

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

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

**Professor:
Rafael Guimarães Rodrigues**

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	16
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 ++.....	22
17. Interferindo no loop	22
18. Escopo das variáveis.....	23
19. Outras coisas a saber.....	24
20. Exercícios.....	24
21. Orientação a Objetos Básica.....	24
22. Classes x Objetos.....	27
23. Construindo um Sistema Orientado a Objetos	29
24. Escrevendo uma classe em Java.....	31
25. Instanciando e usando um Objeto.....	31
26. Métodos de uma Classe	33
27. Objetos e Referências	36
28. Classes, métodos, objetos.....	39
29. Princípio da Alta Coesão.....	39
30. Um pouco mais sobre atributos e ... Agregação.....	50
31. Arrays em Java	62
32. Arrays de referências	63

33. Conhecendo e usando o foreach com arrays:	65
34. Controlando o acesso através de modificadores	69
35. Encapsulamento	69
36. Métodos get e set	69
37. Construtores e Java Bean	69
38. Atributo da classe	71
39. Exercícios sobre encapsulamento, construtores e static	73
40. Herança	74
41. Reescrita de Métodos	79
42. Polimorfismo	82
43. Facilidades da IDE Eclipse	89
44. Classes Abstratas	106
45. Interfaces e Sobrecarga	110
46. Tratamento de Exceções	123
47. Organizando suas Classes e Bibliotecas com Pacotes	140
48. Pacote java.io	144
49. Collections Framework	154
50. Programação Concorrente e Threads	170
51. Ferramentas: jar e javadoc	180
52. O pacote Java.lang	184

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 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.

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, totalmente diferente do paradigma imperativo/procedural, que é o que vocês conhecem 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 atrás, nas primeiras versões do Java.

O Java foi desenvolvido e mantido pela Sun Microsystems, mas essa empresa foi adquirida pela Oracle em 2010. A página principal é: 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 qualquer destes aparelhos independente 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 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, ao invés de apenas renderizar HTML, os navegadores seriam capazes de realizar operações. Surgia 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.

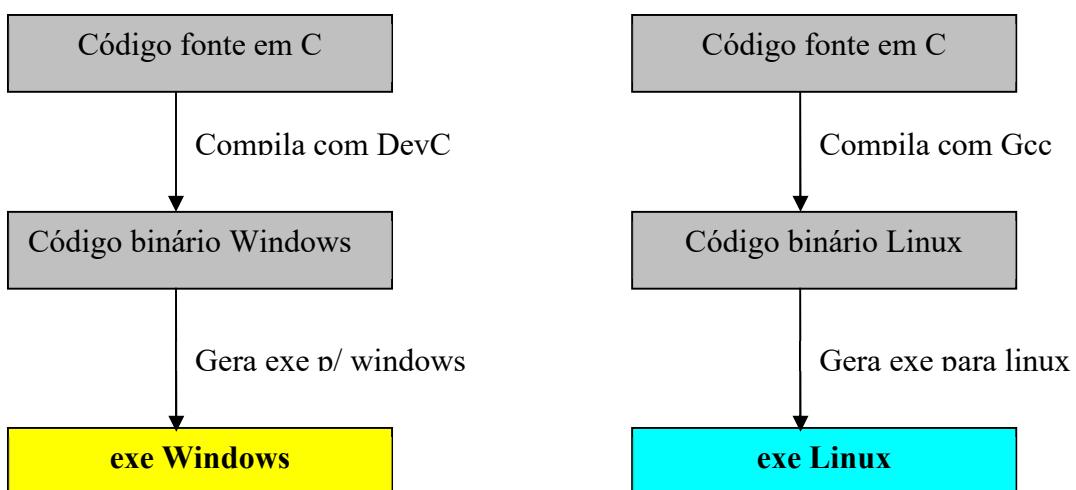
→2001: Java 1.4

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

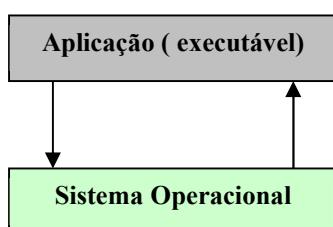
→2006/2007: Java 6. Muitas melhorias na performance da JVM. A partir deste momento a linguagem Java começa a ser amplamente utilizada e começa a dominar o mercado. A linguagem Java “ressuscitou”!

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:



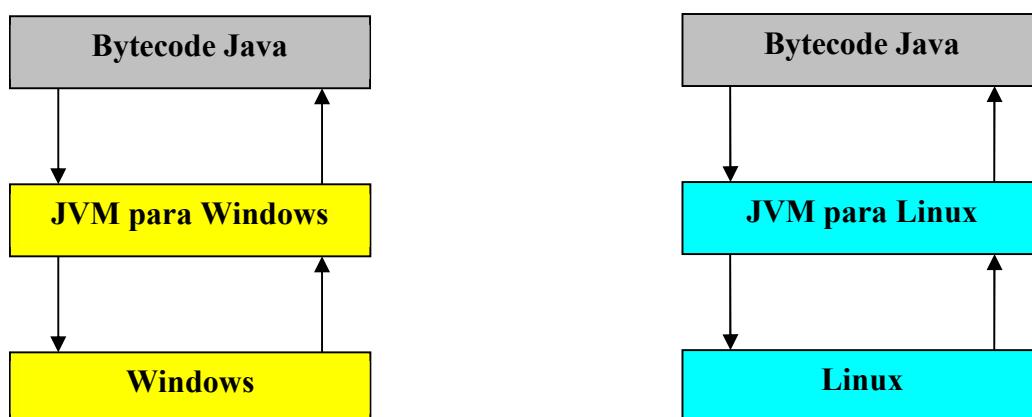
Como acontece com o Java?

O Java utiliza uma máquina virtual que consiste em uma camada extra entre a aplicação e o Sistema Operacional. Entre 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 por uma máquina virtual para Windows e por uma máquina virtual para Linux, ou seja: 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 é 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.

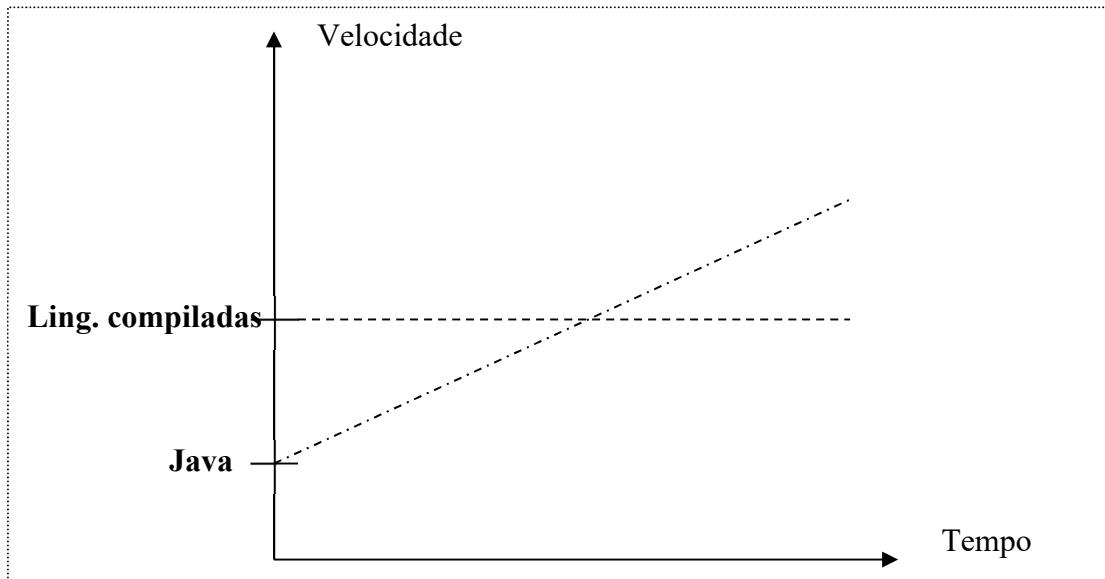
A JVM não compila tudo antes de executar a aplicação por que compilar dinamicamente pode gerar um desempenho melhor. Lembre-se que a JVM conversa com o Sistema Operacional o tempo todo e vai colhendo cada vez mais informações sobre ele.

Um .exe gerado pelo Delphi, pelo C ou pelo Visual Basic é estático. Ele já foi gerado baseado 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, 8 e 9, 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.

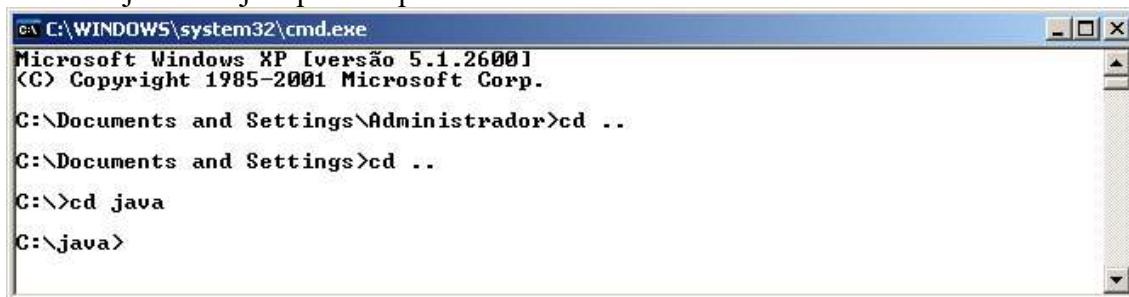
```
public class OlaMundo {  
    public static void main(String[] args) {  
        // Início do miolo do programa  
        System.out.println("Ola Mundo!");  
        // Fim do miolo do programa  
    }  
}
```

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("Ola 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  
Ola 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 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:\>_
```

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:\>_
```

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
int idadeDoAluno;

// 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 inicializar uma variável no momento da declaração. Veja:

```
int idade=15;
```

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 tem 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.

<= → 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 → DE :	Byte	Short	Char	Int	Long	Float	double
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 FaturamentoTrimestral 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 booleanas 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 e 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) { //CONDIÇÃO
    System.out.println(i);
    i++; //INCREMENTO
}
// Imprime de 0 a 9

Outro exemplo:
int i = 9; //CONTADOR
while (i > 0) { //CONDIÇÃO
    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 boleanas 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 ao invés 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 numeros 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, ela 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 estará desenvolvendo 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 pseudo-código 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 ao invés 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 nova 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? Desta 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 objetos do mundo real, bem como suas características e seu comportamento.

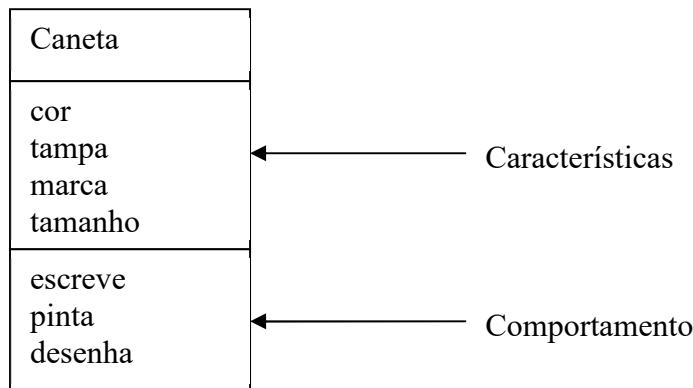
Uma caneta possui que características?

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?

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” ao invés 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 vai ser 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 á 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:

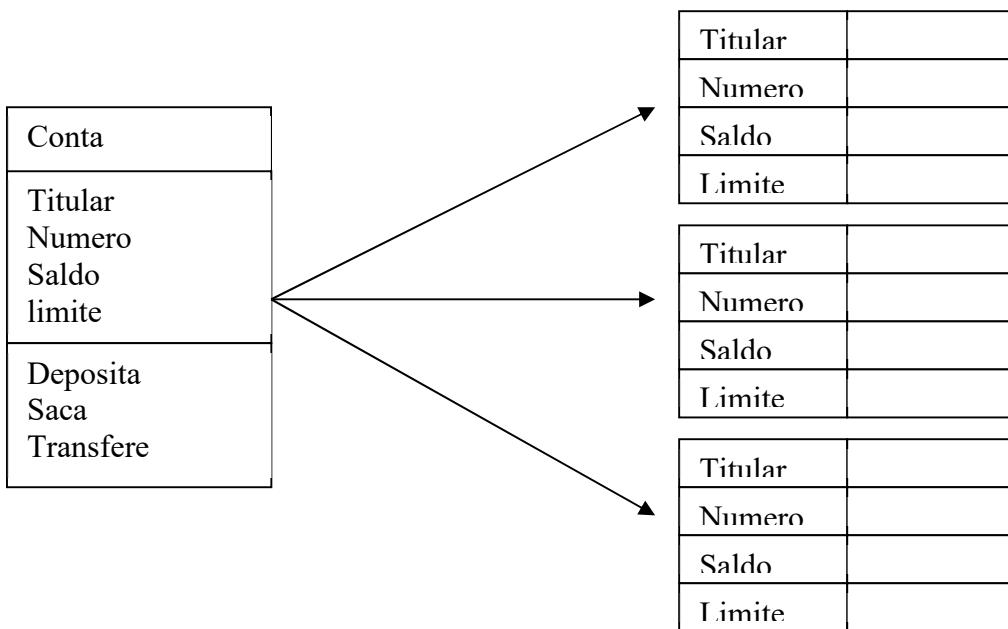
O que toda conta tem e é importante para nós?

- Número
- Nome do Titular
- Saldo
- Limite

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, um Titular e um limite, 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 forem 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, limite;  
    ...  
}
```

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 (que veremos, mais adiante, para que servem).

```
public class Programa {  
    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:

```
public class Programa {  
    public static void main(String[] args) {  
        Conta umaConta;  
        umaConta = new Conta();  
        umaConta.titular = "Fulano"; //acesso ao atributo titular  
        umaConta.saldo = 1000.0; //acesso ao atributo saldo  
  
        System.out.println("Saldo atual de "+umaConta.titular+  
": "+umaConta.saldo);  
    }  
}
```

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 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 acessei 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;

Limite = 0.0;

Numero = 0;

Exercício:

Escreva um programa que crie duas contas e acesse seus atributos. Divida a folha do seu caderno em 3 partes: Classe Conta, Programa e Memória. Faça uma simulação do que acontece.

25. 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:

```
public class Conta {  
    int numero;  
    String titular;  
    double saldo, limite;  
  
    void saca(double quantidade){  
        double novoSaldo = this.saldo - quantidade;  
        this.saldo = novoSaldo;  
    }  
}
```

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

Ao chamar o método saca devemos informar a quantia a ser sacada. Para isso usamos o argumento (ou parâmetro) quantidade. Quantidade 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 quantidade, 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 isso é opcional.

Perceba que, por enquanto, nada impede que tanto o saldo, quanto o limite fixado pelo banco, sejam estourados. Mais adiante trataremos essa situação de forma muito elegante.

Também devemos ter um método para depositar uma quantia:

```
void deposita(double quantidade) {
    this.saldo += quantidade;
    // O mesmo que this.saldo = this.saldo + quantidade;
}
```

Repare que desta vez não usamos uma variável auxiliar.

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

```
public class Programa {
    public static void main(String[] args) {
        Conta umaConta; // declara variável umaConta do tipo Conta
        umaConta = new Conta();

        umaConta.titular = "Fulano"; // acesso ao atributo titular
        umaConta.saldo = 1000.0; // acesso ao atributo saldo

        umaConta.saca(300);
        umaConta.deposita(200);

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

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:

```
public class Programa {
    public static void main(String[] args) {
        Conta umaConta;
        umaConta = new Conta();
        umaConta.titular = "Fulano"; // acesso ao atributo titular
        umaConta.saldo = 1000.0; // acesso ao atributo saldo

        boolean consegui;

        consegui = umaConta.saca(300);
        if (consegui) {
            System.out.println("Consegui sacar!");
        } else {
            System.out.println("Não consegui sacar.");
        }

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

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.

Ao invés 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 + this.limite) < valor) {
        return false;
    } else {
        this.saldo -= valor;
        return true;
    }
}

void deposita(double quantidade) {
    if (quantidade > 0)
        this.saldo += quantidade;
    // O mesmo que this.saldo = this.saldo + quantidade;
}
```

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

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);
    }
}
```

26. 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!

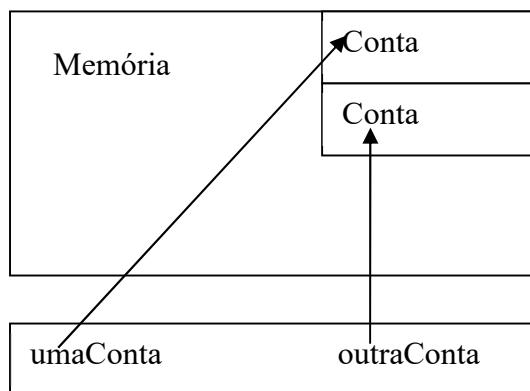
```
public static void main(String[] args) {  
    Conta umaConta; // declarando uma variável de referência  
    para um objeto do tipo Conta.  
    umaConta = new Conta();  
}
```

É comum os programadores dizerem que tem um objeto umaConta do tipo Conta. Isso é só pra 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 nº 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?

Exercícios:

```
if(c1 == c2)
    System.out.println("Iguais");
else
    System.out.println("Diferentes");
}
```

1)Qual será a resposta para o código acima?

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 este 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?

27.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.

28.Princípio da Alta Coesão

Até aqui criamos uma classe Conta que possui atributos (titular, numero, saldo, limite) 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 por que 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 freqüentemente.
- 25% das rotinas são usadas ocasionalmente.
- 30% das rotinas nunca são usadas.

29. 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:

```
public class Conta {  
    int numero;  
    Cliente titular = new Cliente();  
    double saldo, limite;  
  
    void saca(double quantidade) {  
        this.saldo -= quantidade;  
    }  
}
```

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

```
public class TestaContaEstouro1 {  
    public static void main(String[] args) {  
        Conta minhaConta = new Conta();  
    }  
}
```

```
        minhaConta.saldo = 1000;
        minhaConta.limite = 1000;
        minhaConta.saca(5000); // saldo + limite é só 2000!!
    }
}
```

Para evitar uma conta com saldo negativo após um saque, podemos utilizar um IF no método saca().

Na verdade, já fizemos isso anteriormente. Isso melhora bastante as coisas. Contudo, quem garante que o usuário da classe vai sempre usar o método saca() para alterar o saldo da conta?

O código a seguir ultrapassa o limite diretamente. Veja:

```
public class TestaContaEstouro2 {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.saldo = 1000;
        minhaConta.limite = 1000;
        minhaConta.saldo = -3000; // meu saldo agora é de - 1000, ou
        seja,
                                            // negativo!!
    }
}
```

Como evitar isso? Uma forma simples de se resolver isso seria testar se não estamos ultrapassando o saldo+limite 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 acessasse 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 double saldo;
    private double limite;
//...
```

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 um atributo limite que está sendo checado.

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 quantidade) {  
    if ((this.saldo + this.limite) >= 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 e métodos 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 faríamos 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.

30.O ambiente BlueJ

Utilizando o software BlueJ para ajudar na compreensão da diferença entre classe e objeto.

O BlueJ é um ambiente de desenvolvimento Java desenvolvido e mantido na Universidade de Deakin, Austrália, e na Universidade de Kent, Reino Unido, explicitamente como um ambiente de ensino introdutório à programação orientada a objetos.

Mais adiante trabalharemos com o ambiente Eclipse (ou Eclipse IDE). Contudo, neste momento, o BlueJ apresenta-se como um ambiente mais adequado, que facilita e muito o aprendizado do paradigma orientado a objetos e da linguagem Java em comparação a outros ambientes de desenvolvimento, por diversas razões:

- **A interface com o usuário é muito mais simples:** Iniciantes conseguem utilizar esta ferramenta depois de poucos minutos de aprendizado. Sua interface é muito amigável e simples de se utilizar.
- **O BlueJ suporta importantes ferramentas de ensino não-disponíveis em outros ambientes.**

Uma delas é a visualização da estrutura de classes. O BlueJ automaticamente exibe um diagrama UML que representa as classes e seus relacionamentos em um determinado projeto. Isso é de grande ajuda para professores e alunos!

É difícil assimilar o conceito de um objeto quando tudo que se vê na tela são linhas de código! A visualização de um diagrama UML facilita muito o entendimento do aluno iniciante. Outra vantagem é que o aluno já vai ganhando noção da notação UML para cursos (ou disciplinas) posteriores.

- A grande força do BlueJ está na possibilidade de se criar e visualizar os objetos numa bancada que pode ser encarada como a memória do computador. Com isso o aluno pode interagir diretamente com os objetos. O aluno pode criar e destruir objetos na memória. Pode chamar seus métodos, passando parâmetros para os mesmos, pode visualizar o estado de seus atributos. Tudo isso de forma visual e não através de linhas de comando! A distinção entre classe e objeto é uma das grandes dificuldades dos alunos iniciantes. Utilizando o BlueJ, não há como fazer confusão entre classe e objeto. Em minha opinião esse é o grande trunfo do BlueJ! O aluno pode escrever classes e imediatamente criar objetos a partir delas, sem a necessidade de uma classe de testes, por exemplo!

Infelizmente grande parte das instituições de ensino desconhece ou ignora essa poderosa ferramenta de aprendizado. É uma ferramenta que desacelera um pouco o andamento da matéria e infelizmente, hoje em dia, há uma preocupação muito grande em introduzir conceitos e cada vez mais conteúdo de forma acelerada (goela abaixo) para os alunos. Eu discordo completamente desta prática. Penso que não se aprende a programar sem muita prática e tempo adequado para tratar cada conceito.

Eu optei pelo uso do BlueJ no início deste curso por que entendo que a assimilação dos conceitos da orientação a objetos é a base de tudo e se não gastarmos tempo e as ferramentas adequadas nesta parte do processo, todo o aprendizado posterior tende a ficar comprometido. Vamos construir uma base sólida para que o aluno se sinta confortável para ir adiante!

Mais adiante abandonaremos o BlueJ, aos poucos (afinal, trata-se de uma ferramenta introdutória), para começar a utilizar o Eclipse IDE, ferramenta líder de mercado e que possui uma produtividade muito interessante. Tudo no momento adequado! Aproveitaremos ao máximo o que cada ferramenta pode nos oferecer de melhor!

O BlueJ pode ser baixado no endereço: <http://www.bluej.org/>
Há um tutorial interessante nesta página. Contudo, gosto muito de um outro tutorial, mais simples e resumido (em português), na página:
<https://4friendstech.wordpress.com/2011/05/30/java-com-bluej-01/>
Recomendo fortemente essa pequena leitura!

Importante:

Neste momento, é importante seguir o seguinte roteiro para instalação em casa:

- 1) Baixar a última versão do JDK (atualmente estamos na versão 8) em:
<http://www.oracle.com/technetwork/java/javase/downloads/>
- 2) Instalar o JDK e configurar as variáveis de ambiente (apêndice no fim dessa apostila);
- 3) Baixar e instalar o BlueJ em: <http://www.bluej.org/> Lembre-se de escolher a opção sem o JDK (BlueJ Installer);

O Eclipse pode ser baixado e instalado posteriormente.

Exercício p/ fazer com o professor:

- 1) Execute o BlueJ e Abra o projeto chamado formas (será fornecido pelo professor) e brinque um pouco com os objetos e seus métodos. Acredite: Isso vai ajudá-lo a compreender a diferença entre classe e objeto, de uma vez por todas!

Exercícios:

- 1) No BlueJ, crie um projeto chamado projeto-banco. Adicione a classe Conta a este projeto (CTRL+C CTRL+V);
- 2) Faça testes com a classe conta. Observe a bancada de objetos (que simula a memória do computador).
- 3) Crie várias contas e chame seus métodos usando o botão direito do mouse.

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.

Veja como ficaria:

```
public void transfere(Conta contaDestino, double valor) {  
    this.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) {  
    boolean sacou = this.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.

Ao invés 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!

Exercício (Usando o BlueJ):

- 1) O método transfere pode ser escrito de forma mais sucinta. Reescreva-o já fazendo a mudança do nome também!
- 2) Escreva (ou reescreva) a classe TestaConta para instanciar duas contas, sacar, depositar e transferir dinheiro de uma conta para outra.
- 3) Crie uma pasta projeto-banco e coloque as classes Conta e TestaConta dentro desta pasta.
- 4) Execute TestaConta e faça todos os testes possíveis!

31.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
---	--

Interface
(o que faz! Porção visível para o mundo externo).

Implementação
(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.

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 receiveCpf. 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.

32. 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 {  
    private double saldo;  
    private double limite;  
    int numero;  
    String titular;  
  
    public double obtemSaldo() {  
        return this.saldo;  
    }  
    //saca(), deposita() e transfere() omitidos
```

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

```
public class TestaPegaSaldo {  
    public static void main(String[] args) {  
        Conta c1 = new Conta();  
        c1.deposita(200);  
        System.out.println("O saldo atual é: " + c1.obtemSaldo());  
    }  
}
```

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:

```
public class Conta {  
  
    private double saldo;  
    private double limite;  
    int numero;  
    String titular;  
  
    public double getSaldo() {  
        return this.saldo;  
    }  
  
    public void setsaldo(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public double getLimite() {  
        return limite;  
    }  
  
    public void setLimite(double limite) {  
        this.limite = limite;  
    }  
  
    public int getNumero() {  
        return numero;  
    }
```

```
public void setNumero(int numero) {
    this.numero = numero;
}

public String getTitular() {
    return titular;
}

public void setTitular(String titular) {
    this.titular = titular;
}

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

public void deposita(double quantidade) {
    this.saldo += quantidade;
}

public boolean transferePara(Conta destino, double valor) {
    boolean sacou = this.saca(valor);
    if (sacou) {
        destino.deposita(valor);
        return true;
    } else {
        return false;
    }
}
}
```

Agora vamos pensar em uma conta de verdade: existe a possibilidade do 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, criamos um método setSaldo() que permite que o saldo seja modificado sem estar atrelado a uma operação bancária. Este método não deveria ter sido 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 quer 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 ao apagar o método setSaldo() e modificar o método getSaldo() estará tudo bem com a nossa conta né? Existe a chance de ela ficar com menos dinheiro que o limite? Ao primeiro olhar pode parecer que não, mas existe.

E se depositarmos um valor negativo na conta? Não esperávamos por isso. Para proteger nossa conta desta situação, basta alterarmos a implementação do método `deposita()` para que ele verifique se o valor é positivo. Feito isso precisamos modificar mais alguma coisa? Graças 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.

33.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;  
  
    // ...  
}
```

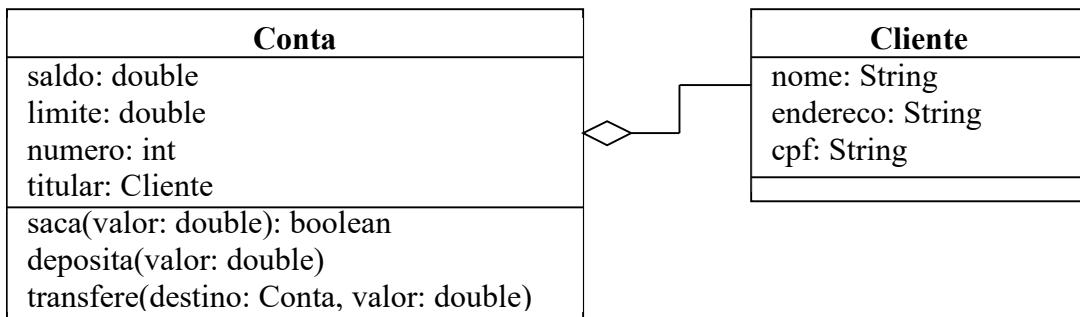
No caso acima, quando você criar uma conta, seus atributos já estarão populados. Mas essa não é a melhor opção nesse caso.

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!

Exercícios no BlueJ

- 1) Esqueça a classe TestaCarro por enquanto. Vamos criar apenas um carro e um motor.
Vamos adicionar o motor ao carro e brincar com seus métodos!

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:
 - o 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, portal1, portal2, 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!

34.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**.

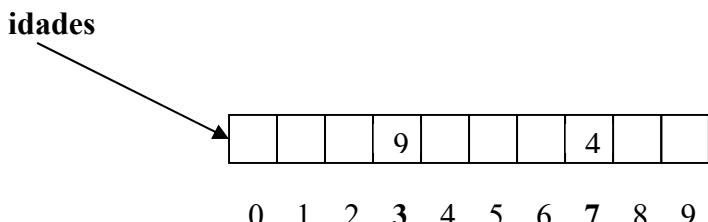
Como criar o **objeto-array**?

```
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^a e 8^a 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.

35. 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.

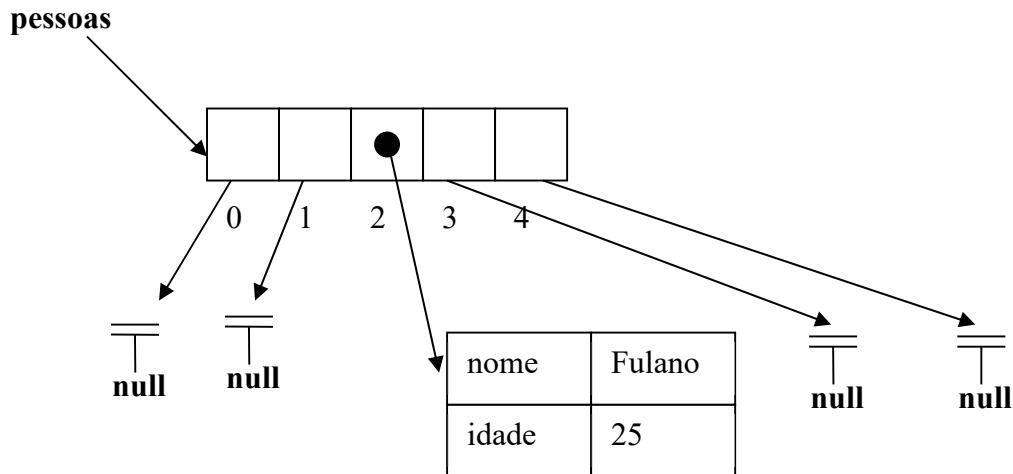
Veja:

```
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^a 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.

36. 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 mostra empregados 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.

37. 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.

38. 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.

39.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);
    }
}
```

40.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**.

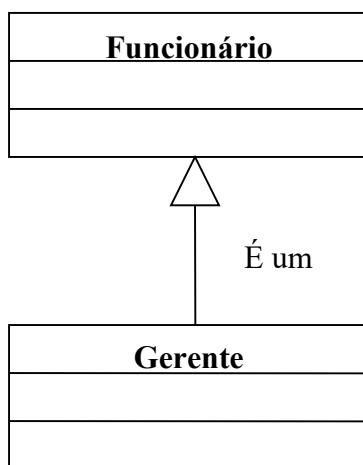
```
public class Gerente extends Funcionario{
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    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**.

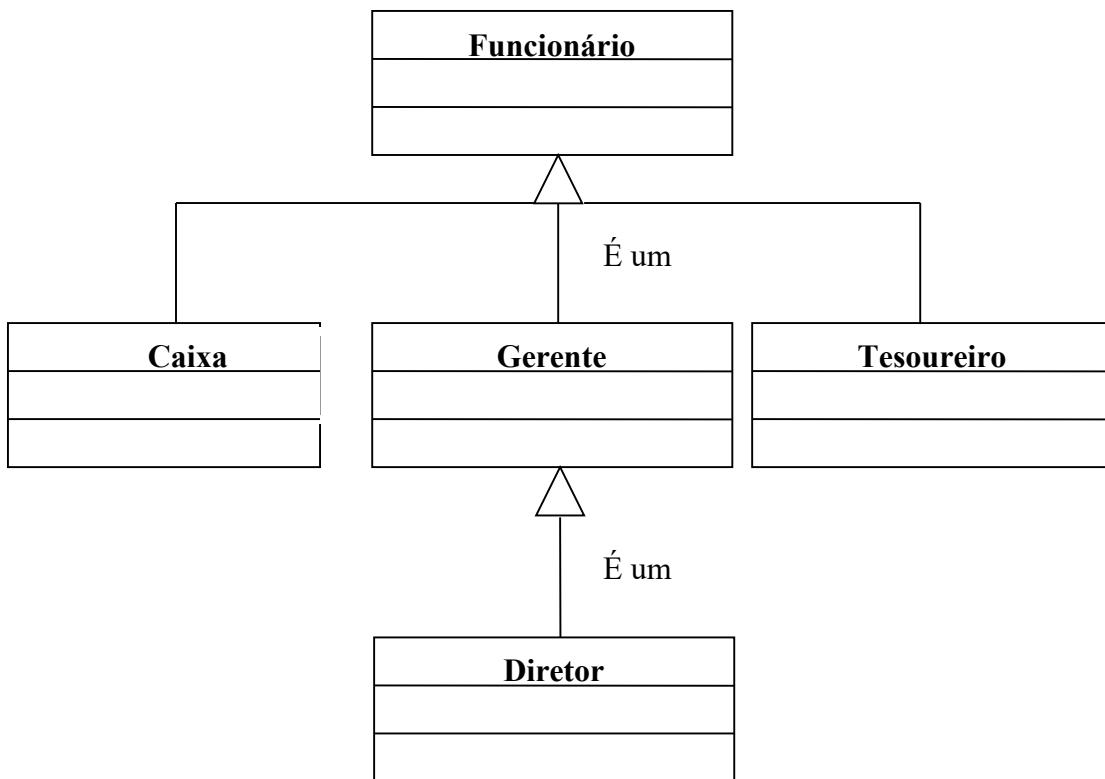
Perceba que há outras possibilidades para o nosso banco. Podemos ter:

- 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.

41. 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?

42. Polimorfismo

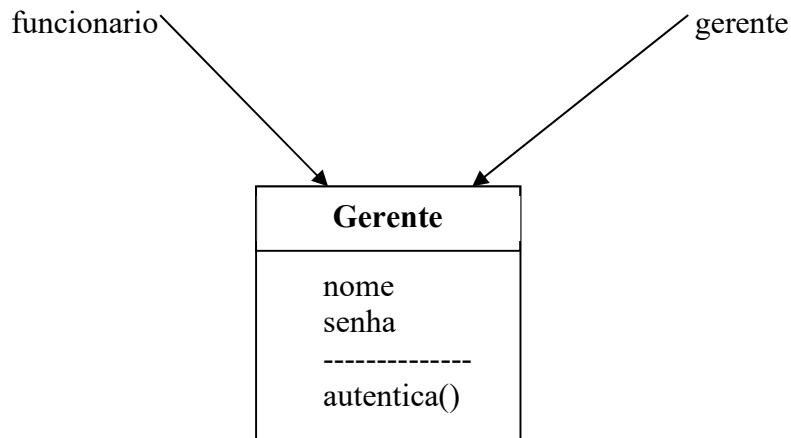
Em uma variável do tipo Funcionário você não guarda um objeto do tipo Funcionário. Você guarda, na verdade, **uma referência** para um Funcionário.

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 selic;

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

    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?

43. 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). 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).

Nota: Hoje em dia a Sun pertence a Oracle.

