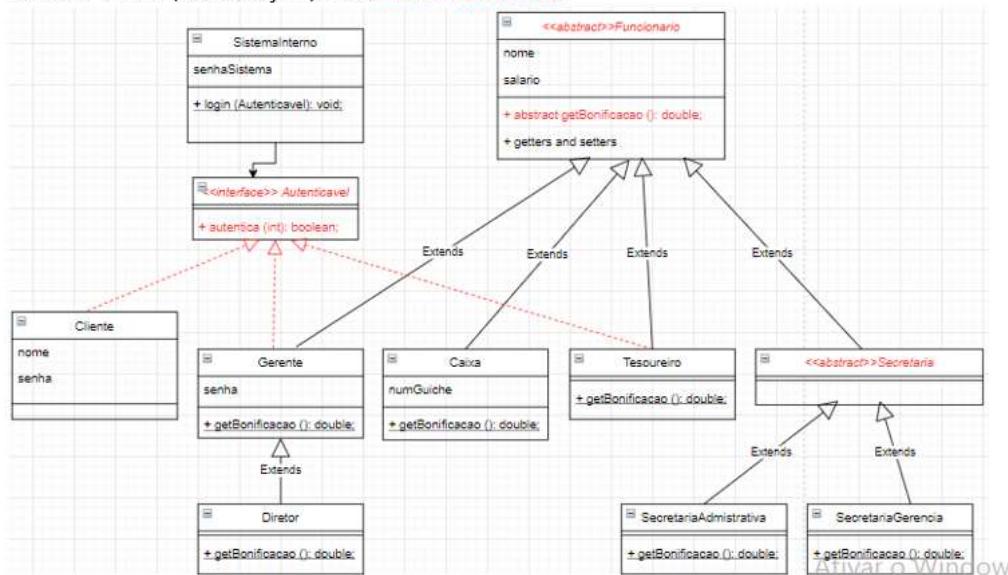
```
public class Gerente extends Funcionario implements Autenticavel{
    private int senha;

    //outros atributos e métodos...

    public boolean autentica(int senha) {
        if(this.senha==senha) {
            return true;
        }else{
            return false;
        }
    }
}
```

- 3) Faça com que Tesoureiro e Cliente tenham o atributo senha e seus métodos get e set. Em seguida, faça com que as duas classes implementem a interface Autenticável e seu método autentica.

Cliente, Gerente, Diretor e Tesoureiro precisam implementar a interface Autenticavel garantindo fornecer uma implementação para o método autentica.



Podemos interpretar o implements da seguinte maneira: “A classe Gerente pode ser tratada como autenticável, se comprometendo a ter todos os métodos especificados no contrato, implementados.”.

Agora além de poder ser tratado como um Funcionário, o Gerente também pode ser tratado como um Autenticável. **Ganhamos polimorfismo!** Temos mais uma forma de referenciar um Gerente. Quando crio uma variável do tipo Autenticável, estou criando uma referência para qualquer objeto de uma classe que implementa Autenticável, direta ou indiretamente. Veja:

Autenticável a = new Gerente();  
**//Posso aqui chamar o método autentica!**

- 4) Novamente, a solução mais adequada seria receber um Autenticável como argumento. Veja como ficaria nosso SistemaInterno e faça as alterações necessárias.

```
public class SistemaInterno {  
    private int senha = 123;  
    public void login(Autenticavel a) {  
        boolean ok = a.autentica(senha);  
        //Nem se sabe para que objeto a referência "a" está apontando,  
        //mas ainda sim podemos chamar o método autentica. Flexibilidade!  
        if(a.autentica(this.senha))  
            ; //Instruções  
        else  
            ; //Instruções  
    }  
}
```

**Pronto!** Agora podemos passar qualquer Autenticável para o SistemaBancario.

Por herança, Diretor também implementa indiretamente essa interface e herda o método autentica.

- 5) Altere a classe TestaSistemaInterno para que ela instancie Gerente, Diretor, Cliente, tesoureiro e SistemaInterno. Faça com que todos os objetos criados sejam passados como argumento para o método login de SistemaInterno e faça os testes necessários.
- 6) Faça uma leitura atenta dos parágrafos abaixo para fixar melhor o conhecimento.

No dia em que tivermos mais um outro tipo de Funcionário que precise acessar o Sistema, basta que ele implemente essa interface.

Perceba que só sabemos que se trata de um Autenticável e isso basta. Não nos interessa saber quem ele é, mas sim o que é capaz de fazer. Nos interessa que ele seja capaz de se autenticar, não importa se é um Gerente, um Diretor ou até mesmo um Cliente. Veja:

```
Autenticável diretor = new Diretor();  
Autenticável gerente = new Gerente();  
Autenticável cliente = new Cliente();
```

Se agora acharmos que Caixa precisa acessar o Sistema, basta que ele implemente Autenticável. Olha só o tamanho do desacoplamento: quem escreveu SistemaInterno só precisa saber que ele é Autenticável.

É justamente isso que queremos: **alta coesão e baixo acoplamento!**

Não importa se é um Gerente, um Diretor ou um Cliente, basta cumprir o contrato! Mais ainda. Cada um pode se autenticar de maneira completamente diferente!

Lembre-se: a interface define que todos vão saber se autenticar (o que faz), enquanto a implementação define exatamente como deve ser feito (como ele faz).

**IMPORTANTE:** O que um objeto faz é mais importante do que como ele faz. Siga essa regra e terá Sistemas mais fáceis de manter e modificar.

### Herança entre interfaces:

Diferentemente das Classes, uma interface pode herdar de mais de uma interface. É como um contrato que depende que outros contratos sejam fechados antes deste valer. Você não herda métodos e atributos, mas sim responsabilidades.

### Qual é o objetivo de usar interfaces?

O maior objetivo é flexibilizar o código e diminuir o acoplamento entre as classes, possibilitando mudança de implementação sem grandes traumas. O uso de interface em vez de herança é amplamente aconselhado.

No livro Design Patterns, logo no início, os autores citam 2 regras de ouro do Java:

- “Evite herança, prefira composição”;
- “Programe voltado à interface e não à implementação”. Dê mais importância ao “o que faz” e não ao “como faz”.

Mais adiante veremos mais sobre interfaces.

### Mais exercícios obrigatórios sobre interfaces:

- 1) Crie um projeto chamado interfaces e crie a interface AreaCalculavel:

```
public interface AreaCalculavel {  
    double calculaArea();  
}
```

- 2) Crie algumas classes que são AreaCalculavel:

```
public class Quadrado implements AreaCalculavel {  
    private int lado;  
  
    public Quadrado(int lado) {  
        this.lado = lado;  
    }  
  
    public double calculaArea() {  
        return this.lado * this.lado;  
    }  
}  
  
public class Retangulo implements AreaCalculavel {  
    private int largura, altura;  
  
    public Retangulo(int largura, int altura) {  
        this.largura = largura;  
        this.altura = altura;  
    }  
  
    public double calculaArea() {  
        return this.altura * this.largura;  
    }  
}
```

```
    }  
}
```

Se você tivesse usado herança, não iria ganhar muito, já que cada implementação é totalmente diferente da outra: Quadrado, Retângulo e Círculo são Figuras Geométricas e poderiam ter uma superclasse com esse nome. O problema é que elas possuem atributos e métodos bem diferentes. O que é comum aqui é **o que elas fazem**. Elas calculam sua Área.

