AUTOR (ERA SÓ O QUE FALTAVA)...

Fábio Burch Salvador

O PRIMEIRO E ÚNICO

LOUCADEMIA DE JAVA

J2SE

OU

ENCICLOPEDIAE JAVALISTICUS UNIVERSALIS

UM GUIA PARA QUEM NÃO SABE NADA DA LINGUAGEM JAVA, MAS QUER APRENDER. VOCÊ COMEÇA A LER APENAS COM CONHECIMENTOS BEM BÁSICOS EM LÓGICA DA PROGRAMAÇÃO, E TERMINA A LEITURA DESENVOLVENDO SISTEMAS EMPRESARIAIS COMPLETOS – SÓ QUE VAI DAR TRABALHO, CLARO!

ENTÃO, SE VOCÊ ESTIVER AFIM DE VAGABUNDAGEM, VÁ TENTAR UMA CARREIRA POLÍTICA OU ALGO ASSIM. OU VIRE HIPPIE.

“Homem de preto, qual é a sua missão?”

“É aprender logo Java e tirar Certificação!”

“Homem de preto, o que é que você faz?”

“Escrevo um código fonte que apavora o Satanás!”

Prefácio

Este livro serve para as pessoas aprenderem a programar em Java. Especificamente, J2SE, programação para desktop. Não é um livro para ser lido na cama ou no banheiro. Não é um livro para ler no ônibus. É um livro que tem muitos códigos e portanto é preciso baixar uma IDE e testar os exemplos. Só se aprende a programar programando.

O conteúdo deste volume foi ordenado seguindo o critério mais correto de todos - o critério da prática. Ele apresenta as aulas de Java ministradas pelo autor, da maneira como ele gostaria de ter feito caso tivesse tempo e o curso não fosse curto demais. As explicações usam uma linguagem informal, solta, e possivelmente possuem erros de concordância em diversos pontos.

Este livro é antes de mais nada um livro escrito às pressas, em horários livres, de intervalos para o lanche ou na madrugada antes do início das aulas. Portanto, mancadas no português devem ser perdoadas. É o livro escrito por alguém que passa o tempo todo na mais absoluta pressa. É, na verdade, o último trabalho feito por mim sob estas condições. As próximas obras serão mais calmamente escritas.

Espero que sirva para os iniciantes e até para quem já conhece alguma coisa de Java.

PreDifícil

Você pode achar Java uma linguagem complicada. Não é. Você pode achar que Orientação a Objetos é um bicho de sete cabeças. Não é. Tudo isso é muito simples. Eu vou tentar fazer parecer ainda mais simples. Trata-se de lógica natural.

Mas eu aviso: é preciso abstrair algumas noções para compreender o que se vai ler aqui. Não se pode pensar em classes como salas ou como caixas. Em objetos como variáveis, como valores ou sei lá o que. É preciso compreender que estamos lidando com coisas abstratas, que a princípio representam coisas reais, mas não SÃO reais. Portanto, comportam-se como aquilo que são: idéias. Uma classe é uma idéia de um tipo de coisa. Um objeto é a idéia de um objeto. E é uma instância de sua classe. Um método, é o quê? Uma função? Mais ou menos. Mas também é uma ção em potencial do objeto. Ou da classe, se for estático.

Pode parecer difícil mas não é. Poderia ser pior. Os comandos, todos palavras em inglês, poderiam ser escritos em alemão. Aí sim a cobra iria fumar!

“And we should consider every day lost on which we have not danced at least once. And we should call every truth false which was not accompanied by at least one laugh.”
(“Devemos considerar perdido um dia em que não dançamos pelo menos uma vez. E mentirosa toda a verdade que não vier acompanhada de pelo menos uma risada”)
Friedrich Nietzsche – Em inglês

“The book is on the table.”
(“Penso, logo existo.”)
Sócrates – Em portunhol

O autor – quem é esse cara?

Fábio Burch Salvador nasceu em 1981, em Porto Alegre. É viciado em informática desde guri, mais precisamente desde o dia em que colocou as mãos em um XT na firma onde o pai trabalhava. Programador desde os 13 anos (o pai tinha um 286) começou no Basic desenvolvendo Games cabulosos tirando sarro de amigos e conhecidos. Mais tarde, evoluiu, desenvolvendo então outras coisas medonhas em outras linguagens. Mas também desenvolveu muita coisa boa, sistemas empresariais principalmente - e apostilas!

Programa Basic, VB, JavaScript, Pascal, PHP, Java. Monta sites escrevendo HTML e CSS no Bloco de Notas. Aprendeu Flash sozinho. E também Photoshop, Corel Draw, SQL, e tudo mais. Incrivelmente, tem apenas um curso - de AutoCAD (?) - ele que rodou na sétima série exatamente por não saber nada de geometria e nem a Fórmula de Báskhara (que ele aliás não domina até hoje).

É formado em Jornalismo desde 2005 pela PUC de Porto Alegre. Por quê logo Jornalismo? Ninguém sabe. Nem ele. Apaixonado pelas câmeras, teve até uma curta carreira como cineasta, dirigindo 4 curtas-metragens, todos dignos de um Cine Trash .

Defensor implacável do DOS nos anos 90, acabou rendendo-se. Pródigo em criar bordões, Fábio criou frases de efeito que ficaram na História de todos os lugares pode onde passou. Iniciou sua vida profissional "pegando" uma série de empregos variados, mais ou menos no estilo Seu Madruga. Foi, entre outras coisas, digitador de cadastro nas Lojas Renner, técnico de fiscalização da Anatel, estagiário na Prefeitura de Porto Alegre, monitor de laboratório na PUC, e só não foi astronauta porque o Brasil não tinha programa espacial.

Em 2004, fundou um pasquim em Viamão/RS chamado "A Cidade". O jornal sagrou-se como o mais polêmico no município, tinha o desenho mais estiloso e foi o primeiro a ter um site. Bagunçou a vida dos políticos locais e estampou manchetes gritantes em capas surrealistas até seu fechamento no início de 2006. Fábio então passou a colaborar no "de-vez-em-quandário" Correio Viamonense, outro pasquim ainda mais mefítico do que o primeiro. Virou sub-celebridade local e passou a ser sistematicamente convidado pelos partidos políticos em tudo que é eleição.