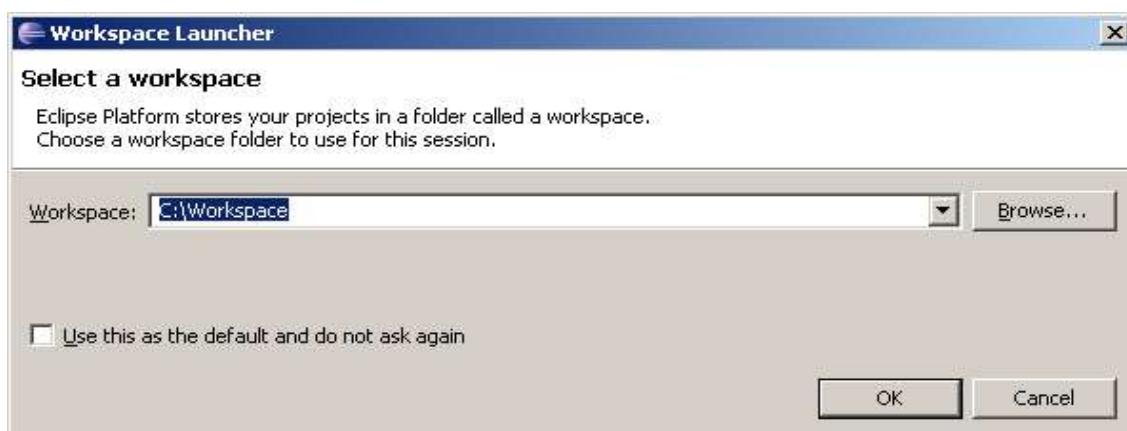
Particularmente eu prefiro o Eclipse. O Netbeans tem o “arrastar e soltar” que acaba gerando trechos de código por conta própria. O Eclipse nos deixa mais donos do nosso próprio código. Por isso é líder de mercado!

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.



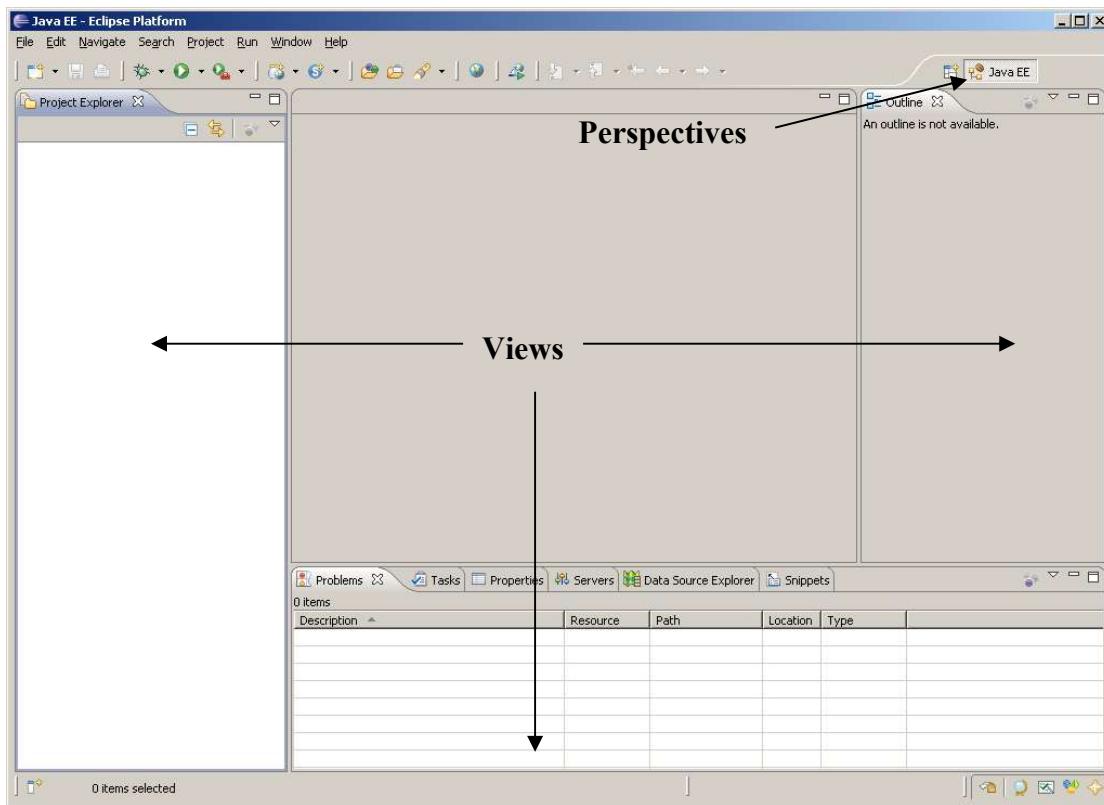
Você tem a opção de deixar o diretório pré-definido. No nosso caso, podemos criar um diretório WorkspaceEnsinoMedio dentro de c:\dev.

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.

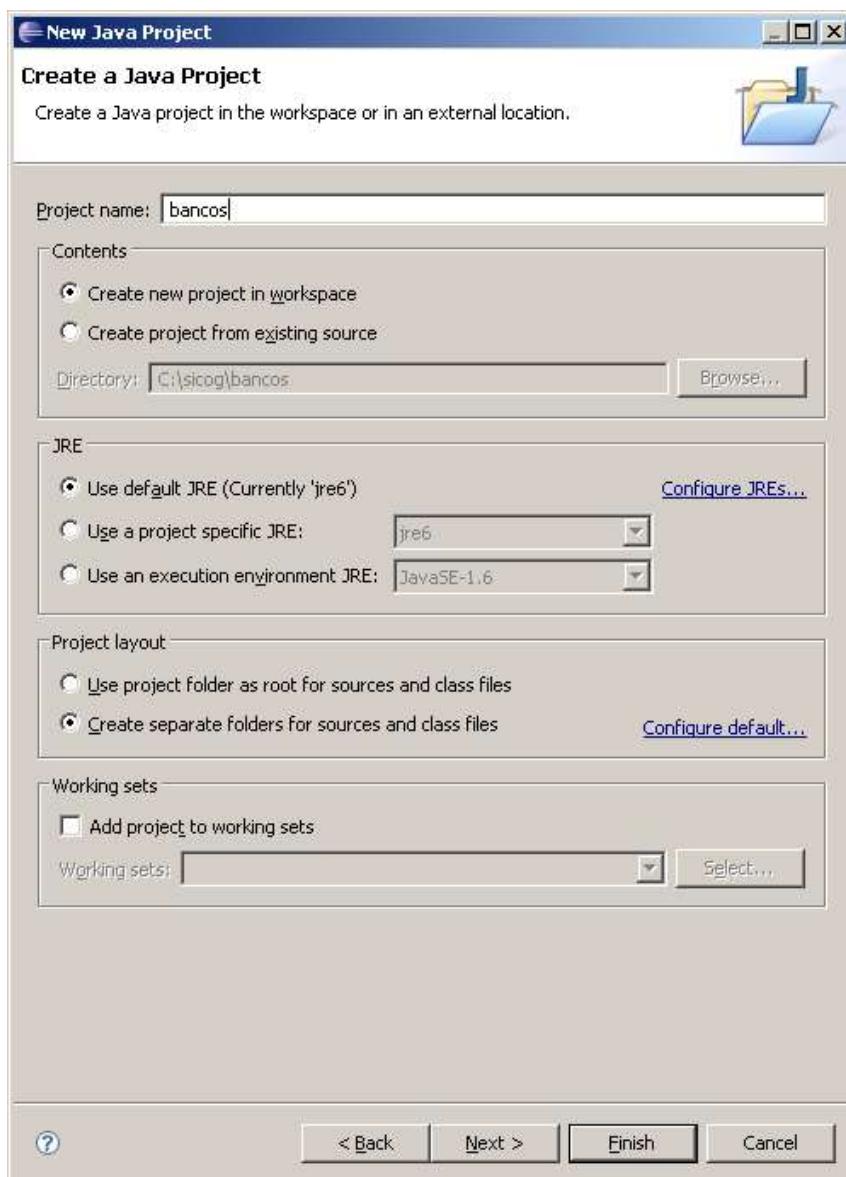
Criando um novo Projeto:

Vá em **File → New → Project**. Selecione (ou digite) Java Project e clique em Next.



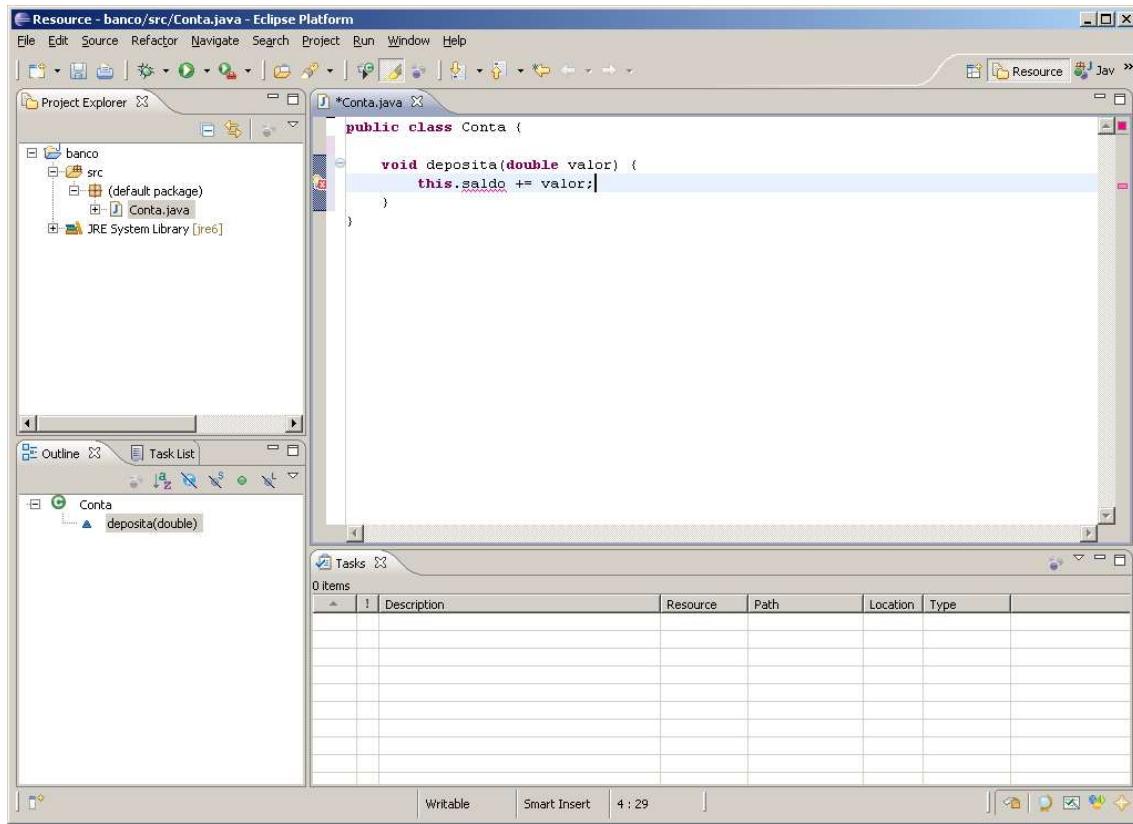
Crie um projeto chamado banco.

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:

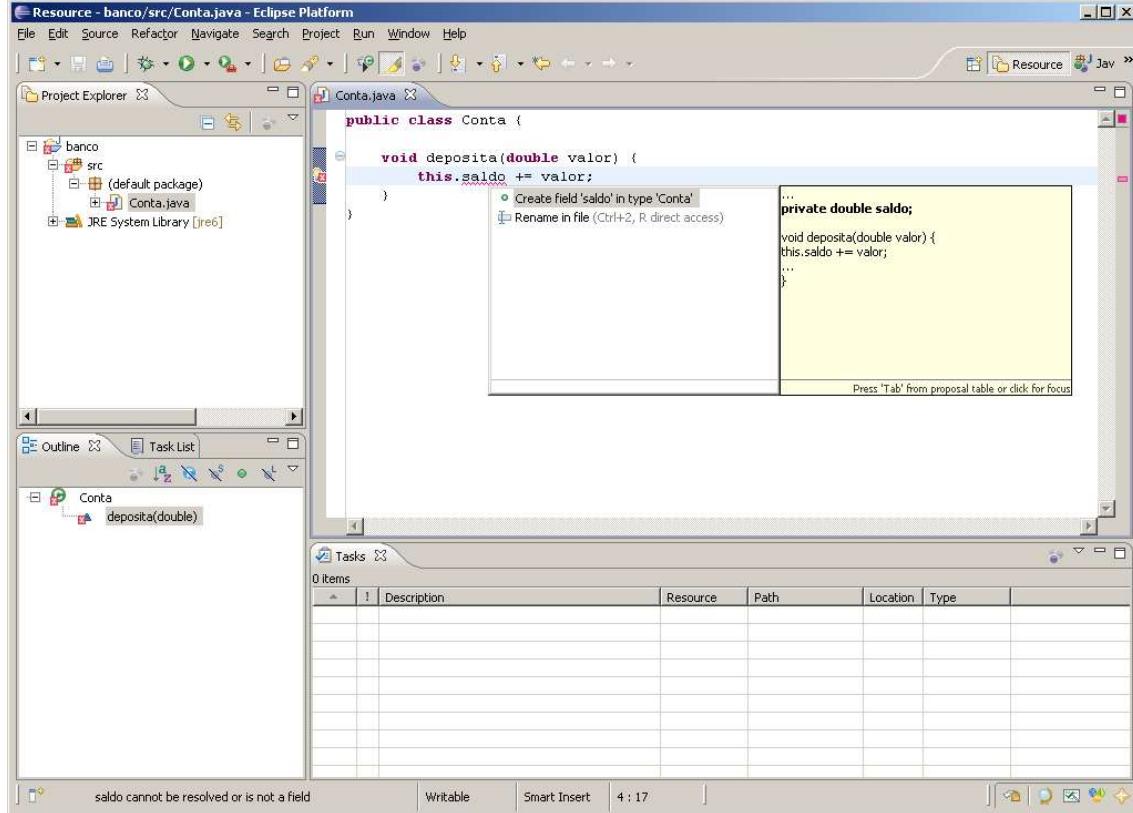


Clique em Finish. O Eclipse possui diversos wizards, mas usaremos o mínimo deles. O que interessa é usar o **code assist** e **quick fixes** que a ferramenta possui.

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.



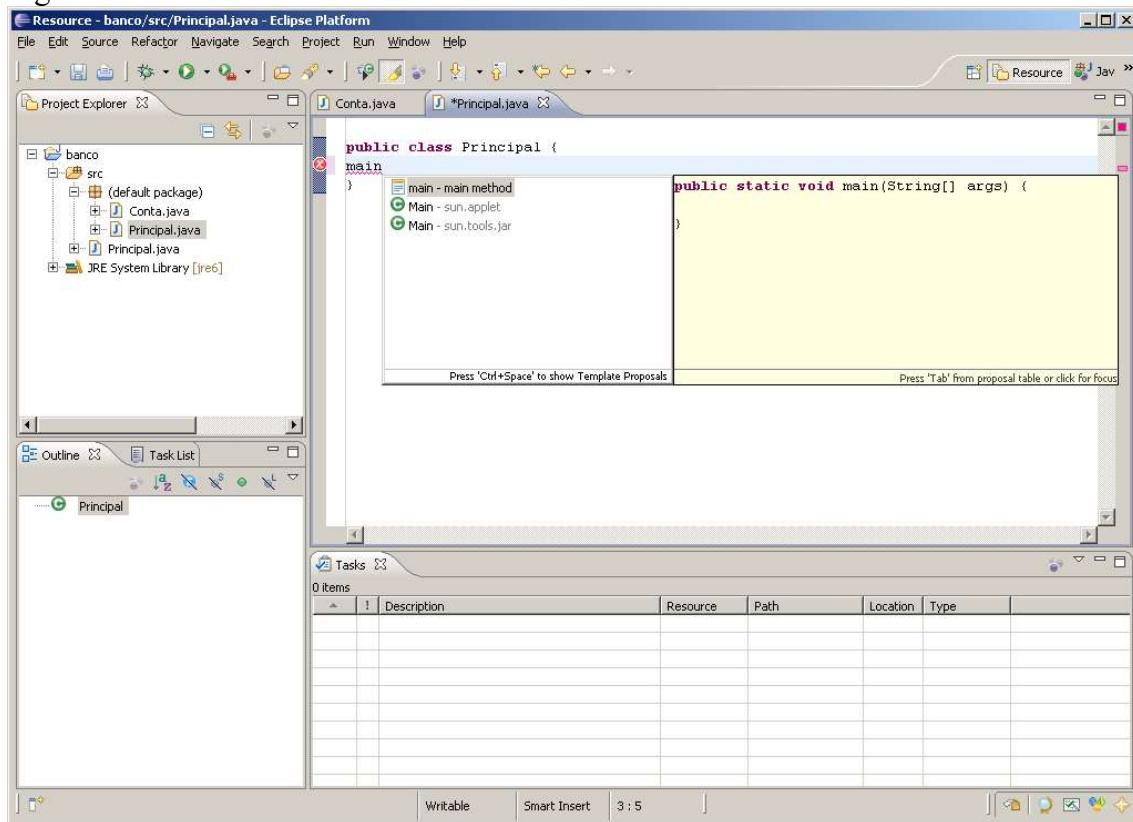
Perceba que ele vai sugerir possíveis formas de consertar o erro. Uma delas é, justamente, criar o campo saldo na classe Conta, que é nosso objetivo. Escolha esta opção e ele vai criar o campo saldo pra você.

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:

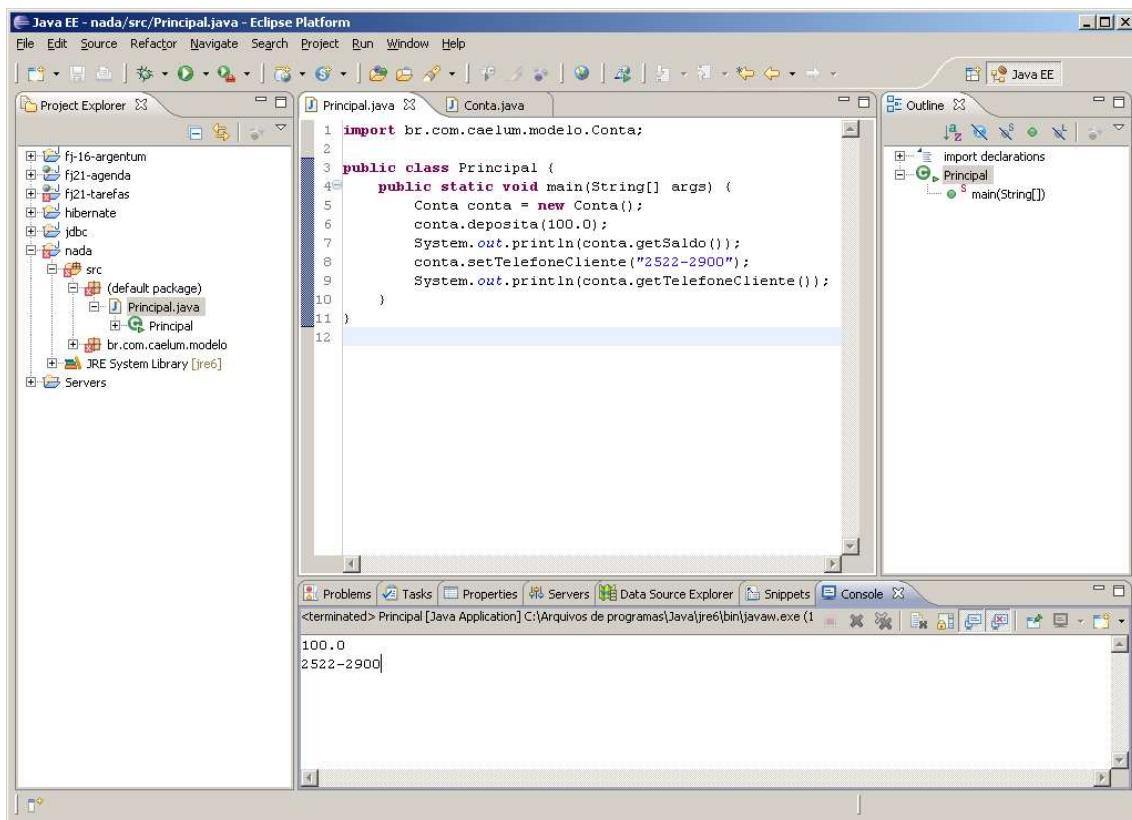
```
System.out.println(conta.getTelefoneCliente());
```

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();

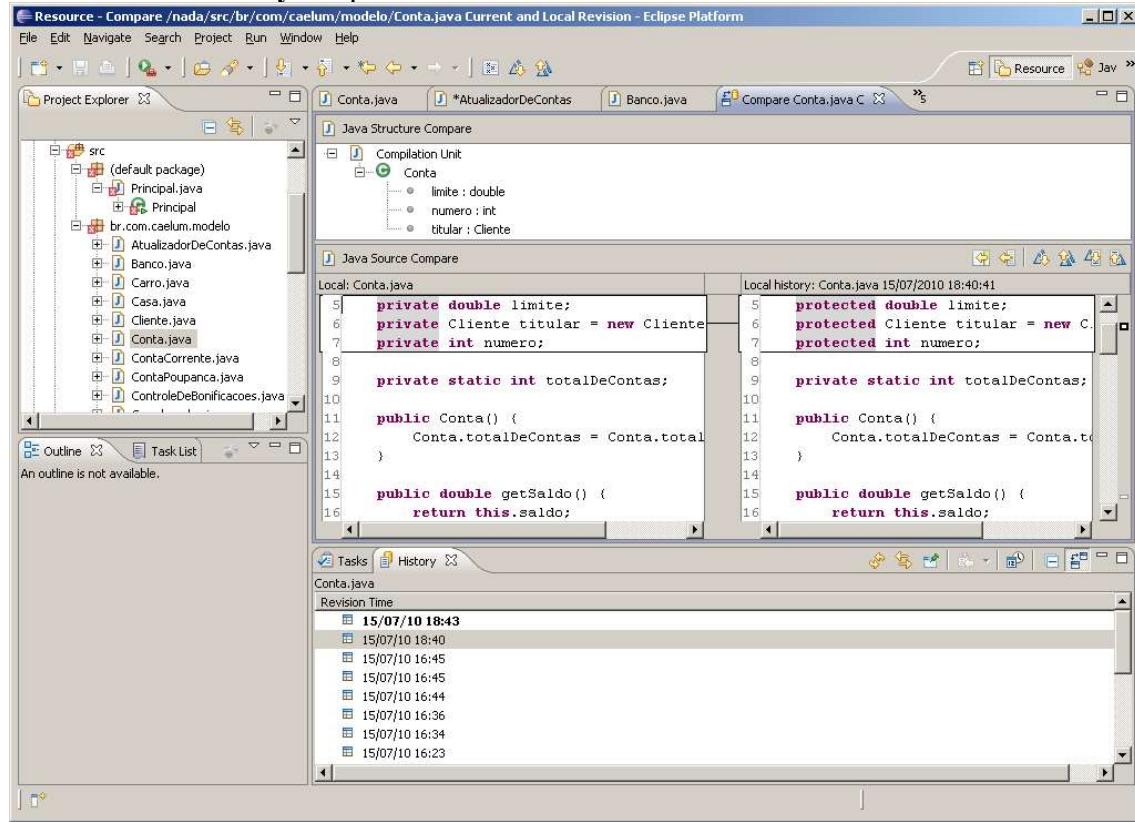
        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);

        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
());
    }
}
```

44.Classes Abstratas

Volte umas duas páginas e relembre como ficaram as classes Funcionario, Gerente, Caixa e ControleDeBonificações.

O que é realmente mais importante, herança ou polimorfismo?

Repare no método registra. 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 serem 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 registra 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 um cara que é simplesmente um Funcionário do banco. Ele 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 Diretor, um Gerente, algo mais **concreto** que um Funcionário.

Veja o código abaixo:

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();
Funcionario f = new Funcionario();
controle.registra(f); //Isso faz sentido?????
```

Outros exemplos:

Imagine a classe Pessoa e duas filhas: PessoaFísica e PessoaJurídica. Existe uma pessoa que não seja física ou Jurídica? Nesse caso, a classe Pessoa estaria sendo usada 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 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 “plaquinha” 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 façá 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 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 resolvermos 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 registra(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 programa-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.

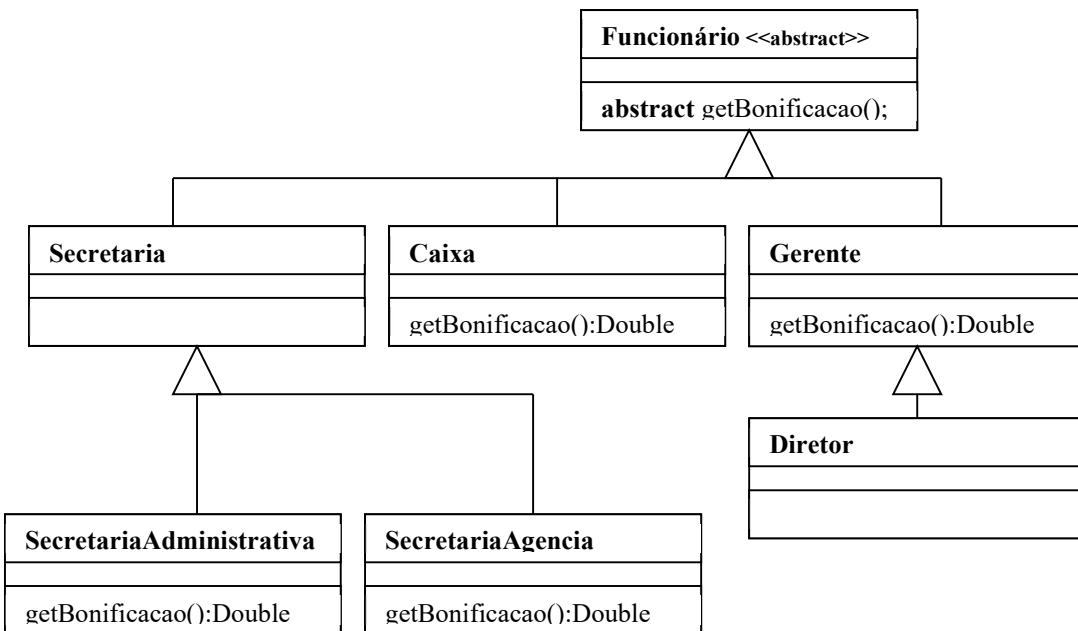
Mas então eu continuo tendo que escrever um método getBonificação na classe Funcionário? Não necessariamente. Basta escrever a palavra chave abstract antes da assinatura do método e colocar um ponto e vírgula em vez de abre e fecha chaves. Você só escreve a assinatura. A implementação do método fica a cargo das subclasses!

```
abstract double getBonificacao();
```

Como esse método nunca vai ser invocado por um objeto do tipo Funcionário, não há necessidade 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.

Exercícios sobre classes abstratas:

- 1) 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.
- 2) 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?
- 3) 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?

- 4) Reescreva o método atualiza na classe ContaPoupanca para que a classe possa compilar normalmente. O Eclipse também sugere isso com um quick fix.
- 5) Existe outra maneira da classe ContaCorrente compilar se você não escrever o método abstrato?
- 6) 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?
- 7) Posso chamar um método abstrato de dentro de outro método da própria classe abstrata? Exemplo: O mostra() de Funcionário pode chamar o this.getBonificação?
- 8) Não podemos dar new em Conta, mas porque então, podemos dar new em Conta[10], por exemplo?

45. Interfaces e Sobrecarga

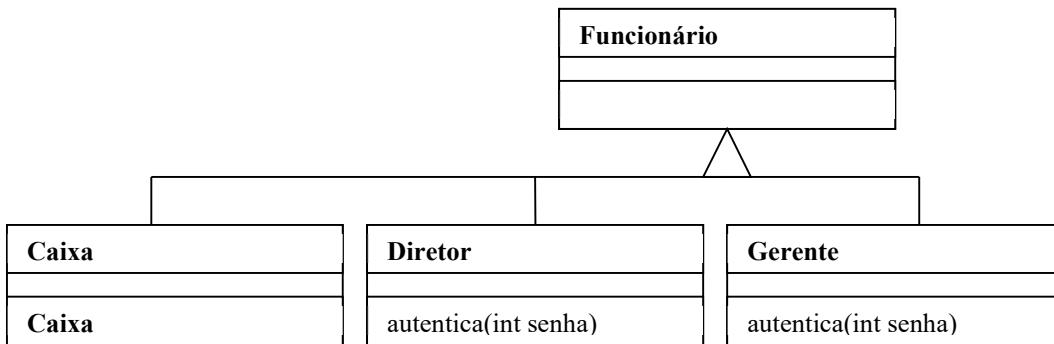
Vamos imaginar que um Sistema de Controle do Banco pode ser acessado pelo Gerente e agora também pelo Diretor do Banco. Vamos supor que ambos estendam funcionário.

```
public class Diretor extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public boolean autentica(int senha) {  
        //Verifica se a senha confere com a recebida por parâmetro.  
    }  
    //...  
  
public class Gerente extends Funcionario {
```

```
int senha;

public boolean autentica(int senha) {
    //Verifica se a senha confere com a recebida por parâmetro.
    //No caso do gerente, verifica também se o departamento dele tem acesso
}

//...
```



Repare que o método de autenticação de cada tipo de funcionário pode variar muito. Também é importante observar que nem todo tipo de funcionário tem o referido método. Vamos aos problemas. Considere a classe SistemaBancario e seu controle: precisamos receber um Gerente ou um Diretor como argumento, verificar se ele se autentica e permitir seu acesso ao Sistema.

```
public class SistemaBancario {
    private int senha=123456;
    void login(Funcionario funcionario) {
        funcionario.autentica(senha);
        //Não compila! Nem todo Funcionário tem o método autentica();
    }
}
```

O SistemaBancario aceita qualquer tipo de Funcionário, tendo ele acesso ao Sistema ou não, mas o problema é que nem todo Funcionário tem o método autentica. Isso nos impossibilita de chamar esse método apenas com uma referência a Funcionário. O que devemos fazer então?

Uma possibilidade é criar dois métodos login no SistemaBancario. Um para receber um Diretor e outro para receber um Gerente. Mas já vimos anteriormente que essa não é uma boa opção. Por que?

```
public class SistemaBancario {
    private int senha=123456;
    //Design problemático
    void login(Gerente funcionario) {
        funcionario.autentica(this.senha);
    }
    //Design problemático
    void login(Diretor funcionario) {
        funcionario.autentica(this.senha);
    }
}
```

Cada vez que criarmos uma nova classe filha de Funcionário, que é autenticável, precisaríamos adicionar um novo método login **sobreescrito** no SistemaBancario.

Sobreescrita de métodos:

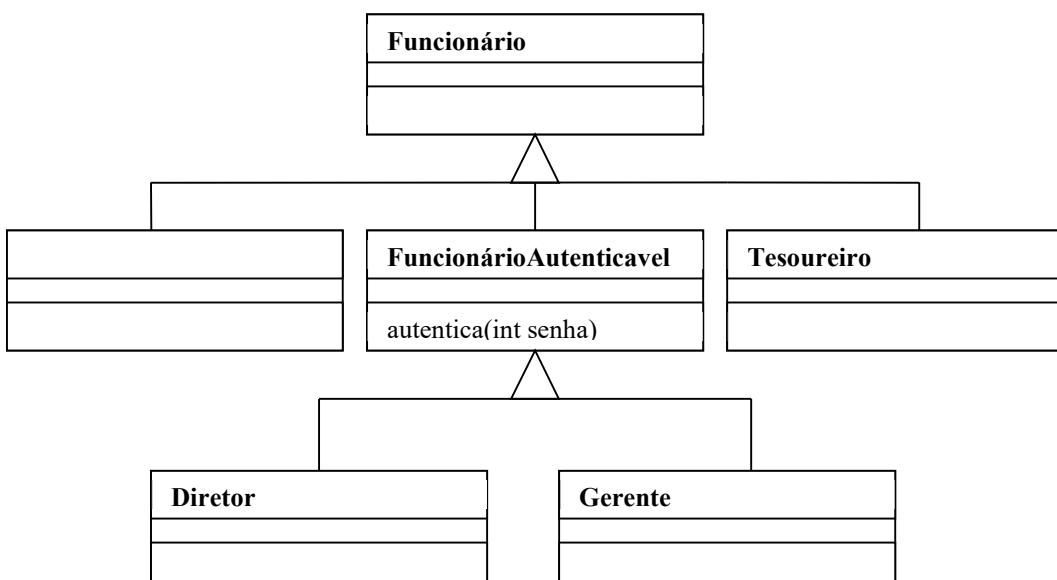
Repare no código acima que temos dois métodos com o mesmo nome. Em Java métodos podem ter o mesmo nome, desde que não sejam ambíguos, isto é, que exista uma maneira de distinguir no momento da chamada. Perceba que o nome é o mesmo, mas a assinatura não é. O tipo de parâmetro é diferente. O mesmo acontece quando criamos mais de um construtor, como fizemos anteriormente. Isso se chama **Overload (sobreescrita)** que é diferente de **Override (reescrita)**.

Voltando ao nosso problema, uma solução mais interessante seria criar uma classe no meio da árvore de herança chamada FuncionarioAutenticavel:

```
public class FuncionarioAutenticavel extends Funcionario {  
    public boolean autentica(int senha) {  
        // faz autenticação padrão  
    }  
}
```

As classes Diretor e Gerente agora ao invés de estender Funcionário, estenderiam FuncionarioAutenticavel. Desse modo o SistemaBancario ficaria assim:

```
public class SistemaBancario {  
    private int senha = 123456  
    //Design problemático  
    void login(FuncionarioAutenticavel fa) {  
        //Aqui eu posso chamar autentica, pois todo  
        FuncionarioAutenticavel tem este método  
        boolean ok = fa.autentica(senha);  
    }  
}
```



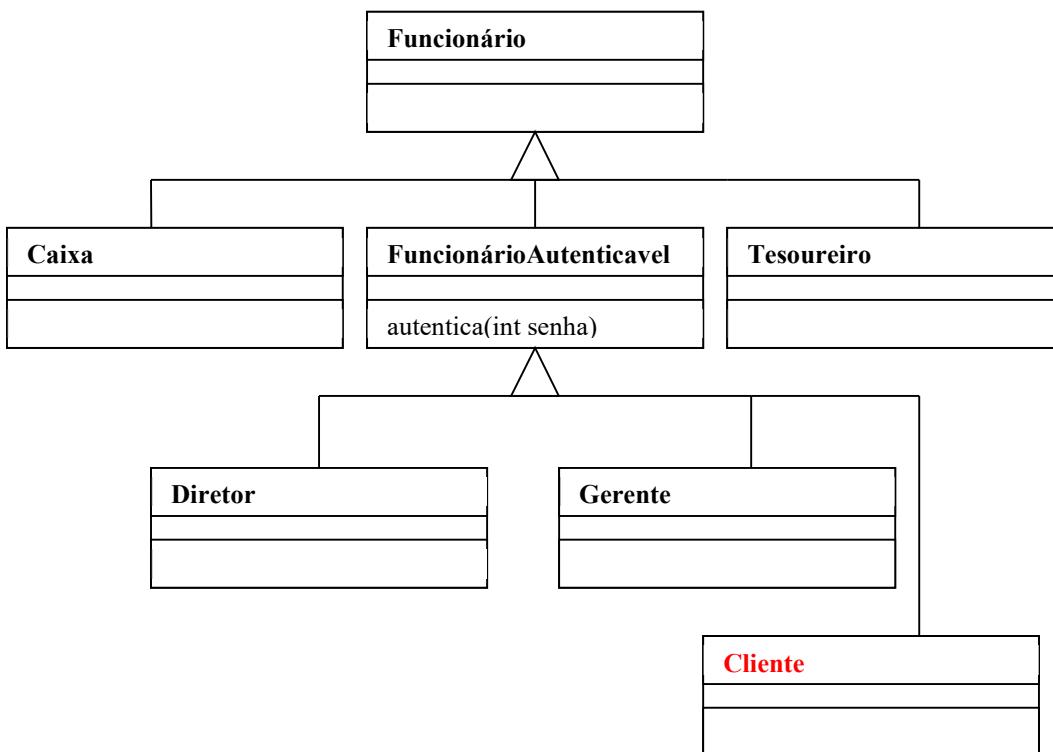
Repare que FuncionarioAutenticavel é forte candidata a classe abstrata. O método autentica também é forte candidato a método abstrato.