Mesmo que elas tivessem atributos em comum, usar interfaces é uma madeira mais elegante de modelar suas classes. A grande vantagem é o desacoplamento. Herança traz muito acoplamento, o que pode quebrar o encapsulamento, lembra?

- 3) Crie a classe Circulo:

```
public class Circulo implements AreaCalculavel {  
    private double raio;  
  
    public Circulo(double raio) {  
        this.raio = raio;  
    }  
  
    public double calculaArea() {  
        return Math.PI * this.raio;  
    }  
}
```

- 4) Crie uma classe Teste. Repare no polimorfismo. Poderíamos estar passando esses objetos para algum método que aceitasse AreaCalculavel como argumento:

```
public class Teste {  
    public static void main(String[] args) {  
        AreaCalculavel a = new Retangulo(3, 2);  
        System.out.println(a.calculaArea());  
    }  
}
```

- 5) Adicione um método imprimirArea à interface AreaCalculavel.
- 6) Crie uma classe chamada MostradorDeArea. Esse método deve poder receber qualquer das figuras acima e imprimir sua área.
- 7) Crie uma classe de testes para instanciar e utilizar os objetos criados acima.

## 50.Exercícios práticos 11

- 1) Nosso banco precisa tributar dinheiro de bens que nossos clientes possuem. Para isso vamos criar uma interface no nosso projeto já existente:

```
public interface Tributavel {  
    double calculaTributos();  
}
```

Regra: “todos que quiserem ser Tributável precisam saber calcular tributos devolvendo um double”.

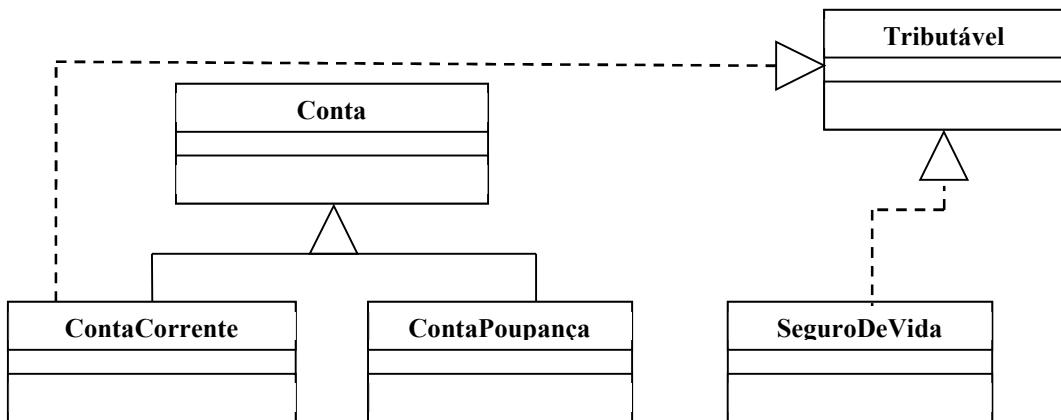
Alguns bens são tributáveis, outros não. ContaPoupanca não é Tributável. Já para ContaCorrente, calculaTributos() deve retornar 1% de seu saldo. A classe que representa o seguro de vida tem uma taxa fixa de 50 reais.

Aproveite os recursos do Eclipse! Quando você escrever implements Tributável na classe ContaCorrente, o quick fix vai sugerir que você reescreva o método. Escolha essa opção e depois preencha o corpo do método adequadamente:

```
public class ContaCorrente extends Conta implements Tributavel {  
    @Override  
    public void atualiza(double taxa) {  
        this.saldo -= (2 * taxa);  
    }  
  
    @Override  
    public boolean deposita(double valor) {  
        if (valor > 0) {  
            this.saldo += valor;  
            this.saldo -= 0.10;  
            return true;  
        }  
        return false;  
    }  
  
    public double calculaTributos() {  
        return this.saldo * 0.01;  
    }  
}
```

2) Crie a classe SeguroDeVida, aproveitando o quick fix para obter:

```
public class SeguroDeVida implements Tributavel{  
    @Override  
    public double calculaTributos() {  
        return 50;  
    }  
}
```



3) Vamos criar a classe TestaTributavel:

```
public class TestaTributavel {
    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        System.out.println(cc.calculaTributos());

        SeguroDeVida sv = new SeguroDeVida();
        System.out.println(sv.calculaTributos());

        // Testando o polimorfismo
        Tributável t = cc;
        System.out.println(t.calculaTributos());
    }
}
```

Agora tente chamar o método `getSaldo` através da referência `t`. O que ocorre? Por quê?  
A linha `Tributável t = cc;` é apenas para você enxergar que é possível fazê-lo. Nesse nosso exemplo não tem nenhuma utilidade. Isso será útil no exercício a seguir.

4) Crie um `GerenciadorDeImpostoDeRenda` que recebe todos os `Tributáveis` de uma pessoa e soma seus valores. Inclua também um método para devolver o total:

```
public class GerenciadorDeImpostoDeRenda {
    private double total;

    void adiciona(Tributável t) {
        System.out.println("Adicionando tributável+t");
        this.total = t.calculaTributos();
    }

    public double getTotal() {
        return this.total;
    }
}
```

- 5) Escreva um programa onde vamos instanciar diversas classes que implementam Tributável e passar como argumento para o método adiciona de um GerenciadorDeImpostoDeRenda. Perceba que você não pode passar qualquer tipo de Conta para o método adiciona, apenas a que implementa Tributável. Pode passar também o SeguroDeVida.

```
public class TestaGerenciadorDeImpostoDeRenda {  
    public static void main(String[] args) {  
        GerenciadorDeImpostoDeRenda gerenciador = new GerenciadorDeImpostoDeRenda();  
  
        SeguroDeVida sv = new SeguroDeVida();  
        gerenciador.adiciona(sv);  
  
        ContaCorrente cc = new ContaCorrente();  
        cc.deposita(1000);  
        gerenciador.adiciona(cc);  
  
        System.out.println(gerenciador.getTotal());  
    }  
}
```

Repare que você não pode acessar o método getSaldo de dentro do GerenciadorDeImpostoDeRenda, pois você não tem nenhuma garantia de que o Tributável que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface Tributável.

Interessante observar que a interface é capaz de ligar classes muito distintas, unindo-as por uma característica que elas têm em comum. SeguroDeVida e ContaCorrente são entidades completamente distintas, porém ambas possuem a característica de serem Tributáveis.

Se amanhã o governo resolver tributar TituloDeCapitalizacao, basta que essa classe implemente Tributável. Perceba o grau de desacoplamento que temos aqui.

- 6) Crie a classe TituloDeCapitalização com o atributo valorTitulo e métodos get e set. Faça com que TituloDeCapitalização implemente Tributável e que seu calculaImpostos() devolva 2% de seu valor.  
7) Faça com que TestaGerenciadorDeImpostoDeRenda utilize um objeto da classe acima.