Ainda buscando uma carreira de comunicador, tentou a sorte na redação do site Terra, como redador temporário. Em seguida, cansou-se de nunca ter grana para nada e resolveu abandonar o jornalismo. Voltou a dedicar-se à informática. Deu o arriscado passo da troca de carreira e acertou na mosca.

Como desenvolvedor de softwares, sua falta de instrução formal na área foi facilmente esquecida por todos os que o empregaram porque, afinal de contas, seus programas funcionam.

Agraciado com o Prêmio McGyver da Programação, Fábio foi professor no Senac em São Leopoldo, ensinando sua arte softwarística e tornando seus alunos tão pirados quanto ele. Acabou também ensinando design, liderança, oratória, administração e outras coisas mais. Saiu do Senac por vontade própria em 2008 para dedicar-se a programar em turno integral. Não que não goste de dar aulas. Teve seu$ motivo$ pe$$oai$.

Desde o começo e sempre, teve principalmente a ajuda indispensável de seus pais, que sempre o incentivaram a ir em frente e até a levantar nas inúmeras vezes em que caiu. Mais tarde, iria somar-se neste esforço de apoio sua namorada que virou esposa. Apesar de serem todos pessoas bem mais "normais" e discretas do que o autor deste livro, sempre tentaram compreendê-lo. Ele que por vezes é incompreensível. E que incompreensivelmente conseguiu achar seu caminho na vida.

Fábio é casado, tem uma filha, é um gaúcho urbano criado na Capital e nunca andou a cavalo. Não sabe tocar gaita. E não usa pilcha. Mas sabe assar churrasco, provando mais uma vez que a conveniência é a mãe da sabedoria.

Continua programando, atuando na área do ensino e dedica-se também a escrever. E ainda por cima mantém seu "lado jornalista" vivo no site que mantém na Internet e cujo jabá já foi feito neste livro. Projeta carros espalhafatosos em AutoCAD 3D (e publica os projetos na Internet). Às vezes, filosofa. Também mete-se na política. E até escreve sobre teologia.

Ainda não plantou uma árvore. Mas já teve um filho. E agora acaba de escrever um livro.

Agradecimentos

Este livro não teria sido escrito sem a ajuda de algumas pessoas. Em primeiro lugar, quero agradecer à minha esposa Fabiana por me aturar nos tempos em que passei concentrado escrevendo, à minha filha Camila por me trazer bolachinhas e beijos enquanto eu escrevo, aos meus pais que sempre me apoiaram e no final são os verdadeiros responsáveis por eu estar aqui hoje escrevendo. Ao meu cunhado Ismael por consertar meu PC nas vezes em que ele apresentou problemas. À minha sogra por me aturar. Ao meu sobrinho Raphael por me alegrar, e à minha irmã Ângela por ter crescido ao meu lado e ter participado de quase todos os momentos mais importantes da minha vida. Novamente à minha mãe por sempre me dar força e ao meu pai por me incentivar a aprender Basic.

Também quero homenagear meus alunos do Senac, tanto a galera do Java e do PHP, sempre formando turmas de pessoas divertidas, o que me estimulava a aprender mais para ensinar mais. E também à gurizada da Aprendizagem Comercial: Manu, Luan, Douglas, Alemão, Frajola, Tainara, Francine, Alisson, Elvis, Pri, e todo o resto da gurizada medonha. Ah, e claro, a Deize que veio da capital tecnológica do mundo Ocidental, Crissiumal. Essa gurizada, com seu jeito barulhento e impulsivo, acabou tornando a mim mesmo mentalmente mais jovem (o papel de professor estava a envelhecer-me aos poucos) e me fez ver que eu não poderia acomodar-me tão cedo, que deveria ser mais confiante e construir meu futuro com ambições maiores e, no fim, causaram eles mesmos a perda de seu professor de todas as tardes.

Também homenageio aos colegas do Senac, a Raquel, o Alexandre, o Antonio, o Alisson, a Pati, a Su, e todos os outros malucos que dividiram comigo um ano absolutamente divertido e edificante. E, claro, ao Nelson e à Carla, que na direção da escola souberam lidar com pessoas como, por exemplo, eu – o que sei ser uma missão casca-grossa.

Mando também um alô especial para a galera do Espiral, da WaySys, da Memo e de todos os lugares onde eu pude desenvolver sistemas, aprimorando-me no decorrer do caminho.

Também quero agradecer a Steve Wozniak, que inventou o computador pessoal. Ao Bill Gates, por ter criado um sistema operacional simplesmente maravilhoso. E ao Linus Torvalds pelo mesmo motivo.

Quero agradecer a Jesus Cristo, sem o qual nós hoje seríamos Zoroastristas ou mais provavelmente Mitraístas – se não fôssemos uns tribais animistas dançando seminus em vota de uma pedra e sacrificando virgens ao deus Dagon – quando se sabe que virgens têm outra utilidade bem melhor.

Não posso também deixar de agradecer a mim mesmo, pessoa sem a qual este texto não se auto-escreveria sozinho.

E quero agradecer a vocês, que já leram umas quantas páginas e ainda não viram nada de Java. Mas verão, e não se arrependerão. Então, vamos ao Java!

Luis, Michel, Bruno, Mauricio, Sergio e Diego (faltou o lendário Daniel) – turma de Java de 2008

Dejair, Clauber, Pozzebon e eu - turma de Java em 2007

```
Import java.livro.JIndice
class Livro_do_Java {
        static JIndice ind;
        public static void main (String[]args) {
                ind = new JIndice();
                ind.listar();
```

Índice

Nomenclaturas e badulaques

Vamos ver agora um pequeno glossário de coisas que precisamos saber antes de começar.

Java a JavaScript

A primeira confusão que existe no mundo Java é entre a linguagem Java e o JavaScript. Vamos ver isso de forma bem clara e objetiva:

JAVA é uma linguagem completa, enorme e complexa utilizada para montar aplicações completas com acesso a Bancos de Dados constante e fins mais completos. Programar Java exige alguma prática, e é exatamente para ensinar Java que eu escrevi este livro.