Aparentemente o uso de Herança resolve esse caso.

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 “gambiarra”, 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”.



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.

Interfaces:

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 boleano.

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!**

```
public interface Autenticavel {  
    boolean autentica(int senha);  
}
```

O nome disso é **interface**!

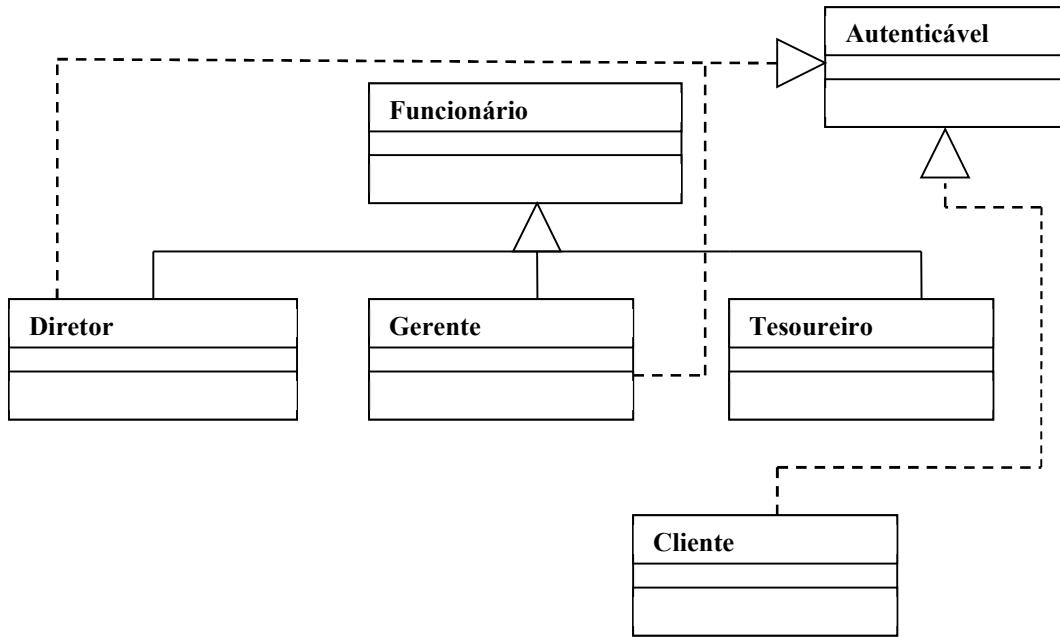
Interface é a maneira pela qual poderemos conversar com um Autenticável. **Interface** é a maneira através da qual conversamos com um objeto.

Lemos a interface da seguinte maneira: “*quem desejar ser Autenticável precisa saber autenticar, recebendo um inteiro e retornando um boleano*”. É um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Uma interface pode definir uma série de métodos, mas nunca 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).

O Gerente pode “assinar” o contrato, ou seja, implementar a interface. Ao implementar essa interface ele precisa escrever os métodos pedidos pela interface. Isso é muito parecido com o efeito de herdar métodos abstratos. Aliás, métodos de uma interface são públicos e abstratos, sempre! Para implementar, usamos a palavra chave **implements** na classe.

```
public class Gerente extends Funcionario implements Autenticavel{  
    private int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    //outros atributos e métodos...  
  
    public boolean autentica(int senha){  
        if(this.senha==senha){  
            System.out.println("Acesso Permitido!");  
            return true;  
        }else{  
            System.out.println("Acesso Negado.");  
            return false;  
        }  
    }  
}
```



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!
  
```

Novamente, a solução mais adequada seria receber um Autenticável como argumento. Veja como ficaria nosso SistemaBancario:

```

public class SistemaBancario {
    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(ok==true) {
            ; // Instruções
        }
    }
}
  
```

Pronto! Agora podemos passar qualquer Autenticável para o SistemaBancario.

Suponha que agora precisamos fazer com que Diretor também implemente essa interface. Fica fácil resolver isso! Basta fazer com que Diretor implemente Autenticavel e automaticamente ele será obrigado a fornecer uma implementação do método autentica!

No dia em que tivermos mais um 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 Fornecedor precisa acessar o Sistema, basta que ele implemente Autenticável. Olha só o tamanho do desacoplamento: quem escreveu SistemaBancario só precisa saber que ele é Autenticável.

Não é 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 ao invés de herança é amplamente aconselhado.

No livro Design Patterns, logo no inicio, os autores citam 2 regras de ouro do Java:

- “Evite herança, prefira composição”;
- “Programe voltado a interface e não à implementação”. Dê mais importância ao “o que faz” e não ao “como faz”.

Mais adiante veremos mais sobre interfaces.

Exercícios 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 ia 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 como argumento para alguém 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.

Exercícios de interface parte 2 (O professor deverá fornecer um projeto iniciado)

- 1) Nossa 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”.

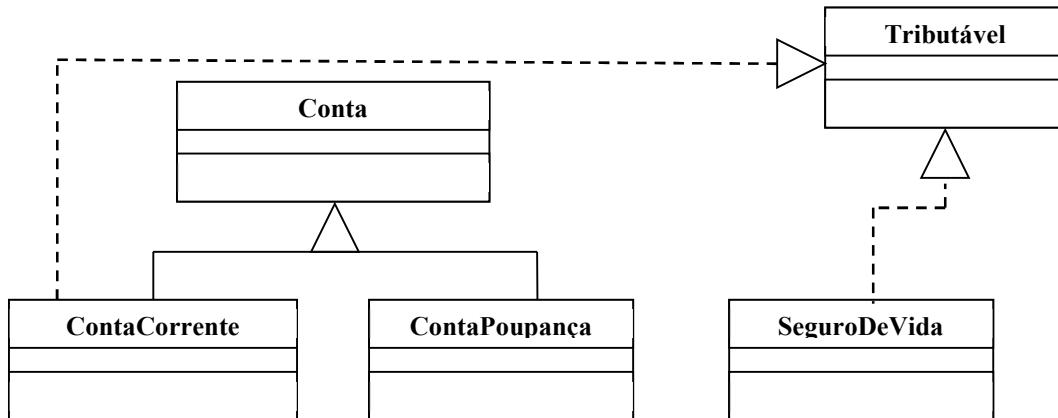
Alguns bens são tributáveis, outros não. ContaPoupanca não é Tributável. Já para ContaCorrente você precisa pagar 1% da conta. A classe que representa o seguro de vida tem uma taxa fixa de 50 reais.

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;  
    }  
}
```

Crie a classe SeguroDeVida, aproveitando o quick fix para obter:

```
public class SeguroDeVida implements Tributavel{
    @Override
    public double calculaTributos() {
        return 50;
    }
}
```



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
        Tributavel t = cc;
        System.out.println(t.calculaTributos());
    }
}
```

Agora tente chamar o método getSaldo através da referência t. O que ocorre? Por quê? A linha `Tributavel 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.

- 2) 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(Tributavel t) {  
        System.out.println("Adicionando tributável+t");  
        this.total = t.calculaTributos();  
    }  
  
    public double getTotal() {  
        return this.total;  
    }  
}
```

- 3) Escreva um programa onde vamos instanciar diversas classes que implementam Tributável e passar como argumento para 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.

- 4) Use o método printf para imprimir o total com exatamente duas casas decimais:

```
System.out.printf("O saldo é: %.2f", gerenciador.getTotal());
```

Mais exercícios sobre interfaces:

- 5) 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? Isso é tão ruim?

6) 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;
- * Não pode ter métodos estáticos, nem final, nactive, 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;
- * A classe que implementa uma interface pode declarar exceções em tempo de execução em qualquer método. Mas não pode declarar uma exceção nova ou mais abrangente.

Pode ter as seguintes declarações:

```
public final double MINIMO = 622; //Constante  
public static final int numero=5; //Constante da Classe
```

Não pode ter as seguintes declarações:

```
public static int contador=0; //Variavel da classe  
public String nome="Fulano"; //Variável comum
```

46. 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 + this.limite) >= quantidade) {  
        this.saldo -= quantidade;  
        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 auto-atendimento 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 onde 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:

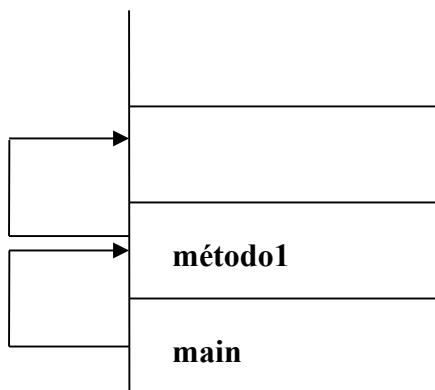
```
public class TesteErro {
    public static void main(String[] args) {
        System.out.println("Inicio do main");
        metodo1();
        System.out.println("Fim do main");
    }

    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);
        }
    }
}
```

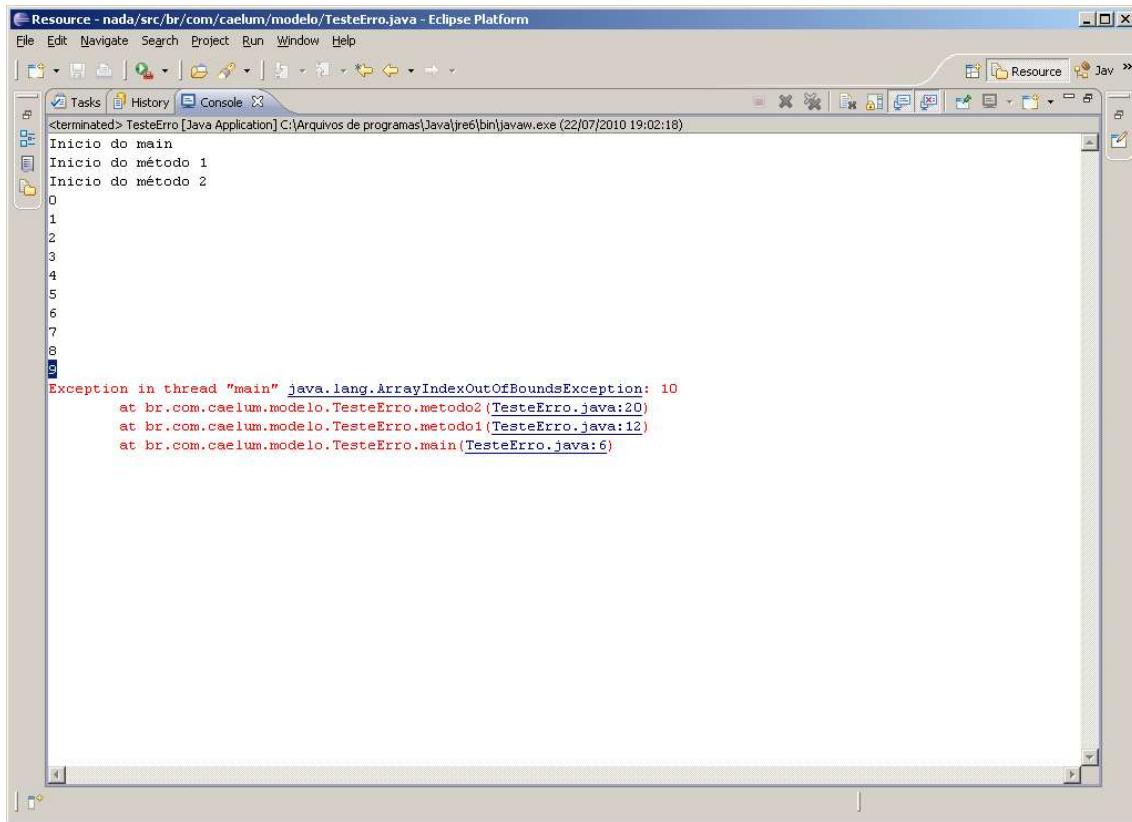
```
        }
        System.out.println("Fim do método 2");
    }
}
```

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?



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.

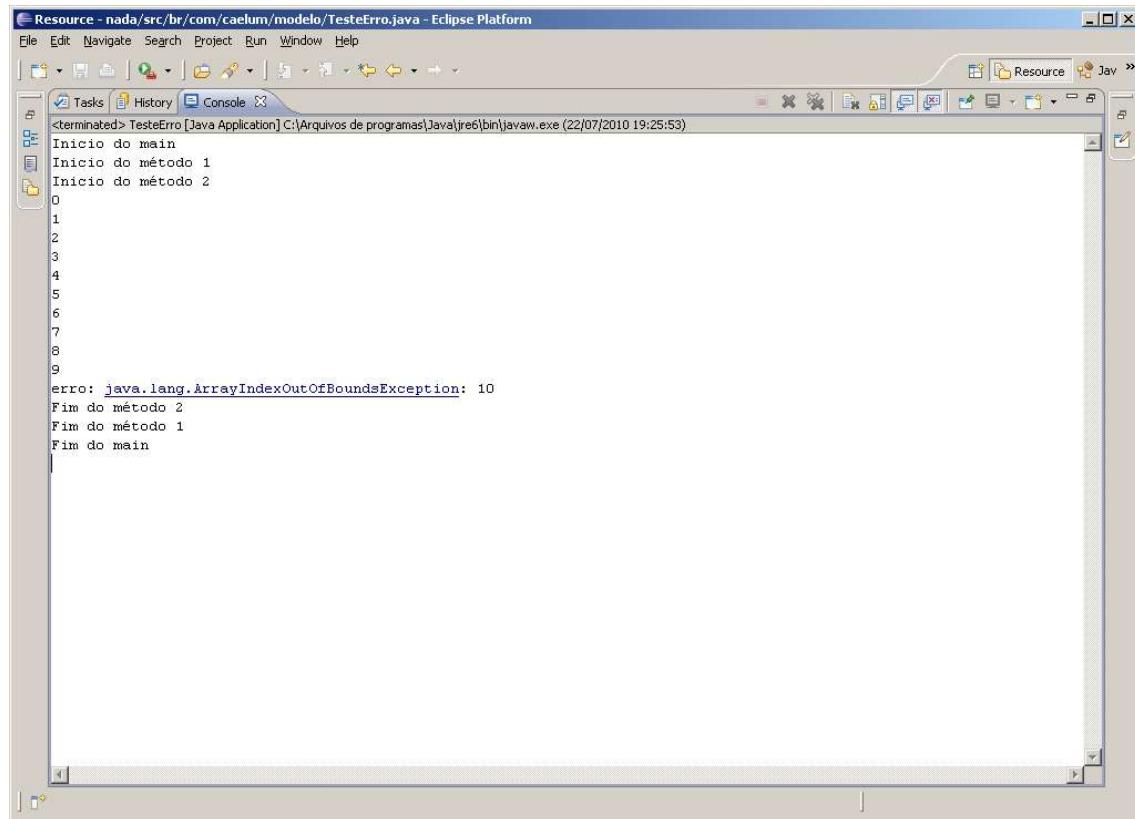
Para entender o controle de fluxo da exceção, vamos colocar o código que vai **tentar** (try) executar o bloco perigoso e, caso o problema seja do tipo *ArrayIndexOutOfBoundsException*, ele será **pego** (caught). Perceba que é interessante que cada exceção tenha um tipo... ela pode ter atributos e métodos.

- 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);
    }
```

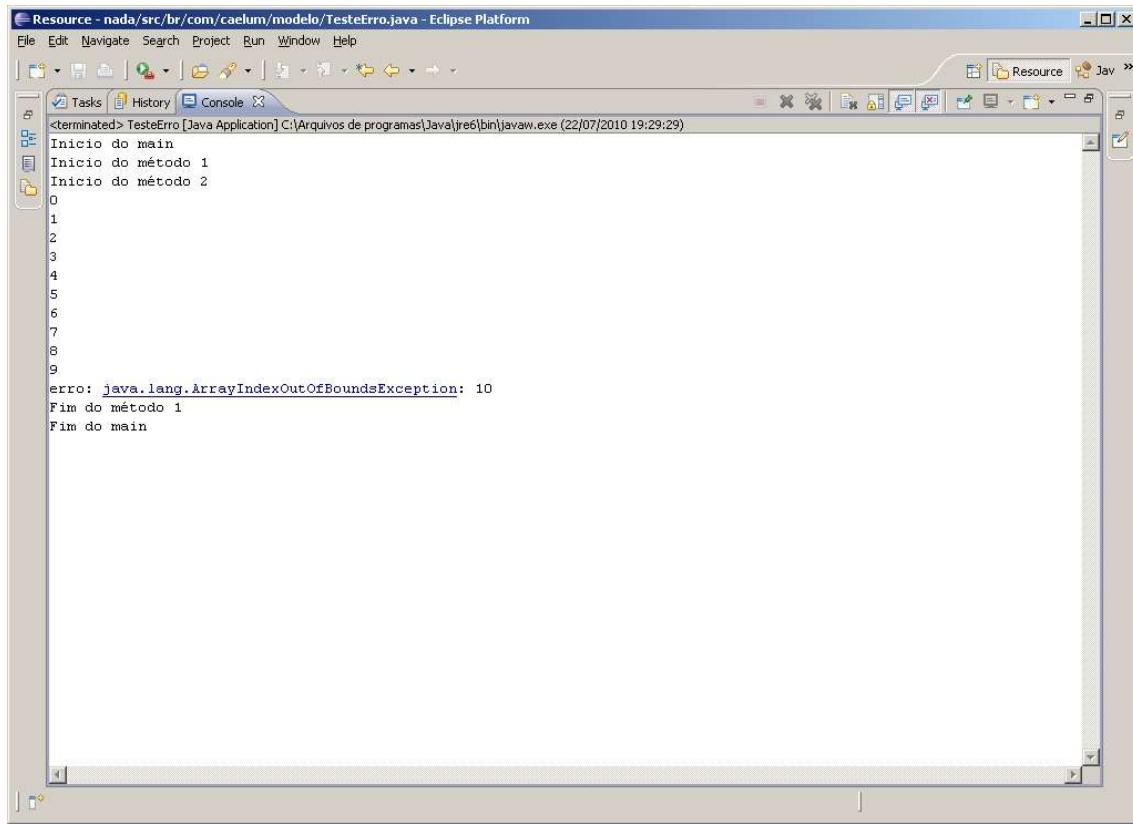
Qual é a diferença?



```
<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
```

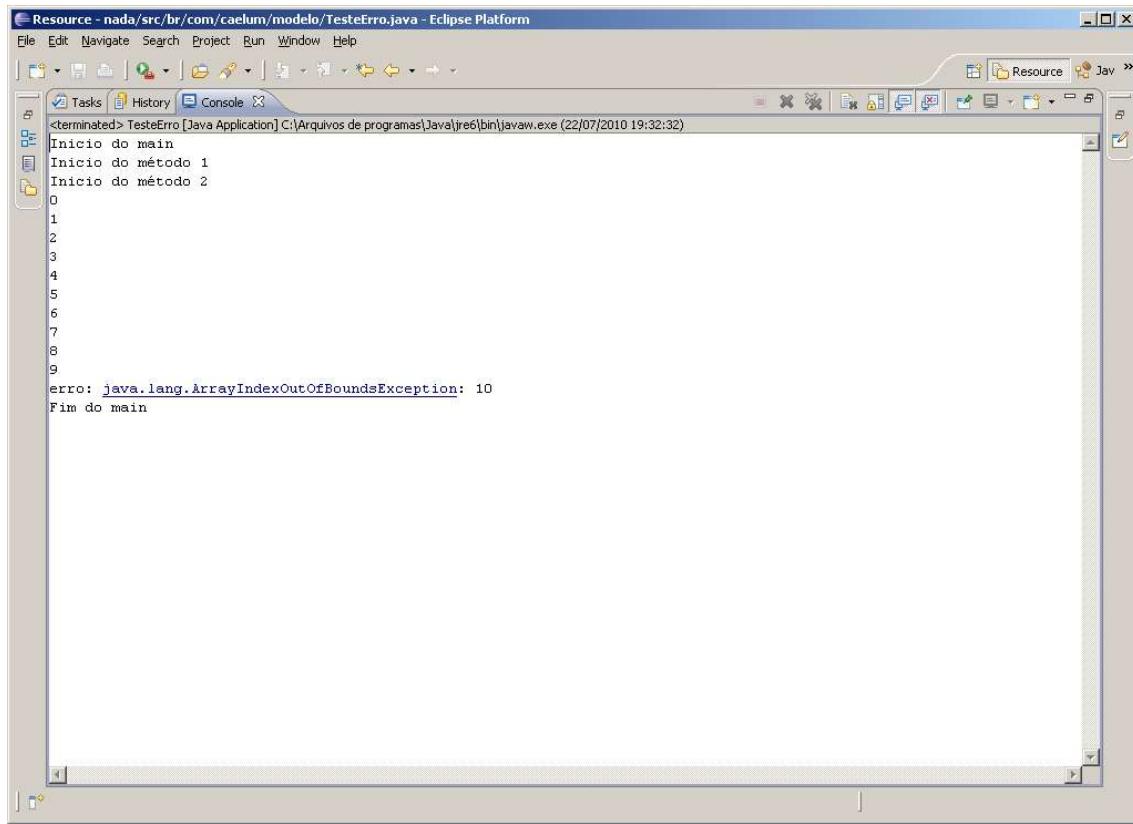
Agora retire o try/catch e coloque em volta da chamada do método2.

```
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");
```



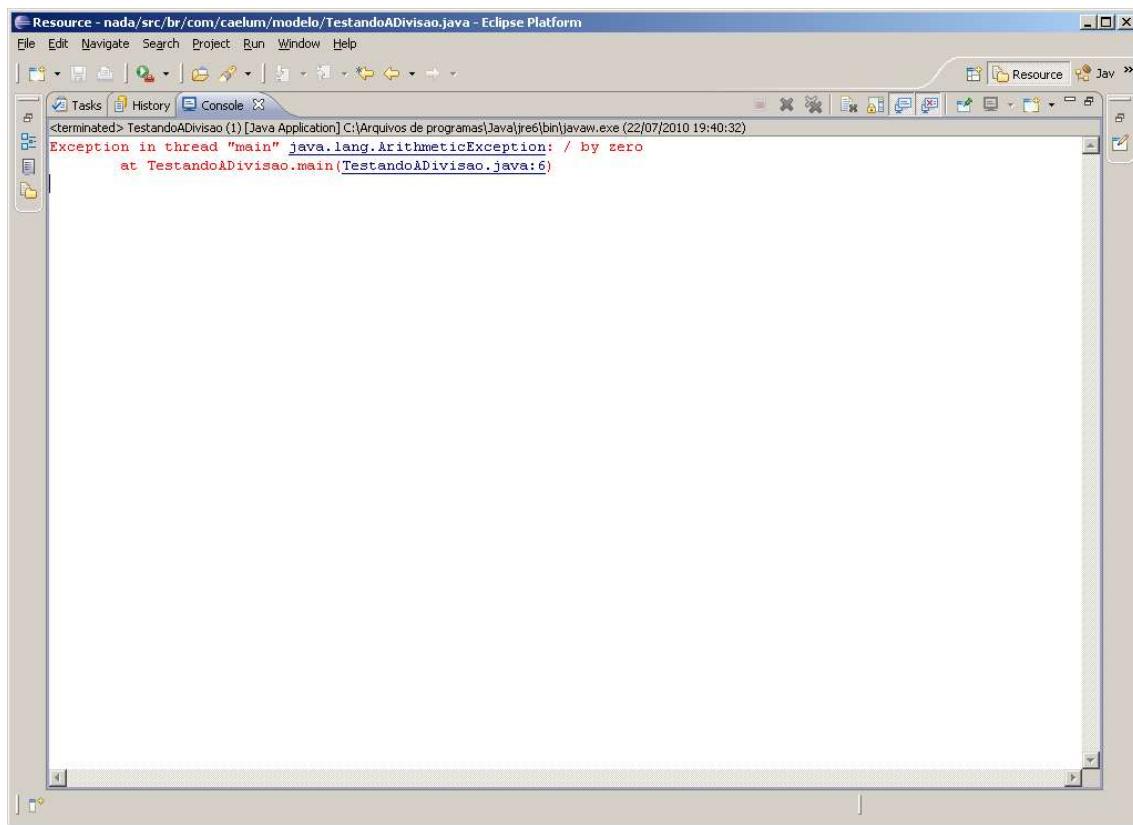
Repare que, a partir do momento que uma exception foi catched (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

Exceções de Runtime mais comuns

Tente dividir um número por zero. Será que o computador consegue fazer algo que nós definimos que não existe?

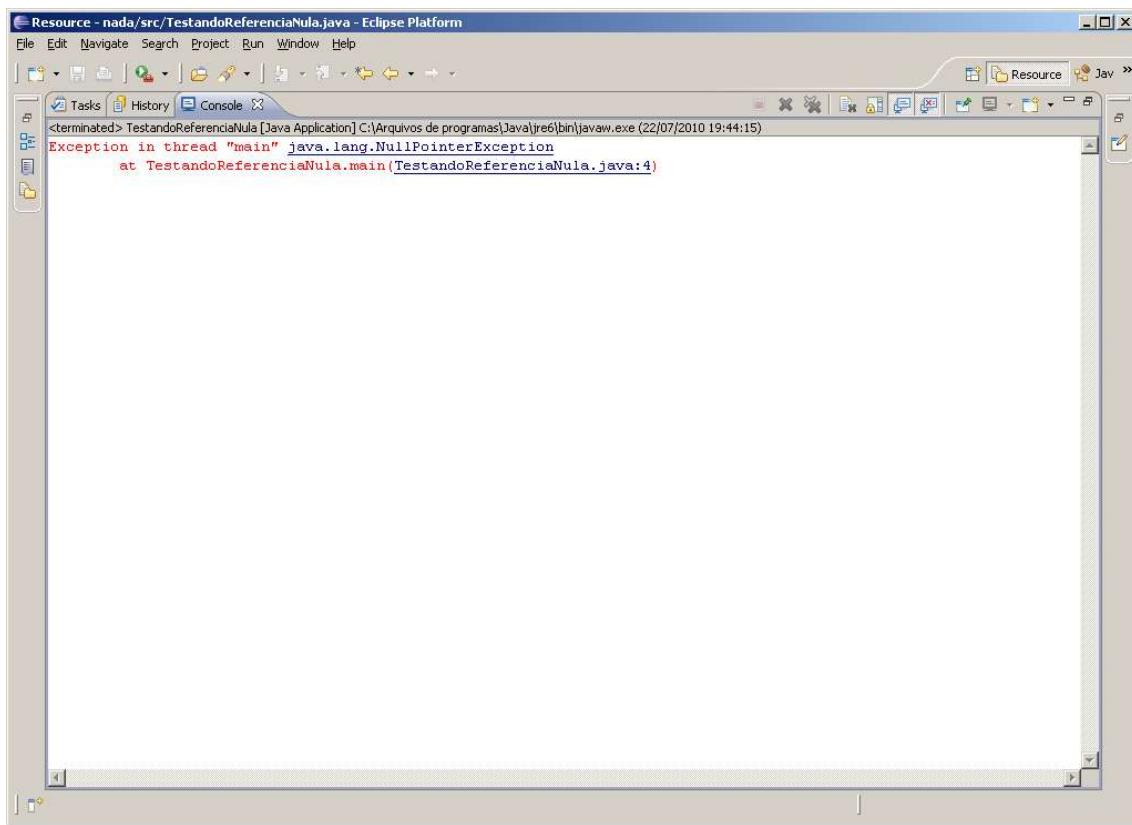
```
public class TestandoADivisao {  
    public static void main(String[] args) {  
        int i = 5000;  
        i = i / 0;  
        System.out.println("O resultado é " + i);  
    }  
}
```

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 uma 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.");
    }
}
```

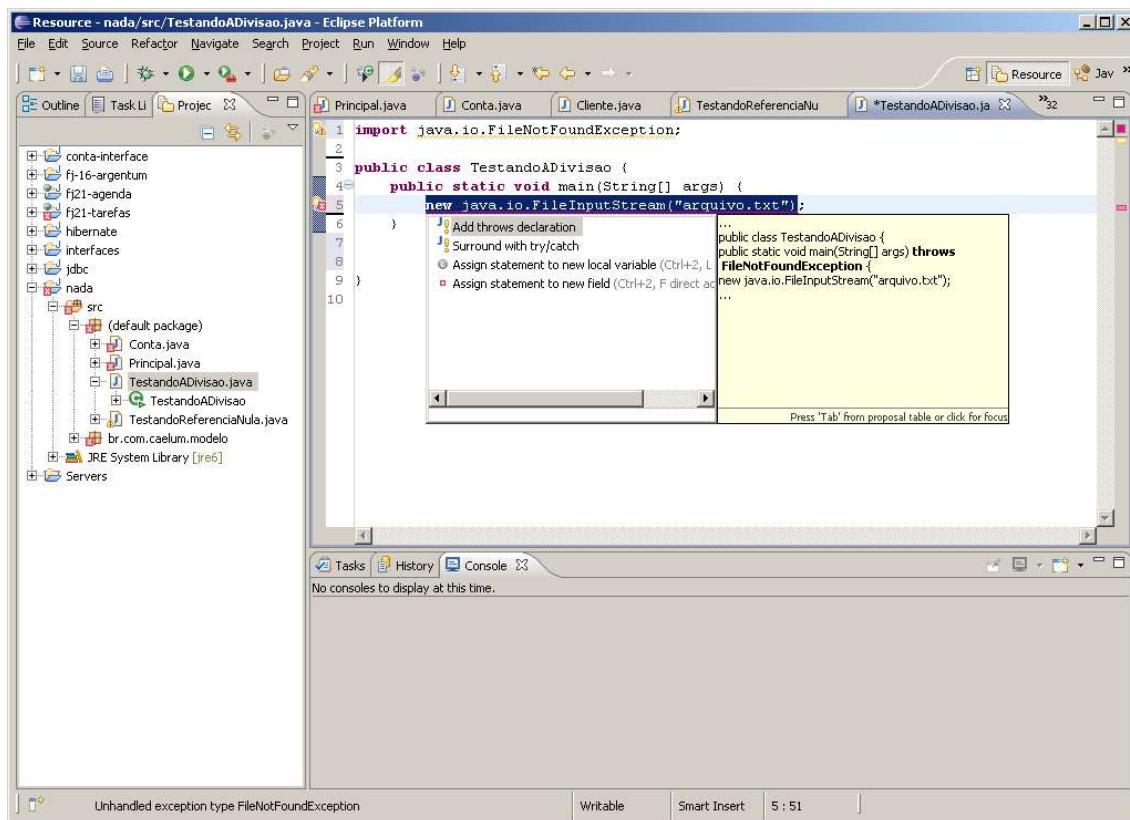
A outra forma de tratar esse erro é delegar ele para quem chamou nosso método, ou seja, passar adiante.

```
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.

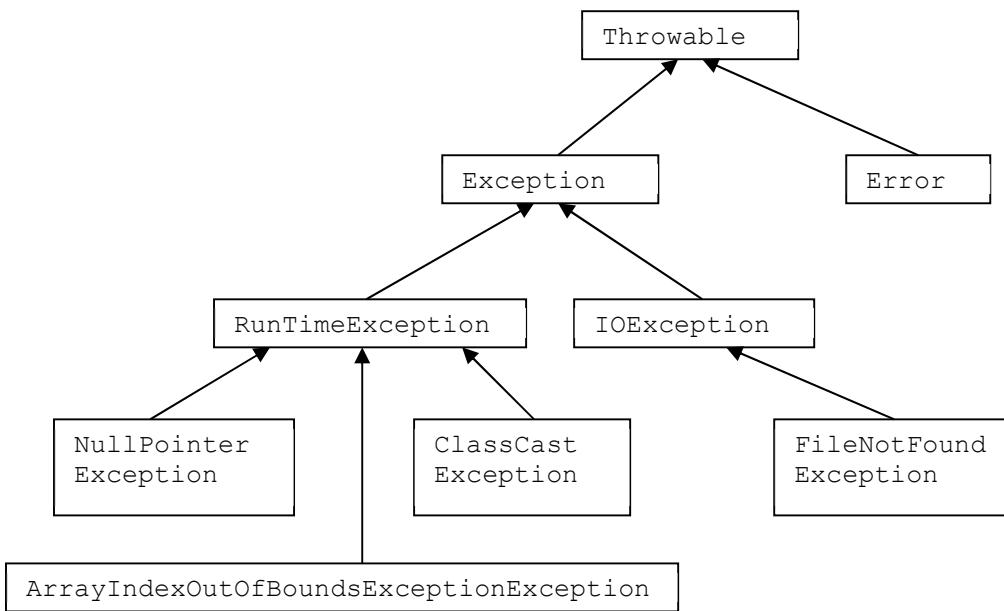
Não existe uma regra para decidir em que momento do programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, provavelmente você vai preferir passar para frente, delegando a responsabilidade 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 + this.limite) >= 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 + this.limite) >= 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 + this.limite) >= 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 + this.limite) >= 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 ao invés de lançar uma IllegalArgumentException, vamos lançar nossa própria exceção, com a seguinte mensagem: “Saldo Insuficiente.”

```
public void saca(double quantidade) {
    if ((this.saldo + this.limite) >= 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

- 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 ContaPoupança 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` ao invés 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.

47.Organizando suas Classes e Bibliotecas com Pacotes

Quando um programador utiliza classes feitas por outro programador, surge um problema clássico: Como escrever duas classes com o mesmo nome?

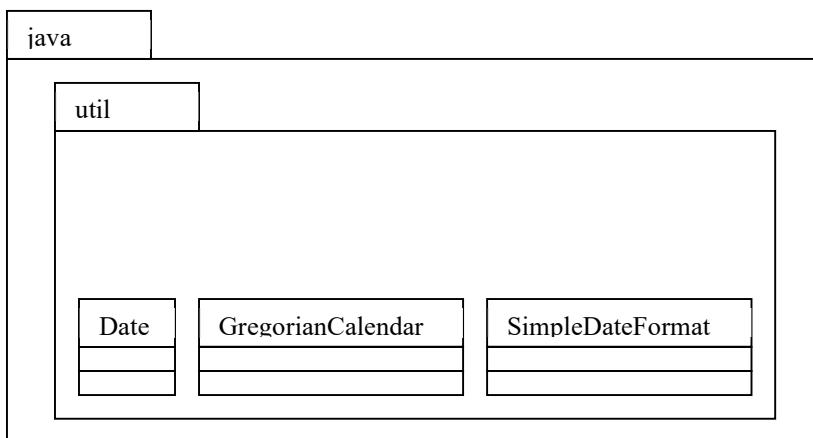
Minha classe `Data` pode estar escrita de um jeito, a classe `Data` de um segundo programador pode estar escrita de outro jeito e a classe `Data` de uma outra biblioteca funcione de uma maneira diferente das duas primeiras. Como controlar quem quer usar qual classe `Data`?