### Mais exercícios sobre interfaces (Esse é só para os curiosos. Não é obrigatório):

- 8) Crie um projeto chamado conta-interface. Crie a classe Conta como uma Interface:

```
public interface ContaI {  
    double getSaldo();  
    boolean deposita(double valor);  
    boolean saca(double valor);  
    boolean transferePara(ContaI contaDestino, double valor);  
    void atualiza(double taxa);  
}
```

Adapte ContaCorrente e ContaPoupança para essa modificação:

```
public class ContaCorrente2 implements ContaI {
    private double saldo;

    @Override
    public void atualiza(double taxa) {
        this.saldo -= (2 * taxa);
    }
    public boolean deposita(double valor) {
        if (valor > 0) {
            this.saldo += valor;
            return true;
        }
        return false;
    }
    public boolean saca(double valor) {
        if (valor > 0 && valor <= this.saldo) {
            this.saldo -= valor;
            return true;
        } else
            return false;
    }
    public boolean transferePara(ContaI contaDestino, double valor) {
        ContaI contaOrigem = (ContaI) this;
        if (contaOrigem.saca(valor))
            return contaDestino.deposita(valor);
        return false;
    }
    @Override
    public double getSaldo() {
        return this.saldo;
    }
}
```

```
public class ContaPoupanca2 implements ContaI {
    private double saldo;

    public void atualiza(double taxa) {
        this.saldo -= (3 * taxa);
    }
    public boolean deposita(double valor) {
        if (valor > 0) {
            this.saldo += valor;
            return true;
        }
        return false;
    }
    public boolean saca(double valor) {
        if (valor > 0 && valor <= this.saldo) {
            this.saldo -= valor;
            return true;
        } else
            return false;
    }
    public boolean transferePara(ContaI contaDestino, double valor) {
        ContaI contaOrigem = (ContaI) this;
        if (contaOrigem.saca(valor))
            return contaDestino.deposita(valor);
        return false;
    }
    public double getSaldo() {
        return this.saldo;
    }
}
```

Algum código vai ter que ser copiado e colado? Sim. Essa é a maneira de obter polimorfismo se desvincilhando do alto acoplamento da herança. Vale a pena todo esse trabalho? Cabe ao programador decidir. As duas formas são válidas.

9) Subinterfaces:  
Às vezes é interessante criarmos uma interface que herda de outras interfaces.

```
public interface ContaTributavelI extends ContaI{
    double calculaTributos();
}
```

Desta forma, quem for implementar essa nova interface precisa implementar todos os métodos herdados das suas superinterfaces (e talvez ainda novos métodos declarados dentro dela):

```
public class ContaCorrente2 implements ContaTributavelI {
    //métodos de ContaI e de ContaTributavelI
}
```

Desta forma, as 2 linhas abaixo compilam sem problemas. O único porém é que através da referência c não é possível chamar o método calculaTributos().

```
ContaI c = new ContaCorrente2();
ContaTributável ct = new ContaCorrente2();
```

Apesar de uma interface poder herdar de várias interfaces, uma classe só pode herdar de uma única classe (herança simples).

### Coisas importantes a lembrar a respeito de Interfaces:

- \* Pode ter constantes (public final);
- \* É como uma classe 100% abstrata (Isso mudou um pouco com o surgimento da palavra-chave default);
- \* Não pode ter métodos estáticos, nem final, native, stricted, synchronized;
- \* Não pode ser e nem ter private e protected;
- \* Só pode ter constantes (public static final ou public final);
- \* Não implementa outra interface;
- \* Todos os métodos são implicitamente public e abstract (agora há também a possibilidade de métodos implementados);

### Pode ter as seguintes declarações:

```
public final double MINIMO = 1222; //Constante
public static final int numero=5; //Constante da Classe

default public static int contador=0; //Variavel da classe
default public String nome="Fulano"; //Variável comum
```

### Não pode ter as seguintes declarações:

```
public static int contador=0; //Variavel da classe
public String nome="Fulano"; //Variável comum
```

## 51. Tratamento de Exceções

Voltando à primeira versão das Contas que criamos nos itens anteriores, o que aconteceria ao tentar chamar o método saca com valor fora do limite? Como avisar aquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário de que o saque não foi efetuado. No exemplo abaixo, vamos, propositalmente, forçar a Conta a estar em um estado inconsistente de acordo com nossa modelagem, fazendo com que ele tenha um saldo negativo.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
```

// O saldo é -900? É 100? É 0? A chamada do método saca funcionou?

Em Sistemas de verdade, quem sabe tratar o erro é aquele que chamou o método e não a própria classe. Portanto, o normal é a classe sinalizar que um erro ocorreu.

A solução mais simples, muito usada antigamente, é a de retornar verdadeiro ou falso. True ou False:

```
public boolean saca(double quantidade) {
    if (this.saldo >= quantidade) {
        this.saldo -= quantidade;
        return true;
    } else {
        return false;
    }
}
```

```
        return true;
    } else {
        System.out.println("Não posso sacar fora do limite!");
        return false;
    }
}
```

Um outro exemplo de chamada ao método acima seria:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
if(!minhaConta.saca(1000)) {
    System.out.println("Não saquei.");
}
```

Tivemos que lembrar de testar o retorno do método, mas não somos obrigados a fazer isso através da linguagem. Esquecer de fazer esse teste traria consequências drásticas para o nosso Sistema. A máquina de autoatendimento poderia liberar um dinheiro mesmo sem ter conseguido efetuar o saque.

Mesmo invocando o método e lembrando de tratar seu retorno de maneira correta, o que faríamos se fosse necessário sinalizar quando o usuário passasse um valor negativo como quantidade? Uma solução seria alterar o retorno, mudando de boolean para int e retornar o código do erro ocorrido. Isso é uma má prática conhecida como uso de “magic numbers”.

Além de você perder o retorno do método, o valor retornado é mágico e só legível perante extensa documentação, além de não obrigar o programador a tratar o retorno. Caso o programador se esqueça de tratar, o programa continuará rodando como se tudo estivesse normal.

Para evitar esse tipo de situação, utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: o caso em que acontece algo que normalmente não iria acontecer. O exemplo do argumento do saque inválido ou do id inválido de um cliente é uma exceção à regra.

**Exceção** → uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

Exercícios para fixar os conceitos:

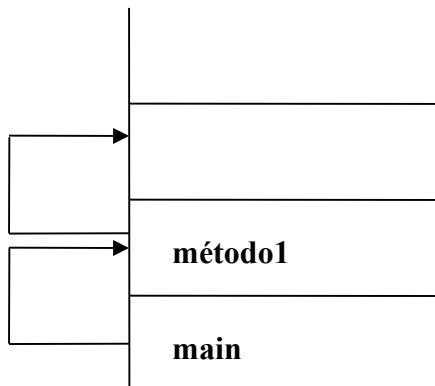
- 1) Teste o seguinte código você mesmo:

```
public class TesteErro {
    public static void main(String[] args) {
        System.out.println("Inicio do main");
        metodo1();
        System.out.println("Fim do main");
    }

    static void metodo1() {
        System.out.println("Inicio do método 1");
        metodo2();
        System.out.println("Fim do método 1");
    }
}
```

```
static void metodo2() {
    System.out.println("Inicio do método 2");
    int[] array = new int[10];
    for (int i = 0; i <= 15; i++) {
        array[i]=i;
        System.out.println(i);
    }
    System.out.println("Fim do método 2");
}
}
```

O método main chama o método1 que chama o método2. Cada um destes métodos pode ter suas próprias variáveis locais, sendo que, por exemplo, o método1 não enxerga as variáveis declaradas no método main. Como o Java (e muitas outras linguagens) faz isso? Toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da pilha de execução (stack). Basta jogar fora um gomo da pilha (stackframe):



Propositalmente nosso método2 tem um enorme problema: está acessando um índice de array indevido. O índice estará fora dos limites do array quando chegar em 10!