O código JAVA deve ser desenvolvido, testado e depois convertido em Byte-Code, em uma operação de compilação. O código torna-se então ilegível e o usuário não tem acesso às fontes que deram origem às funcionalidaes.

JAVASCRIPT é uma pequena linguagem de scripts usados dentro do código HTML de um site e que executam pequenas tarefas a serem executadas do lado do cliente na conexão via Internet. O código é aberto e as funcionalidades raramente passam de pequenas validações de campos em formulários HTML ou efeitos de animação limitados.

A única coisa legal sobre o JavaScript, do ponto de vista do programador Java, é que as sintaxes dos comandos são iguais. Laços de repetição, funções e outros badulaques têm a mesma forma de escrever que os da linguagem Java.

O que é uma API

Uma API é uma coleção de componentes de software que já vêm prontos e o programa só precisa usar. Muita gente fala das APIs do Windows, pois é possível chamar estas APIs nos programas. Nós vamos usar uma API do sistema operacional, lá no final do livro.

Vamos criar um arquivo PDF, e daí vamos rodar uma API que identificará qual o programa que o Windows do usuário está utilizando para abrir este tipo de arquivo.

A API do Java é uma coleção imensa, absurdamente imensa, de classes reutilizáveis com as mais bizarramente variadas utilidades. Temos, por exemplo, as classes da bibliotecas JAVA.AWT e JAVA.SWING, que servem para criarmos telas gráficas de apresentação para o nosso usuário. Temos a JAVA.IO, com suas classe usadas para manipular entrada e saída de dados, por exemplo, para arquivos-texto. Enfim.

As APIs nos economizam tempo, porque não precisamos criar as coisas “do zero”, fazendo um movimento semelhante a reinventar a roda. Não. Nós vamos usar tudo o que já vier pronto e desenvolver nossos softwares sem, por exemplo, ter que reinventar o botão clicável.

J2SE, J2EE e J2ME

São denominações que damos a três caminhos para o desenvolvimento de aplicações na linguagem Java.

J2SE (Standard Edition) consiste em criar classes, montando um programa executável, construindo a interface gráfica do programa nessas classes. No final, compila-se o código e temos um arquivo JAR, que funciona como um executável em qualquer computador devidamente preparado. Usa-se para fazer softwares desktop, que operam na tela do computador normalmente.

J2EE (Enterprise Edition) consiste em criar alguns códigos em arquivos JSP e construir umas classes. Parece muito com a programação J2SE. Porém a interface gráfica é feita com o uso de páginas HTML. Estas páginas têm FORMS cujo botão de ativação aciona as funcionalidades feitas em Java. É usado para desenvolver para a Web.

J2ME (Micro Edition) é parecido com o desenvolvimento J2SE, mas não se pode usar uma série de

funcionalidades, especialmente no que se refere à área de interface gráfica. Serve para desenvolver aplicações para telefones celulares, palmtops e outros dispositivos móveis.

Java Virtual Machine (JVM)

A Máquina Virtual do Java é um software que deve ser baixado e instalado no computador do usuário para que os programas em Java rodem.

Parece um problema, a princípio, mas é aqui que mora a verdadeira vantagem da linguagem Java sobre as linguagens mais tradicionais.

Uma linguagem tradicional cria um arquivo EXE, executável, que roda em contato direto com o Sistema Operacional do computador do usuário. Assim, se o EXE foi compilado para ser usado no Windows, ele não roda no Linux. Por isso os programas precisam ter várias versões, uma para Linux, uma para Windows, uma para Solaris, uma para o Unix, e assim por diante.

Já no Java, o programa roda em contato direto com a JVM, e a JVM é quem fala diretamente com o Sistema Operacional. Assim, se eu tiver 3 clientes, um com Linux, um com Windows e um com Solaris, e os três forem á internet baixar a JVM (que é gratuita), meu programa poderá ser compilado uma única vez, pois funciona em qualquer sistema operacional. Isso é maravilhoso. O lema da linguagem java é “Write Once, Run Everywhere”, ou seja, escreva (o código) uma vez, e rode em todos os lugares.

Java Developer Kit (JDK)

Ah, isso aqui é uma mão na roda! O kit de desenvolvimento Java consiste de uma JVM completa e mais as fontes das classes que formam a API do Java. Assim, temos todo o necessário para montar, n computador, um ambiente propício ao desenvolvimento de aplicações Java, porque essas classes prontas da API já vão ficar disponíveis para o nosso programa usar.

O que é uma IDE

Integrates Development Environments, as IDEs, são, traduzindo, Ambientes de Desenvolvimento Integrado. Mas o que diabos é isso?

São programas muito bons, que nos ajudam a fazer programas. A gente pode escrever código no Bloco de Notas e depois compilar, mas isso já é loucura. Nem eu sou maluco o suficiente para fazer isso.

Existem IDEs muito simples, como o BlueJ, que apenas tem um interpretador capaz de rodar nosso programa com ele ainda não fechado no arquivo JAR. Assim, podemos testar tudo antes de acabar o projeto.

Existem IDEs bem mais complexas, como o NetBeans e o Eclipse. No Eclipse, por exemplo, quando a gente escreve o nome de uma classe qualquer ou de um objeto pertencente a uma classe, aparecem na tela os métodos daquela classe em uma lista bem bonitinha, para a gente clicar. Isso nos poupa de ter que decorar, “de cabeça”, todos os métodos. Essas IDEs também corrigem nossos erros, marcando eles com sublinhados vermelhos, e possuem assistentes de compilação que nos ajudam a montar o arquivo JAR sem que nós tenhamos que escrever um arquivo de parâmetros “na mão”.

IDEs são, a grosso modo, editores que ajudam a gente a fazer código sem errar tanto.