A solução está nos diretórios: O sistema operacional não permite a existência de dois arquivos com o mesmo nome sob o mesmo diretório. Portanto, precisamos organizar nossas classes em diretórios diferentes.

Os diretórios estão diretamente relacionados aos chamados pacotes e costumam agrupar classes de funcionalidades similares ou relacionadas.

No pacote `java.util` por exemplo, temos as classes `Date`, `SimpleDateFormat` e `GregorianCalendar`, todas trabalham com datas de formas diferentes.

Em um outro pacote chamado `java.sql` também existe uma classe `Date`. Os pacotes é que nos permitem diferenciar uma da outra.



Diretórios:

Se a classe `Cliente` está no pacote `banco`, ela deverá estar no diretório com o mesmo nome: `banco`. Se ela se localiza no diretório `br.com.cefet`, significa que ela está no diretório `br/com/cefet/banco`.

A classe `Cliente` deve ser escrita da seguinte forma:

```
package br.com.cefet.banco;  
  
public class Cliente {  
    //...  
}
```

A palavra chave `package` indica qual pacote/diretório contém esta classe. Um pacote pode conter nenhum ou mais subpacotes e/ou classes dentro dele.

Padrão de nomenclatura de pacotes:

O padrão da Sun para nomes de pacotes é relativo ao nome da empresa que desenvolveu a classe:

`br.com.nomedaempresa.nomedoprojeto.subpacote`
`br.com.nomedaempresa.nomedoprojeto.subpacote2`
`br.com.nomedaempresa.nomedoprojeto.subpacote2.subpacote3`

Não importa quantas palavras estejam contidas nele, um pacote só aceita letras minúsculas.

Import:

Para usar uma classe do mesmo pacote basta fazer como vimos fazendo até agora, referenciando-a escrevendo simplesmente o seu nome. Se existe uma classe banco dentro do pacote br.com.cefet.banco, ela deve ser escrita assim:

```
package br.com.cefet.banco;

public class Banco {
    String nome;
    Cliente clientes[];
    //...
}
```

A classe Cliente deve ficar no mesmo pacote da seguinte maneira:

```
package br.com.cefet.banco;

class Cliente {
    String nome, cpf, endereco;
    //...
}
```

A novidade chega ao tentar utilizar a classe Banco (ou Cliente) em uma outra classe que esteja em outro pacote, por exemplo, no pacote br.com.ffsd.banco.util:

```
package br.com.cefet.banco.util;

public class TesteDoBanco {
    public static void main(String[] args) {
        br.com.ffsd.banco.Cliente cliente = new
br.com.ffsd.banco.Cliente();
        cliente.setNome("Maria");
        System.out.println(cliente.getNome());
    }
}
```

Perceba que precisamos referenciar a classe Cliente com todo o nome do pacote na frente. Esse é o conhecido Full Qualified Name de uma classe. Em outras palavras, é o verdadeiro nome de uma classe, por isso duas classes com mesmo nome em pacotes diferentes não conflitam.

Mesmo assim, ao tentar compilar a classe acima, surge um erro reclamando que a classe Cliente não está visível.

Isso acontece por que as classes só são visíveis para outras do mesmo pacote e, para permitir que a classe TesteDoBanco veja a classe Cliente em outro pacote, precisamos alterar essa última e transformá-la em pública:

```
package br.com.cefet.banco;

public class Cliente {
    String nome, cpf, endereco;
    //...
}
```

A palavra chave public libera o acesso para classes de outros pacotes.

Voltando ao código de TesteDoBanco, preciso mesmo escrever todo o nome do pacote para identificar qual classe quero usar? Existe uma maneira mais simples de se referenciar a classe Cliente. Basta importá-la do pacote br.com.ffsd.banco:

```
package br.com.cefet.banco.util;

import br.com.cefet.banco.Cliente; //agora podemos referenciar a classe diretamente

public class TesteDoBanco {
    public static void main(String[] args) {
        Cliente cliente = new Cliente();
        cliente.setNome("Maria");
        System.out.println(cliente.getNome());
    }
}
```

É muito importante manter a ordem! Primeiro aparece uma (ou nenhuma) vez o package; depois pode aparecer um ou mais importS; e, por último, as declarações de classes.

É possível importar um pacote inteiro (todas as classes do pacote, exceto os subpacotes) através do *:

```
import java.util.*;
```

Importar todas as classes de um pacote não implica em perda de performance, mas pode trazer problemas com classes de mesmo nome! Além disso, importar de um em um é considerado boa prática, pois facilita a leitura para outros programadores.

Acesso aos atributos, construtores e métodos:

No Java existem 4 modificadores de acesso. Até o momento vimos 3, mas só explicamos 2.

- **public** – Todos podem acessar aquilo que foi definido como public. Classes, atributos, Construtores e Métodos podem ser public.
- **protected** – Aquilo que é protected pode ser acessado por todas as classes do mesmo pacote e por todas as classes que o estendam. Somente Atributos, Construtores e Métodos podem ser protected.
- **padrão** (sem nenhum modificador) – Se nenhum modificador for utilizado, todas as classes do mesmo pacote têm acesso aos Atributos, Construtores, Métodos ou Classe.
- **private** – A única classe capaz de acessar os Atributos, Construtores e Métodos privados é a própria classe. Classes, como conhecemos, não podem ser private, mas atributos, construtores e métodos sim.

Classes Públicas:

O Java não permite mais de uma classe pública por arquivo. De qualquer forma, mais de uma classe por arquivo é uma má prática de programação.

Classes podem ser protected ou private, mas esse é um tópico avançado que só veremos mais adiante chamado **classes internas**.

Usando pacotes no Eclipse:

A partir de agora já podemos mudar a perspectiva para Java, no Eclipse. A view principal de navegação é o package Explorer, que agrupa as classes pelos pacotes ao invés de diretórios (Você pode usá-la em conjunto com a Navigator, basta também abri-la pelo Windows>Show View/Package Explorer).

Você pode mover uma classe de um pacote a outro arrastando-a. O Eclipse já declara packages e imports necessários.

No Eclipse nunca precisamos declarar um import. Ele sempre vai sugerir isso quando usarmos o Ctrl+Espaço no nome da classe, ou vai importá-las direto quando apertarmos Ctrl+Shift+O.

Você também pode usar Ctrl+1 no caso de a classe possuir erros.

Exercícios com Pacotes

Atenção: utilize os quick fixes quando o Eclipse reclamar dos diversos problemas. Da pra fazer esse exercício inteiro sem modificar uma linha de código manualmente. Aproveite para praticar e descobrir o Eclipse, evitando usá-lo apenas como editor de text.

- 1) Altere seu projeto-banco para que ele utilize pacotes. Clique com o botão direito do mouse sobre src e escolha File/New/Package, ponha seu sistema de Contas para utilizar pacotes. Respeite a convenção de código da Sun:

br.com.cefet.banco.testes → colocar classes com o método main aqui (os Testes).
br.com.cefet.banco.modelo → Colocar a classe Conta, suas filhas aqui.
br.com.cefet.banco.modelo.exceptions → Colocar suas Exceptions aqui.
br.com.cefet.banco.modelo.sistema → Colocar AtualizadorDeContas e GerenciadorDeImpostoDeRenda aqui.

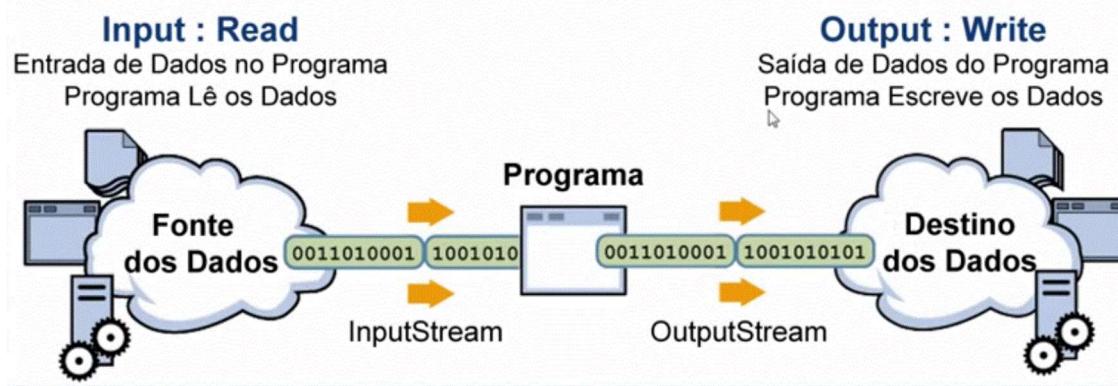
Antes de corrigir qualquer erro de compilação, primeiro move todas as suas classes, sem deixar nenhuma no pacote default.

- 2) O código não compilar prontamente, pois muitos métodos que declaramos são package-friendly quando, na verdade, precisaríamos que eles fossem public. O mesmo vale para as classes: algumas delas precisarão ser públicas.
Use o quick fix do Eclipse aqui: ele mesmo vai sugerir que o modificador de acesso deve ser public. Para isso, clique no quick fix que aparecerá no erro do import e das invocações de métodos.
- 3) Abra a view navigator para ver como ficaram os arquivos no sistema de arquivos do Windows. Para isso, use o menu Window, Show View.

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada e saída de dados (conhecida como io) é orientada a objetos e usa os principais conceitos mostrados até agora: interfaces, classes abstratas e polimorfismo.

A idéia por trás do polimorfismo é utilizar fluxos de entrada (InputStream) e de saída (OutputStream) bem genéricos para que posam servir para qualquer operação, seja ela relativa a um arquivo, a um campo blob do banco de dados, a uma conexão remota via sockets, ou até mesmo às entrada e saída padrão de um programa (normalmente o teclado e o console).

I/O : Input / Output : Read / Write



As classes abstratas InputStream e OutputStream definem, respectivamente, o comportamento padrão dos fluxos em Java (ler e escrever bytes).

InputStream / OutputStream --> Classes abstratas que definem o comportamento padrão dos fluxos (entrada/saída) em Java.

Veja o código para ler um byte de um arquivo:

```
public class TestaEntradaByte {  
    public static void main(String[] args) throws IOException {  
        InputStream is = new FileInputStream("arquivo.txt");  
        int b= is.read(); //read() retorna um int  
    }  
}
```

FileInputStream --> Classe concreta, filha de InputStream, que recebe, como argumento, o nome do arquivo a ser lido. Ela vai procurar o arquivo dentro do diretório raiz do projeto, a menos que se informe o caminho completo do arquivo.

Algumas classes filhas de InputStream:

FileInputStream, ObjectInputStream, AudioInputStream, ByteArrayInputStream, entre outras.

InputStreamReader--> Filha da classe abstrata Reader, que possui outras filhas que manipulam chars. InputStreamReader traduz bytes em caracteres para que possamos trabalhar. Esta classe pode receber como argumento, o encoding a ser utilizado. Ex: UTF-8, ISO-8859-1 etc.

```
public class TestaEntradaByte {  
    public static void main(String[] args) throws IOException {  
        InputStream is = new FileInputStream("arquivo.txt");  
        InputStreamReader isr = new InputStreamReader(is, "UTF-8" );  
        int c= isr.read(); //read() retorna um int  
    }  
}
```

O código acima pode lançar FileNotFoundException ou, de forma mais genérica, IOException.

BufferedReader--> É um Reader que recebe outro Reader pelo construtor e concatena os diversos chars para formar uma String através do método readLine().
Esta classe lê o Reader por pedaços (usando o buffer) para evitar muitas chamadas ao Sistema operacional.

Veja um exemplo de uso leitura de arquivo texto:

```
public class TestaLeituraArquivo {  
    public static void main(String[] args) throws IOException {  
        //Trabalha com bytes  
        InputStream is = new FileInputStream("arquivo.txt");  
        //Obtem um InputStreamReader a partir de um InputStream  
        //Trabalha com chars  
        InputStreamReader isr = new InputStreamReader(is);  
        //Obtem um BufferedReader a partir de um Reader  
        //Trabalha com Strings  
        BufferedReader br= new BufferedReader(isr);  
  
        //readLine() converte os chars em String para nós  
        String s=br.readLine();  
  
        while(s!=null){  
            //Imprime a String lida  
            System.out.println(s);  
            //Recebe a próxima String  
            s=br.readLine();  
        }  
        //Fecha o BufferedReader  
        br.close();  
    }  
}
```

Veja um exemplo de leitura a partir do teclado. Perceba que só muda a linha do InputStream:

```
public class TestaEntradaTeclado {  
    public static void main(String[] args) throws IOException {  
        //Trabalha com bytes  
        InputStream is = System.in;  
        //Obtem um InputStreamReader a partir de um InputStream  
        //Trabalha com chars  
        InputStreamReader isr = new InputStreamReader(is);  
        //Obtem um BufferedReader a partir de um Reader  
        //Trabalha com Strings  
        BufferedReader br= new BufferedReader(isr);  
  
        //readLine() converte os chars em String para nós  
        String s=br.readLine();
```

```
    while(s!=null) {
        //Imprime a String lida
        System.out.println(s);
        //Recebe a próxima String
        s=br.readLine();
    }
    //Fecha o BufferedReader
    br.close();
}
}
```

Veja a composição, também conhecida como **Decorator Pattern**:

InputStream ("arquivo.txt" ou System.in) --> **InputStreamReader** (converte os bytes em char) --> **BufferedReader** (converte os chars em String).

Percebam a vantagem de usar o polimorfismo!!!

Repare no código que poderíamos ter qualquer InputStream, seja ObjectInputStream, AudioInputStream, ByteArrayInputStream, FileInputStream (1º exemplo) ou o System.in (conforme exemplo acima).

O nosso InputStream (a forma mais genérica) serve para representar qualquer uma das entradas acima (filhas de InputStream). É o **polimorfismo facilitando nossa vida!!**

Independentemente do que estamos lendo em bytes no InputStream, usamos InputStreamReader para converter em chars e BufferedReader para converter em Strings.

Analogia para escrita: OutputStream

Como você já deve imaginar, escrever em um arquivo é o mesmo processo:

```
public class TestaSaida {
    public static void main(String[] args) throws IOException
    //Sobrescreve
    OutputStream os = new FileOutputStream("arquivo.txt");
    //O true abaixo indica que vc quer manter o que já estiver escrito
    //OutputStream os = new FileOutputStream("arquivo.txt",true);
    OutputStreamWriter osw = new OutputStreamWriter(os);
    BufferedWriter bw = new BufferedWriter(osw);
    //O write não efetua a quebra de linha por isso chamamos newLine()
    bw.newLine();
    bw.write("CEFET");
    bw.close();
}
}
```

Veja a composição, também conhecida como **Decorator Pattern**:

OutputStream ("arquivo.txt") --> **OutputStreamWriter** (converte os bytes em char) --> **BufferedWriter** (converte os chars em String).

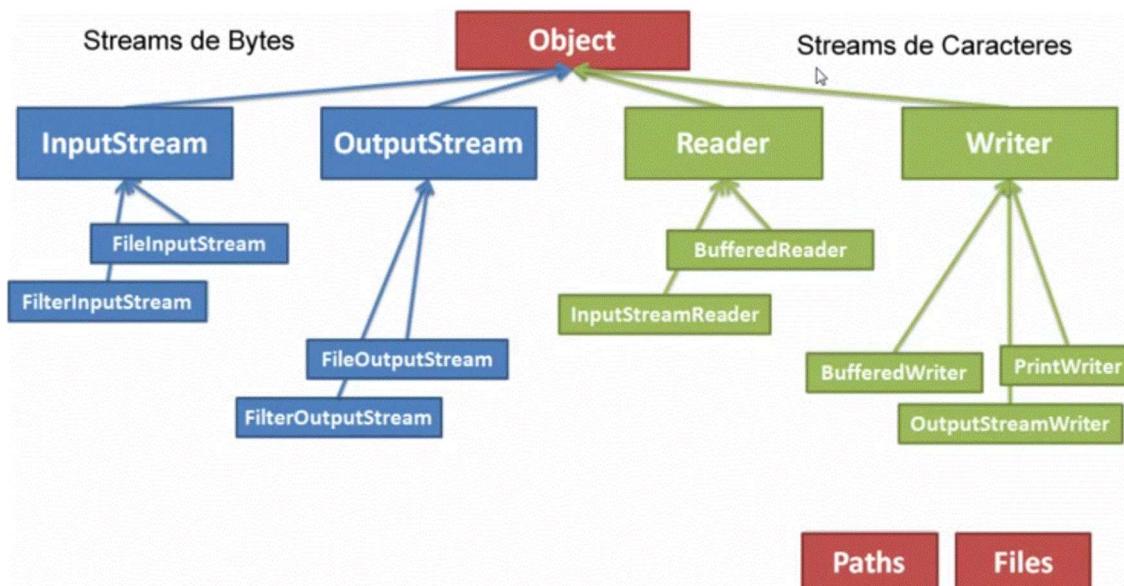
Fechando o arquivo

É importante você fechar o arquivo. Você pode fazer isso chamando diretamente o método close de FileInputStream/OutputStream, ou ainda chamando o close de

BufferedReader/Writer. Nesse caso o close será cascadeado para os outros objetos, além de fazer o flush dos buffers no caso da escrita. É comum que o close esteja dentro de um bloco Finally.

As classes InputStream, OutputStream, Reader e Writer implementam a interface Closeable!

Veja a hierarquia das classes:



Uma maneira mais fácil: Scanner e PrintStream

A classe `java.util.Scanner` facilita bastante o trabalho de ler de um `InputStream`. Além disso, a classe `PrintStream` possui um construtor que já recebe o nome de um arquivo como argumento. Desta forma, a leitura do teclado com saída para um arquivo ficou muito simples:

```
public class EscreveEmArquivoAPartirDoTeclado {
    public static void main(String[] args) throws
FileNotFoundException {
        PrintStream ps = new PrintStream("arquivo.txt");
        // System.in é um InputStream
        Scanner s = new Scanner(System.in);
        while(s.hasNextLine()) {
            //Escrevendo no arquivo a partir do teclado
            ps.println(s.nextLine());
        }
        ps.close();
    }
}
```

Estas duas classes também implementam Closeable.
Scanner é uma public final class. Não pode ser estendida.
Scanner também implementa Iterator. Veremos mais adiante.

Nenhum dos métodos acima lançam `IOException`: `PrintStream` liga `FileNotFoundException` se você o construir passando uma `String`. `Scanner` considerará que chegou ao fim se uma `IOException` for lançada, mas o `PrintStream` simplesmente

engole exceptions desse tipo. Ambos possuem métodos para você verificar se algum problema ocorreu.

Análogamente temos o mesmo processo através do Decorator Pattern:

```
public class EscreveEmArquivoAPartirDoTeclado {
    public static void main(String[] args) throws IOException {
        PrintStream ps = new PrintStream("arquivo.txt");
        //Todo o código de Scanner pode ser trocado por:
        InputStream is=System.in;
        InputStreamReader isr=new InputStreamReader(is);
        BufferedReader br=new BufferedReader(isr);
        String texto=br.readLine();
        while(texto!=null){
            //Escrevendo no arquivo a partir do teclado
            ps.println(texto);
            texto=br.readLine();
        }
        ps.close();
    }
}
```

A classe Scanner possui métodos para trabalhar com Strings, em especial, diversos métodos já preparados para pegar números e palavras já formatadas através de expressões regulares. Fica fácil parsear um arquivo com qualquer formato dado.

System.out

O atributo out da classe System é do tipo PrintStream e, portanto, é um OutputStream.

Um pouco mais...

Existem duas classes chamadas java.io.FileReader e java.io.FileWriter. Elas são atalhos para leitura e escrita de arquivos.

Experimente usar o do{ .. }while(condicao); para ler um arquivo. O código vai ficar mais suscinto já que você não precisará ler a primeira linha fora do laço.

Exercícios: Java I/O

1) Crie um projeto chamado teste-io

Crie um programa que leia da entrada padrão (teclado). Para tal você vai precisar de um BufferedReader, que leia do System.in da mesma forma como fizemos.

Cuidado! Queremos a classe InputStream do pacote java.io. Existem outros..

```
public class TestaEntrada {
    public static void main(String[] args) throws IOException {
        InputStream is = System.in;
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String linha;
        do {
            linha = br.readLine();
            if (linha != null)
                System.out.println(linha);
        } while (linha != null);
    }
}
```

```
    }  
}  
EOF
```

Quando rodar a aplicação, para encerrar a entrada de dados do teclado, é necessário enviarmos um sinal de fim de stream. É o famoso EOF, End Of File.

No Linux/Mac/Solaris/Unix você faz isso com o CONTROL+D. No Windows, use o CONTROL + Z.

2)Vamos ler de um arquivo, ao invés do teclado. Antes, vamos criar o arquivo que será lido pelo programa, obviamente, acrescentando algumas linhas de texto a ele:
new --> file --> arquivo.txt.

Troque, na classe anterior, o **System.in** por **new FileInputStream("arquivo.txt");**
Graças ao polimorfismo, essa é a única alteração que você vai precisar fazer!!

3) Repare que, no final, só usamos mesmo o BufferedReader. As referências para InputStream e para InputStreamReader são utilizadas apenas temporariamente. Portanto, é comum encontrarmos o seguinte código nesses casos:

```
BufferedReader br = new BufferedReader(new InputStreamReader(new  
FileInputStream("arquivo.txt")));
```

É claro que devemos evitar este tipo de código que compromete a legibilidade do programa.

O resultado final deve ser assim:

```
public class TestaEntrada {  
    public static void main(String[] args) throws IOException{  
        InputStream is = new FileInputStream("arquivo.txt");  
        InputStreamReader isr = new InputStreamReader(is);  
        BufferedReader br = new BufferedReader(isr);  
        //BufferedReader br = new BufferedReader(new  
InputStreamReader(new FileInputStream("arquivo.txt")));  
        String linha;  
        do {  
            linha = br.readLine();  
            if (linha != null)  
                System.out.println(linha);  
        } while (linha != null);  
    }  
}
```

4) Altere o programa anterior para que ele leia do arquivo e, ao invés de jogar na tela, jogue em outro arquivo. Você vai precisar do código para escrever em um arquivo. Para isso, você pode usar o BufferedWriter ou o PrintStream, sendo este último de mais fácil manipulação.

Se for usar o BufferedWriter, fazemos assim para abri-lo:

```
OutputStream os= new FileOutputStream("saída.txt");  
OutputStreamWriter osw=new OutputStreamWriter(os);  
BufferedWriter bw = new BufferedWriter(osw);
```

Agora, dentro do loop de leitura do teclado, você deve usar bw.write(x); onde x é a linha que você leu. Use bw.newLine() para pular de linha. Não se esqueça de, ao término do loop, dar um bw.close(). Você pode seguir o modelo abaixo:

```
String linha;  
do {  
    linha = br.readLine();
```

```
        if (linha != null) {
            System.out.println(linha);
            bw.write(linha);
            bw.newLine();
        }
    } while (linha != null);
br.close();
bw.close();
```

Após rodar o programa, dê um refresh no seu projeto e veja que ele criou um arquivo saída.txt no diretório.

No final deve ficar mais ou menos assim:

```
public class TestaEntrada {
    public static void main(String[] args) throws IOException{
        InputStream is = new FileInputStream("arquivo.txt");
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        //BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream("arquivo.txt")));
        OutputStream os= new FileOutputStream("saída.txt");
        OutputStreamWriter osw=new OutputStreamWriter(os);
        BufferedWriter bw = new BufferedWriter(osw);
        String linha;
        do {
            linha = br.readLine();
            if (linha != null){
                System.out.println(linha);
                bw.write(linha);
                bw.newLine();
            }
        } while (linha != null);
        br.close();
        bw.close();
    }
}
```

5) Altere novamente seu programa para ele virar um pequeno editor que lê do teclado e escreve no arquivo. Repare que a mudança é mínima!

```
public class TestaEntrada {
    public static void main(String[] args) throws IOException{
        //InputStream is = new FileInputStream("arquivo.txt");
        InputStream is = System.in;
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        //BufferedReader br = new BufferedReader(new
InputStreamReader(new FileInputStream("arquivo.txt")));
        OutputStream os= new FileOutputStream("saída.txt");
        OutputStreamWriter osw=new OutputStreamWriter(os);
        BufferedWriter bw = new BufferedWriter(osw);
        String linha;
        do {
            linha = br.readLine();
            if (linha != null){
                System.out.println(linha);
                bw.write(linha);
                bw.newLine();
            }
        } while (linha != null);
        br.close();
        bw.close();
    }
}
```

```
    }  
}
```

Percebam que só precisamos mudar o nosso InputStream.

6) A classe Scanner é muito poderosa. Procure se informar mais a respeito dela no javadoc. Leia sobre o delimiter, o método next, entre outros.

Manipulando arquivos com as classes java.io.File, FileReader e FileWriter

A classe java.io.File representa um arquivo, porém não representa o conteúdo do arquivo. Trata-se da representação de um arquivo ou diretório, não dos dados contidos nele. A linha abaixo não cria o arquivo propriamente dito, simplesmente referencia um arquivo com tal nome.

```
File f=new File("C:\\java\\meuArquivo.txt");
```

Para criar o arquivo acima podemos chamar:

```
f.createNewFile(); //Cria se ele não existir
```

Esta função retorna true, caso ele tenha criado o arquivo, e false, caso ele não tenha criado. Se o arquivo já existir na chamada do método por exemplo.

Para verificar se um arquivo existe no HD use:

```
f.exists();
```

Para lermos um arquivo utilizamos a classe FileReader e passamos a representação do arquivo(File) como argumento. Também podemos passar o caminho do arquivo diretamente para a classe FileReader.

Como já vimos, a classe BufferedReader melhora a performance da leitura do arquivo. Ela recebe como argumento um Reader (classe abstrata) ou uma de suas filhas, no nosso caso, FileReader.

BufferedReader também fornece o método readLine() que facilita a leitura.

Exemplo de leitura de arquivo com as classes e métodos acima, além de outros:

```
public class TestaFileReader {  
    public static void main(String[] args) throws IOException {  
        File diretorio = new File("C:\\\\java\\\\");  
        if (!diretorio.isDirectory()) {  
            System.out.println("Diretório não existe.");  
            if (diretorio.mkdir())  
                System.out.println("Diretório " + diretorio + "  
criado.");  
        }  
  
        File arquivo = new File(diretorio, "meuArquivo.txt");  
        if (diretorio.exists()) {  
            System.out.println("Diretório " + diretorio + "  
existe");  
            if (arquivo.exists())
```

```
        System.out.println("Arquivo " + arquivo + " existe");
    else if (arquivo.createNewFile())
        System.out.println("O arquivo " + arquivo
                           + " acaba de ser criado.");
}

FileReader fr = new FileReader(arquivo);
BufferedReader br = new BufferedReader(fr);

String linha = br.readLine();
System.out.println(linha);
while ((linha = br.readLine()) != null)
    System.out.println(linha);
br.close();
}
}
```

Gravação de arquivos

Para gravarmos em arquivo utilizamos a classe `FileWriter` e passamos como argumento o nome do arquivo para o qual vamos escrever ou um objeto do tipo `File`.

Na classe `FileWriter` temos o método `write("String")` para escrevermos no arquivo. No entanto, para facilitar, usaremos a classe `BufferedWriter` que recebe como argumento um objeto do tipo `Writer` (classe abstrata), como por exemplo sua filha `FileWriter`. No caso, ao trabalhar com `BufferedWriter`, usaremos o método `newLine()` ao invés de `write()`.

Para adicionar conteúdo a um arquivo existente, sem apagar as informações contidas nele, usamos:

`new FileWriter(File, true);`

Se quisermos escrever por cima do que já existe:

`new FileWriter(File, false);`

Veja um exemplo de gravação de arquivo com as classes e métodos citadas acima, além de outros:

```
public class TestaFile {
    public static void main(String[] args) throws IOException {
        File diretorio=new File("C:\\\\java\\\\");
        if(!diretorio.isDirectory()){
            System.out.println("Diretório não existe.");
            if(diretorio.mkdir())
                System.out.println("Diretório "+diretorio+
criado.");
        }

        File arquivo=new File(diretorio, "meuArquivo.txt");
        if(diretorio.exists()){
            System.out.println("Diretório "+diretorio+" existe");
            if(arquivo.exists())
                System.out.println("Arquivo "+arquivo+
existe");
        }
        else
            if(arquivo.createNewFile())
                System.out.println("O arquivo "+arquivo+
acaba de ser criado.");
    }
}
```

```
}

File files=new File("C:\\java\\");
//Verifica arquivos e diretórios em files
for (File file : files.listFiles()) {
    System.out.println(file);
}

FileWriter fw=new FileWriter(arquivo,true);
PrintWriter pw=new PrintWriter(fw);
pw.println("Linha 1");
pw.println("Linha 2");
//Libera a escrita do arquivo
pw.flush();
//fecha o arquivo
pw.close();
}
}
```

Teste usar o BufferedWriter no lugar de PrintWriter.

49. Collections Framework

Trabalhar com arrays é complicado sob vários aspectos:

- * Não podemos redimensionar um array em java;
- * Fica impossível buscar por um determinado elemento sem saber o seu índice;
- * A não ser que criemos métodos auxiliares, não conseguimos saber quantas posições do array foram populadas.

Supondo que os dados armazenados representem contas cabe perguntar:

- * O que acontece quando precisarmos inserir uma nova conta no array?
- * Precisaremos procurar por um espaço vazio?
- * Guardaremos, em alguma estrutura de dados externa, as posições vazias?
- * E se não houver espaço vazio? Teremos que criar um array maior e copiar os dados do antigo para ele?
- * Como posso saber quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array para conseguir esta informação?

Para resolver estes problemas, a Sun criou um conjunto de classes e interfaces conhecido como Collections Framework, que reside no pacote `java.util` desde o Java2 (1.2). A API do Collections é robusta e possui diversas classes que representam estruturas de dados avançadas. Não precisamos, por exemplo, criar uma lista ligada, mas sim utilizar aquela que a Sun disponibilizou.

❖ Listas (`java.util.List`)

Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos. A ordem de inserção é respeitada e é a base para ordenação.

Isso resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho "infinito" ..).

Este código já está pronto!

A API de Collections traz a interface `java.util.List`, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.

A implementação mais utilizada de `List` é a `ArrayList`, que trabalha com um array interno para gerar uma lista. Ela é mais rápida na pesquisa que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.

`ArrayList` não é um array! Internamente ela usa um array como estrutura para armazenar os dados, porém, este atributo está propriamente encapsulado e você não tem como acessá-lo. Você também vai perceber que não pode usar o `[]` com um `ArrayList`, nem acessar o atributo `length`. Não há nenhuma relação!!

Para criar um `ArrayList`, basta chamar o construtor:

```
ArrayList lista = new ArrayList();
```

É boa prática abstrair a lista a partir da interface `List`:

```
List lista = new ArrayList();
```

Para criar uma lista de nomes (`String`) podemos fazer o seguinte:

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("José");
lista.add("Marcia");
```

A interface `List` possui dois métodos `add`. Um que recebe um objeto a ser inserido e o coloca no final da lista e um segundo que permite adicionar o objeto em qualquer posição. Toda lista (na verdade, toda `Collection`) trabalha do modo mais genérico possível. Não há um `ArrayList` específico para `Strings`, outra para números, outra para datas etc. Todos os métodos trabalham com `Object`.

Assim, é possível criar, por exemplo, uma lista de Contas Correntes:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(1000);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(2000);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(3000);

List contas = new ArrayList();
contas.add(c1);
contas.add(c2);
contas.add(c3);
```

```
System.out.println(contas.size());  
  
for (int i = 0; i < contas.size(); i++) {  
    contas.get(i); //Código não muito útil...  
}
```

Para saber quantos elementos existem na lista, utilizamos o método size().

Para recuperar um elemento utilizamos o método get(int) que recebe como argumento o índice do elemento a ser recuperado.

Podemos iterar na lista utilizando um for, como pode se observar acima.

Mas... como fazemos para imprimir o saldo destas contas? Podemos acessar o getSaldo() diretamente após fazer o contas.get(i)??? Não podemos. Lembre-se que toda lista trabalha com Object. Sendo assim, faz-se necessário realizar um cast.

```
for (int i = 0; i < contas.size(); i++) {  
    ContaCorrente cc = (ContaCorrente) contas.get(i);  
    System.out.println(cc.getSaldo());  
}
```

Há ainda outros métodos como o remove(Object) que recebe um objeto que se deseja remover da lista; e contains(Object) que recebe um objeto como argumento e devolve um booleano indicando se o mesmo está ou não na lista.

Existem várias classes concretas que implementam a interface List. São elas: LinkedList, ArrayList e Vector.

Através do método get(int) fizemos um acesso aleatório e obtivemos o elemento de forma imediata, sem que fosse necessário percorrer a lista inteira (que chamamos de acesso sequencial).

Contudo, ao percorrer uma lista, devemos sempre utilizar Iterator ou enhanced for (foreach).

❖ **LinkedList**

A LinkedList (outra implementação bastante utilizada) fornece métodos adicionais para obter e remover o primeiro e último elemento da lista. Ela também tem o funcionamento interno diferente, o que pode impactar performance. Veremos mais adiante.

❖ **Vector**

Vector deve ser tratada com cuidado já que lida de maneira diferente com processos correndo em paralelo. Ela será mais lenta que a ArrayList quando não houver acesso simultâneo a dados.

❖ **Listas com Genéricos**

Como Object é a forma mais genérica, podemos ter listas heterogêneas. Veja:

```
List lista = new ArrayList();  
lista.add(c1);  
lista.add("Uma String..");
```

```
lista.add(new Integer(56));
```

O grande problema é na hora de percorrer e fazer os casts para cada um dos tipos diferentes. A partir do Java5 surgiu o recurso Generics para restringir as listas a um determinado tipo de objeto (e não qualquer Object).