Rode o código. Qual é a saída? O que isso representa? O que ela indica?

The screenshot shows the Eclipse IDE interface with the title bar "Resource - nada/src/br/com/caelum/modelo/TesteErro.java - Eclipse Platform". The menu bar includes File, Edit, Navigate, Search, Project, Run, Window, and Help. The toolbar has icons for New, Open, Save, Cut, Copy, Paste, Find, and others. The left sidebar has sections for Tasks, History, and Console. The main area is the Console view, which displays the output of a Java application named "TesteErro". The output shows the following sequence of events:

```
<terminated> TesteErro [Java Application] C:\Arquivos de programas\Java\jre6\bin\javaw.exe (22/07/2010 19:02:18)
Inicio do main
Inicio do método 1
Inicio do método 2
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at br.com.caelum.modelo.TesteErro.metodo2(TesteErro.java:20)
at br.com.caelum.modelo.TesteErro.metodo1(TesteErro.java:12)
at br.com.caelum.modelo.TesteErro.main(TesteErro.java:6)
```

O que você vê acima é conhecido como **rastro da pilha** (stacktrace). É uma saída importantíssima para o programador – tanto que em qualquer fórum ou lista de discussão é comum programadores enviarem, juntamente com a descrição do problema, essa stacktrace.

Por que isso ocorreu? Quando uma exceção é lançada a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao tentar executar esse trecho de código. Como podemos ver o método2 não toma nenhuma medida.

Como o método2 não está **tratando** esse problema, a JVM para a execução abruptamente, sem esperar ele terminar, e volta um stackframe para baixo, onde será feita nova verificação: o método1 está se precavendo de um problema chamado *ArrayIndexOutOfBoundsException*? Não.... volta para o main, onde também não há proteção, então a thread corrente morre.

O erro aqui foi proposital. Para arrumar isso bastaria fazer com que o array navegasasse no máximo até o seu length.

Para entender o controle de fluxo da exceção, vamos colocar o código que vai **tentar** (try) executar o bloco perigoso e, caso o problema seja do tipo *ArrayIndexOutOfBoundsException*, ele será **pego** (caught). Perceba que é interessante que cada exceção tenha um tipo... ela pode ter atributos e métodos.

- 2) Adicione um try/catch em volta do for, pegando *ArrayIndexOutOfBoundsException*. O que o código imprime agora?

```
try {
    for (int i = 0; i <= 15; i++) {
        array[i] = i;
        System.out.println(i);
    }
} catch (ArrayIndexOutOfBoundsException e) {
```

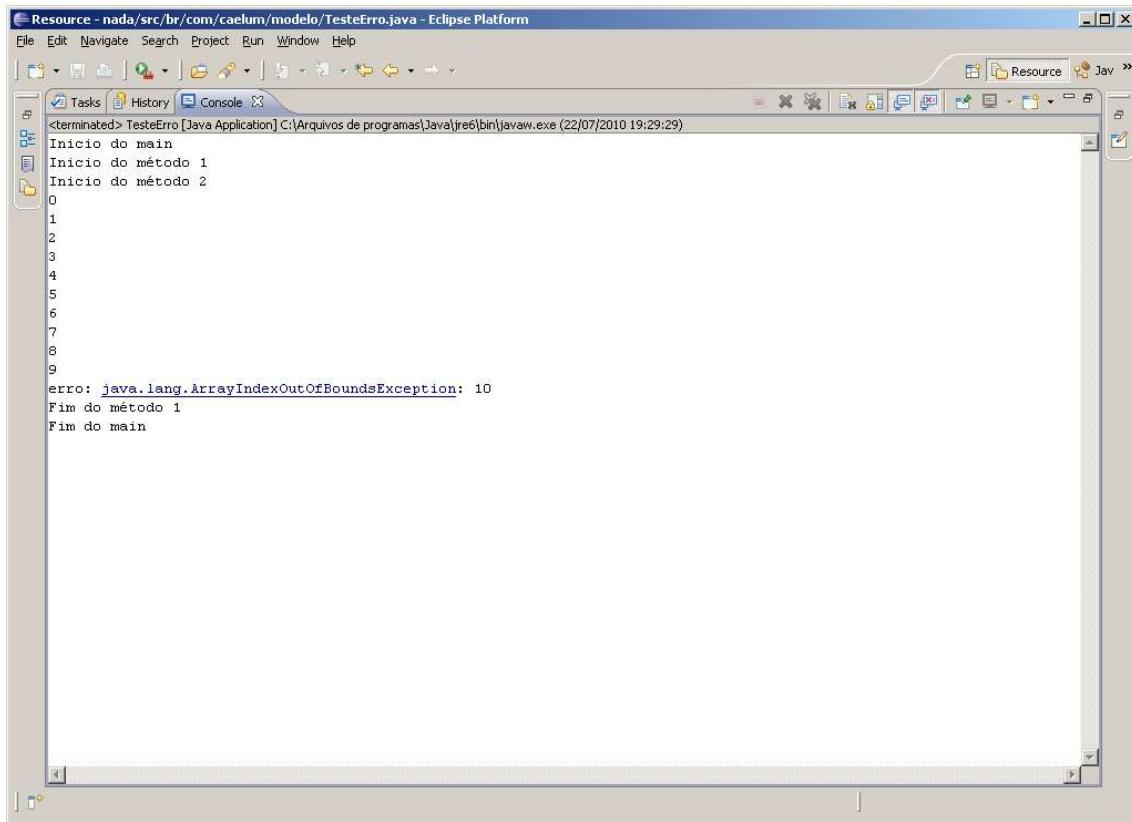
```
        System.out.println("erro: " + e);
    }
```

Qual é a diferença?

```
<terminated> TesteErro [Java Application] C:\Arquivos de programas\Java\jre6\bin\javaw.exe (22/07/2010 19:25:53)
Inicio do main
Inicio do método 1
Inicio do método 2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
Fim do método 2
Fim do método 1
Fim do main
```