Palavras Reservadas

Existem algumas palavras em Java que não poderemos usar para designar classes, variáveis ou qualquer outro tipo de coisa. Essas palavras são comandos nativos da linguagem:

abstract continue for new assert default goto package boolean do if private break double implements protected byte else import public case enum instanceof return catch extends int short char final interface static class finally long strictfp const float native super true false null

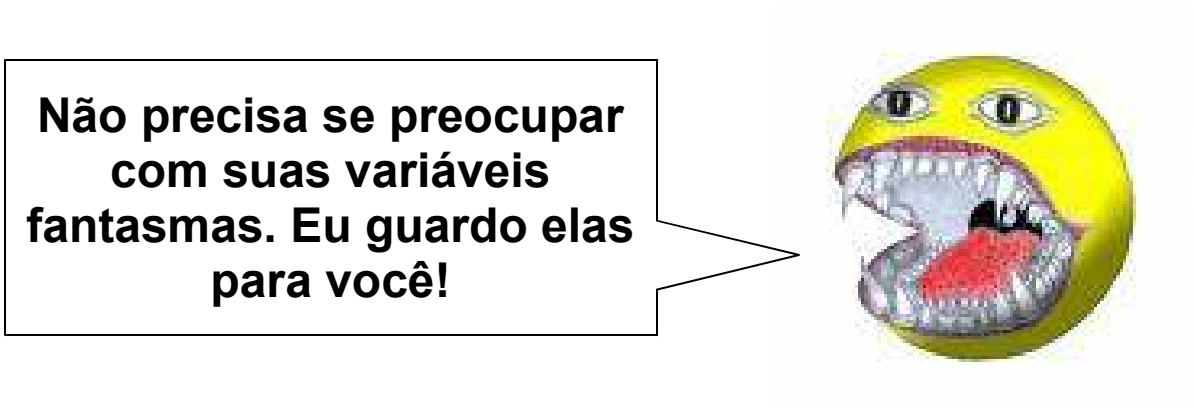
Coleta automática de lixo

Java é uma linguagem para lá de inteligente. Surgiu em 1995, e veio com essa história de lixo. O que é isso?

Simples! Nas linguagens de antigamente, nós criávamos a variável X e precisávamos, quando não mais a quiséssemos usar, destruí-la para desocupar memória. No Java, isso não acontece. Quando X deve cair fora, ela cai.

E mais: se antes a variável X ficava com lixo das atribuições de valor anteriores, hoje ao colocar um valor novo nela, ela é automaticamente limpa.

No fim, ficou muito mais fácil gerenciar memória.



Classes e objetos

O que vem a ser uma classe? Uma classe é aquilo que será instanciado para a criação de objetos. Não entendeu nada? Vamos então à explicação acadêmica para essa coisa toda:

Explicação Wikipédica

Em orientação a objeto, uma classe abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar.

Classes não são diretamente suportadas em todas as linguagens, e são necessárias para que uma linguagem seja orientada a objetos. Classes são os elementos primordiais de um diagrama de classes.

Estrutura da classe

Uma classe define estado e comportamento de um Objeto geralmente implementando métodos e atributos (nomes utilizados na maioria das linguagens modernas). Os atributos, também chamados de campos (do inglês fields), indicam as possíveis informações armazenadas por um objeto de uma classe, representando o estado de cada objeto. Os métodos são procedimentos que formam os comportamentos e serviços oferecidos por objetos de uma classe.

Outros possíveis membros de uma classe são:

* Construtores - definem o comportamento no momento da criação de um objeto de uma classe.

* Destrutor - define o comportamento no momento da destruição do objeto de uma classe. Normalmente, como em C++, é utilizado para liberar recursos do sistema (como memória).

* Propriedades - define o acesso a um estado do objeto.

* Eventos - define um ponto em que o objeto pode chamar outros procedimentos de acordo com seu comportamento e estado interno.

Encapsulamento

Em linguagens orientadas a objetos, é possível encapsular o estado de um objeto. Em termos práticos, isso se realiza limitando o acesso a atributos de uma classe exclusivamente através de seus métodos. Para isso, as linguagens orientadas a objeto oferecem limitadores de acesso para cada membro de uma classe.

Tipicamente os limitadores de acesso são:

* público (public) - o membro pode ser acessado por qualquer classe. Os membros públicos de uma classe definem sua interface.

* protegido (protected) - o membro pode ser acessado apenas pela própria classe e suas sub-classes.

* privado (private) - o membro pode ser acessado apenas pela própria classe.

Cada linguagem de programação pode possuir limitadores de acesso próprios. Por exemplo, em Java, o nível de acesso padrão de um membro permite que qualquer classe de seu pacote (package) possa ser acessado. Em C#, o limitador de acesso interno (internal) permite que o membro seja acessado por qualquer classe do Assembly (isto é, da biblioteca ou executável).

No exemplo abaixo, implementado em Java, a classe Pessoa permite o acesso ao atributo nome somente através dos métodos setNome e getNome.

```
public class Pessoa {
   public String getNome() {
      return nome;
   }
   public void setNome(String nome) {
      this.nome = nome;
   }
   private String nome;
}
```

Herança

A herança é um relacionamento pelo qual uma classe, chamada de sub-classe, herda todos comportamentos e estados possíveis de outra classe, chamada de super-classe ou classe base. É permitido que a sub-classe estenda os comportamentos e estados possíveis da super-classe (por isso este relacionamento também é chamado de extensão). Essa extensão ocorre adicionando novos membros a sub-classe, como novos métodos e atributos. É também possível que a sub-classe altere os comportamentos e estados possíveis da super-classe. Neste caso, a sub-classe sobrescreve membros da super-classe, tipicamente métodos.

Quando uma classe herda de mais de uma super-classe, ocorre uma herança múltipla. Esta técnica é possível em C++ e em Python, mas não é possível em Java e C#, no entanto estas linguagens permitem múltipla tipagem através do uso de interfaces.

Explicação "no popular"

Uma classe é um pedaço de código que vai definir um molde para a gente construir objetos.

Uma classe tem, basicamente, métodos e atributos. Atributos são como variáveis. Métodos são como funções. Se você não sabe o que são Variáveis e Funções, volte a ler o material sobre Lógica da Programação.

Um exemplo prático:

Vamos construir a classe PESSOA.

ATRIBUTOS ==> NOME (texto), IDADE (número inteiro) e SEXO (texto)

MÉTODOS ==>

Método ANIVERSARIO – A idade da pessoa é igual à idade +1 (se era 20, passa a ser 21).

Método CIRGURGIA_NO_MARROCOS – O sexo muda para "Transexual"

Ótimo, Agora, temos, uma classe com 3 atributos e 2 métodos. Então, eu vou escrever o código principal do programa, aquele que será executado linearmente (como acontecia nas linguagens Pascal, Clipper, Cobol, Basic, e todas as antigas).