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();  
contas.add(c1);  
contas.add(c2);  
contas.add(c3);
```

O uso de genérics elimina a necessidade de casting, já que, seguramente, todos os elementos inseridos na lista serão do tipo ContaCorrente:

```
for (int i = 0; i < contas.size(); i++) {  
    ContaCorrente cc = contas.get(i); //Sem casting!  
    System.out.println(cc.getSaldo());  
}
```

No código anterior você deve ter reparado que usamos a interface List para especificar nossa lista de ContasCorrente. Declarar nossa lista através de sua forma mais genérica ou até mesmo retornar um ArrayList (por exemplo) na forma de um List em um método, nos da flexibilidade. Afinal, se nosso método devolve ArrayList, somos obrigados a trabalhar com ArrayList. Se nosso método devolve um List, podemos optar por trabalhar com Vector, LinkedList e ArrayList. Muito mais prático! Aliás, sempre que for possível é boa prática trabalhar com a forma mais genérica possível. Isso vale também para declarações e argumentos. Veja:

```
public void atualizaContas(List<ContaCorrente> contas){  
    //....  
}
```

❖ Ordenação (Collections.sort)

Já vimos que as listas são percorridas de maneira pré-determinada, de acordo com a inclusão dos itens. Mas, muitas vezes, queremos percorrer nossa lista de maneira ordenada.

A classe Collections traz um método estático sort que recebe um List como argumento e o ordena por ordem crescente. Veja:

```
List lista = new ArrayList();  
lista.add("Manoel");  
lista.add("José");  
lista.add("Marcia");  
  
System.out.println(lista); //Repare que o toString de  
ArrayList foi sobreescrito  
Collections.sort(lista);  
System.out.println(lista);
```

Funcionou!!

Contudo, toda lista em Java pode ser de qualquer tipo de objeto, por exemplo, ContaCorrente. E se quisermos ordenar uma lista de ContaCorrente??? Em que ordem a classe Collections ordenará? Pelo saldo? Pelo número da conta??

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
```

```
contas.add(c1);
contas.add(c2);
contas.add(c3);

Collections.sort(contas); //qual seria o critério para esta ordenação??
```

Precisamos instruir o método sort sobre como comparar nossas ContaCorrente a fim de determinar uma ordem na lista. O método sort necessita que todos os objetos da lista sejam **comparáveis** e possuam um método que se compara com outra ContaCorrente. Como é que o método sort terá a garantia de que sua classe possui este método? Isso será feito, novamente, através de um contrato, uma interface!!

Vamos fazer com que os elementos da nossa coleção implementem a interface `java.lang.Comparable`, que define o método `int compareTo(Object)`. Este método deverá devolver zero, se o objeto comparado for **igual** a este objeto, um número negativo, se este objeto for **menor** que o objeto dado, e um número positivo, se este objeto for **maior** que o objeto dado.

Para ordenar as `ContaCorrenteS` por saldo, basta implementar o `Comparable` e fornecer uma implementação para o método `compareTo`. Veja:

```
public class ContaCorrente extends Conta implements Comparable<ContaCorrente>{

    public int compareTo(ContaCorrente outra) {
        if(this.getSaldo()<outra.getSaldo())
            return -1;
        else if(this.getSaldo()>outra.getSaldo())
            return 1;
        else
            return 0;
    }
}
```

Desta forma nossa classe se tornou um comparável. Repare que o critério de ordenação é totalmente aberto, definido pelo programador. Se quisermos ordenar por outro/outros atributo/atributos, basta modificar a implementação do método `compareTo` na classe.

Agora sim, o método `sort` de `Collections` saberá ordenar nossas contas.

No caso anterior, com uma lista de `Strings`, conseguimos obter a ordenação sem ter que fazer nada. Isso aconteceu por que, certamente, alguém reescreveu a classe `String` implementando a interface `Comparable` e fornecendo uma implementação para o método `compareTo`, fazendo comparações pela ordem alfabética. O mesmo acontece com outras classes como `Integer`, `BigDecimal`, `Date` entre outras.

Veja o exemplo de ordenação que funciona:

```
public class ContaCorrente extends Conta implements Comparable<ContaCorrente>{
    public int compareTo(ContaCorrente outra) {
        if(this.getSaldo()<outra.getSaldo())
            return -1;
        else if(this.getSaldo()>outra.getSaldo())
            return 1;
```

```
        else
            return 0;
    }

    public static void main(String[] args) throws
DepositoInvalidoException {
    ContaCorrente c1 = new ContaCorrente();
    c1.deposita(3000);

    ContaCorrente c2 = new ContaCorrente();
    c2.deposita(2000);

    ContaCorrente c3 = new ContaCorrente();
    c3.deposita(1000);

    List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
    contas.add(c1);
    contas.add(c2);
    contas.add(c3);
    Iterator<ContaCorrente> it=contas.iterator();
    while(it.hasNext())
        System.out.println(it.next().getSaldo());
    Collections.sort(contas); //qual seria o critério para esta
ordenação??
    it=contas.iterator();
    while(it.hasNext())
        System.out.println(it.next().getSaldo());
}
}
```

também podemos colocar a ordenação em forma decrescente. Para tal, basta mudar o retorno das nossas comparações. Veja:

```
public int compareTo(ContaCorrente outra) {
    if(this.getSaldo()<outra.getSaldo())
        return 1; //Antes retornava -1
    else if(this.getSaldo()>outra.getSaldo())
        return 1;
    else
        return -1; //Antes retornava 1
}
```

❖ Ordenado com a classe Comparator

Caso você não queira mudar o critério de ordenação definido para ContaCorrente, você pode escrever uma classe que implementa Comparator e que reescreva o método compare de forma a fornecer outro critério de ordenação.

Isso permite que você chame uma outra versão de Collections.sort que recebe além da lista, um argumento do tipo Comparator. Veja abaixo:

```
public class ContaCorrenteComparatorSaldo implements
Comparator<ContaCorrente>{

    public int compare(ContaCorrente c1, ContaCorrente c2) {
        if(c1.getSaldo()<c2.getSaldo())
            return -1;
        else if(c1.getSaldo()>c2.getSaldo())
            return 1;
    }
}
```

```
        return 1;
    else
        return 0;
}

public static void main(String[] args) throws
DepositoInvalidoException {
    ContaCorrente c1 = new ContaCorrente();
    c1.deposita(3000);

    ContaCorrente c2 = new ContaCorrente();
    c2.deposita(2000);

    ContaCorrente c3 = new ContaCorrente();
    c3.deposita(1000);

    List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
    contas.add(c1);
    contas.add(c2);
    contas.add(c3);
    Iterator<ContaCorrente> it=contas.iterator();
    while(it.hasNext())
        System.out.println(it.next().getSaldo());
    Collections.sort(contas,
                     new
ContadoraComparadorSaldo());
    it=contas.iterator();
    while(it.hasNext())
        System.out.println(it.next().getSaldo());
}
}
```

❖ Outros métodos da classe Collections:

- * binarySearch(List, Object): Realiza uma busca binária por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado.
- * max(Collection): Retorna o maior elemento da coleção.
- * min(Collection): Retorna o menor elemento da coleção.
- * reverse(List): Inverte a lista.

❖ Outros:

```
Collections.shuffle(contas); //Embaralha
Collections.rotate(contas, 5); //Rotaciona a lista em 5 posições
```

Consulte a documentação para conhecer outros métodos.

Existe uma classe análoga, a java.util.Arrays, que faz operações similares com arrays.
É importante conhecê-la para evitar escrever código já existente.

❖ Exercícios com Ordenação:

- 1) Faça sua classe ContaPoupanca (exercícios anteriores) implementar a interface Comparable<ContaPoupanca>. Utilize o critério de ordenar pelo número da conta.

```
public class ContaPoupanca extends Conta implements
Comparable<ContaPoupanca>{

    public void atualiza(double taxa) {
        this.saldo = this.saldo + (this.saldo * (taxa * 3));
    }
}
```

```
    }

    public int compareTo(ContaPoupanca outraConta) {
        if(this.getNumero()<outroConta.getNumero())
            return -1;
        else if(this.getNumero()>outroConta.getNumero())
            return 1;
        else
            return 0;
    }
}
```

O que acha da implementação abaixo?

```
public int compareTo(ContaPoupanca outraConta) {
    return this.getNumero()-outroConta.getNumero();
}
```

2) Crie uma classe TestaOrdenacao, onde você vai instanciar diversas contas e adicioná-las a uma List<ContaPoupanca>. Use o Collections.sort() nesta lista:

```
public class TestaOrdenacao {
    public static void main(String[] args) {
        List<ContaPoupanca> contas = new ArrayList<ContaPoupanca>();

        ContaPoupanca c1 = new ContaPoupanca();
        c1.setNumero(1970);
        contas.add(c1);

        ContaPoupanca c2 = new ContaPoupanca();
        c2.setNumero(1520);
        contas.add(c2);

        ContaPoupanca c3 = new ContaPoupanca();
        c3.setNumero(1880);
        contas.add(c3);

        Collections.sort(contas);
    }
}
```

3) Faça um laço para imprimir todos os números das contas já ordenadas.

```
for(int i=0;i<contas.size();i++) {
    Conta atual= contas.get(i);
    System.out.println("Número: "+atual.getNumero());
}
```

Repare que, na verdade, quem chama o método compareTo que reescrevemos é o Collections.sort(), que o usa como base para o algoritmo de ordenação. Isso cria um sistema extremamente coeso e, ao mesmo tempo, com baixo acoplamento: a classe Collections nunca imaginou que ordenaria objetos do tipo ContaPoupanca, mas já que eles são Comparable, o seu método sort os aceita!! Outra maneira mais suscinta de escrever o for acima é:

```
for(int i=0;i<contas.size();i++) {
    System.out.println("Número: "+ contas.get(i).getNumero());
```

}

4) Use a classe Random para criar contas aleatoriamente, conforme demonstrado abaixo:

```
//Criando 20 contas
    for (int i = 0; i < 20; i++) {
        Conta c=new ContaPoupanca();
        Random r=new Random();
        c.setNumero(r.nextInt()/1000000);
        contas.add((ContaPoupanca) c);
    }

    for(int i=0;i<contas.size();i++){
        Conta atual= contas.get(i);
        System.out.println("Número: "+atual.getNumero());
    }
    Collections.sort(contas);
    System.out.println("::::::::::::::::::Lista
Ordenada::::::::::::::::::");
    for(int i=0;i<contas.size();i++){
        Conta atual= contas.get(i);
        System.out.println("Número: "+atual.getNumero());
    }
```

5) O que teria acontecido se a classe ContaPoupanca não implementasse Comparable<ContaPoupanca>, mas tivesse o método compareTo implementado?

Faça este teste: Remova temporariamente a sentença implements Comparable<ContaPoupanca>, não remova o método compareTo e veja o que acontece. Basta ter o método, sem assinar a interface?

6) Utilize LinkedList ao invés de ArrayList. Precisamos alterar mais algum código para que esta substituição funcione? Rode o programa. Alguma diferença?

7) Imprima a referência para esta lista. Repare que o toString de uma ArrayList/LinkedList é reescrito:

```
System.out.println(contas);
```

8) Mude o critério de ordenação.

9) Crie um outro critério de comparação através da interface Comparator e use o mesmo no Collections.sort().

❖ Conjunto (java.util.Set)

Um conjunto (Set) funciona de forma análoga aos conjuntos da matemática. Trata-se de uma coleção que não permite elementos duplicados. A ordem em que os elementos são armazenados pode não ser a ordem em que eles foram inseridos. A interface não define como deve ser este comportamento. tal ordem varia de implementação para implementação. Ordem desconhecida.

Um conjunto é representado pela interface Set e tem como suas principais implementações as classes HashSet, LinkedHashSet e TreeSet.

O código a seguir cria um conjunto e adiciona diversos elementos (alguns repetidos):

```
Set<String> conjunto=new HashSet<String>();  
conjunto.add("José");  
conjunto.add("Pedro");  
conjunto.add("Paulo");  
conjunto.add("Pedro");//repetido  
conjunto.add("Maria");  
conjunto.add("Roberto");  
  
System.out.println(conjunto);
```

O elemento repetido "Pedro" não será adicionado e o método add lhe retornará false.

O uso de um Set pode parecer desvantajoso. Afinal, ele não armazena a ordem e não aceita elementos repetidos. Não há métodos que trabalham com índices, como por exemplo, get(int) que as listas possuem.

A grande vantagem do Set é que existem implementações, como a HashSet, que possuem uma performance incomparável com as Lists quando usadas para pesquisa (método contains por exemplo). Veremos esta enorme diferença durante os exercícios.

❖ Ordem de um Set

A implementação TreeSet insere os elementos de tal forma que, quando forem percorridos, eles apareçam em uma ordem definida pelo método de comparação entre seus elementos. Esse método é definido pela interface java.lang.Comparable. Ou ainda, pode se passar um Comparator para seu construtor.

Já o LinkedHashSet mantém a ordem de inserção dos elementos.

❖ Principais interfaces (java.util.Collection)

Uma coleção pode implementar diretamente a interface Collection, no entanto, normalmente se usa uma das duas subinterfaces mais famosas: Set e List (**ambas filhas de Collection**).

Set --> define um conjunto de elementos únicos;

List --> permite elementos duplicados além de manter a ordem em que os mesmos foram adicionados.

A busca em um **Set** pode ser mais rápida já que diversas implementações utilizam-se de tabelas de espalhamento (hash tables).

A interface **Map** faz parte do Framework, mas **não estende Collection** (Veremos Map mais adiante).

No java 5 temos ainda outra filha de Collection: a Queue, que define métodos de entrada e de saída e cujo critério será definido pela sua implementação (por exemplo LIFO, FIFO ou ainda um heap onde cada elemento possui sua chave de propriedade).

❖ Percorrendo coleções no Java

Como percorrer os elementos de uma coleção? Se for uma lista, podemos usar o for invocando o método get para obter cada elemento.

Mas e no caso de um Set, por exemplo, que não possui um método capaz de indexar o conjunto?

Neste caso podemos usar o enhanced-for (foreach) para percorrer qualquer Collection sem se preocupar com estas coisas. Veja um exemplo:

```
Set conjunto = new HashSet();
conjunto.add("Item 1");
conjunto.add("Item 2");
conjunto.add("Item 3");

for (Object elemento : conjunto) {
    String palavra = (String) elemento;
    System.out.println(palavra);
}
```

O Java vai usar o iterator da Collection dada para percorrer a coleção. Se você já estiver usando uma coleção parametrizada, o for pode ser feito utilizando um tipo mais específico:

```
Set<String> conjunto = new HashSet<String>();
conjunto.add("Item 1");
conjunto.add("Item 2");
conjunto.add("Item 3");

for (Object palavra : conjunto) {
    System.out.println(palavra);
}
```

Em que ordem os elementos serão acessados?

Diferente do que acontece com as listas, em um conjunto a ordem depende da implementação da interface Set.

Por que o Set é, então, tão importante e tão usado?

Para saber se um item já existe em uma lista, é muito mais rápido usar algumas implementações de Set do que de um List. Os TreeSetS, por exemplo, já vem ordenados de acordo com as características que desejarmos!

Quando não houver a necessidade de guardar os elementos em uma ordem pré-determinada e buscá-los através de um índice, considere usar um Set!

❖ Iterando sobre as coleções com java.util.Iterator

Antes de o Java5 introduzir o enhanced-for, iterações em coleções eram feitas com o Iterator. Toda coleção fornece acesso a um iterator, um objeto que implementa a interface Iterator, que conhece internamente a coleção e dá acesso a todos os seus elementos.

na verdade, quando usamos o enhanced-for, na verdade estamos utilizando Iterator por trás dos panos. Veja como percorrer um conjunto com Iterator:

```
Set conjunto = new HashSet();
conjunto.add("Item 1");
conjunto.add("Item 2");
```

```
conjunto.add("Item 3");

Iterator it=conjunto.iterator();
while(it.hasNext())
    System.out.println((String) it.next());
```

Uma vantagem de se usar Iterator é a possibilidade de remover elementos da coleção durante a iteração. Veja a seguir:

```
Set conjunto = new HashSet();
conjunto.add("Item 1");
conjunto.add("Item 2");
conjunto.add("Item 3");

Iterator it=conjunto.iterator();
while(it.hasNext()){
    if(it.next().equals("Item 2")){
        it.remove();
        break;
    }
}

it=conjunto.iterator();
while(it.hasNext())
    System.out.println(it.next());
```

❖ ListIterator

O Iterator pode ser fornecido tanto por uma lista quanto por um conjunto. Uma lista fornece, ainda, um listIterator, que oferece recursos adicionais, específicos para listas. Usando ListIterator você pode adicionar um elemento na lista ou ainda voltar para o elemento que foi iterado anteriormente.

❖ Mapas (java.util.Map)

Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. O conceito é o mesmo das matrizes assossiativas usadas em Perl e Php.

java.util.Map é um mapa, pois é possível usá-lo para mapear uma chave, por exemplo: mapeie o valor "Rafael" à chave "Professor", o valor "Rua Argentina" à chave "Endereço".

O método **put(Object, Object)** da interface Map recebe a chave e o valor de uma nova assossiação.

Para saber o valor assossiado a um objeto chave, usamos **get(Object objetoChave)**.

Veja o exemplo:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(1000);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(2000);
ContaCorrente c3 = new ContaCorrente();
c3.deposita(3000);

Map mapaDeContas= new HashMap();
```

```
mapaDeContas.put("diretor", c3);
mapaDeContas.put("gerente", c2);

//Qual é a conta associada a um diretor?
Object elemento=mapaDeContas.get("diretor");
ContaCorrente contaDoDiretor=(ContaCorrente) elemento;
System.out.println(contaDoDiretor.getSaldo());
```

Um mapa é utilizado para "indexar" objetos de acordo com determinado critério, para podermos buscá-los rapidamente através deste critério. Ele, assim como as coleções, trabalha diretamente com Object (tanto para chave quanto para valor), o que torna necessário o casting no momento de recuperar os elementos.

Suas principais implementações são o HashMap, o TreeMap e o HashTable.

Apesar do mapa fazer parte do Framework, ele não estende Collection, por ter um comportamento bem diferente.

As coleções internas de um mapa (a de chaves e a de valor) são acessíveis por métodos definidos na interface Map.

O método **keySet()** retorna um Set com todas as chaves daquele mapa.
O método **values()** retorna a Collection com todos os valores que foram associados a alguma das chaves.

❖ Usando Genéricos com os mapas:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(1000);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(2000);
ContaCorrente c3 = new ContaCorrente();
c3.deposita(3000);

Map<String, ContaCorrente> mapaDeContas= new HashMap<String,
ContaCorrente>();

mapaDeContas.put("diretor", c3);
mapaDeContas.put("gerente", c2);

//Neste caso, dispensa casting
ContaCorrente contaDoDiretor=mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

❖ Properties

Um mapa importante é a tradicional classe Properties, que mapeia Strings e é muito utilizada para configuração de aplicações.

A Properties possui, também, métodos para ler e gravar o mapeamento com base em um arquivo text, facilitando muito a sua persistência.

```
Properties config=new Properties();
config.setProperty("database.login","root");
config.setProperty("database.password","");

```

```
config.setProperty("database.host", "jdbc:mysql://localhost/teste");
" );
//muitas linhas depois....
```

```
String login=config.getProperty("database.login");
String senha=config.getProperty("database.password");
String host=config.getProperty("database.host");
DriverManager.getConnection(login,senha,host);
```

A classe Properties foi desenhada com o propósito de trabalhar com associação entre Strings. Por isso não houve necessidade de casting.

❖ Collections Framework - Resumo:

Listas (Implementam a interface List): Vector, ArrayList e LinkedList

Conjuntos (Implementam a interface Set): HashSet, LinkedHashSet e TreeSet

Collections (**Classe Concreta**)

Collection (**Interface**)

List e Set são subinterfaces (filhas) de Collection.

Assinaturas do método sort:

Collections.sort(Object)

Collections.sort(Object, Comparator)

ArrayList x LinkedList

ArrayList é mais rápida na pesquisa (get(i)) e LinkedList é mais rápida na inserção/remoção!

Listas x Conjuntos

Para pesquisa utilizando contains(Object obj), os conjuntos são mais rápidos que as listas.
Listas permitem elementos duplicados. Conjuntos não permitem!

Exercícios: Collections

1) Crie um código que insira 30 mil números num ArrayList e pesquise-os. Vamos usar um método de System para cronometrar o tempo gasto:

```
System.out.println("Iniciando...");
long inicio=System.currentTimeMillis();
Collection<Integer> numeros=new ArrayList<Integer>();
int total=300000;
for (int i = 0; i < total; i++) {
    numeros.add(i);
}
for (int i = 0; i < total; i++) {
    numeros.contains(i);
}
long fim=System.currentTimeMillis();
long tempo=fim-inicio;
System.out.println("Tempo gasto: "+tempo);
```

Troque ArrayList por HashSet e verifique o tempo que vai demorar.

O que torna a operação com ArrayList tão lenta? A inserção ou a pesquisa?
Evite o uso de List em casos que queiramos utilizá-la essencialmente para pesquisa.

2) faça testes com o Map, como visto anteriormente:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(1000);
ContaCorrente c2 = new ContaCorrente();
c2.deposita(2000);
ContaCorrente c3 = new ContaCorrente();
c3.deposita(3000);

Map<String, ContaCorrente> mapaDeContas= new HashMap<String,
ContaCorrente>();

mapaDeContas.put("diretor",c3);
mapaDeContas.put("gerente",c2);

//Neste caso, dispensa casting
ContaCorrente contaDoDiretor=mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

Depois altere para trabalhar sem genéricos. Vai precisar fazer casting nesse caso.

3) Assim como no exercício 1, crie uma comparação entre ArrayList e LinkedList, para ver qual é a mais rápida para adicionar elementos na primeira posição (list.add(0, elemento), como por exemplo:

```
long inicio,fim;
double tempo;

System.out.println("Iniciando com LinkedList...");
inicio=System.currentTimeMillis();
List<Integer> lista=new LinkedList<Integer>();
for(int i=0;i<100000;i++){
    lista.add(0, i);
}
fim=System.currentTimeMillis();
tempo=(fim-inicio)/1000.0;
System.out.println("Tempo gasto com LinkedList: "+tempo);
System.out.println("#####");
System.out.println("Iniciando com ArrayList...");
inicio=System.currentTimeMillis();
lista=new ArrayList<Integer>();
for(int i=0;i<100000;i++){
    lista.add(0, i);
}
fim=System.currentTimeMillis();
tempo=(fim-inicio)/1000.0;
System.out.println("Tempo gasto com ArrayList: "+tempo);
```

Você vai perceber que a LinkedList é muito mais rápida para esta operação (inserção).

Que tal testar agora a operação get(int)?

Apesar de o certo ser utilizar um Iterator ou o enhanced-for, vamos testar com o for:

```
long inicio,fim;
double tempo;

System.out.println("Iniciando com LinkedList...");
inicio=System.currentTimeMillis();
List<Integer> lista=new LinkedList<Integer>();
for(int i=0;i<30000;i++){
    lista.add(i);
}
for(int i=0;i<30000;i++){
    lista.get(i);
}
fim=System.currentTimeMillis();
tempo=(fim-inicio)/1000.0;
System.out.println("Tempo gasto com LinkedList: "+tempo);
System.out.println("#####");
System.out.println("Iniciando com ArrayList...");
inicio=System.currentTimeMillis();
lista=new ArrayList<Integer>();
for(int i=0;i<30000;i++){
    lista.add(i);
}
for(int i=0;i<30000;i++){
    lista.get(i);
}
fim=System.currentTimeMillis();
tempo=(fim-inicio)/1000.0;
System.out.println("Tempo gasto com ArrayList: "+tempo);
```

Percebeu como ArrayList é muito mais rápida para este caso (pesquisa)?

4) Crie uma classe Banco que possui uma List de Conta chamada contas. Essa List poderá receber tanto ContaCorrente quanto ContaPoupança por causa do polimorfismo.

Crie um método void adiciona(Conta c), um método Conta pega(int i) e outro int pegaTotalDeContas(), muito similar à relação anterior de Empresa-Funcionários. Basta usar a sua lista para delegar as chamadas para os métodos de coleções que estudamos.

5) No Banco, crie um método Conta buscaPorNome(String nome) que procura por uma conta cujo nome seja equals ao nome dado.

Você até pode implementar esse método com um for, porém não terá uma performance eficiente.

Adicionando um atributo privado do tipo Map<String, Conta> terá um impacto significativo. Toda vez que adiciona(Conta c) for invocado, você deve invocar put(c.getNome(), c) no seu mapa. Dessa maneira, quando alguém invocar o método buscaPorNome(String nome), basta você fazer o get no seu mapa, passando nome como argumento!

6) Gere todos os números de 1 a 1000 e ordene em ordem decrescente utilizando um TreeSet.

7) Gere todos os números de 1 a 1000 e ordene em ordem decrescente utilizando um ArrayList.

50. Programação Concorrente e Threads

A necessidade de fazer várias coisas simultaneamente (**paralelamente**) aparece frequentemente na computação. Para vários programas distintos, normalmente o próprio sistema operacional gerencia isso através de vários processos em paralelo.

Em um programa só (um processo só), se queremos executar coisas em paralelo, normalmente falamos de **Threads**.

Em Java usamos a classe `java.lang.Thread` para criarmos linhas de execução paralelas. A classe `Thread` recebe, como argumento, um objeto com o código que queremos rodar. Veja o exemplo de um programa que "gera um pdf" e mostra uma barra de progresso:

```
public class GeraPDF {
    public void rodar() {
        //Logica para gerar um PDF
        for(int i=0;i<10000;i++) {
            ;//simula a geração de um arquivo pdf
        }
    }
}

public class BarraDeProgresso {
    public void rodar() {
        //Mostra a barra de progresso e vai atualizando ela
        for(int i=10000;i>0;i--) {
            System.out.println(i);
        }
    }
}
```

No método main da nossa classe principal, criamos os objetos e passamos para a classe `Thread`. O método `start` é responsável por iniciar a execução da Thread:

```
public class MeuPrograma {
    public static void main(String[] args) {
        GeraPDF geraPdf = new GeraPDF();
        Thread threadDoPdf = new Thread(geraPdf);
        threadDoPdf.start();

        BarraDeProgresso barraDeProgresso = new BarraDeProgresso();
        Thread threadDabarra = new Thread(barraDeProgresso);
        threadDabarra.start();

    }
}
```

O código acima não vai compilar. Como a classe `Thread` vai saber que deve chamar o método `roda()`? Como ela vai saber que nome daremos ao método? Como ela vai saber qual método chamar?

Falta estabelecer um contrato entre nossas classes e a classe `Thread`, estabelecendo qual método deve rodar.

Este contrato já existe e é declarado na interface `Runnable`: devemos dizer que nossas classes são "executáveis" e que seguem este contrato. O contrato diz que devemos fornecer uma implementação para o método `run`.

Resumindo: basta fazer com que nossas classes implementem `Runnable` e forneçam uma implementação para o método `run`.

E é justamente no método run que devemos colocar o nosso código que queremos executar em paralelo.

Feito isso, a Thread saberá como executar a nossa classe!

```
public class GeraPDF implements Runnable{
    public void run() {
        //Logica para gerar um PDF
        for(int i=0;i<10000;i++) {
            ;//simula a geração de um arquivo pdf
        }
    }
}

public class BarraDeProgresso implements Runnable {
    public void run() {
        // Mostra a barra de progresso e vai atualizando ela
        for (int i = 10000; i > 0; i--) {
            System.out.println(i);
        }
    }
}
```

A classe Thread recebe no construtor um objeto que é um Runnable, e seu método start chama o método run da nossa classe. Repare que a classe Thread não sabe o tipo específico da nossa classe; para ela, basta saber que a classe segue o contrato estabelecido por Runnable e possui o método run implementado.

É o bom uso de interface e polimorfismo na prática!!

Estendendo a classe Thread

Outra forma de se trabalhar com Threads é estender a classe Thread.

A classe Thread implementa Runnable e possui o método run que, neste caso, não faz nada.

Basta estender Thread e reescrever o método run! Veja:

```
public class GeraPDFT extends Thread {
    public GeraPDFT(String nomeDaThread) {
        //Este valor vai para o atributo name de Thread
        super(nomeDaThread);
    }

    public void run() {
        // Logica para gerar um PDF
        for (int i = 0; i < 10000; i++) {
            // simula a geração de um arquivo pdf
            System.out.println(this.getName()+" executando o
passo "+i);
        }
    }
}

public class BarraDeProgressoT extends Thread{
    public BarraDeProgressoT(String nomeDaThread) {
        //Este valor vai para o atributo name de Thread
        super(nomeDaThread);
    }
}
```

```
public void run() {
    // Logica para gerar um PDF
    for (int i = 10000; i > 0; i--) {
        // simula a geração de um arquivo pdf
        System.out.println(this.getName() + " executando o
    passo "+(i-10000));
    }
}
```

E como nossa classe é uma Thread, podemos chamar o start diretamente:

```
public class MeuProgramaT {
    public static void main(String[] args) {
        GeraPDFT threadDoPdf = new GeraPDFT("Thread do Pdf...");  

        BarraDeProgressoT threadDaBarra = new  

        BarraDeProgressoT("Thread da barra...");  

        threadDoPdf.start();
        threadDaBarra.start();
    }
}
```

Apesar de ser um código mais simples, estamos usando herança por preguiça e não por polimorfismo, que seria a grande vantagem. Estamos herdando um monte de métodos para usar apenas o run.

Prefira sempre implementar Runnable!!

Para que a thread atual durma, basta chamar o método a seguir, se quiser que ela durma por 3 segundos por exemplo:

threadDaBarra.sleep(3000);

Problemas de Concorrência

Imagine um banco atualizando o saldo de cada uma de suas milhões de contas uma a uma a cada 1º dia de cada mês, de acordo com determinada taxa. Este processo provavelmente demoraria horas. Ao mesmo tempo, como um banco não pode parar de funcionar, teríamos clientes sacando e depositando paralelamente.

A atualização de contas se dá através do método atualiza da classe conta sendo chamado pela classe AtualizadorDeContas que vimos anteriormente.

Em uma Thread pegamos todas as contas e chamamos o método atualiza. Em outra podemos estar sacando ou depositando dinheiro. Estamos compartilhando objetos entre múltiplas Threads (as contas neste caso).

Agora imagine a seguinte possibilidade (mesmo que muito remota): No exato instante que o atualizador está atualizando uma conta X, o cliente dono desta conta resolve efetuar um saque. Como sabemos, a qualquer instante o escalonador pode parar uma Thread para executar outra, e você não tem controle sobre isso.

Veja a classe Conta:

```
public abstract class Conta {
    protected double saldo;
    private double limite;
```

```
private Cliente titular = new Cliente();
private int numero;

public void saca(double quantidade) throws Exception {

    if ((this.saldo + this.limite) >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new SaldoInsuficienteException("Novo:Saldo
insuficiente. Você só pode sacar "+this.getSaldo()+this.getLimite());
    }
}

public void deposita(double quantidade) throws
DepositoInvalidoException {
    double novoSaldo=this.saldo+quantidade;
    if (quantidade >= 0) {
        this.saldo =novoSaldo;
    }else{
        throw new DepositoInvalidoException("Erro ao
depositar");
    }
}

void transferePara(Conta destino, double valor) {
    try {
        this.saca(valor);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

    try {
        destino.deposita(valor);
    } catch (DepositoInvalidoException e) {
        e.printStackTrace();
    }
}

public void atualiza(double taxa) {
    this.saldo = this.saldo + (this.saldo * (taxa * 2));
}
//Outros métodos
```

Situação hipotética: Uma conta tem saldo de 100 reais. Um cliente faz um depósito de 1000 reais. Isso dispara uma Thread que chama o método deposita(); Ele começa calculando (sem fazer a atribuição) o novo saldo que passa a ser 1100 reais veja:

```
public void deposita(double valor) throws DepositoInvalidoException {
    double novoSaldo=this.saldo+valor; //novoSaldo = 1100,00
    //A Thread PARA aqui!!!! (saldo ainda continua valendo = 100,00)
    if (quantidade >= 0) {
        this.saldo =novoSaldo;
    }else{
        throw new DepositoInvalidoException("Erro ao depositar");
    }
}
```

O escalonador para esta Thread e começa a executar outra Thread que chama o método atualiza da mesma conta. Isso quer dizer que o novo saldo vai para 101 reais. Neste

instante o escalonador troca de Threads novamente. A Thread que fazia o depósito executa a seguinte linha:

`this.saldo = novoSaldo; //Nesse momento o saldo passa a valer 1100 reais.`

Acabando o depósito, o escalonador volta para a Thread que atualiza as contas e executa a seguinte linha do método atualiza:

`this.saldo = this.saldo + (this.saldo * 0.01); //saldo vai valer 101,00`

Isso faz com que o saldo passe a valer 101 reais. Afinal, na outra Thread essa era a última informação do saldo!!

Resultado: o depósito de 1000 reais foi totalmente ignorado e o cliente não vai ficar nem um pouco feliz. Perceba que não é possível detectar este erro, já que o código foi executado perfeitamente, sem problemas. **O problema aqui foi o acesso simultâneo de duas Threads ao mesmo objeto.**

Dizemos que esta classe não é Thread safe, isto é, não está pronta para ter uma instância compartilhada entre várias threads concorrentemente.

O ideal é que não seja possível atualizar uma conta enquanto outra pessoa deposita ou saca. Queremos que não seja possível uma Thread acessar uma conta enquanto outra Thread está mexendo nela. Precisamos colocar uma espécie de trava. No momento em que uma Thread entrasse em um destes métodos, ela trancaria a entrada com uma chave.

Essa idéia é chamada de **região crítica**. É um pedaço de código que definimos como crítico e que não pode ser executado por duas Threads ao mesmo tempo.

Podemos fazer isso em Java. Podemos sincronizar um bloco ou método. Nesse caso, apenas uma Thread poderá acessar este bloco de código por vez. A palavra chave synchronized dá essa característica ao bloco de código. Este bloco só será liberado no momento que a Thread que o está acessando sair do bloco, seja por return, disparo de uma Exceção ou na atualização do método (wait()).

Queremos bloquear o acesso simultâneo a uma mesma Conta:

```
public void atualiza(double taxa) {  
    synchronized (this) {  
        double novoSaldo=this.saldo + (this.saldo * (taxa * 2));  
        //Aqui a Thread pode parar...  
        this.saldo = novoSaldo;  
    }  
}  
  
public void deposita(double quantidade) throws  
DepositoInvalidoException {  
    synchronized (this) {  
        if (quantidade >= 0) {  
            this.saldo += quantidade - 0.10;  
        } else {  
            throw new DepositoInvalidoException("Erro ao  
depositar");  
        }  
    }  
}
```

Estes blocos agora são mutuamente exclusivos e só executam de maneira atômica. Threads que tentarem acessá-los enquanto outra já está mexendo neles, ficarão em um conjunto especial esperando pela liberação do lock (não necessariamente numa fila).

Sincronizando o método

É comum sincronizarmos o método inteiro ao invés de definir um bloco sincronizado. Veja:

```
public synchronized void atualiza(double taxa) {  
    this.saldo = this.saldo + (this.saldo * (taxa));  
}
```

Se um método for estático, será sincronizado usando o lock do objeto que representa a classe (NomeDaClasse.class). Veja:

```
public static void testaSincronização() {  
    synchronized (Conta.class) {  
        //instruções....  
    }  
}
```

O pacote java.util.concurrent, conhecido como JUC, entrou no Java 5.0 para facilitar uma série de trabalhos comuns que costumam aparecer em uma aplicação concorrente.

Vector e HashTable

Vector e HashTable são duas collections muito famosas. A diferença entre elas e suas irmãs, respectivamente, ArrayList e Hashmap é que as primeiras são thread safe.

Você pode se perguntar porque não usamos sempre estas classes thread safe. Adquirir um lock tem um custo e, se um objeto não vai ser usado entre diferentes threads, não há porque usar estas classes que consomem mais recursos. Nem sempre é fácil enxergar se devemos sincronizar ou não. Hoje em dia estes recursos já não consomem tanto. Mas isso não é motivo para você sair sincronizando tudo sem necessidade.

Exercícios

1) Vamos implementar o teste a respeito da atualização e depósitos que utilizam concorrentemente a classe Conta.

a) Na classe Conta, crie um construtor que receba o saldo inicial como argumento e sincronize os métodos deposita e atualiza conforme demonstrado abaixo:

```
public class ContaCorrente extends Conta implements Tributavel {  
  
    public ContaCorrente(double saldoInicial) {  
        this.saldo=saldoInicial;  
    }  
  
    public synchronized void atualiza(double taxa) {  
        this.saldo = this.saldo - (this.saldo * (taxa));  
    }  
  
    public void deposita(double quantidade) throws  
DepositoInvalidoException {  
        synchronized (this) {  
            if (quantidade >= 0) {  
                this.saldo += quantidade;  
            } else {  
                throw new DepositoInvalidoException("Erro ao  
depositar");  
            }  
        }  
    }  
}
```

b) Se não encontrar a que fizemos anteriormente, crie a classe Atualizador de Contas:

```
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 atual: " + 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;
    }
}
```

c) Crie as classes EfetuarAtualizacao e EfetuarDeposito, ambas implementando Runnable. Estas classes deverão ser passadas para nossas Threads.