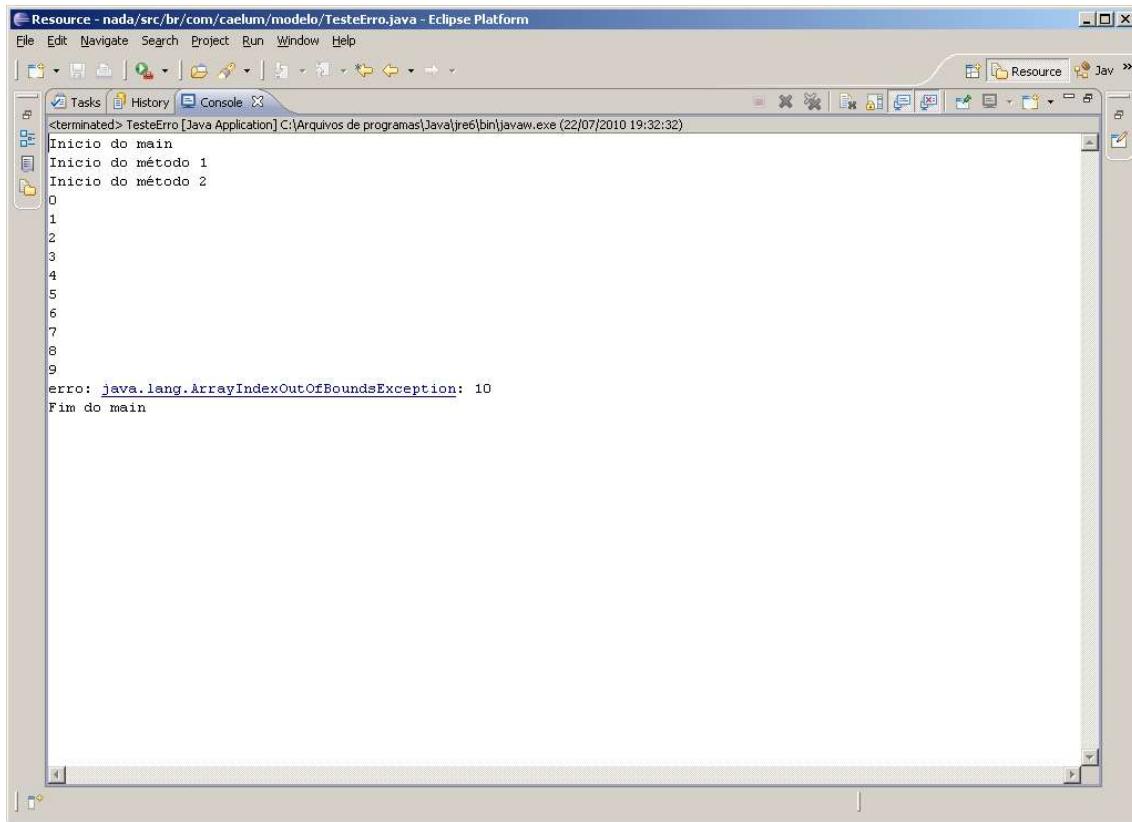
Agora retire o try/catch e coloque em volta da chamada do método2.

```
static void metodo1() {
    System.out.println("Inicio do método 1");
    try {
        metodo2();
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("erro: " + e);
    }
    System.out.println("Fim do método 1");
}
```



Faça a mesma coisa retirando o try/catch novamente e colocando em volta da chamada do método1. Rode os códigos, o que acontece?

```
System.out.println("Inicio do main");
try {
    metodo1();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
System.out.println("Fim do main");
```



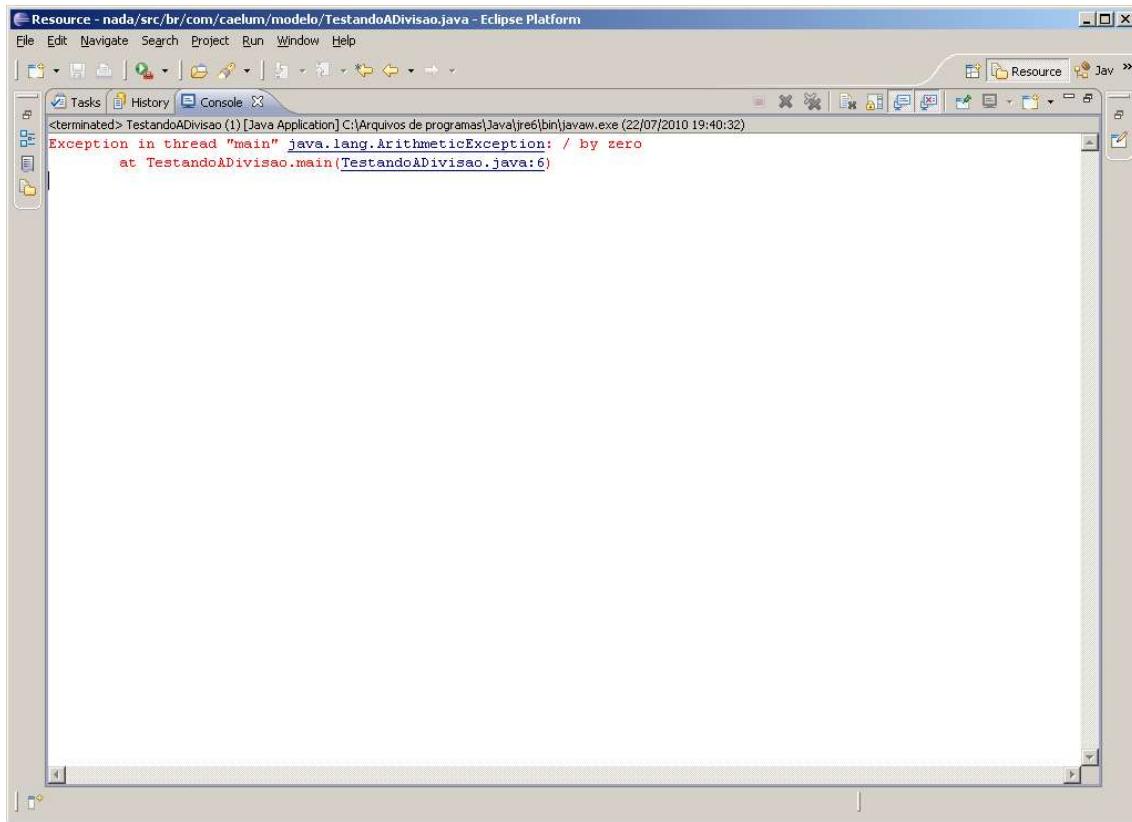
Repare que, a partir do momento que uma exception foi catched (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

### Exceções de Runtime mais comuns

Tente dividir um número por zero. Será que o computador consegue fazer algo que nós definimos que não existe?