No código principal, eu vou colocar uma instrução:

A Variável P armazenará um objeto do tipo PESSOA.

P = Nova PESSOA;

P.NOME = "Joãozinho";

P.IDADE = 30;

P.SEXO = "Homem";

Bom. Temos então a variável P, armazenando nossa PESSOA, o “Joãozinho”, com 30 anos, Homem.

Agora, vamos fazer o programa executar um método do objeto P:

P.ANIVERSARIO();

Ao executar este método, o P vai auto-aumentar sua idade em 1 ano, ficando com 31 anos.

Acontece que P é um OBJETO do tipo PESSOA. Ele não está armazenado dentro da CLASSE PESSOA, ele foi gerado a partir dela, mas existe de forma independente.

Depois disso, vou criar outro objeto, o P2, que também é uma PESSOA:

P2 = Nova PESSOA;

P2.NOME = “Claudinei”;

P2.IDADE = 20;

P2.SEXO = “Homem”;

Temos então agora:

Joãozinho – Homem – 31 anos

Claudinei – Homem – 20 anos

Se eu mandar que o objeto P2 execute um método:

P2.ANIVERSARIO();

Então agora, P2 aumentou sua idade em 1 ano. Ele tem 21 anos. Mas a idade de P continua em 31.

E se eu ordenar que:

P2.CIRURGIA_NO_MARROCOS();

Então P2 mudará seu sexo para “Transexual”, embora P não sofra conseqüência alguma.

Para entender:

P e P2 são independentes. Eles podem fazer aniversário em épocas diferentes, um deles pôde trocar de sexo, e as funções executadas por um não vão afetar o outro. Eles são, contudo, semelhantes porque ambos têm os mesmos 3 atributos e 2 métodos, embora cada um armazene atributos com valores diferentes e execute seus métodos de forma completamente independente.

Por quê é interessante trabalhar com Orientação a Objetos?

Porque eu posso criar a mesma classe PESSOA que eu criei no exemplo anterior, e criar um VETOR (se você não lembra o que é isso, volte a consultar o material de Lógica da Programação).

Vetor V[100] de PESSOA (vai armazenar até 100 objetos do tipo PESSOA).

V[0].NOME = “Fredegundeson”;

V[1].NOME = “Godofredilson”;

V[2].NOME = “Juvenal Antena”;

V[3].NOME = “Mendigo Pop”;

E assim por diante...

Aliás, é este o mecanismo usado pelos carrinhos de compras dos sites de e-commerce: existe uma classe PRODUTO que determina que todo produto vai ter certos atributos (preço, tipo, descrição, etc.), e terá alguns métodos (somar_frete, adicionar_ao_carrinho, e assim por diante). E daí o programa alimenta um Vetor de PRODUTOs e vai guardando na memória, sem envolver gravação de Banco de Dados.

No final, quando o usuário resolve efetivar a compra, há a gravação de um dado efetivamente, sendo então os dados armazenados no vetor de objetos dentro da base de dados.

Em Java, tudo é classe e objeto

Em Java, até o próprio programa principal, aquele que roda e faz o resto das classe se mexer, é uma classe ele próprio, e o código central do software é um método. Veremos depois como fazer um programa funcional.

Você vai ter que criar classes para tudo e usar objetos de um monte de classes. A interface gráfica, por exemplo, usa uma classe chamada JFrame. A JFrame é uma classe que vem com a JDK e que monta janelinhas no estilo Windows. Esta classe tem atributos (cor de fundo, tamanho, etc.). Então, eu crio um objeto a partir dela e manipulo estes atributos para obter a aparência que eu quiser para o meu programa.

Se você ainda não entendeu nada

Não fique preocupado. Quem não tem facilidade para a teoria muitas vezes tem para a prática. Vamos então começar a trabalhar.

Padrões básicos de programação Java

Primeiro, é preciso saber que Java é uma linguagem “case sensitive”, ou seja, faz diferença digitar alguma coisa em letras maiúsculas ou minúsculas. Então, se quisermos usar uma JFrame e escrevermos ali JFRAME, nada vai dar certo.

Principais tipos de dados

Java trabalha com alguns tipos de variáveis e dados. Se você não sabe o que é uma variável volte a estudar o material de Lógica da Programação.

int	Número inteiro.
double	Número com casas decimais.
boolean	Apenas pode valer Verdadeiro ou Falso.
char	Um caracter, que pode ser letra ou número ou sinal gráfico.
String	Uma série de “chars”, formando uma frase ou texto.

Usando chaves

Segundo, é preciso saber como delimitamos blocos. Blocos são coisas, como por exemplo, o conteúo de uma expressão condicional.

Linguagens como o Pascal e o Basic utilizam palavras-chave para abrir e fechar blocos. Assim, um IF em Basic fica:

```
IF X = 1 THEN
        // ALGUMA COISA ACONTECE
END IF
```

No java, porém, usamos chaves. Exatamente, chaves, aquelas que nós aprendemos a usar na quarta série para delimitar cálculos.

```
If (x == 1) {
        // ALGUMA COISA ACONTECE
}
```

Fazendo cálculos

Depois, precisamos saber como fazer cálculos. Os sinais matemáticos são os seguintes:

+	Soma
-	Subtração
/	Divisão
*	Multiplicação

Só que para fazer cálculos envolvendo uma certa hierarquia, não usamos parênteses, colchetes e chaves. Usamos apenas parênteses, uns dentro dos outros.