```
public class EfetuarAtualização implements Runnable{
    private List<ContaCorrente> contas;
    private AtualizadorDeContas atualizador;

    public EfetuarAtualização(List<ContaCorrente> contas,
                               AtualizadorDeContas atualizador) {
        this.contas = contas;
        this.atualizador = atualizador;
    }

    public void run() {
        Iterator it = contas.iterator();
        while (it.hasNext()) {
            System.out.println("Thread EfetuarAtualização
rodando...");
            ContaCorrente cc = (ContaCorrente) it.next();
            atualizador.roda(cc);
        }
    }
}

public class EfetuarDeposito implements Runnable{
    private final double deposito;
    private final ContaCorrente conta;

    public EfetuarDeposito(ContaCorrente cc, double valor) {
        this.deposito = valor;
        this.conta = cc;
    }
}
```

```
    }

    public void run() {
        try {
            System.out.println("Thread EfetuarDepósito
rodando...");  
            conta.deposita(deposito);
        } catch (DepositoInvalidoException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

d) Crie a classe TestaThreads, que deverá criar e rodar nossas Threads:

```
public class TestaThread {

    public static void main(String[] args) {
        ContaCorrente c1=new ContaCorrente(10000);
        ContaCorrente c2=new ContaCorrente(10000);
        ContaCorrente c3=new ContaCorrente(10000);
        AtualizadorDeContas atualizador=new
AtualizadorDeContas(0.01);
        List<ContaCorrente> contas=new ArrayList<ContaCorrente>();
        contas.add(c1);
        contas.add(c2);
        contas.add(c3);

        Thread tAtualiza = new Thread(new
EfetuarAtualização(contas, atualizador));
        Thread tDepositac1 = new Thread(new EfetuarDeposito(c1,
1000));
        Thread tDepositac2 = new Thread(new EfetuarDeposito(c2,
1000));
        Thread tDepositac3 = new Thread(new EfetuarDeposito(c3,
1000));

        tDepositac1.start();
        tAtualiza.start();
        tDepositac2.start();
        tDepositac3.start();

        for (int i = 0; i <100000; i++) {
            ; //Para esperar as Threads acabarem de executar...
        }
        System.out.println("O saldo esperado para c1 é 9900. O
saldo de c1 é "+c1.getSaldo());
        System.out.println("O saldo esperado para c2 é 9900. O
saldo de c2 é "+c2.getSaldo());
        System.out.println("O saldo esperado para c3 é 9900. O
saldo de c3 é "+c3.getSaldo());
    }
}
```

Rode esta classe e confira o resultado.

Se tivéssemos muitas contas este código faria mais sentido para simular os depósitos:

```
for (int i = 0; i < contas.size(); i++) {
new Thread(new
EfetuarDeposito(contas.get(i),Math.random()/100)).start();
```

}

2) Vamos imprimir números em paralelo. Escreva a classe Programa:

```
public class Programa implements Runnable {
    private int id;

    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println("Programa " + id + " valor: " +
i);
        }
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

Escreva a classe de teste:

```
public class Teste {
    public static void main(String[] args) {
        Programa p1 = new Programa();
        p1.setId(1);
        Thread t1 = new Thread(p1);
        t1.start();

        Programa p2 = new Programa();
        p1.setId(2);
        Thread t2 = new Thread(p2);
        t2.start();
    }
}
```

Rode várias vezes a classe de teste e observe os diferentes resultados em cada execução.

Exercícios Avançados

1) Vamos enxergar o problema ao se usar uma classe não thread safe: a LinkedList por exemplo.

Imagine que temos um objeto que guarda todas as SQLs que uma aplicação precisa executar. Vamos usar uma `LinkedList<String>` para armazená-los. Nossa aplicação é multi-thread, então diferentes Threads vão inserir diferentes SQLs para serem executados. Não importa a ordem que elas sejam executadas, desde que elas um dia sejam!! Vamos gerar a seguinte classe para adicionar as queries:

```
public class ProduzSQL implements Runnable {
    private int começo, fim;
    private Collection<String> sqls;

    public ProduzSQL(int começo, int fim, Collection<String> sqls) {
        this.começo = começo;
        this.fim = fim;
        this.sqls = sqls;
    }

    public void run() {
        for (int i = começo; i < fim; i++)
    }
}
```

```
        sqls.add("SQL" + i);
    }
}

Vamos criar 3 threads que rodem esse código, todas adicionando SQLs no mesmo
HashSet. Em outras palavras, teremos threads compartilhando e acessando um mesmo
objeto: é aqui que mora o perigo.

public class BancoDeDados {
    public static void main(String[] args) throws InterruptedException
    {
        Collection<String> sqls = new HashSet<String>();
        ProduzSQL p1 = new ProduzSQL(0, 10000, sqls);
        Thread t1 = new Thread(p1);
        ProduzSQL p2 = new ProduzSQL(10000, 20000, sqls);
        Thread t2 = new Thread(p2);
        ProduzSQL p3 = new ProduzSQL(20000, 30000, sqls);
        Thread t3 = new Thread(p3);

        t1.start();
        t2.start();
        t3.start();

        // Faz com que se espere o fim das threds. Pode lançar
        // IninterruptedException
        t1.join();
        t2.join();
        t3.join();
        System.out.println("Threads produtoras de SQL finalizadas");

        // Verifica se todas as sqls foram geradas
        for (int i = 0; i < 15000; i++) {
            if (!sqls.contains("SQL" + i)) {
                throw new IllegalStateException("Não encontrei o
SQL" + i);
            }
        }

        // Verifica se null não se encontra
        for (int i = 0; i < 15000; i++) {
            if (sqls.contains(null)) {
                throw new IllegalStateException("Não deveria ter
null aqui! Não inserimos null.");
            }
        }
        System.out.println("Fim da execução com sucesso!");
    }
}
```

2) Teste o código anterior, mas agora usando o synchronized ao adicionar na coleção:

```
public void run() {
    for (int i = começo; i < fim; i++)
        synchronized (sqls) {
            sqls.add("SQL" + i);
        }
}
```

3) Sem usar o synchronized teste com a classe Vector (que é uma Collection).

4) Novamente, sem usar o synchronized, teste usar HashSet e ArrayList, em vez de Vector. Faça vários testes, pois as threads vão se entrelaçar cada vez de uma forma diferente, podendo ou não ter um efeito inesperado.

51.Ferramentas: jar e javadoc

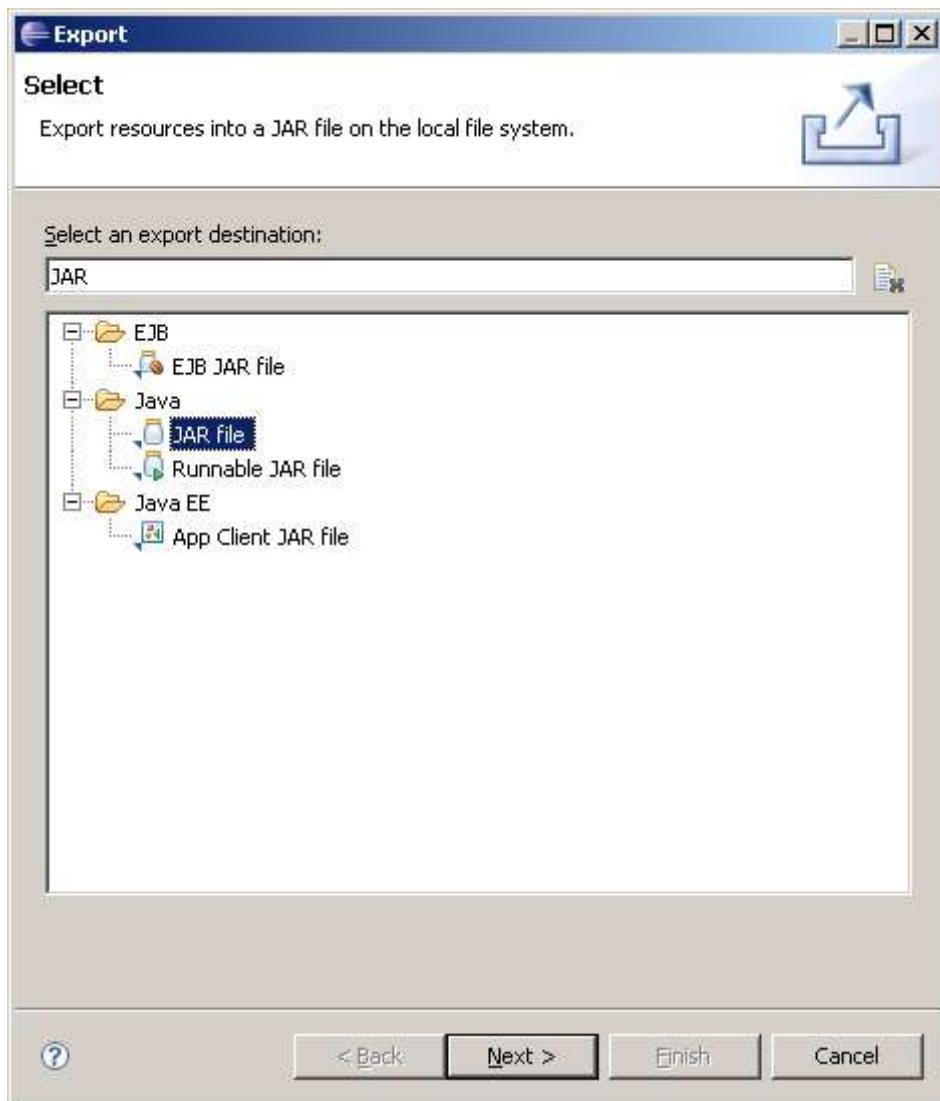
Quando um programa fica pronto, é complicado enviar dezenas ou centenas de classes para cada cliente que quer utilizá-lo. É mais simples compactá-las em um único arquivo. O formato de compactação padrão é o ZIP com a extensão do arquivo compactado JAR.

O arquivo jar (Java Arquive) → possui um conjunto de classes (e arquivos de configurações) compactados no estilo de um arquivo zip. O arquivo JAR pode ser criado com qualquer compactador zip disponível no mercado, inclusive o programa jar que vem junto com o JDK.

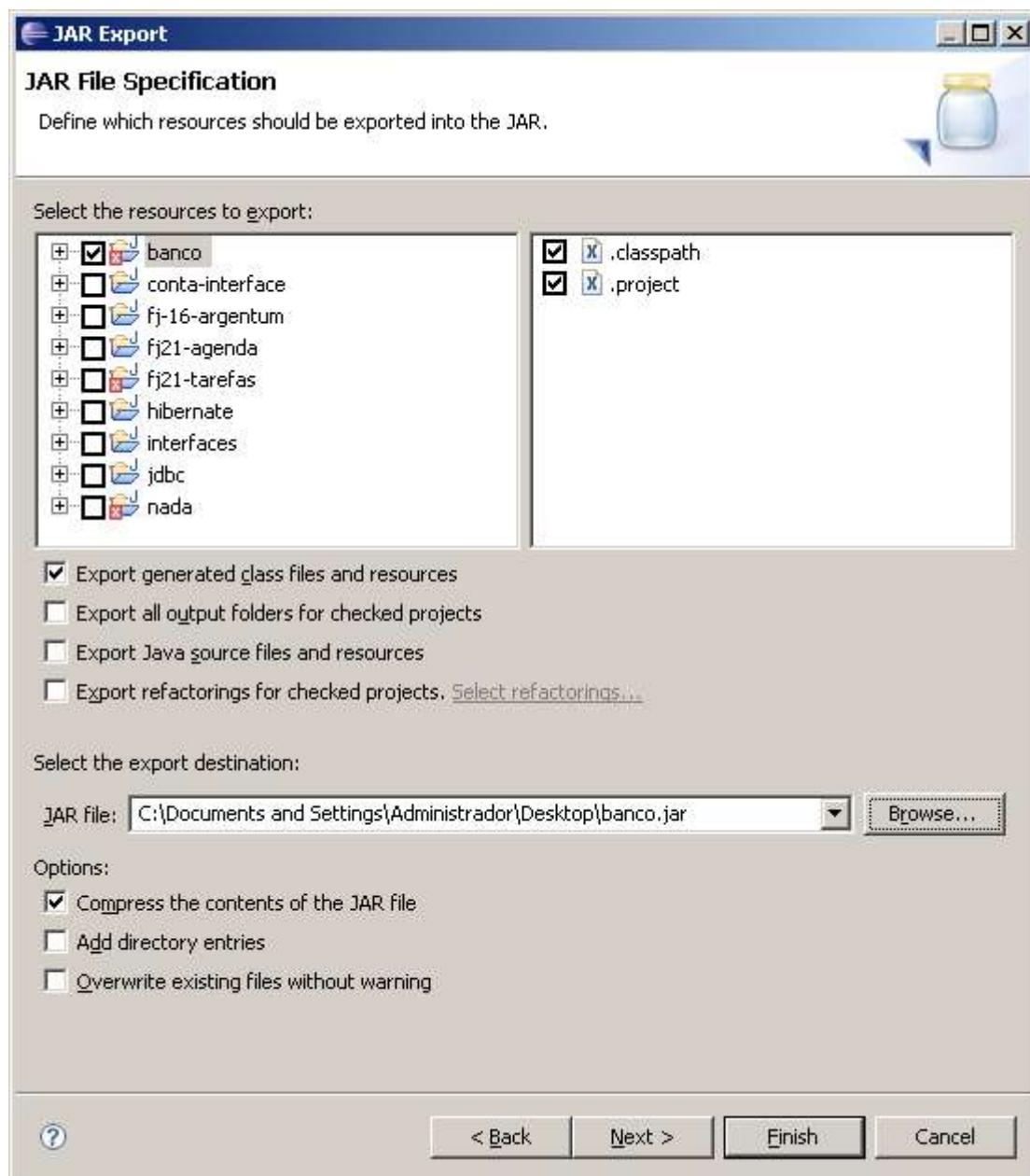
Gerando um jar pelo Eclipse:

Vamos gerar um jar a partir do Eclipse, passo-a-passo:

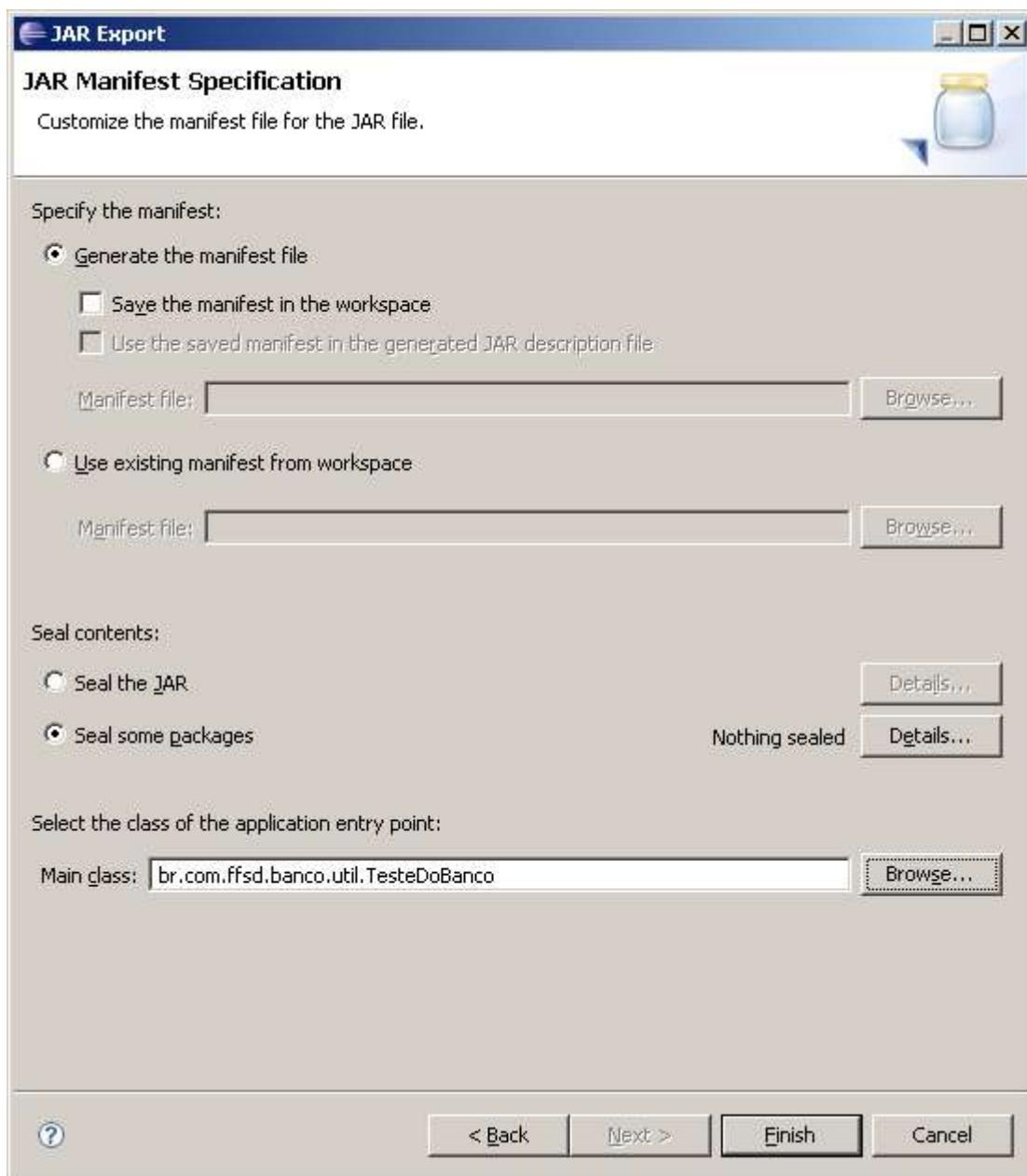
- 1) Clique com o botão direito em cima do seu projeto e selecione a opção export.
- 2) Na tela Export (como mostra a figura abaixo), selecione a opção “JAR file” e aperte o botão “Next”.



- 3) na opção “JAR file”, selecione o local onde você deseja salvar o arquivo JAR e aperte “Next”.



- 4) Na próxima tela, simplesmente clique em next, pois não há nenhuma configuração a ser feita.
- 5) Na tela abaixo, na opção “select the class of the application entry point”, você deve escolher qual será a classe que vai rodar automaticamente quando você executar o JAR.



6) Entre na linha de comando: `Java -jar banco.jar`
É comum dar um nome mais significativo aos jars, incluindo nome da empresa, do projeto e versão, como cefet-banco-1.0.jar.

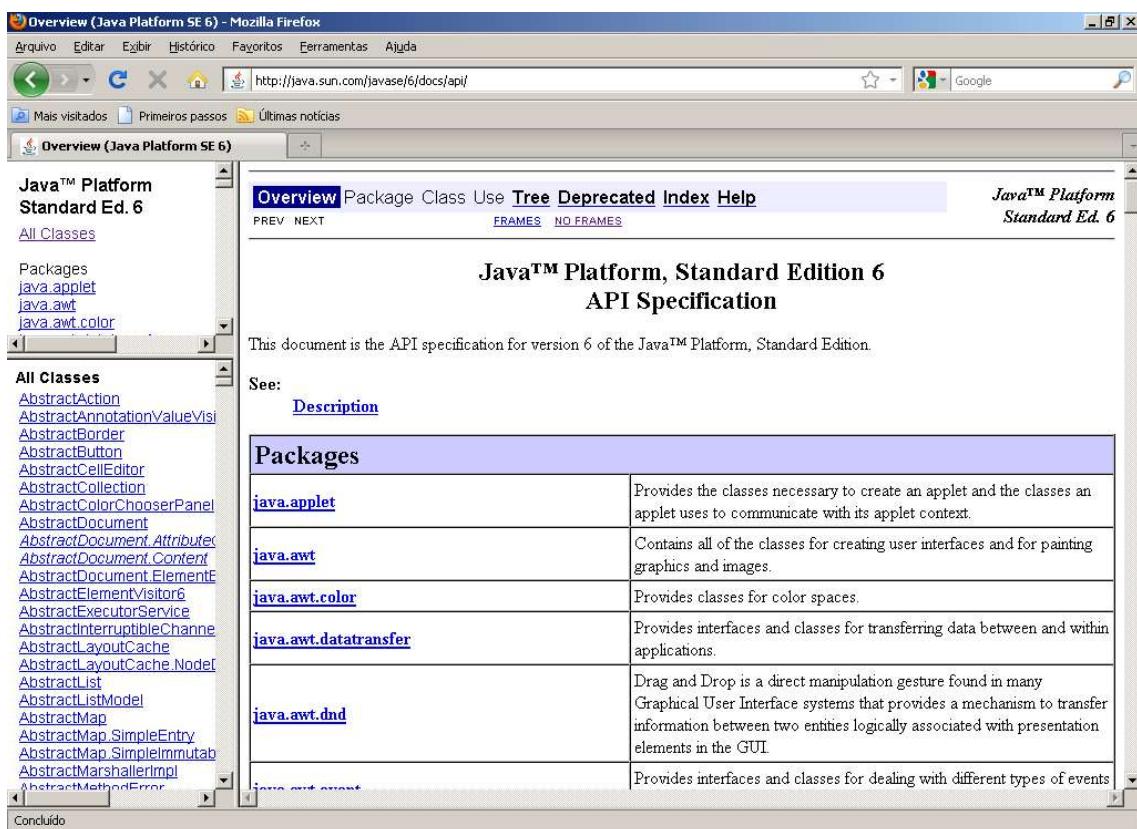
Javadoc

Como saber o que cada classe tem no Java? Quais são seus métodos, o que eles fazem?

Para conhecer melhor a api do Java entre no link abaixo:

<http://java.sun.com/javase/6/docs/api/>

No site da Sun você pode e deve baixar a documentação das bibliotecas do Java, frequentemente referida como javadoc ou API (sendo na verdade a documentação da API).



No quadro superior esquerdo você encontra os pacotes. No quadro inferior esquerdo está a listagem das classes e interfaces do respectivo pacote (ou de todos, caso nenhum tenha sido especificado).

Clicando em uma classe ou interface, o quadro da direita passa a detalhar todos os atributos e métodos.

Perceba que métodos e atributos privados não estão aí. O importante é documentar o que sua classe faz e não como ela faz: detalhes da implementação como atributos e métodos privados, não interessam ao desenvolvedor que usará sua biblioteca (ou, ao menos, não deveriam interessar).

Você também pode gerar o javadoc a partir da linha de comando com o comando: javadoc. Contudo, é muito mais simples gerar a partir do Eclipse! Pesquise a respeito.

52. O pacote Java.lang

Já usamos, por diversas vezes, as classes String e System. Vimos o sistema de pacotes do Java e nunca precisamos dar um import nessas classes. Isso ocorre porque elas estão dentro do pacote Java.lang, que é automaticamente importado para você. É o único pacote com essa característica. Vamos ver a seguir, um pouco de suas principais classes.

System

A classe System possui uma série de atributos e métodos estáticos. Já usamos o atributo System.out, para imprimir.

Olhando a documentação, você irá perceber que o atributo `out` é do tipo `PrintStream` do pacote `Java.io`. Veremos sobre essa classe mais adiante. Já percebemos que podemos quebrar o `System.out.println` em duas linhas:

```
PrintStream saída = System.out;  
Saída.println("Olá Mundo!");
```

Ela também possui o atributo `in`, que lê a entrada padrão, porém só consegue captar bytes:

```
Int i = System.in.read();
```

O código acima deve estar dentro de um bloco `try/catch`, pois pode lançar uma exceção `IOException`. É útil ficar lendo de byte em byte? Trabalharemos mais com a entrada padrão também no próximo capítulo.

O `System` também tem um método que simplesmente desliga a JVM, retornando um código de erro para o sistema operacional, é o `exit`.

```
System.exit(0);
```

Também veremos mais sobre a classe `System` no decorrer das outras disciplinas. Consulte a documentação do Java e veja outros métodos úteis da classe `System`.

Java.lang.Object

Sempre quando declaramos uma classe, ela é obrigada a herdar de outra. Isto é, para toda classe que declararmos, existe uma superclasse. Porém, criamos diversas classes sem herdar de ninguém:

```
//Está implícito  
Class MinhaClasse extends Object{  
}
```

Quando o Java não encontra a palavra chave `extends`, ele considera que você está herdando da classe `Object`, que também se encontra no pacote `Java.lang`. Você pode até mesmo escrever essa herança que é a mesma coisa:

```
Class MinhaClasse extends Object{  
}
```

Todas as classes, sem exceção, herdam de `Object`, seja direta ou indiretamente, pois ela é a mãe, avó, bisavó, de qualquer classe.

Também podemos afirmar que qualquer objeto em Java é um `Object`, podendo ser referenciado como tal. Sendo assim, podemos dizer que qualquer objeto possui todos os métodos declarados em `Object`. Veremos alguns deles.

Casting de Referências

A habilidade de poder se referir a qualquer objeto como Object nos traz muitas vantagens. Podemos criar um método que recebe um Object como argumento, isto é, qualquer coisa! Melhor, podemos armazenar qualquer objeto:

```
public class GuardadorDeObjetos {  
    private Object[] arrayDeObjetos = new Object[100];  
    private int posicao = 0;  
  
    public void adicionaObjeto(Object objeto) {  
        this.arrayDeObjetos[this.posicao] = objeto;  
        this.posicao++;  
    }  
  
    public Object pegaObjeto(int indice) {  
        return this.arrayDeObjetos[indice];  
    }  
}
```

Mas, e no momento em que eu for recuperar esse objeto? Como vou acessar os métodos e atributos deste objeto? Se estamos referenciando-o como Object, não podemos acessá-lo como sendo Conta. Veja o exemplo:

```
public class TestaGuardador {  
    public static void main(String[] args) {  
        GuardadorDeObjetos guardador = new GuardadorDeObjetos();  
        ContaCorrente conta = new ContaCorrente();  
        guardador.adicionaObjeto(conta);  
  
        // ...  
        Object objeto = guardador.pegaObjeto(0);  
        System.out.println(objeto.getSaldo()); // Não compila!  
        // Object não tem o método getSaldo().  
    }  
}
```

Poderíamos então atribuir essa referência de Object para Conta? Tentemos:

```
Conta contaResgatada = objeto;
```

Java não tem garantias de que objeto é do tipo Conta! Portanto, não compila! Nem todo Object é uma Conta.

Para conseguirmos fazer essa atribuição, devemos avisar o Java que realmente queremos fazer isso, sabendo do risco que podemos estar correndo. Fazemos então, o casting de referência, parecido com o que fazíamos com os tipos primitivos:

```
Conta contaResgatada = (Conta) objeto;
```

Agora esse código compila e roda sem nenhum problema, pois em tempo de execução ele verificará se essa referência é realmente do tipo Conta. Se não fosse, uma exceção do tipo ClassCastException seria lançada.

Poderíamos fazer o mesmo com as classes Funcionário e Gerente. Tendo uma referência para um Funcionário que temos certeza de ser um Gerente, pois nem todo Funcionário é um Gerente.

O correto seria:

```
Gerente gerente = (Gerente) funcionario;
```

Misturando um pouco o código.

```
Object objeto = new Conta();
//...e depois
Gerente gerente = (Gerente) objeto;
```

Esse código compila? Roda?

Compila, mas não roda. Ele vai lançar uma exceção ClassCastException em tempo de execução. É importante diferenciar tempo de compilação de tempo de execução.

Nós garantimos ao Java que nosso objeto objeto era um Gerente com o casting, por isso compilou, mas na hora de rodar, o Java percebeu que a referência não era do tipo Gerente, era de uma Conta. Daí ele reclamou lançando um ClassCastException!

Métodos equals e toString:

toString

As classes podem reescrever o método `toString` para mostrar uma mensagem, uma String, que a represente. Podemos usar o `toString` assim:

```
Conta c = new Conta();
System.out.println(c.toString());
```

O método `toString` do `Object` retorna o nome da Classe @ um número de identificação:

Conta@34f5d74a

Mas isso não é interessante para nós. Então, podemos reescrevê-lo:

```
public class Conta {
    protected double saldo;
    private double limite;

    public String toString() {
        return "Uma conta com valor: "+this.saldo;
    }
    //...
}
```

Rode o código agora e veja o resultado!

Melhor! Se você apenas jogar na tela, nem precisa chamar o `toString`! Ele já é chamado pra você! Veja:

```
Conta c = new Conta();
c.deposita(100);
System.out.println(c);
```

Dá no mesmo! Você também pode concatenar Strings com classes:

```
Conta c = new Conta();
c.deposita(100);
System.out.println("Descrição: "+ c);
```

Equals

Outro método muito importante é o equals. Quando comparamos duas variáveis referência no Java o == verifica se as duas se referem ao mesmo objeto:

```
Conta c1 = new Conta(100);
Conta c2 = new Conta(100);
if(c1!=c2){
    System.out.println("Objetos      referenciados      são
diferentes.");
}
```

Nesse caso são mesmo diferentes. Mas e se fosse preciso comparar os atributos? Quais atributos ele deveria comparar? O Java por si só não faz isso, mas existe um método na classe Object que pode ser reescrito para criarmos esse critério de comparação. Esse método é o equals.

O equals recebe um Object como argumento e verifica se ele mesmo é igual ao Object recebido para retornar um boolean. Se você não reescrever esse método, o comportamento herdado é fazer um == com o objeto recebido como argumento.

```
public class Conta {
    protected double saldo;
    private double limite;

    public boolean equals(Object obj) {
        Conta outraConta = (Conta) obj;
        if (this.saldo == outraConta.saldo) {
            return true;
        } else {
            return false;
        }
    }
    public String toString() {
        return "Uma conta com valor: "+this.saldo;
    }
    //...
}
```

Um exemplo clássico do uso do equals é para datas. Se você criar duas datas, isto é, dois objetos diferentes, contendo 06/11/1974, ao comparar com o == receberá false, pois são referências para objetos diferentes. Seria correto, então, reescrever este método, fazendo as comparações dos atributos, e o usuário passaria a invocar equals ao invés de comparar com ==.

Você até poderia criar um método com outro nome para fazer esse tipo de comparação, mas o equals é importante porque muitas bibliotecas o chamam através de polimorfismo, como veremos mais adiante.

Regras para reescrita do método equals:

Pelo contrato definido pela classe Object devemos retornar false também no caso do objeto passado não ser de tipo compatível com a sua classe. Então, antes de fazer o casting devemos verificar isso usando a palavra instanceof, ou teríamos uma exception sendo lançada. Além disso, podemos resumir nosso equals de tal forma a não usar um if:

```
public boolean equals(Object obj) {  
    if (!(obj instanceof Conta))  
        return false;  
    Conta outraConta = (Conta) obj;  
    return this.saldo == outraConta.saldo;  
}
```

java.lang.Integer e classes wrappers (Box)

Como transformar um número em String e vice-versa?

Cuidado! Apesar de termos usado aqui o termo “transformar”, o que ocorre não é uma transformação de tipos e sim uma forma de obtermos uma String a partir de um int e vice-versa. O jeito mais simples de transformar um número numa String é concatená-lo da seguinte maneira:

```
int i = 100;  
String s = "" + i;  
System.out.println(s);  
  
double d = 1.2;  
String s2 = "" + d;  
System.out.println(s2);
```

Para formatar o número de uma maneira diferente, com vírgula e número de casas decimais devemos utilizar outras classes de ajuda (NumberFormat e Formatter).

Para transformar uma String em número, utilizamos as classes de ajuda para os tipos primitivos correspondentes. Por exemplo, para transformar a String s em um número inteiro utilizamos o método estático da classe Integer:

```
String s = "200";  
int i = Integer.parseInt(s);
```

As classes Double, Short, Long, Float etc contêm o mesmo tipo de método: parseDouble, parseFloat, etc.

Essas classes também são muito utilizadas para fazer o **wrapping** (embrulho) de tipos primitivos como objetos, pois referências e tipos primitivos são incompatíveis.

Imagine que precisamos passar um inteiro para o nosso GuardadorDeObjetos. Um inteiro não é um Object. Como fazer?

```
int i = 5;  
Integer x = new Integer(i);  
guardador.adiciona(x);
```

E, dado um integer, podemos pegar o int que está dentro dele (desembrulhá-lo):

```
int i = 5;  
Integer x = new Integer(i);  
int numeroDeVolta = x.intValue( ); //Desembrulhando
```

Autoboxing no Java 5.0

Fazer wrapping e unboxing é chato. O java 5.0 em diante traz um recurso chamado de **autoboxing**, que faz isso sozinho pra você, custando legibilidade:

```
Integer x = 5;  
int y = x;
```

No Java 1.4 esse código era inválido. A partir do Java 5.0 funciona perfeitamente. Isso não significa que tipos primitivos e referências agora são a mesma coisa. Isso é simplesmente um “açúcar sintático” (syntax sugar) para facilitar a codificação.

Agora você pode fazer todos os tipos de operações matemáticas com os wrappers, correndo o risco de tomar um NullPointerException, é claro!

Você também pode fazer um autoboxing diretamente para Object, possibilitando passar um tipo primitivo para um método que recebe Object como argumento:

```
Object o = 5;
```

java.lang.String

String é uma classe em Java. Variáveis do tipo String guardam referências a um objeto e não um valor, como acontece com os tipos primitivos.

Aliás, podemos criar uma String utilizando o new:

```
String x = new String("ffsd");  
String y = new String("ffsd");
```

Acabamos de criar dois objetos diferentes. O que acontece quando comparamos essas duas referências com ==?

```
if(x == y){  
    System.out.println("referências para o mesmo objeto");  
}  
else{  
    System.out.println("referências para objetos diferentes");  
}
```

Temos aqui, dois objetos diferentes! Então, como faríamos para verificar se o conteúdo dos objetos é o mesmo? Utilizamos o método equals, que foi reescrito pela classe String, para fazer a comparação de char em char.

```
if(x.equals(y)){  
    System.out.println("Consideramos iguais no critério de igualdade");  
}  
else{
```

```
        System.out.println("Consideramos diferentes no critério de igualdade");
    }
```

Aqui, a comparação retorna verdadeiro, pois quem implementou a classe String reescreveu o método equals onde decidiu que este seria o melhor critério de comparação. Você pode descobrir os critérios de igualdade de cada classe pela documentação.

A classe String também conta com um método chamado split, que divide a String em um array de Strings, dado determinado critério.