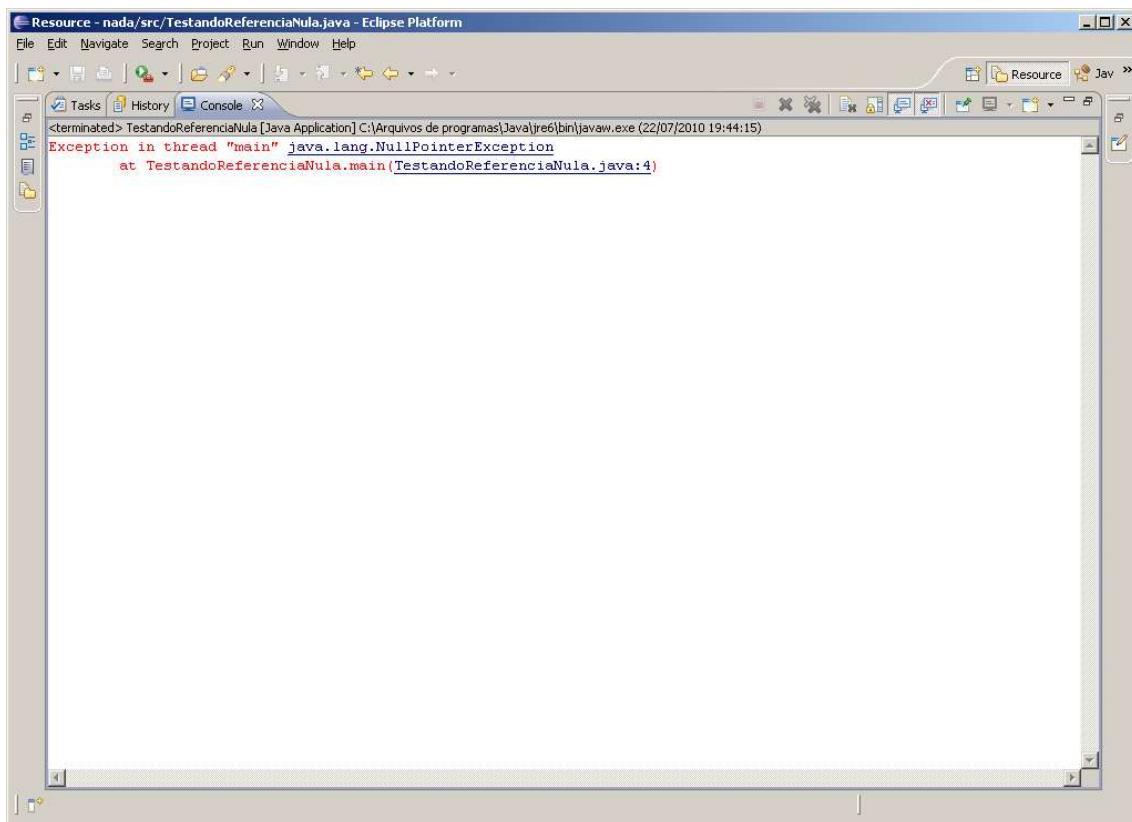
```
public class TestandoADivisao {
    public static void main(String[] args) {
        int i = 5000;
        i = i / 0;
        System.out.println("O resultado é " + i);
    }
}
```

Tente executar o programa acima. O que acontece?



```
public class TestandoReferenciaNula {
    public static void main(String[] args) {
        Conta c = null;
        System.out.println("Saldo atual " + c.getSaldo());
    }
}
```

E agora? O que acontece?



Outro caso comum é quando um cast errado é feito. Veremos mais adiante. Em todos os casos, tais erros provavelmente poderiam ser evitados pelo programador. É por esse motivo que o Java não te obriga a dar try/catch nessas exceptions. Chamamos essas exceções de unchecked. Em outras palavras, o compilador não checa se você está tratando essas exceções.

### Checked Exceptions:

Todos os exemplos que vimos até agora poderiam ter ficado sem o try/catch que iriam compilar e rodar. Sem usar o try/catch o erro terminou o programa, usando o try/catch o erro foi tratado. Mas, no Java não existe só esse tipo de exceção onde o tratamento é opcional. Existe também um tipo de exceção que obriga quem chama o método ou construtor a tratá-la. Chamamos esse tipo de exceção de **checked**, pois o compilador irá checar se ela está sendo devidamente tratada.

Um exemplo clássico é o de abrir um arquivo para leitura, onde pode ocorrer o erro de o arquivo não existir (veremos como tratar arquivos mais adiante, não se preocupe com isso agora):

```
public static void metodo() {
    new java.io.FileInputStream("arquivo.txt");
}
```

O código não compila. O compilador avisa que é necessário tratar o FileNotFoundException que pode ocorrer.

Para compilar e fazer o programa funcionar, precisamos tratar o erro de um dos dois jeitos. O primeiro, é tratá-lo com o try e catch do mesmo jeito que usamos no exemplo anterior com um array:

```
public static void metodo() {
    try {
        new java.io.FileInputStream("arquivo.txt");
    } catch (FileNotFoundException e) {
        System.out
            .println("Não foi possível abrir o arquivo
para leitura.");
    }
}
```

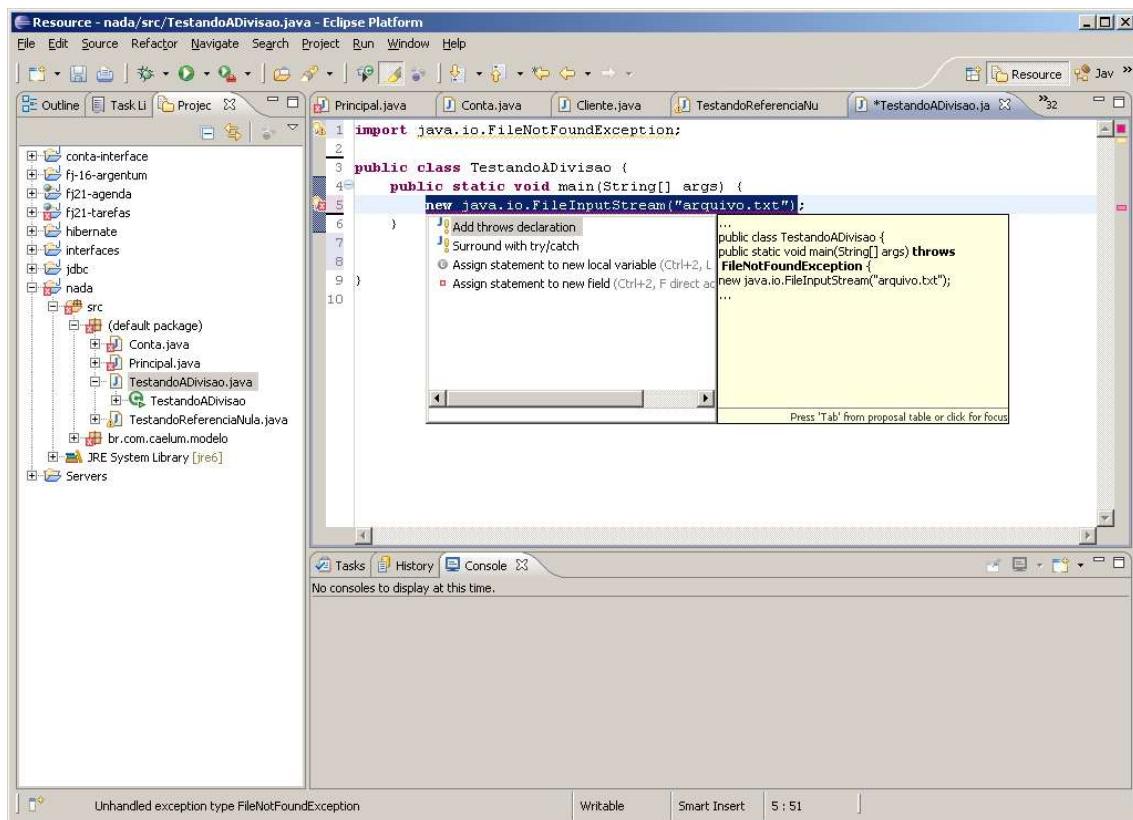
A outra forma de tratar esse erro é delegar ele para quem chamou nosso método, ou seja, passar adiante.

```
public static void metodo() throws FileNotFoundException{
    new java.io.FileInputStream("arquivo.txt");
}
```