Comparações

Em Java, faremos algumas comparações de valores entre variáveis. Então, devemos conhecer os sinais usados:

X == Y	X é igual a Y
X != Y	X é diferente de Y
X < Y	X é menor que Y
X > Y	X é maior que Y
X <= Y	X é menor ou igual a Y
X >=Y	X é igual ou maior do que Y

Concatenando condições

Eu posso exigir que, para alguma coisa ser executada, duas ou mais condições sejam verdadeiras. Para isso, uso a concatenação das condições utilizando principalmente três sinais:

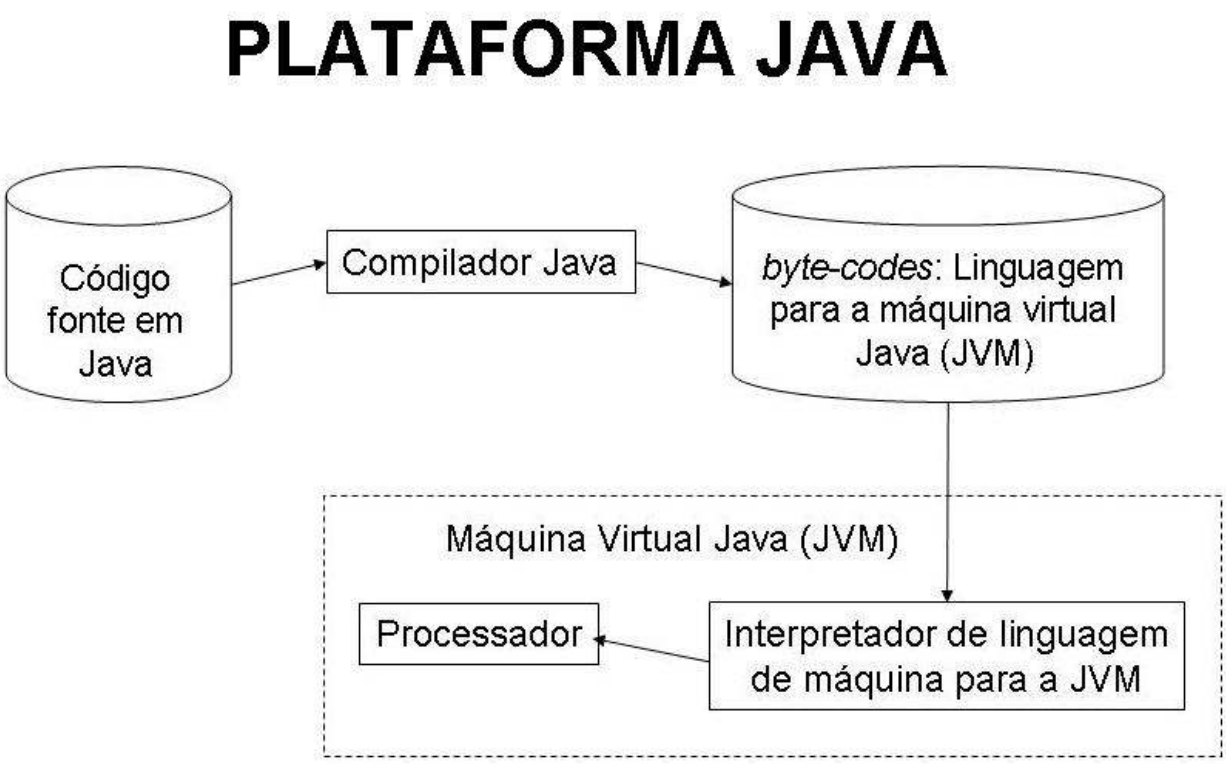
&&	If ((Condição1) && (Condição2)) {	Será verdadeiro se a primeira condição for verdadeira e a segunda também for. Caso alguma delas falhe, toda a expressão será FALSE.
		
		Exemplo:
		SE ((TIVER SOL) && (EU ESTIVER COM GRANA)) {
		VOU À PRAIA
		}
		
		Ou seja, se fizer sol mas eu estiver "duro", não vou à praia. Se eu tiver dinheiro e chover, eu também não vou. Só vou se estiver com dinheiro e fizer sol.

||	If ((Condição1) || (Condição2)) {	Será verdadeiro se qualquer das condições for verdadeira, mesmo que a outra não seja.
		
		Exemplo: SE ((HOUVER UM ÔNIBUS) || (ARRUMAR CARONA)) {
		EU VOU
		}
		
		Ou seja, se eu arrumar um jeito de viajar, seja ele qual for, eu vou viajar.

XOR	If ((Condição1) && (Condição2)) {	Será verdadeiro se uma, e apenas uma, das condições for verdadeira.
		
		Exemplo: SE ((ELE É GAY) XOR (ELE É MULHER)) {
		ELE GOSTA DE HOMEM
		}
		
		Ou seja, para gostar de homem, OU o sujeito é gay, OU é mulher. Se for mulher E gay, nada feito – é lésbica e não gosta de homem.
		Se não for mulher nem gay, então é homem heterossexual e não gosta de homem.

Montando um programa simples

Antes de mais nada, vamos ver um pequeno diagrama que mostra como funciona o esquema que eu tentei explicar nos textos que lemos até agora:



Agora, vamos montar um programa finalmente. Abra a sua IDE e vamos começar a escrever.

Desde os anos 80, há uma tradição que diz que o primeiro software desenvolvido por qualquer pessoa é o famoso “Olá Mundo”. Nós vamos fazer um pouco diferente. Nosso primeiro programa exibirá na tela uma frase que não seja essa tradicional, pois já está muito manjada.

Então, vamos lá:

```
public class prog {
	public static void main (String[]args) {
		System.out.println("Minha vingança será maligna!");
	}
}
```

O programa acima é simplíssimo. Na primeira linha, eu crio a classe PROG. Na segunda, inicio o método MAIN, que será o principal do nosso programa. Não se preocupe em entender as palavras PUBLIC, STATIC e VOID pois há um farto material sobre isso mais adiante neste mesmo livro (ver Modificadores). O método chamado MAIN é sempre rodado como principal, por um padrão existente na linguagem Java.

Depois, eu tenho uma chamada à classe SYSTEM.OUT, que lida com saídas de dados para o sistema. Eu rodo o método PRINTLN, que significa “mande essa frase para o prompt e quebre a linha”, sendo que eu passei uma frase para ele exibir. Pronto! Temos um programa funcional!

Sintaxes principais

Laços de repetição

Os principais laços de repetição usados em Java são o FOR e o WHILE. Se você não faz a mínima idéia do que seja um Laço de Repetição, por favor reveja a matéria de Lógica da Programação.

FOR

O FOR roda um determinado código por um número limitado de vezes. A sintaxe de sua declaração é um pouco confusa para quem nunca programou em Java, C ou PHP, mas é bem simples. Ela é composta de 3 elementos separados por ponto-e-vírgula.