```
String frase = "Fluminense é campeão";
String palavras[] = frase.split(" ");
```

Assim teremos:

```
frase[0] contém a String "Fluminense";
frase[1] contém a String "é";
frase[2] contém a String "campeão";
```

Se quisermos comparar duas Strings, utilizamos o método compareTo, que recebe uma String como argumento e devolve um inteiro indicando se a String vem antes, é igual ou vem depois da String recebida. Se forem iguais é recebido 0; Se for anterior à String do argumento, devolve um inteiro negativo; Se for posterior, recebe um inteiro positivo.

Fato importante: **uma String é imutável**. O Java cria um pool de Strings para usar como cache e, se a String não fosse imutável, mudando o valor de uma String afetaria todas as Strings de outras classes que tivessem o mesmo valor.

Repare no código abaixo:

```
String palavra = "ffsd";
palavra.toUpperCase();
System.out.println(palavra);
```

Pode parecer estranho, mas ele imprime ffsd em minúsculo. Todo método que parece alterar o valor de uma String, na verdade, cria uma nova String com as mudanças solicitadas e a retorna! Tanto que esse método não é void. O código realmente útil ficaria assim:

```
String palavra = "ffsd";
String outra = palavra.toUpperCase();
System.out.println(outra);
```

Ou você pode eliminar a criação de uma variável temporária se achar conveniente:

```
String palavra = "ffsd";
palavra = palavra.toUpperCase();
System.out.println(palavra);
```

Funciona da mesma forma para todos os métodos que parecem alterar o conteúdo de uma String.

Se você ainda quiser trocar um caractere específico por outro, basta usar o replace:

```
String palavra = "linha";
palavra = palavra.replace("a","o");
System.out.println(palavra); //Vai imprimir "linho".
```

Ou ainda podemos concatenar invocações de métodos, já que uma String é devolvida a cada invocação:

```
String palavra = "linha";
palavra = palavra.toUpperCase().replace("a","o");
System.out.println(palavra); //Vai imprimir "LINHO".
```

O funcionamento do pool inteiro de Strings do Java tem uma série de detalhes. Você pode encontrar mais informações na documentação da classe String e no seu método intern().

Outros métodos interessantes da classe String:

charAt(i) → devolve o caracterer existente na posição i da String.

length → devolve o número de caracteres da String.

substring(i) → devolve a cadeia de caracteres a partir da posição i na String.

indexOf("nense") → recebe um char ou uma String e devolve o índice (posição) onde aparece pela primeira vez na String.

lastIndexOf("nense") → recebe um char ou uma String e devolve o índice (posição) onde aparece pela última vez na String.

toUpperCase() → devolve uma String em maiúscula.

toLowerCase() → devolve uma String em minúsculo.

isEmpty → devolve true se a String estiver vazia ou false caso contrário.

Alguns métodos úteis para busca são o contains e o matches.

Consulte o javadoc da classe!

java.lang.Math

Na classe Math, existe uma série de métodos estáticos que fazem operações com números como, por exemplo, arredondar(round), tirar o valor absoluto(abs), tirar a raiz(sqrt), calcular o seno(sin), entre outros.

Faça os testes abaixo e veja as saídas:

```
double d = 4.6;
long i = Math.round(d);
System.out.println(i);

int x = -4;
int y = Math.abs(x);
System.out.println(y);
```

No Java 5.0, podemos tirar proveito do import static aqui:

```
import static java.lang.Math.*;

public class Nada {
    public static void main(String[] args) {
        double d = 4.6;
        long i = round(d);
```

```
        System.out.println(i);

        int x = -4;
        int y = abs(x);
        System.out.println(y);
    }
}
```

Exercícios com Java.lang

Aqui faremos diversos testes, além de modificar a classe Conta. Você pode fazer todos estes exercícios dentro do próprio projeto banco.

- 1) Teste os exemplos desse capítulo, para ver que uma String é imutável. Por exemplo:

```
public class TestaString {
    public static void main(String[] args) {
        String s = "camisa 11";
        s.replaceAll("1", "2");
        System.out.println(s);
    }
}
```

Como fazer para ele imprimir “camisa 22”?

- 2) Como fazer para saber se uma String se encontra dentro da outra? E para tirar os espaços em branco de uma String? E para saber se uma String está vazia? E para saber quantos caracteres tem uma String?

Adquira o hábito saudável de sempre pesquisar o javadoc. Conhecer a API, aos poucos, é fundamental para que você não precise reinventar a roda!

- 3) Crie a classe TesteInteger para fazermos comparações com Integers:

```
Integer x1 = new Integer(10);
Integer x2 = new Integer(10);

if (x1 == x2) {
    System.out.println("iguais");
} else {
    System.out.println("diferentes");
}
```

E se testarmos com o equals?

O que podemos concluir? Como verificar se a classe Integer também reescreve o método `toString`?

A maioria das classes do Java que são muito utilizadas terão seus métodos `equals` e `toString` reescritos convenientemente.

Faça um teste com o método estático `parseInt`, recebendo uma String válida e uma inválida (com caracteres alfabéticos). Veja o que acontece!

- 4) Na documentação do Java, descubra de que classe é o objeto referenciado pelo atributo `out` da `System`. Repare que com o devido import, poderíamos escrever:

```
//Falta a declaração da saída
```

```
saida = System.out;  
saida.println("Olá");
```

Se você jogar esse código no Eclipse, ele vai te sugerir um quick fix e declarará a variável para você. Estudaremos esta classe mais adiante.

- 5) Crie e imprima uma referência para Conta (ContaCorrente ou ContaPoupança, no caso de sua Conta ser abstrata):

```
Conta c = new ContaCorrente();  
System.out.println(c);
```

O que acontece?

- 6) Reescreva o método `toString` da sua classe Conta fazendo com que uma mensagem mais apropriada seja devolvida. Use os recursos do Eclipse para isto.

```
abstract class Conta {  
    protected double saldo;  
    private double limite;  
    //..  
    public String toString() {  
        return "Este objeto é uma conta com saldo R$" + this.saldo;  
    }  
    //Restante da classe...  
}
```

Agora imprima novamente uma referência a Conta. O que aconteceu?

- 7) Reescreva o método `equals` da classe Conta para que duas contas com o mesmo número sejam consideradas iguais. Esboço:

```
abstract class Conta {  
    protected double saldo;  
    private double limite;  
    //..  
    private int numero;  
  
    public boolean equals(Object obj) {  
        if(obj instanceof Conta){  
            Conta outraConta = (Conta) obj;  
            return this.numero==outroConta.numero;  
        }  
        return false;  
    }  
    //resto da classe. Colocar get e set para numero, se ainda não houver.  
}
```

Você pode usar Ctrl+Espaço do Eclipse para escrever o esqueleto do método `equals`, basta digitar que e pressionar Ctrl+Espaço.

- 8) Crie uma classe TestaComparacaoContas e dentro do main compare as duas instâncias de ContaCorrente com `==` e depois com `equals`, sendo que as instâncias são diferentes mas possuem o mesmo numero de conta. Use o `setNumero`.

- 9) Agora faça com que o equals da sua classe Conta também leve em consideração a String do nome do Cliente a qual ela pertence. Se sua Conta não possuir o atributo nome do tipo String ou um atributo titular do tipo Cliente, crie. Teste se o método criado está funcionando corretamente.
- 10) Crie a classe GuardadorDeObjetos como vimos há algumas páginas atrás. Crie uma classe TestaGuardadorDeObjetos e dentro do main crie uma ContaCorrente e adicione-a num GuardadorDeObjetos. Depois, teste pegar essa referência como ContaPoupanca, usando casting. Repare na Exception que é lançada. Teste também o autoboxing do Java 5.0, passando um inteiro para o GuardadorDeObjetos.
- 11) Escreva um método que usa os métodos charAt e length de uma String para imprimir a mesma caractere a caractere, com cada caractere em uma linha diferente.
- 12) Repita o método do exercício anterior, agora imprimindo de trás para frente.
- 13) Dada uma frase, reescreva essa frase com as palavras na ordem invertida. “O Fluminense é campeão” deve retornar “campeão é Fluminense O”. Utilize o método split da String para auxiliá-lo.
- 14) (opcional) Pesquise sobre a classe StringBuilder (ou StringBuffer no Java 1.4). Ela é mutável. Por que usá-la ao invés da String? Quando usá-la? Repare que ela tem um método reverse e outros muito úteis.

- 15) (opcional) Um Double não está sendo suficiente para guardar a quantidade de casas necessárias em uma aplicação. Preciso guardar um número decimal muito grande! O que eu poderia usar? (Consulte a documentação, tente adivinhar onde você pode encontrar algo desse tipo pelos nomes dos pacotes, veja como é intuitivo).

O double também tem problemas de precisão ao fazer contas, por causa de arredondamentos da aritmética de ponto flutuante definido pela IEEE 754:
http://en.wikipedia.org/wiki/IEEE_754

Ele não deve ser usado se você precisa de muita precisão (casos que envolvam dinheiro, por exemplo).

Lembre-se: No Java existe muita coisa pronta. Seja na biblioteca padrão, seja em bibliotecas open source que você pode encontrar na internet.

- 16) Converta uma String para um número sem usar as bibliotecas do Java que já fazem isso. Isso é, uma string x = “750” deve gerar um int i = 750.

Para ajudar, saiba que um char pode ser “transformado” em int com o mesmo valor numérico fazendo:

```
char c = '3';
```

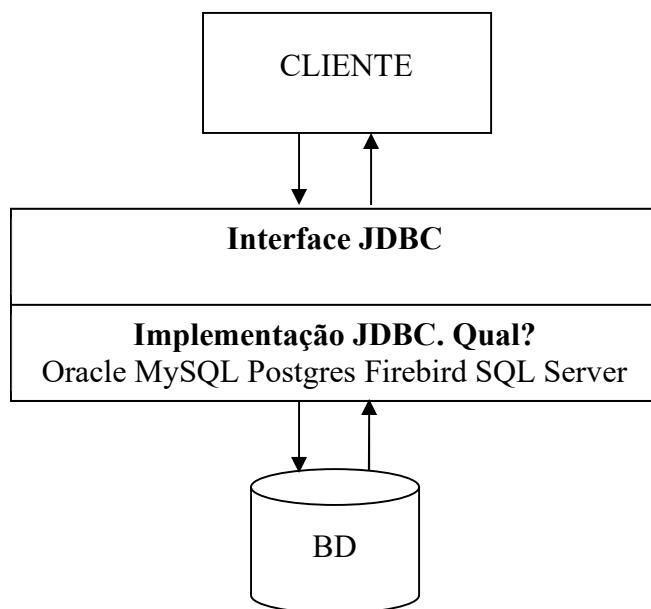
```
int i = c - '0'; // i vale 3!
```

Aqui estamos nos aproveitando do conhecimento da tabela Unicode: os números de 0 a 9 estão em sequência! Você poderia usar o método estático Character.getNumericValue(char) em vez disso.

Finalizando: O que você precisa fazer em Java? Gerar gráficos 3D? Relatórios em PDF? Gerar código de barra? Gerar boletos? Validar CPF?
Saiba que para a maioria absoluta das suas necessidades, alguém já fez uma biblioteca e a disponibilizou.

53.JDBC

JDBC é um conjunto de interfaces muito bem definidas que devem ser implementadas. Esse conjunto de interfaces, também chamada de API, fica no pacote Java.sql. Através da API JDBC podemos acessar todos os bancos de maneira igual, apesar de suas implementações serem bem diferentes uma da outra. A API JDBC facilita a vida do desenvolvedor de forma que ele não seja obrigado a conhecer as bibliotecas de cada um dos bancos de dados. O acesso é feito de forma padronizada. Veja:



Dentre as diversas interfaces do pacote Java.sql encontramos a interface Connection que define métodos para executar uma query, comitar uma transação, fechar uma conexão, etc.

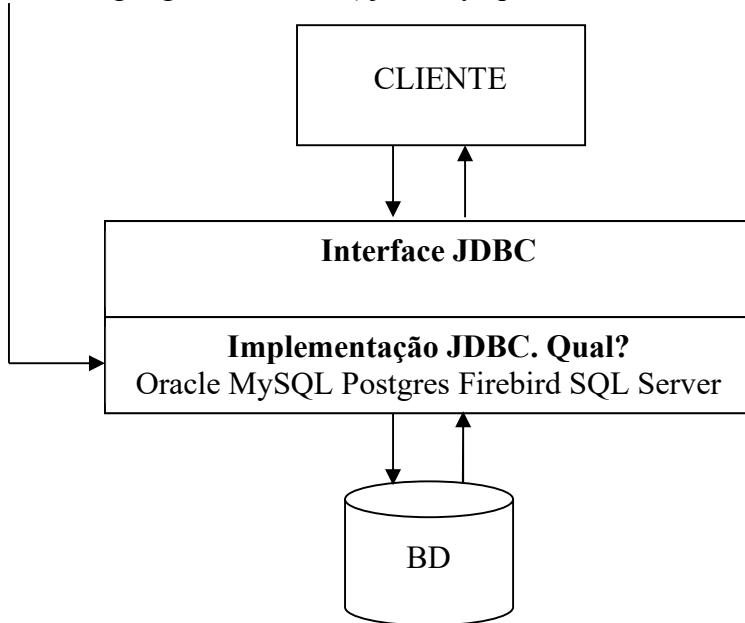
Para fazer uma conexão com o banco devemos usar o driver (conjunto de classes concretas que implementam interfaces do pacote Java.sql). Ex:

- Driver JDBC MySql;
- Driver JDBC Postgres;
- Driver JDBC Oracle;

Para obter uma conexão com o banco de dados desejado devemos usar o drive apropriado. Para isso devemos usar uma classe em Java que se comunique com os drives.

Esta classe se chama DriverManager e ela possui métodos estáticos (Não precisa ser instanciada. Você não consegue ter um objeto deste tipo). A conexão é obtida através do método getConnection. Este método recebe 3 argumentos (Um String contendo o driver JDBC, o ip do servidor e o banco, Um String contendo o usuário do banco e outra String contendo a senha do banco. Veja no exemplo:

```
DriverManager.getConnection("jdbc:mysql://localhost/teste","root","123");
```



Veja como ficaria uma classe de teste:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestaConexaoJDBC {

    public static void main(String[] args) throws SQLException {
        Connection conexao; // pacote java.sql
        conexao = DriverManager.getConnection("jdbc:mysql://localhost/test",
                                             "root", "");
        System.out.println("Conectado!");
        conexao.close();
    }
}
```

Perceba que estamos deixando passar a SQLException, que é uma checked exception lançada por muitos dos métodos da API JDBC. Numa aplicação real devemos usar try/catch onde julgarmos haver possibilidade de recuperar de uma falha com o banco.

Ao testar o código acima, poderemos receber uma exception: *No suitable driver found for jdbc:mysql://localhost/test*. Por quê?

Antes de obter a conexão, digite a seguinte linha para garantir que o Driver vai ser localizado:

```
Class.forName("com.mysql.jdbc.Driver");
```

O sistema ainda não achou uma implementação de driver JDBC que pode ser usada para abrir a conexão indicada.

O que precisamos fazer é adicionar o driver do MySQL ao classpath. O arquivo jar contendo a implementação JDBC do MySQL (mysql connector) precisa ser colocado em um lugar visível pelo seu projeto ou adicionado à variável de ambiente CLASSPATH. No Eclipse fazemos isso através de um clique da direita em nosso projeto, Properties/Java Build Path e em Libraries adicionamos o jar do driver JDBC do MySQL. Faremos isso passo a passo nos exercícios.

Onde encontrar os drivers?

- <http://developers.sun.com/product/jdbc/drivers>
- <http://www.mysql.org> (No caso do MySQL)

❖ 3 – O design pattern factory

Às vezes queremos controlar um processo muito repetitivo e trabalhoso, como obter uma conexão com o banco. Veja o exemplo da classe a seguir que seria responsável por obter uma conexão com o banco de dados:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {
    // Este método deverá retornar um objeto do tipo Connection ou uma Exceção
    public Connection getConexao() {
        try {
            System.out.println("Conectando com o banco....");
            return DriverManager.getConnection("jdbc:mysql://localhost/test",
                "root", "");
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Poderíamos fazer com que os programadores da nossa aplicação fossem notificados a obter uma conexão:

```
Connection conexão = new ConnectionFactory().getConexao();
```

Perceba que o método getConexao() é uma fábrica de conexões. Ele fabrica conexões para nós, não importando de onde vieram. Encapsulando a fabricação de uma conexão, desta forma, mais tarde, se precisarmos fazer alguma mudança, como o usuário ou senha do banco, só precisamos fazê-lo em um único lugar: na classe ConnectionFactory! Até mesmo se precisarmos mudar o banco de dados utilizado! Perceberam a vantagem?

Tratamento de exceções:

Ao fazermos um try/catch em SQLException e relançando-a como uma RuntimeException evitamos que seu código, que chamará a fábrica de conexões, fique acoplado com a API JDBC. Sempre que precisarmos lidar com uma SQLException, vamos relançá-la como RuntimeException.

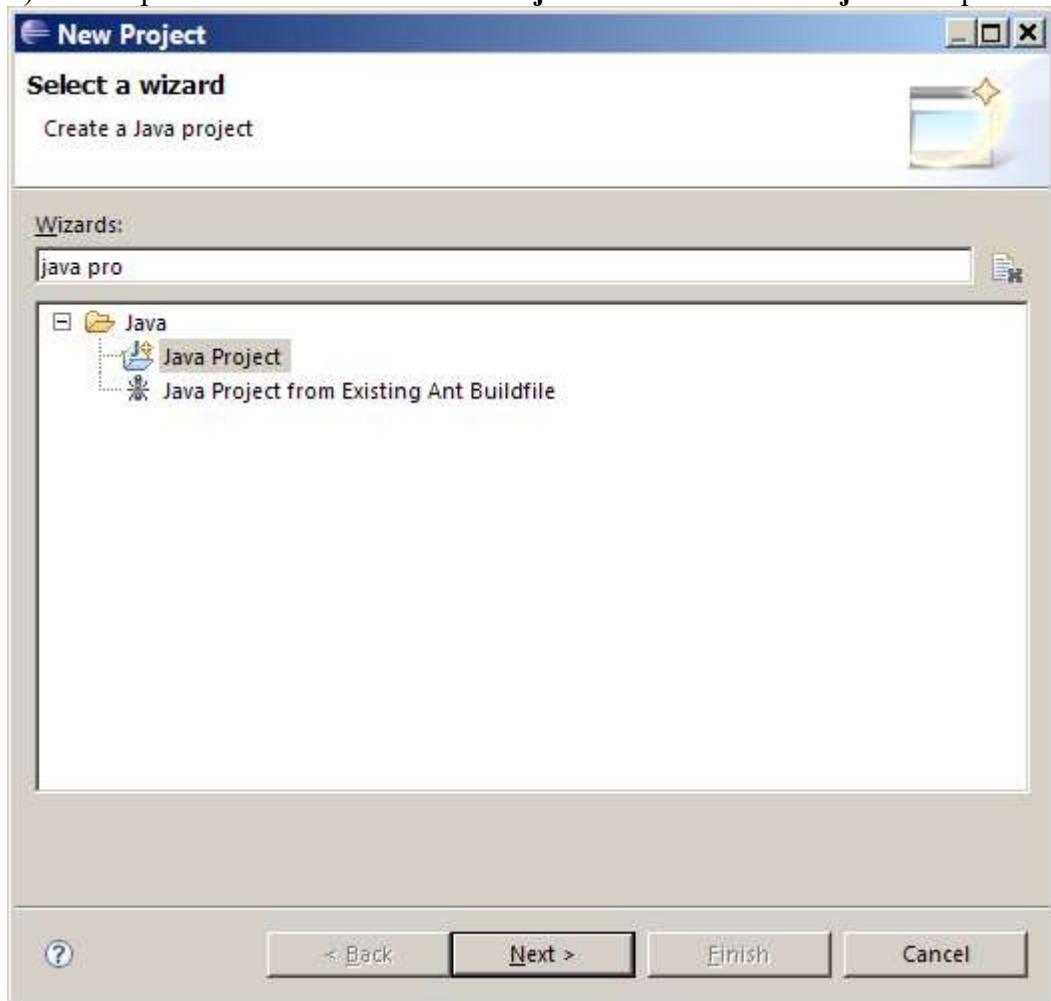
Também poderíamos criar nossa própria exceção que indica que ocorreu um erro dentro da nossa Factory, algo chamado ConnectionFactoryException.

Factory → Design Pattern que prega o encapsulamento da construção (fabricação) de objetos complexos e processos repetitivos.

Pergunte a mim sobre Design Patterns.

❖ **5 – Exercícios sobre ConnectionFactory, JDBC, etc:**

1) No Eclipse vá em **File → New → Project** selecione **Java Project** e clique em **next**:

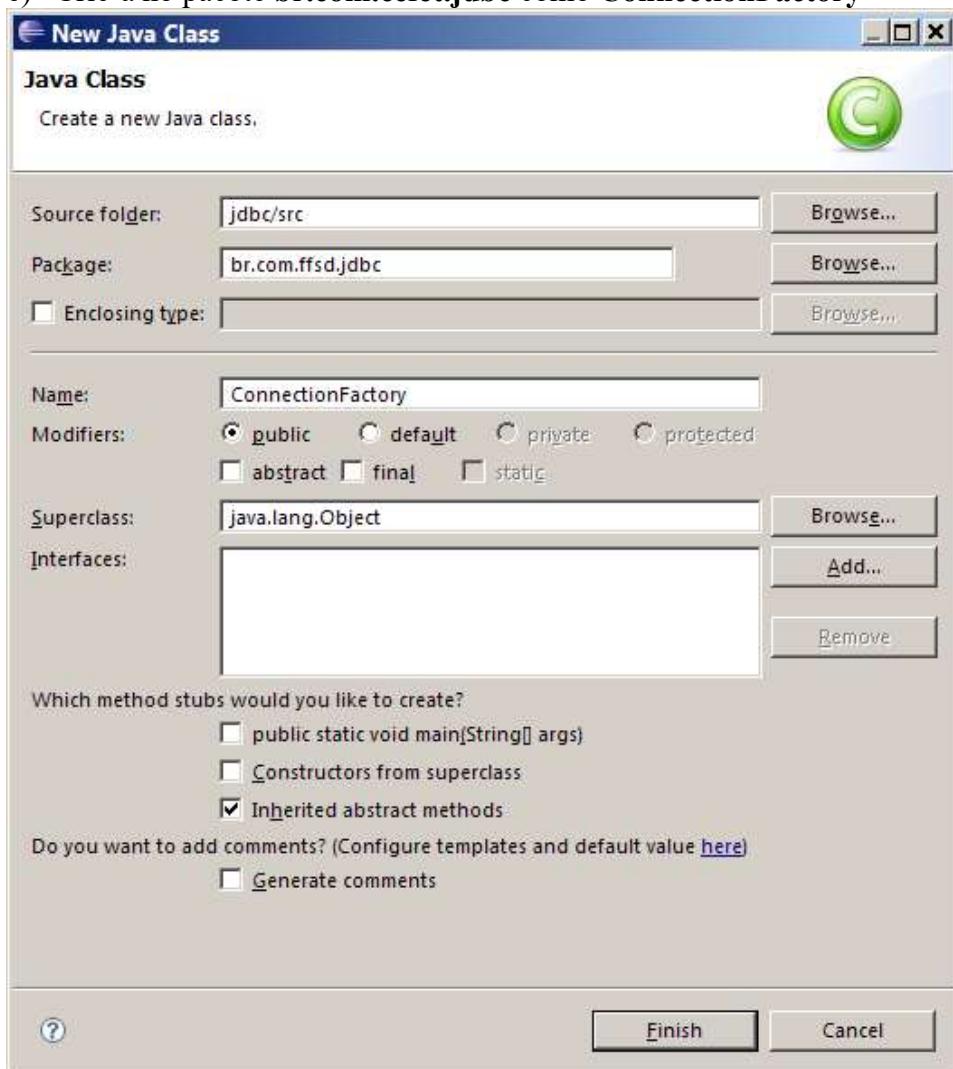


Coloque o nome do projeto como **jdbc** e clique em **finish**.
Aceite a mudança de perspectiva:



2) Copie o driver do Mysql para o seu projeto:

- Em nosso material, na pasta arquivos\, copie o arquivo mysql-connector-java-5.1.7-bin.jar.
- Entre no seu diretório workspace\jdbc e cole este arquivo lá.
- No Eclipse clique com o botão direito do mouse sobre o projeto e atualize-o.
- Ainda no Projeto, vamos criar a fábrica de conexões. Clique em **File → New → Class**:
- Crie-a no pacote **br.com.cefet.jdbc** como **ConnectionFactory**



- f) No código, crie o método getConnection() que obtém uma nova conexão. Quando perguntado, importe as classes do pacote java.sql (cuidado para não importar de com.mysql).

```
public class ConnectionFactory {  
    // Este método deverá retornar um objeto do tipo Connection ou uma Exceção  
    public Connection getConexao() {  
        try {  
            System.out.println("Conectando com o banco....");  
            return DriverManager.getConnection("jdbc:mysql://localhost/test",  
                "root", "");  
        } catch (SQLException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

3) Crie uma classe chamada TestaConexao no pacote br.com.ffsd.jdbc.teste. Todas as nossas classes teste deverão ficar neste pacote.

- a) Coloque o método main:

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

- b) Fabrique uma conexão e imprima uma mensagem confirmando que conseguiu:

```
Connection connection = new ConnectionFactory().getConexao();  
System.out.println("Conectando com o banco...");
```

- c) Feche a conexão:

```
connection.close();
```

- d) Trate os erros com throws. (Use Ctrl + 1 e escolha “add throws declaration”).

4) Crie a base de dados teste no mysql.

5) Rode sua classe TestaConexao:

- a) Clique com o botão da direita sobre a classe TestaConexao

- b) Escolha **Run as → Java Application**

6) A aplicação não funciona? O driver não foi encontrado? Esquecemos de colocar o JAR no classpath! (Build Path no Eclipse)

- a) Caso tenha esquecido nos passos anteriores, clique no seu projeto e pressione F5 para executar um refresh,

- b) Clique com o botão direito do mouse sobre o projeto e escolha **Build Path → Add to Build Path**. Em seguida selecione o seu **driver MySQL**.

- c) Rode novamente a aplicação. Desta vez deverá funcionar!

Criando a tabela Contatos e seu Javabean:

A tabela contatos será usada nos próximos exemplos. Veja sua estrutura:

```
create table contatos(
```

```
id BIGINT NOT NULL AUTO_INCREMENT,  
nome VARCHAR(80),  
email VARCHAR(80),  
endereço VARCHAR(80),  
dataNascimento DATE,  
primary key(id)  
);
```

A seguir, você verá um exemplo do Javabean que seria equivalente ao nosso modelo de entidade do banco de dados:

```
import java.util.Calendar;  
  
public class Contato {  
    private long Id;  
    private String nome, email, endereco;  
    private Calendar dataNascimento;  
  
    //métodos get e set....
```

Javabeans

Há uma confusão entre Javabeans e Enterprise Java Beans (EJB).

Javabeans são classes que possuem o construtor sem argumentos e métodos de acesso do tipo `get` e `set`! Mais nada! Simples, não? Já os EJBs costumam ser javabeans com características mais avançadas que veremos mais adiante.

Podemos usar beans por diversos motivos, normalmente as classes de modelo da nossa aplicação costumam ser javabeans.

❖ 4 – Inserindo dados em uma tabela

Para guardar uma informação numa tabela precisamos escrever o nosso comando SQL que certamente será um `INSERT`:

```
String sql= "insert into contatos(nome,email,endereco,dataNascimento) " +  
           "values('"+ nome + "','"+ email + "','" + endereço + "','" + dataNascimento + "')";
```

O código acima apresenta 3 problemas:

- 1) Um outro programador, ao bater os olhos não entenderá o código.
- 2) Difícil saber se faltou uma vírgula, um parêntese, uma aspas. Existem tabelas com muito mais campos, o que torna a missão ainda mais difícil.
- 3) SQL Injection → Se o contato a ser adicionado possuir, em seu nome, uma aspas simples, o SQL quebra todo e para de funcionar. Pior, o usuário final pode colocar uma aspas simples no código para quebrar o SQL e executar aquilo que ele deseja (SQL injection). Tudo porque escolhemos aquela linha de código e não fizemos o escape de caracteres especiais.

Também temos problemas com a data:

A data tem que ser passada em um formato que o banco de dados entenda e como uma String. Portanto, se possui um objeto Java.util.Calendar que é o nosso caso, precisará converter para String.

Por estes 4 motivos, não usaremos esse código SQL. Usaremos algo mais genérico:

```
String sql = "insert into  
contatos(nome,email,endereco,dataNascimento) values(?, ?, ?, ?);
```

As interrogações serão os parâmetros do nosso SQL que será executado, chamado de statement.

As cláusulas serão executadas através da **interface PreparedStatement**. Para receber um PreparedStatement relativo à conexão, basta chamar o método `prepareStatement` da interface **Connection**, passando como argumento, o código SQL.

```
PreparedStatement stmt = conexao.prepareStatement(sql);
```

Em seguida, chamamos o método `setString` do PreparedSatatement para preencher os valores que são do tipo String, passando a posição da interrogação no SQL e o valor a ser colocado naquela posição. A contagem começa de 1.

```
stmt.setString(1, c.getNome());  
stmt.setString(2, c.getEmail());  
stmt.setString(3, c.getEndereco());
```

Precisamos definir também a data de nascimento do nosso contato. Para isso precisaremos de um objeto do tipo Java.sql.Date para passarmos para o nosso PreparedStatement. Neste exemplo passaremos a data atual. Para isso vamos passar um long que representa os milisegundos da data atual para dentro de um Java.sql.Date que é o tipo suportado pela API JDBC. Para obtermos estes milisegundos usaremos a classe Calendar.

```
java.sql.Date dataParaGravar = new  
Date(c.getDataNascimento().getTimeInMillis());  
stmt.setDate(4, dataParaGravar);
```

Ao final, para executar o código, basta uma chamada ao método `execute()` do PreparedStatement:

```
stmt.execute();
```

Imagine todo este processo sendo escrito a cada vez que quer inserir algo no banco. Ainda não consegue perceber o quanto destrutivo isso pode ser?

Veja no exemplo abaixo como ficaria o código que obtém uma conexão e insere um contato no banco:

```
@import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Calendar;

public class TestaJDBCInsere {
    public static void main(String[] args) {
        Contato c = new Contato();
        c.setNome("Fulano");
        c.setEmail("fulano@gmail.com");
        c.setEndereco("Rua das Flores 23");
        // Obtendo a data atual por enquanto
        c.setDataNascimento(Calendar.getInstance());

        // Obtendo uma conexão com o banco
        Connection conexao = new ConnectionFactory().getConexao();
        try {
            // Comando SQL para inserção no bd
            String sql = "INSERT INTO contatos(nome, email, endereco, dataNascimento) VALUES(?, ?, ?, ?)";
            // Instrução Preparada
            PreparedStatement stmt = conexao.prepareStatement(sql);
            // Passando os argumentos
            stmt.setString(1, c.getNome());
            stmt.setString(2, c.getEmail());
            stmt.setString(3, c.getEndereco());

            // transformando o objeto Calendar em java.sql.Date
            java.sql.Date dataParaGravar;
            dataParaGravar = new java.sql.Date(c.getDataNascimento()
                .getTimeInMillis());
            // passando o parâmetro referente a data
            stmt.setDate(4, dataParaGravar);
            // Executando a instrução
            stmt.execute();
            // Fechando a Instrução Preparada (PreparedStatement)
            stmt.close();
            System.out.println("Contato " + c.getNome()
                + " incluído com sucesso!");
            // fechando a conexão
            conexao.close();
        } catch (SQLException e) {
            System.out.println("Não foi possível inserir." + e.getMessage());
        }
    }
}
```

Não é comum utilizar JDBC diretamente hoje em dia. O mais indicado é o uso de alguma API de ORM como Hibernate e JPA. Contudo, aqueles que ainda utilizam o JDBC diretamente devem ter atenção no momento de fechar a conexão.

O exemplo acima não fecha a conexão caso algum erro de banco ocorra. O comum é fechar uma conexão com o Finally.

Criando sua própria biblioteca para converter data

Diante de tantos problemas, resolvi criar minha própria biblioteca de conversão de datas. Escrevi uma classe com métodos de conversão para `java.sql.Date`, `java.util.Date`, `Calendar` e `String`, obviamente recebendo argumentos de todos os tipos. Veja:

```
package br.com.cefet.modelo;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;

public class TratamentoDeDatas {

    //Obtendo um java.sql.Date
    public java.sql.Date calendarParaJavaSqlDate(Calendar cal){
        java.sql.Date dataSql= new java.sql.Date(cal.getTimeInMillis());
        return dataSql;
    }

    public java.sql.Date StringParaJavaSqlDate(String dataTexto) throws ParseException{
        java.util.Date data= new SimpleDateFormat("dd/MM/yyyy").parse(dataTexto);
        Calendar dataEmCalendar = Calendar.getInstance();
        dataEmCalendar.setTime(data);
        java.sql.Date dataSql = new java.sql.Date(dataEmCalendar.getTimeInMillis());
        return dataSql;
    }

    public java.sql.Date javaUtilDateParaJavaSqlDate(java.util.Date dataUtil) throws ParseException{
        Calendar dataEmCalendar = Calendar.getInstance();
        dataEmCalendar.setTime(dataUtil);
        java.sql.Date dataSql = new java.sql.Date(dataEmCalendar.getTimeInMillis());
        return dataSql;
    }

    //Obtendo um java.util.Date
    public java.util.Date StringParaJavaUtilDate(String dataTexto) throws ParseException{
        java.util.Date data= new SimpleDateFormat("dd/MM/yyyy").parse(dataTexto);
        return data;
    }

    public java.util.Date calendarParaJavaUtilDate(Calendar cal) throws ParseException{
        java.util.Date data= cal.getTime();
        return data;
    }

    public java.util.Date javaSqlDateParaJavaUtilDate(java.sql.Date dataSql) throws ParseException{
        java.util.Date data= new java.util.Date(dataSql.getTime());
        return data;
    }

    //Obtendo um Calendar
    public java.util.Calendar StringParaCalendar(String dataTexto) throws ParseException{
        java.util.Date data= new SimpleDateFormat("dd/MM/yyyy").parse(dataTexto);
        Calendar dataEmCalendar = Calendar.getInstance();
        dataEmCalendar.setTime(data);
        return dataEmCalendar;
    }

    public java.util.Calendar javaUtilDateParaCalendar(java.util.Date dataUtil) throws ParseException{
        Calendar dataEmCalendar=Calendar.getInstance();
        dataEmCalendar.setTime(dataUtil);
        return dataEmCalendar;
    }

    //Obtendo um String
    public String calendarParaString(Calendar cal) throws ParseException{
        SimpleDateFormat sdf= new SimpleDateFormat("dd/MM/yyyy");
        String dataEmTexto = sdf.format(cal.getTime());
        return dataEmTexto;
    }

    public String javaUtilDateParaString(java.util.Date dataUtil) throws ParseException{
        SimpleDateFormat sdf= new SimpleDateFormat("dd/MM/yyyy");
        String dataEmTexto = sdf.format(dataUtil);
        return dataEmTexto;
    }

    public String javaSqlDateParaString(java.sql.Date dataSql) throws ParseException{
        SimpleDateFormat sdf= new SimpleDateFormat("dd/MM/yyyy");
        String dataEmTexto = sdf.format(dataSql.getTime());
        return dataEmTexto;
    }
}
```

Fechando a conexão e implementando o controle de transações

O exemplo dado acima não fecha a conexão caso algum erro ocorra no momento de inserir um dado no banco de dados. O comum é fechar a conexão em um bloco finally.

O bloco finally é muito útil para liberar recursos caros como uma conexão por exemplo.

Controle de transações

A conexão trabalha confirmando automaticamente cada instrução enviada ao SGBD. Para o exemplo abaixo não há problemas já que estamos enviando apenas uma instrução ao banco.

Contudo, para já nos acostumarmos a trabalhar da maneira correta, através da chamada do método setAutoCommit(false), informamos que a instrução (ou as instruções) agora dependem de um commit() para confirmar a transação ou de um rollback() para voltar o banco ao estado original caso aconteça algum imprevisto.

Perceba que logo após o execute() confirmamos a transação e dentro do catch estamos cancelando a mesma caso aconteça algum erro. Lembrando que isso é muito útil para instruções aninhadas dentro de um mesmo bloco.