Usando o Eclipse, fica bem simples fazer tanto o try/catch como um throws:  
Digite esse código no Eclipse:

```
public static void main(String[] args) {
    new java.io.FileInputStream("arquivo.txt");
}
```

O Eclipse vai reclamar:



E te oferecer duas opções:

- 1) Add throws declaration que vai gerar:

```
public static void main(String[] args) throws  
FileNotFoundException {  
    new java.io.FileInputStream("arquivo.txt");  
}
```

- 2) Surround with try/catch, que vai gerar:

```
public static void main(String[] args) {  
    try {  
        new java.io.FileInputStream("arquivo.txt");  
    } catch (FileNotFoundException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

No início, a tentação de sempre passar o erro para frente para outros tratarem dele é grande. Dependendo do caso até faz sentido, mas não até o **main**, por exemplo. Quem tentou abrir um arquivo, não sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa não saber resolver isso. Pode ser até pior: quem chamou o método no começo do programa pode estar tentando abrir uns 4 ou 5 arquivos diferentes e não saberá qual deles teve um problema.

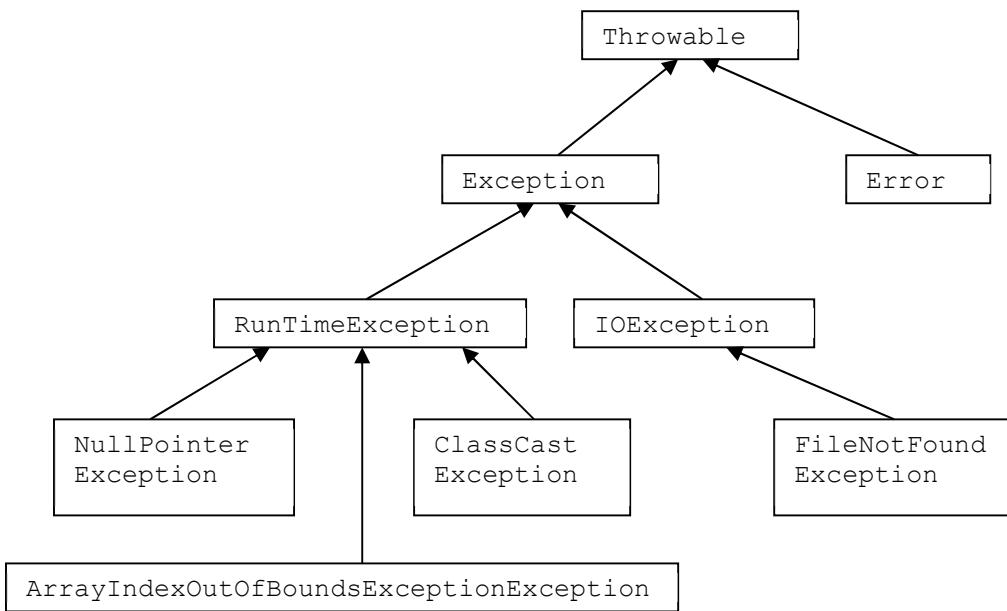
Não existe uma regra para decidir em que momento do programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, provavelmente você vai preferir passar para frente, delegando a responsabilidade para o método que te invocou.

No endereço abaixo você vai encontrar um artigo que discute boas práticas em relação ao tratamento de exceções.

<http://blog.caelum.com.br/2006/10/07/lidando-com-exceptions/>

### A família Throwable:

Eis uma pequena parte da família Throwable:



### Aprendendo a lidar com mais de um erro:

É possível tratar mais de um erro, quase que ao mesmo tempo:

- Com o try/catch:

```
public static void main(String[] args) {  
    try {  
        objeto.metodQuePodeLancarException();  
    } catch (IOException e) {  
        //...  
    } catch (SQLException e) {  
        //...  
    }  
}
```

- Com o throws:

```
public void abre(String arquivo) throws IOException, SQLException{  
    //...  
}
```

- Tratando algumas exceções e declarando outras no throws:

```
public void abre(String arquivo) throws IOException{  
    try {  
        objeto.metodQuePodeLancarIOeSQLException();  
    } catch (SQLException e) {  
        //...  
    }  
}
```

Declarar, no throws, as exceptions que são unchecked, é desnecessário, porém é permitido e, às vezes, facilita a leitura e a documentação do seu código.

### Lançando Exceções:

Lembra do nosso método saca da classe Conta? Conseguindo ou não sacar, ele devolve um boolean:

```
public boolean saca(double quantidade) {  
    if (this.saldo >= quantidade) {  
        this.saldo -= quantidade;  
        return true;  
    } else {  
        return false;  
    }  
}
```

Caso ocorra algo inesperado, como um valor negativo para saque, podemos lançar uma Exception, o que é extremamente útil. Assim, resolvemos o problema de alguém poder esquecer-se de fazer um IF no retorno do método.

A palavra chave throw lança uma Exeption. Não é o mesmo caso que throws, que apenas avisa da possibilidade daquele método lançá-la, obrigando o outro método, que chamou este, a se preocupar com a referida exception.

```
public void saca(double quantidade) {  
    if (this.saldo >= quantidade) {  
        this.saldo -= quantidade;  
    } else {  
        throw new RuntimeException();  
    }  
}
```

No código acima estamos lançando uma exceção do tipo unchecked. RunTimeException é a exceção mãe de todas as exceptions unchecked. A desvantagem, nesse caso, é que ela é muito genérica. Quem receber esse erro não saberá dizer exatamente qual foi o problema. Podemos então usar uma Exception mais específica:

```
public void saca(double quantidade) {  
    if (this.saldo >= quantidade) {  
        this.saldo -= quantidade;  
    } else {  
        throw new IllegalArgumentException();  
    }  
}
```

IllegalArgumentException já esclarece um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma Exception unchecked, pois estende de RunTimeException. Quando um argumento sempre é inválido (números negativos, referências nulas etc), IllegalArgumentException é a melhor escolha!

Por fim, para pegar esse erro, não usaremos um IF/else e sim um try/catch, porque faz mais sentido já que a falta de saldo é uma exceção:

```
Conta cc = new ContaCorrente();  
cc.deposita(100);  
try {  
    cc.saca(300);  
} catch (IllegalArgumentException e) {  
    System.out.println("saldo insuficiente.");  
}
```

Podemos melhorar ainda mais o código, passando para o construtor de `IllegalArgumentException` o motivo da exceção.

```
public void saca(double quantidade) {
    if (this.saldo >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new IllegalArgumentException("saldo
insuficiente.");
    }
}
```

O método `getMessage()` definido na classe `Throwable` (mãe de todos os tipos de erros e exceptions) vai retornar a mensagem que passamos ao construtor da `IllegalArgumentException`.