O primeiro elemento é a criação e atribuição de valor iniciar a uma variável que servirá de contador.

O segundo elemento é a condição que precisa ser verdadeira para o laço continuar rodando.

O terceiro elemento é o comando de incremento da variável.

```
public class prog {
	public static void main (String[]args) {
		for (int x = 0; x<10; x++) {
			System.out.println("Estou rodando pela " + x + "ª vez!");
		}
	}
}
```

O resultado será:

```
Estou rodando pela 0ª vez!
Estou rodando pela 1ª vez!
Estou rodando pela 2ª vez!
Estou rodando pela 3ª vez!
Estou rodando pela 4ª vez!
Estou rodando pela 5ª vez!
Estou rodando pela 6ª vez!
Estou rodando pela 7ª vez!
Estou rodando pela 8ª vez!
Estou rodando pela 9ª vez!
```

WHILE

A instrução WHILE é simplesmente a imposição de uma condição para que o laço continue a repetir-se por um número infinito de vezes até que aquela condição não seja mais cumprida. Trata-se de algo bem mais simples do que o FOR.

Vamos pegar como exemplo o seguinte: eu vou ficar perguntando ao usuário, pedindo que ele digite um número. Ele vai ficar digitando números até o momento em que ele digitar um 9, que é o número de SAIR DO SISTEMA, porque eu vou definir isso no WHILE.

```
public class prog {
	public static void main (String[]args) {
		int x = 0;
		while (x != 9) {
			// Aqui dentro vai ter uma rotina para o usuário digitar um número
		}
	}
}
```

Eu não coloquei o código responsável pela interação do usuário porque ele contém muitos comandos que nós não vimos ainda e isso iria embaralhar a cabeça dos leitores.

Estruturas condicionais

IF

O uso da instrução IF nos permite fazer algumas escolhas, o que é bem interessante. Estruturas condicionais equivalem á instrução SE do Português Estruturado que você deve ter visto em Lógica da Programação.

Um IF funciona de forma simples. Eu coloco dentro de parênteses a condição que deve ser verdadeira para que eu execute o código que está dentro do bloco IF.

Um exemplo de código que, executando, não satisfará à condição do IF e portanto não fará nada:

```
public class prog {
	public static void main (String[]args) {
		int x = 5;
		if (x == 4) {
			System.out.println("Se esta linha aparecer, algo muito errado foi digitado!");
		}
	}
}
```

ELSE

O Else significa “senão”, e é usado em um IF para dar comandos que acontecerão caso a condição do IF não seja atendida.

No programa de exemplo, eu coloquei o valor FALSE em uma variável boolean que determina se um sujeito é viado ou não. Se ele for, uma mensagem é exibida. SENÃO, outra dizendo “Muy Macho” aparecerá.

```
public class prog {
	public static void main (String[]args) {
		boolean viado = false;
		if (viado == true) {
			System.out.println("Ah, viadão!");
		}
		else {
			System.out.println("HUm, muy macho!");
		}

	}
}
```

ELSE IF

O Else If é usado para colocarmos uma alternativa dirigida dentro de uma estrutura condicional. É o caso, por exemplo, do seguinte sistema:

Eu vou cadastrar ali, em uma String, que tipo de político um determinado deputado é. Vou ter várias opções listadas com ELSEIF, e finalmente, no finalzinho, um ELSE para servir caso a opção marcada seja “Nenhuma das Anteriores”.

```
public class prog {
	public static void main (String[]args) {
		String falcatrua = "Dossiê";
		if (falcatrua == "Mensalão") {
			System.out.println("Esse é do Mensalão!");
		}
		else if (falcatrua == "Dossiê") {
			System.out.println("Esse é da turma do Dossiê!");
		}
		else if (falcatrua == "Bolsa-Esmola") {
			System.out.println("Esse é Populistão!");
		}
		else {
			System.out.println ("Ainda não descobriram o que o cara faz");
		}
	}
}
```

SWITCH

O SWITCH é um comando usado para descobrir dentre várias opções o que deve acontecer. Um exemplo:

```
// Antes, o programa pergunta ao usuário qual opção, entre 1 e 3, ele quer ver.
Int x = // A escolha do usuário
Switch (x) {
	case 1:
		System.out.println(“Você escolheu 1”);
	break;
	case 2:
		System.out.println(“Você escolheu 2”);
	break;
	case 3:
		System.out.println(“Você escolheu 3”);
	break;
	default:
		System.out.println(“Nenhuma das anteriores”);
}
```

Onde os Breaks fazem a execução parar e o programa passa a executar o que vem imediatamente após o Switch. Já o Default é o que acontece se nenhuma das opções válidas for escolhida.

Aplicando Orientação a Objetos

Instanciando uma classe

Eu já disse que uma classe é um molde para criarmos objetos. Certo. Agora, vamos criar uma classe bem simples para fazermos uma experiência prática.

```
public class pessoa {
	String nome;
	int idade;
	pessoa (String n, int i) {
		this.nome = n;
		this.idade = i;
	}
	void aniversario () {
		this.idade ++;
	}
	String dados() {
		return this.nome + " tem " + this.idade + " anos.";
	}
}
```

Bom. Eu criei uma classe Pessoa, que possui como ATRIBUTOS um Nome e uma Idade.

Daí, parti para a criação de métodos. O primeiro método tem o mesmo nome da classe, o que pode parecer bizarro, mas é perfeitamente plausível. Quando um método tem o mesmo nome da classe, ele passa a ser executado no próprio ato de criação de um objeto daquela classe. Assim, chamamos este método de "construtor". Veja que o construtor de Pessoa recebe uma String (um texto) e um int (um número), que ele atribui a THIS.NOME e THIS.IDADE. Mas afinal, o que é THIS? This, em inglês, significa "este", e quando eu digo THIS.NOME, estou dizendo (aliás, o OBJETO está dizendo) "meu nome será" - e aí vem o valor da variável n, que contém o nome da Pessoa.