```
// Obtendo uma conexão com o banco
Connection conexao = new ConnectionFactory().getConexao();
PreparedStatement stmt = null; //Declarando uma instrução preparada
try {
    conexao.setAutoCommit(false); //Impedindo o banco de confirmar automaticamente cada instrução
    // Comando SQL para inserção no bd
    String sql = "INSERT INTO contatos(nome, email, endereco, dataNascimento) VALUES(?, ?, ?, ?)";
    stmt = conexao.prepareStatement(sql); //Preparando a instrução
    // Passando os argumentos
    stmt.setString(1, c.getNome());
    stmt.setString(2, c.getEmail());
    stmt.setString(3, c.getEndereco());

    // transformando o objeto Calendar em java.sql.Date
    java.sql.Date dataParaGravar;
    dataParaGravar = new java.sql.Date(c.getDataNascimento()
        .getTimeInMillis());
    // passando o parâmetro referente a data
    stmt.setDate(4, dataParaGravar);
    // Executando a instrução
    stmt.execute();
    // Fechando a Instrução Preparada (PreparedStatement)
    conexao.commit(); //Confirmando a transação
    System.out.println("Contato " + c.getNome()
        + " incluido com sucesso!");
} catch (SQLException e) {
    conexao.rollback(); //Cancelando a transação e voltando o banco de dados ao estado original
    System.out.println("Não foi possível inserir." + e.getMessage());
} finally{
    // Permitindo ao banco confirmar automaticamente cada instrução novamente
    conexao.setAutoCommit(true);
    conexao.close(); // fechando a conexão
    stmt.close(); //Fechando a instrução
}
```

Assim, mesmo que o código dentro do try lance exception, o conexao.close() será executado. Garantimos que não deixaremos uma conexão pendurada sem uso. Esse código pode ficar muito maior se quisermos ir além.

Afinal, o que acontece no caso de con.close lançar uma exception? Quem a tratará?

Exercícios com o Professor:

- 1) No Eclipse, criar o projeto (Java Project) projeto-contatos;

2) Criar a base de dados agenda e dentro dela a tabela contatos (Id (chave primária auto-incrementável), nome (varchar(80), email (varchar(80), telefone (varchar(13), dataNascimento (Date));

3) Criar os pacotes br.com.cefet, br.com.cefet.modelo, br.com.cefet.modelo.factory e br.com.cefet.programa.

4) Usando o design pattern Factory, criar (no pacote br.com.cefet.modelo.factory) uma classe ConnectionFactory (fábrica de conexões) que deverá ser capaz de retornar uma conexão com a base de dados agenda do Mysql.

5) No Eclipse (pacote br.com.cefet.modelo), criar o javabean da tabela acima.

6) Criar dentro do pacote br.com.cefet.programa uma classe chamada ProgramaControle onde você vai instanciar um contato, atribuir valores para seus atributos e em seguida fazer uma inserção na tabela contatos da base agenda.

7) Criar o pacote br.com.cefet.exception e dentro dele uma ConnectionFactoryException cujo tratamento deverá ser obrigatório.

8) Fazer com que a classe ConnectionFactory lance uma exceção do tipo que acabamos de criar. Fazer as devidas modificações na classe ProgramaControle.

Exercícios (para tentar fazer sem auxílio):

1) No Eclipse, criar o projeto controle-estoque;

2) Criar a base de dados controle e dentro dela as tabelas categorias (IdCategoria, DescricaoCategoria), fornecedores(IdFornecedor, NomeFornecedor, TelFornecedor) e produtos(IdProduto, IdCategoria, IdFornecedor, NomeProduto, Estoque, PrecoCusto).

3) Criar os pacotes br.com.cefet, br.com.cefet.modelo, br.com.cefet.modelo.factory e br.com.cefet.programa.

4) Usando o design pattern Factory, criar (no pacote br.com.cefet.modelo.factory) uma classe ConnectionFactory (fábrica de conexões) que deverá ser capaz de retornar uma conexão com a base de dados controle do Mysql.

5) No Eclipse (pacote br.com.cefet.modelo), criar o javabean de cada uma das 3 tabelas, não se esquecendo de implementar as agregações.

6) Criar dentro do pacote br.com.cefet.programa uma classe chamada ProgramaControle onde você vai instanciar um produto, atribuir valores para seus atributos e em seguida fazer uma inserção nas devidas tabelas da base controle.

7) Criar o pacote br.com.cefet.exception e dentro dele uma ConnectionFactoryException cujo tratamento deverá ser obrigatório.

8)Fazer com que a classe ConnectionFactory lance uma exceção do tipo que acabamos de criar. Fazer as devidas modificações na classe ProgramaControle.

❖ 5 – DAO – Data Access Object

A idéia do DAO é remover todo o código de acesso a Banco de Dados de suas classes de lógica e colocá-lo em uma única classe responsável pelo acesso a dados. Separando o código de acesso ao banco de suas classes de lógica, a manutenção fica facilitada.

Agora a idéia é poder ter um código que adicione um Contato ao banco. Veja o código a seguir:

```
//Adiciona os dados no banco
ClasseDao dao = new ClasseDao();
dao.adiciona("meu nome","meu email","meu endereço",meuCalendar);
```

Repare que todos os parâmetros que estamos passando são as informações do Contato. Se o Contato tivesse 30 atributos, passaríamos 30 parâmetros? Java é orientado a Strings? Agora vamos tentar um código mais simples e funcional:

```
//Adiciona um contato no banco
ClasseDao dao = new ClasseDao();
//Método muito mais elegante
dao.adiciona(contato);
```

Vamos tentar chegar ao código anterior. O mais elegante seria poder chamar um único método para inclusão, certo?

```
public class TestaInsere {
    public static void main(String[] args) {
        Contato contato = new Contato();
        contato.setNome("Fulano de tal");
        contato.setEmail("fulano@gmail.com");
        contato.setEndereco("Rua Monsenhor Miranda, 86, Centro");
        contato.setDataNascimento(Calendar.getInstance());

        // Grave nesta conexão
        ClasseDAO dao = new ClasseDAO();

        // Método elegante
        dao.adiciona(contato);

        System.out.println("Contato Gravado");
    }
}
```

Repare que estamos isolando todo o acesso a banco de dados em classes bem simples, cuja instância é um objeto responsável por acessar os dados. Desta responsabilidade surgiu o termo Data Access Object ou simplesmente DAO, um dos mais famosos padrões de projeto (design pattern).

O que falta agora é criarmos a classe ClasseDAO, ou melhor, chamaremos de ContatoDAO, que possui um método adiciona. Vamos criar uma que se conecta ao banco ao criarmos uma instância dela:

```
public class ContatoDAO {  
    private Connection conexao;  
  
    public ContatoDAO() {  
        this.conexao = new ConnectionFactory().getConnection();  
    }  
}
```

Agora que todo ContatoDAO possui uma conexão com o banco, podemos focar no método adiciona, que recebe um Contato como argumento e é responsável por adicioná-lo através de código SQL:

```
public void adiciona(Contato c) {  
    //Testando se preciso fabricar uma conexão  
    try {  
        if(conexao.isClosed())  
            conexao=new ConnectionFactory().getConexao();  
    } catch (SQLException e1) {  
        System.out.println("Erro ao obter conexão."+e1.getMessage());  
    }  
    //declarar minha instrução sql  
    this.comandoSQL="INSERT INTO contatos(nome,email,endereco,dataNascimento) VALUES(?, ?, ?, ?)";  
    //Preparando minha instrução através do método prepareStatement da minha conexão  
    try {  
        this.stmt=this.conexao.prepareStatement(comandoSQL);  
        //passando os argumentos  
        stmt.setString(1, c.getNome());  
        stmt.setString(2, c.getEmail());  
        stmt.setString(3, c.getEndereco());  
        //Convertendo a data para gravar  
        java.sql.Date dataParaGravar = new java.sql.Date(c.getDataNascimento().getTimeInMillis());  
        stmt.setDate(4, dataParaGravar);  
        //executa a instrução preparada  
        stmt.execute();  
        System.out.println("Contato "+c.getNome()+" adicionado com sucesso!"); //ferindo o SRP  
    } catch (SQLException e) {  
        System.out.println("Erro ao adicionar"+e.getMessage());  
    } finally{  
        //liberando os recursos  
        try {  
            stmt.close();  
            conexao.close();  
            System.out.println(conexao);  
        } catch (SQLException e) {  
            System.out.println("Não foi possível liberar os recursos"+e.getMessage());  
        }  
    }  
}
```

Perceba que tratamos a SQLException respeitando o SRP (princípio da responsabilidade única).

Exercícios com Javabeans e DAO

1) Crie a classe Contato:

a) No pacote br.com.cefat.modelo, crie uma classe chamada Contato.

```
import java.util.Calendar;  
  
public class Contato {  
    private long Id;  
    private String nome, email, endereco;  
    private Calendar dataNascimento;  
  
    //métodos get e set....
```

- b) Aperte Ctrl + 3 e digite ggas (abreviação de Generate Getters and Setters)
e selecione todos os getters e setters.
- 2) Crie a classe ContatoDAO no pacote br.com.cefat.modelo.dao (utilize CTRL + SHIFT + O para os imports)

```
package br.com.cefat.modelo.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import br.com.cefat.modelo.ConnectionFactory;
import br.com.cefat.modelo.Contato;

public class ContatoDAO {

    //Atributo privado para guardar a conexão (interface Connection)
    private Connection conexao;
    //Instrução preparada (interface PreparedStatement)
    private PreparedStatement stmt;
    //Comando SQL a ser executado pelo PreparedStatement
    private String sql;

    /*Construtor que obtém uma conexão com o bd através da
    ConnectionFactory */
    public ContatoDAO() {
        conexao = new ConnectionFactory().getConexao();
    }

    public void adiciona(Contato c) {
        //Testando se preciso fabricar uma conexão
        try {
            if(conexao.isClosed())
                conexao=new ConnectionFactory().getConexao();
        } catch (SQLException e1) {
            System.out.println("Erro ao obter conexão."+e1.getMessage());
        }
        //declarar minha instrução sql
        this.comandoSQL="INSERT INTO contatos(nome,email,endereco,dataNascimento) VALUES(?, ?, ?, ?)";
        //Preparando minha instrução através do método prepareStatement da minha conexão
        try {
            this.stmt=this.conexao.prepareStatement(comandoSQL);
            //passando os argumentos
            stmt.setString(1, c.getNome());
            stmt.setString(2, c.getEmail());
            stmt.setString(3, c.getEndereco());
            //Convertendo a data para gravar
            java.sql.Date dataParaGravar = new java.sql.Date(c.getDataNascimento().getTimeInMillis());
            stmt.setDate(4, dataParaGravar);
            //executa a instrução preparada
            stmt.execute();
            System.out.println("Contato "+c.getNome()+" adicionado com sucesso!"); //ferindo o SRP
        } catch (SQLException e) {
            System.out.println("Erro ao adicionar"+e.getMessage());
        } finally{
            //liberando os recursos
            try {
                stmt.close();
                conexao.close();
                System.out.println(conexao);
            } catch (SQLException e) {
                System.out.println("Não foi possível liberar os recursos"+e.getMessage());
            }
        }
    }
}
```

Apesar de termos tratado todos os erros de SQL no DAO, o código acima ainda fere o SRP (Princípio da Responsabilidade Única). Por que???

Lembre-se de importar as classes SQL do pacote Java.sql, **inclusive a classe Date!**

- 3) Crie, no pacote br.com.cefat.teste, uma classe chamada TestaInsereDAO com um método main:

```
public class TestaInsereDAO {  
    public static void main(String[] args) {  
        //Instanciando um contato e preenchendo seus atributos  
        Contato contato= new Contato();  
        contato.setNome("Fulano");  
        contato.setEmail("fulano@ffsd.br");  
        contato.setEndereco("Rua Piauí, 25");  
        //opcional  
        Calendar dataNascimento = Calendar.getInstance();  
        try {  
            java.util.Date data = new SimpleDateFormat("dd/MM/yyyy").parse("05/10/1992");  
            dataNascimento.setTime(data);  
            contato.setDataNascimento(dataNascimento);  
        } catch (ParseException e) {  
            System.out.println("Erro ao converter data");  
        }  
        //Instanciando um dao  
        ContatoDAO dao = new ContatoDAO();  
        //Adicionar o contato ao banco de dados  
        dao.adiciona(contato);  
    }  
}
```

- 4) Teste o programa
- 5) Verifique, no banco, se o seu Contato foi adicionado.

❖ 6 – Pesquisando no Banco de Dados

Para pesquisar também utilizamos a interface PreparedStatement. A diferença é que usaremos método executeQuery que retorna todos os contatos de uma determinada Query.

O objeto retornado é uma implementação da interface ResultSet que permite navegar pelos registros através do método next. Esse método irá retornar false quando chegar ao fim da pesquisa, portanto ele é normalmente utilizado para fazer um loop nos registros como no exemplo a seguir:

```
// pega a conexão  
Connection conexao = new ConnectionFactory().getConexao();  
// pega o Statement  
PreparedStatement stmt = conexao.prepareStatement("select * from contatos");  
  
// executa um select  
ResultSet rs = stmt.executeQuery();  
  
// itera no ResultSet  
while (rs.next()) {  
    ;  
}  
//Fecha o ResultSet  
rs.close();  
//Fecha o Statement  
stmt.close();
```

Chamando um dos métodos get do ResultSet (o mais comum é o getString), podemos obter o valor de uma coluna no banco de dados.

```
// pega a conexão
Connection conexao = new ConnectionFactory().getConexao();
// pega o Statement
PreparedStatement stmt = conexao.prepareStatement("select * from contatos");

// executa um select
ResultSet rs = stmt.executeQuery();

// itera no ResultSet
while (rs.next()) {
    System.out.println(rs.getString("nome") + " :: " + rs.getString("email"));
}
//Fecha o ResultSet
rs.close();
//Fecha o Statement
stmt.close();
```

Assim como o cursor do banco de dados, só é possível mover para frente. Para permitir um processo de leitura para trás é necessário especificar na abertura do ResultSet que tal cursor deve ser utilizado.

Mais uma vez podemos aplicar as idéias de DAO e criar um método `getLista()` no nosso `ContatoDAO`. Mas o que esse método retornaria? Um `ResultSet`? E teríamos o código de manipulação de `ResultSet` espalhado por todo o código? Vamos fazer nosso `getLista()` devolver algo mais interessante, uma lista de `Contato`:

```
List<Contato> contatos = new ArrayList<Contato>();
String sql = "select * from contatos";
PreparedStatement s;
s = conexao.prepareStatement(sql);
ResultSet r = s.executeQuery();
while (r.next()) {
    Contato c = new Contato();
    c.setNome(r.getString("nome"));
    c.setEmail(r.getString("email"));
    c.setEndereco(r.getString("endereco"));
    Calendar data = Calendar.getInstance();
    data.setTime(r.getDate("dataNascimento"));
    c.setDataNascimento(data);

    contatos.add(c);
}
r.close();
s.close();
return contatos;
```

Exercícios: Listagem

- 1) Crie o método `getLista` na classe `ContatoDAO`. Import o `List` do `Java.util`:

```
public List<Contato> getLista() {
    try {
        if (conexao.isClosed())
            conexao = new ConnectionFactory().getConexao();
    } catch (SQLException e1) {
        System.out.println("Erro ao obter conexao." + e1.getMessage());
    }
    this.comandoSQL = "SELECT * FROM contatos";
    List<Contato> contatos = new ArrayList<Contato>();
    try {
        this.stmt = conexao.prepareStatement(comandoSQL);
        // Obtenho os dados e guardo em um ResultSet
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Contato c = new Contato();
            c.setNome(rs.getString("nome"));
            c.setEmail(rs.getString("email"));
            c.setEndereco(rs.getString("endereco"));
            c.setId(rs.getInt("id"));
            Calendar dataEmCalendar = Calendar.getInstance();
            dataEmCalendar.setTime(rs.getDate("dataNascimento"));
            c.setDataNascimento(dataEmCalendar);
            contatos.add(c);
        }
        return contatos;
    } catch (SQLException e) {
        System.out.println("Não foi possivel listar" + e.getMessage());
        return null;
    } finally {
        try {
            stmt.close();
            conexao.close();
        } catch (SQLException e) {
            System.out.println("Não foi possível liberar os recursos. "
                + e.getMessage());
        }
    }
}
```

A saber:

List → Interface para coleções do pacote java.util

ArrayList e Vector → Classes (também do pacote Java.util) que implementam a interface List.

<Contato> → Generics (veremos mais adiante) define um tipo de objeto específico para nossa List para que não tenhamos que fazer casting mais adiante.

- 2) Agora vamos usar o método getLista para listar todos os contatos do nosso banco de dados.
- 3) Crie uma classe chamada TestaLista com um método main:
 - a) Crie um ContatoDAO:

ContatoDAO dao = new ContatoDAO();

- b) Liste os contatos com o DAO:

List<Contato> contatos = dao.getLista();

- c) Itere nessa lista e imprima as informações dos contatos:

```
List<Contato> contatos = dao.getLista();
for (Contato c : contatos) {
    System.out.println();
    System.out.println("Id: "+c.getId());
    System.out.println("Nome: "+c.getNome());
    System.out.println("E-mail: "+c.getEmail());
    System.out.println("Endereço: "+c.getEndereco());
    //formatando a data (coisa chata!!!!)
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    String dataEmTexto = sdf.format(c.getDataNascimento().getTime());
    System.out.println("Nascimento: "+dataEmTexto);
}
```

- 4) Rode o programa que acabamos de escrever. Clique em Run → Run as → Java Application.

A saber:

- O Hibernate é a ferramenta ORM que domina o mercado atualmente;
- Se um projeto não usa nenhuma das tecnologias ORM disponíveis no mercado, o mínimo a se fazer é usar o DAO.

Mais exercícios:

- 1) Pesquise sobre a classe SimpleDateFormat.
- 2) Crie uma classe DAOException que estenda de RuntimeException e utilize-a no seu ContatoDAO.
- 3) Utilize a cláusula WHERE para refinar suas pesquisas no banco de dados.
- 4) Crie o método pesquisarId, que receberá um id (inteiro) e retornará um objeto do tipo Contato.
- 5) Crie um método para fazer uma busca aproximada por nome, chamado pesquisarPorNome.

❖ 7 – Adicionando outros métodos ao DAO:

Já sabemos que o PreparedStatement pode executar qualquer tipo de código SQL e que ResultSet recebe dados retornados de sua consulta. Escrever código de diferentes métodos de uma classe típica de DAO fica fácil, porém maçante.

Veja o método altera:

```
public void altera(Contato c) {
    try {
        if (conexao.isClosed())
            conexao = new ConnectionFactory().getConexao();
    } catch (SQLException e1) {
        System.out.println("Erro ao obter conexao." + e1.getMessage());
    }
    this.comandoSQL = "UPDATE contatos SET nome=?,email=?,endereco=?,dataNascimento=? WHERE id=?";
    try {
        this.stmt = this.conexao.prepareStatement(comandoSQL);
        stmt.setString(1, c.getNome());
        stmt.setString(2, c.getEmail());
        stmt.setString(3, c.getEndereco());
        // Convertendo a data para gravar
        java.sql.Date dataParaGravar = new java.sql.Date(c
                .getDataNascimento().getTimeInMillis());
        stmt.setDate(4, dataParaGravar);
        stmt.setInt(5, c.getId());
        stmt.execute();
        System.out.println("Contato " + c.getId()
                + " alterado com sucesso!"); // ferindo o SRP
    } catch (SQLException e) {
        System.out.println("Erro ao alterar" + e.getMessage());
    } finally {
        try {
            stmt.close();
            conexao.close();
            System.out.println(conexao);
        } catch (SQLException e) {
            System.out.println("Não foi possivel liberar os recursos"
                    + e.getMessage());
        }
    }
}
```

O código para remoção baseia-se em um Contato, mas usa apenas o id dele para executar o comando delete:

```
public void remove(Contato c) {
    try {
        if (conexao.isClosed())
            conexao = new ConnectionFactory().getConexao();
    } catch (SQLException e1) {
        System.out.println("Erro ao obter conexao." + e1.getMessage());
    }
    this.comandoSQL = "DELETE FROM contatos WHERE id=?";
    try {
        this.stmt = this.conexao.prepareStatement(comandoSQL);
        stmt.setInt(1, c.getId());
        stmt.execute();
        System.out.println("Contato " + c.getId()
                + " removido com sucesso!"); // ferindo o SRP
    } catch (SQLException e) {
        System.out.println("Erro ao remover" + e.getMessage());
    } finally {
        try {
            stmt.close();
            conexao.close();
            System.out.println(conexao);
        } catch (SQLException e) {
            System.out.println("Não foi possivel liberar os recursos"
                    + e.getMessage());
        }
    }
}
```

Exercícios com DAO:

- 1) Implemente os métodos altera e remove escritos acima.

- 2) Use os métodos acima para fazer testes. Altere e remova um Contato. Verifique as modificações no banco de dados.
- 3) Crie um novo projeto (Java Project) no Eclipse, com o nome de projeto-empresa.
- 4) Crie, no pacote br.com.cefet.modelo a classe Departamento com os campos id (Long) e descrição (String).
- 5) Crie a base de dados empresa e a tabela departamentos no banco de dados.
- 6) Crie, no pacote br.com.cefet.modelo.dao, a classe DepartamentoDAO.
- 7) Crie a classe TestaDepartamentoDAO (contendo um método main) no pacote br.com.cefet.testes.
- 8) Na classe criada acima use a classe DepartamentoDAO, com os 5 métodos vistos anteriormente e/ou outros que julgar necessário, para instanciar novos departamentos e colocá-los no seu banco de dados.
- 9) Crie, no pacote br.com.cefet.modelo, a classe Funcionário com os campos id (Long), nome, login e senha (String) e departamento (Departamento).
- 10) Crie a tabela na base de dados empresa.
- 11) Crie uma classe DAO para Funcionário.
- 12) No pacote br.com.cefet.testes, crie a classe TestaFuncionarioDAO e use a classe FuncionarioDAO, com os 5 métodos vistos anteriormente e/ou outros que julgar necessário, para instanciar novos funcionários e colocá-los no seu banco de dados.

❖ 8 – O design pattern Singleton

O padrão de projeto (Design Pattern) Singleton tem entre outras funções a garantia de apenas uma instância de determinada classe, ou seja, somente um objeto por classe. Para o desenvolvimento de tal implementação existem algumas maneiras, mas a tradicional é baseada no uso de atributos estáticos e da modificação do controle de acesso do construtor, onde um membro externo depende de uma propriedade para “requisitar” uma instância.

Considerando as características temos:

- 1) Variável estática contendo a instância para a classe;
- 2) Construtor declarado **private**.
- 3) Método **estático** e, de preferência, **sincronizado** para requisitar instância.

A implementação em Java seria a seguinte:

```
public class ClasseSingleton {  
    //Atributo que vai representar nossa ÚNICA instância  
    private static ClasseSingleton instancia;  
    //Construtor default PRIVADO para evitar o acesso fora da classe  
    private ClasseSingleton() {}  
    //Método estático e sincronizado (threads) que garante o retorno  
    de apenas uma instância  
    //da nossa classe  
    public static synchronized ClasseSingleton getInstancia(){  
        if(instancia==null)  
            instancia=new ClasseSingleton();  
        return instancia;  
    }  
}
```

Singleton na Prática

O uso do padrão Singleton, normalmente está associado a economia de recursos, ou melhor, quando a resposta para seguinte pergunta é verdadeira:
Quando você precisa de uma única instância?

Por exemplo, quando múltiplas instâncias não fazem sentido, quando estas instâncias apenas desperdiçam memória ou processamento. Na pior das hipóteses, quando múltiplas instâncias causam um comportamento incorreto para o software. Desta forma também garante um ponto de acesso global para acesso a instância.

Considerando o escopo de vida do Singleton, o mesmo é iniciado na primeira requisição e finalizando apenas quando a aplicação é encerrada. O que pode considerar uma longa vida caso o mesmo seja usado somente uma vez, assim o uso do mesmo deve ter uso considerado sem esquecer que o escopo do mesmo é de todo um programa, um sistema ou mesmo um sistema distribuído.

Singleton usado para conexões com o Banco de Dados

Este padrão tem um uso comum em conexões com banco de dados, pois o mesmo proporciona uma única instância, ou seja, canal de comunicação entre a aplicação e o SGBD.

Neste caso, a execução das sintaxes será toda sobre uma única thread, o que pode ocasionar outros problemas à aplicação.

Trocando a ConnectionFactory pela classe BDSingleton

A idéia é que, a partir de agora, utilizemos apenas uma classe Singleton para cuidar da nossa comunicação com o Banco de Dados. Chamaremos esta classe de BDSingleton. Veja como ela ficará:

```
public class BDSingleton {

    private static BDSingleton instancia;
    private Connection conexao = null;

    public static synchronized BDSingleton getInstancia() {
        /*Se uma thread chamar o método getInstancia() e for interrompida
        antes de realizar a instanciação, uma outra thread poderá chamar o
        método e realizar a instanciação. Neste caso, duas instâncias serão
        construídas, o que fere os requisitos do singleton .
        O objetivo do synchronized é evitar o compartilhamento da instância
        entre 2 ou mais threads*/
        if (instancia == null)
            instancia = new BDSingleton();
        return instancia;
    }

    private BDSingleton() {
        try {
            conexao = DriverManager.getConnection(
                "jdbc:mysql://localhost/baseDeDados",
                "root", "");
        } catch (SQLException e) {
            System.out.println("Erro ao conectar com o banco.
Erro: "
                + e.getMessage());
        }
    }

    public Connection getConexao() {
        return conexao;
    }
}
```

Exercícios - Alterando nosso DAO

Agora vamos modificar nosso DAO de forma que ao ser instânciado ele receba uma conexão através da classe BDSingleton. Agora não precisaremos mais ficar preocupados em abrir e fechar conexões com o banco.

1) Altere o construtor:

Troque as linhas:

```
public ContatoDao() {
    conexao = ConnectionFactory.getConnection();
}
```

Por:

```
public ContatoDao() {
    conexao = BDSingleton.getInstancia().getConexao();
}
```

2) Elimine linhas nos métodos adiciona, altera, remove e demais:

Elimine as seguintes linhas no início de cada método:

```
if (this.conexao.isClosed())
    this.conexao
    ConnectionFactory.getConnection(); =
```

Elimine a seguinte linha no final de cada método:

```
this.conexao.close();
```

Pronto! Nossa DAO agora possui apenas uma conexão e não precisamos nos preocupar com a abertura e fechamento dela.

Instalação do JDK no Windows

Para instalar o JDK no Windows, primeiro baixe-o no site da Oracle. É um simples arquivo executável que contém o Wizard de instalação:

<http://www.oracle.com/technetwork/java/>



Instalação

1. Dê um clique duplo no arquivo jdk-<versão>-windows-i586-p.exe e espere até ele



1. Dê um clique duplo no arquivo jdk-<versão>-windows-i586-p.exe e espere até ele entrar no wizard de instalação.
2. Aceite os próximos dois passos clicando em *Next*. Após um tempo, o instalador pedirá para escolher em que diretório instalar o SDK. Pode ser onde ele já oferece como padrão. Anote qual foi o diretório escolhido, vamos utilizar esse caminho

mais adiante. A cópia de arquivos iniciará:

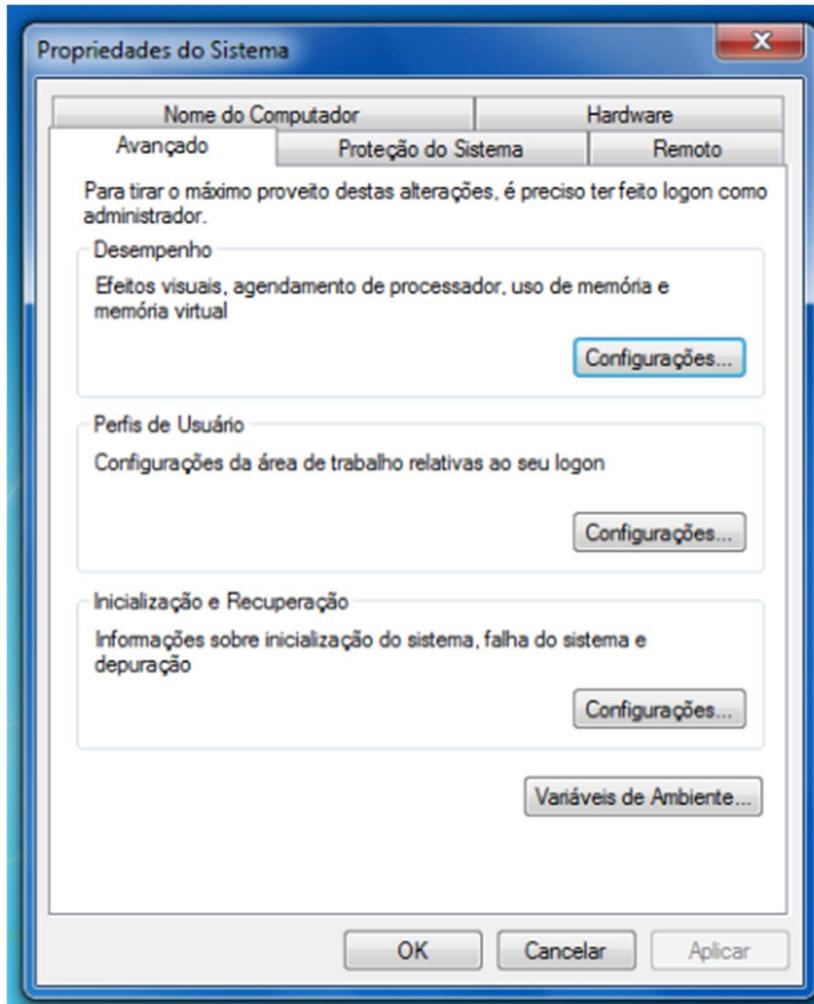


3. O instalador instalará também o JavaFX 2. Após isso, você será direcionado à uma página onde você pode, opcionalmente, criar uma conta na Oracle para registrar sua instalação.

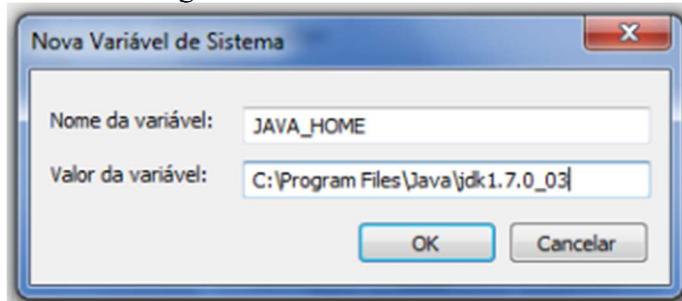
Configurando o ambiente

Precisamos configurar algumas variáveis de ambiente após a instalação, para que o compilador seja acessível via linha de comando. Caso você vá utilizar diretamente o Eclipse, provavelmente não será necessário realizar esses passos.

1. Clique com o botão direito em cima do ícone *Computador* e selecione a opção *Propriedades*.
2. Escolha a aba "Configurações Avançadas de Sistema" e depois clique no botão "Variáveis de Ambiente"

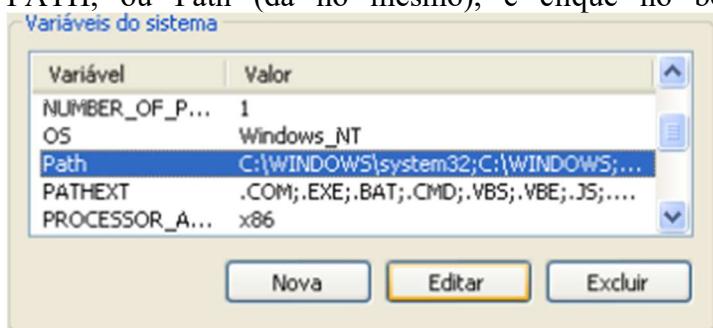


3. Nesta tela, você verá, na parte de cima, as variáveis de ambiente do usuário corrente e, embaixo, as variáveis de ambiente do computador (servem para todos os usuários). Clique no botão *Novo...* da parte de baixo.
4. Em *Nome da Variável* digite JAVA_HOME e, em valor da variável, digite o caminho que você utilizou na instalação do Java. Provavelmente será algo como: C:\Program Files\Java\jdk1.7.0_03:

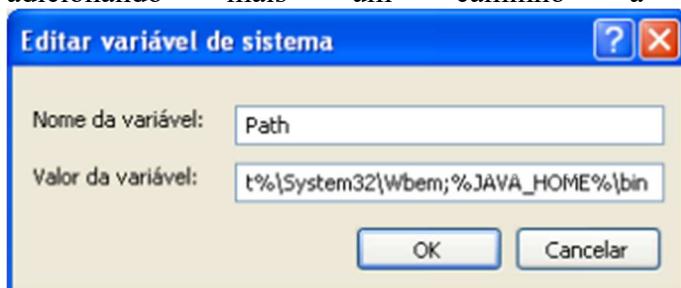


Clique em *Ok*.

5. Não vamos criar outra variável, mas sim *alterar*. Para isso, procure a variável PATH, ou Path (dá no mesmo), e clique no botão de baixo "Editar".



6. Não altere o nome da variável! Deixe como está e adicione no final do valor;%JAVA_HOME%\bin, não esqueça do ponto-e-vírgula - assim, você está adicionando mais um caminho à sua variável Path.



7. Abra o prompt, indo em *Iniciar, Executar* e digite cmd.
8. No console, digite javac -version. O comando deve mostrar a versão do Java Compiler e algumas opções.

The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. It displays the following text:
Microsoft Windows [versão 6.1.7601]
Copyright © 2009 Microsoft Corporation
C:\Users\Caetum> javac -version
javac 1.7.0_03
C:\Users\Caetum>

Você pode seguir para a instalação do Eclipse, conforme visto no seu capítulo, ou utilizar um editor de texto simples como o bloco de notas para os primeiros capítulos de apostila.