```
try {
    cc.saca(300);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

## O que devo colocar dentro do try?

Imagine que vamos sacar dinheiro de diversas contas:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

Conta cp = new ContaPoupanca();
cp.deposita(100);

cc.saca(50);
System.out.println("Consegui sacar da conta corrente.");

cp.saca(50);
System.out.println("Consegui sacar da conta poupança");
```

Onde colocar a mensagem “Consegui sacar”?

Sempre que temos algo que depende do sucesso da linha de cima para ser correto, devemos agrupá-lo no `try`:

```
try {
    cc.saca(300);
    System.out.println("Consegui sacar da conta
corrente.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

try {
    cp.saca(300);
    System.out.println("Consegui sacar da conta
poupança.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

Ainda há uma outra opção: imagine que, para o nosso sistema, uma falha ao sacar da conta poupança deve parar o processo de saques e nem tentar sacar da conta corrente. Para isso, agruparíamos mais ainda:

```
try {
    cc.saca(300);
    System.out.println("Consegui sacar da conta corrente.");
    cp.saca(300);
    System.out.println("Consegui sacar da conta poupança.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

O que você vai colocar dentro do try influencia demais na execução do programa! Pense bem nas linhas que dependem uma da outra para a execução correta de sua lógica de negócios.

## Como criar seu próprio tipo de Exceção?

Para controlar melhor o uso de suas exceções, é bem comum criar uma própria classe de Exceção. Dessa forma, podemos passar valores específicos para ela carregar, que sejam úteis de alguma forma. Vamos criar nossa classe de Exceção SaldoInsuficienteException:

```
public class SaldoInsuficienteException extends RuntimeException {
    public SaldoInsuficienteException(String message) {
        super(message);
    }
}
```

Agora em vez de lançar uma IllegalArgumentException, vamos lançar nossa própria exceção, com a seguinte mensagem: “Saldo Insuficiente.”

```
public void saca(double quantidade) {
    if (this.saldo >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new SaldoInsuficienteException("saldo insuficiente.");
    }
}
```

Para testar, crie uma classe que deposite um valor e tente sacar um valor maior:

```
public static void main(String[] args) {
    Conta cc = new ContaCorrente();
    cc.deposita(10);

    try {
        cc.saca(100);
    } catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
    }
}
```

Podemos também transformar essa exceção de unchecked para checked, obrigando quem chama esse método a dar try/catch, ou throws:

```
public class SaldoInsuficienteException extends Exception {  
    public SaldoInsuficienteException(String message) {  
        super(message);  
    }  
}
```

## Finally

Os blocos try/catch podem conter uma terceira cláusula chamada finally que indica o que deve ser feito após o término do bloco try ou de um catch qualquer.

Geralmente se coloca algo que é imprescindível de ser executado, caso o que você queira fazer tenha dado certo ou não. O mais comum é que o programador use o finally para liberar algum recurso, como um arquivo, uma conexão com o banco de dados, mesmo que algo tenha falhado no decorrer do código.

Veja o código abaixo:

```
try{  
    //Bloco do try  
} catch(IOException ex){  
    //Bloco do catch 1  
} catch(SQLException sqlex){  
    //Bloco do catch 2  
} finally{  
    //Bloco do finally  
}
```

O bloco finally será executado, correndo tudo ok ou dando erro.

## Exercícios com Exceções (Projeto fornecido pelo professor)

- 1) Na classe Conta, modifique o método deposita. Ele deve lançar uma IllegalArgumentException sempre que receber um valor inválido como argumento (um valor negativo por exemplo).

```
public void deposita(double quantidade) {  
    if (quantidade < 0) {  
        throw new IllegalArgumentException();  
    } else {  
        this.saldo += quantidade;  
    }  
}
```

- 2) Crie uma classe TestaDeposita com o método main. Crie uma ContaPoupanca e tente depositar valores inválidos:

```
public class TestaDeposita {  
    public static void main(String[] args) {  
        Conta cp = new ContaPoupanca();  
        cp.deposita(-100);  
    }  
}
```

}

O que acontece? Uma `IllegalArgumentException` é lançada já que tentamos depositar um valor inválido. Adicione o try/catch para tratar o erro:

```
public static void main(String[] args) {
    Conta cp = new ContaPoupanca();
    try {
        cp.deposita(-100);
    } catch (IllegalArgumentException e) {
        System.out.println("Você tentou depositar um valor
inválido");
    }
}
```

**IMPORTANTE:** Se sua classe `ContaCorrente` está reescrevendo o método `deposita` e não utiliza o `super.deposita`, ela não lançará a exception no caso do valor negativo. Você pode resolver isso usando o `super.deposita`, ou fazendo apenas o teste com a `ContaPoupanca`.

- 3) Ao lançar uma `IllegalArgumentException`, passe via construtor uma mensagem a ser exibida. Lembre que a String recebida como parâmetro é acessível depois via método `getMessage()` herdado por todas as Exceptions.

```
public void deposita(double quantidade) {
    if (quantidade < 0) {
        throw new IllegalArgumentException("Você tentou
depositar um valor negativo.");
    } else{
        this.saldo += quantidade;
    }
}
```

- 4) Agora altere sua classe `TestaDeposita` para exibir a mensagem da exceção através da chamada do método `getMessage()`:

```
public static void main(String[] args) {
    Conta cp = new ContaPoupanca();
    try {
        cp.deposita(-100);
    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    }
}
```

- 5) Crie sua própria Exception, `ValorInvalidoException`. Para isso você precisa de uma classe com esse nome que estenda `RuntimeException`. O Eclipse vai sugerir que você serialize esta rotina. Faça isso! Mais adiante discutiremos melhor isso.

```
public class ValorInvalidoException extends RuntimeException {
    /**
     *
     */
    private static final long serialVersionUID = 1L;

    public ValorInvalidoException(String message) {
        super(message);
    }
}
```

```
}
```

**IMPORTANTE:** Nem sempre é interessante criarmos um novo tipo de exceção. Depende do caso. Neste aqui, por exemplo, bastava usar uma `IllegalArgumentException`. Na dúvida prefira as já existentes.

- 6) Mude o construtor de `ValorInvalidoException` para que ele receba como argumento o valor inválido que ele tentou passar. Quando estendemos uma classe, não herdamos seu construtor, mas podemos acessá-lo através da palavra chave `super` de dentro de um construtor. As exceções em Java possuem uma série de construtores úteis para poder populá-las com uma mensagem de erro. No exemplo acima, delegamos para o construtor da classe mãe. Vamos continuar fazendo isso neste exercício.

```
public class ValorInvalidoException extends RuntimeException {  
    /**  
     *  
     */  
    private static final long serialVersionUID = 1L;  
  
    public ValorInvalidoException(double valor) {  
        super("Valor inválido: "+valor);  
    }  
}
```

- 7) Declare a classe `ValorInvalidoException` como filha direta de `Exception` em vez de `RuntimeException`. Ela agora passa a ser checked. No que isso resulta? Você vai precisar avisar que o seu método `deposita()` throws `ValorInvalidoException`, pois ela é uma checked Exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre `try/catch` ou `throws`. Utilize-se do QuickFix do Eclipse novamente!

Depois retorne a exception para unchecked (filha de `RuntimeException`), pois iremos utilizá-la assim mais adiante.

**IMPORTANTE:** Existe uma péssima prática de programação que é a de escrever o `catch` e o `throws` com `Exception`. Isso é muito genérico. Procure classificar as Exceptions para que quem receba saiba qual é o tipo de erro em questão.