Depois, eu tenho o método "aniversario", que apenas adiciona um ano à idade da pessoa. A palavra VOID antes do nome do método significa que este método não retorna nada de dados – ele apenas executa a função de adicionar um ano a mais na idade da pessoa, não dá retorno algum. Void em inglês significa "vazio".

O último método, "dados", retorna um dado do tipo String. Ele só concatena numa String os dados da Pessoa, e devolve para o local de onde o método foi chamado.

Note que os método têm parênteses após seus nomes. Os parênteses estão ali para a gente especificar parâmetros vindos em forma de variáveis (aliás, nós fazemos isso no método construtor, o que já serve de demonstração). É algo muito parecido com o que fazíamos com as FUNÇÕES lá nas aulas de Lógica da Programação. Se você não se lembra, dê uma estudada.

Bom. Agora, chegou a hora de criar a classe principal, que vai ter um programa rodando e instanciando a classe Pessoa.

```
public class principal {
	public static void main(String[]args) {
		pessoa x;
		x = new pessoa("João", 25);
		String texto;
		texto = x.dados();
		System.out.println(texto);
		x.aniversario();
		texto = x.dados();
		System.out.println(texto);
	}
}
```

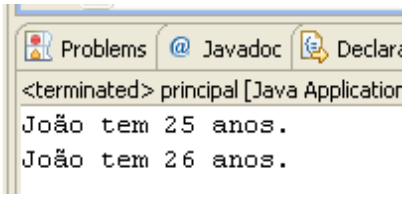
Veja que eu criei uma classe principal que tem o método MAIN – este que será rodado como método principal do programa.

O que é que eu faço primeiro? Eu crio uma variável X que vai receber um objeto do tipo PESSOA. Depois, eu atribuo à variável X um objeto PESSOA. Bom. Mas como eu disse que, ao criar um objeto, o programa rodará também seu método construtor, eu já passei no próprio ato de criação do objeto os parâmetros que vão preencher as variáveis N e I do método construtor de PESSOA – passando o nome e a idade do sujeito.

Depois disso, eu criei uma variável "texto" que é String, e fiz ela receber o RETORNO de dados fornecido pelo método DADOS() do objeto X, que é uma Pessoa. O retorno foi aquela frase "João tem 25 anos". Em seguida, mandei mostrar a variável TEXTO na tela com o comando System.out.println. Então, a frase apareceu na tela.

Só que em seguida, eu mandei o X fazer aniversário. Então, João agora tem 26 anos e não mais 25. Depois disso, mandei exibir os dados novamente.

O resultado será o seguinte:



Este meu código ficou meio longo. Eu poderia encurtá-lo, fazendo o seguinte:

```
pessoa x;
x = new pessoa("João", 25);
```

As duas linhas acima podem ser resumidas em:

```
pessoa x = new pessoa ("João", 25);
```

Esta frase só faz sentido se notarmos que ela resulta da justaposição de três instruções.

PESSOA X – Cria a variável X, para guardar objeto do tipo PESSOA.
X = NEW PESSOA – Cria um objeto PESSOA dentro da variável X.
PESSOA ("João", 25) – Roda o método PESSOA, de dentro da classe PESSOA, especificando dados.

Há ainda outra instrução que poderia ser resumida:

```
String texto;
texto = x.dados();
System.out.println(texto);
```

Eu poderia nem usar a String TEXTO, colocando diretamente o retorno do método DADOS para ser exibido na tela.

```
System.out.println(x.dados());
```

Ao fazer reduções, o código da classe principal ficaria assim:

```
public class principal {
	public static void main(String[]args) {
		pessoa x = new pessoa("João", 25);
		System.out.println(x.dados());
		x.aniversario();
		System.out.println(x.dados());
	}
}
```

Herança

Herança é uma coisa que ocorre em linguagens orientadas a objeto como o Java. Nós podemos criar uma classe que é como um filhote de outra, e que herda todos os seus atributos e métodos, podendo então ter seus próprios métodos e atributos que apenas servem para somar aos que foram herdados.

Vamos ver um exemplo. Eu vou criar uma classe chamada “Veiculo”, que terá duas sub-classes: “Tanque” e “Carro”.

```
public class veiculo {
	String modelo;
	int ano_fabricacao;
	String estado_funcionamento;
	veiculo (String m, int a) {
		this.modelo = m;
		this.ano_fabricacao = a;
		this.estado_funcionamento = "Perfeito";
	}
	void levar_tiro() {
		this.estado_funcionamento = "Estragado";
	}
}
```

Na classe Veículo, eu coloquei 3 atributos:

modelo – para guardar o nome do modelo de veículo;

ano_fabricacao – Para guardar o ano em que o veículo foi fabricado;

estado_funcionamento – para guardar em que estado de funcionamento o veículo encontra-se.

Então, criei um método construtor que recebe por parâmetros um texto com o modelo e o ano do veículo, e que colocam estes valores ou palavras dentro dos atributos do objeto.

Depois, criei um método que será rodado sempre que o veículo levar um tiro, mudando seu estado de conservação de “Perfeito” para “Estragado”, pois vamos assumir que um veículo qualquer, ao ser baleado, fica com grandes danos.

Bom. Até aqui, nenhuma novidade. Vou agora então criar uma classe que será sub-classe de Veículo. Esta classe será Carro – porque Carro é um tipo de veículo específico, pois haveria também a Moto, o Trem, o Trator e por aí em diante.

```
public class Carro extends veiculo {
	String categoria;
	Carro (String m, int a, String cat){
		super(m, a);
		this.categoria = cat;
	}
}
```

Agora, dê uma olhada nesta classe Carro. Ela EXTENDS a classe Veículo. Isso quer dizer que ela herda tudo – atributos e métodos – da sua superclasse (veículos). Um Carro é um tipo de veículo específico.

Após herdar as características de “veículo”, a Carro ainda adiciona um atributo a mais, o “categoria”, usado para dizer se é de luxo, se é popular, etc. Então, um Carro tem tudo o que um Veículo tem: ano, modelo, estado de funcionamento, e... CATEGORIA!

Dê também uma olhada no método construtor desta classe. Ela tem um comando SUPER – este comando faz com que o objeto, pertencente à subclasse, rode o método construtor da superclasse – ou seja, que rode o método construtor de “Veículo”. Depois de o método da superclasse atribuir nome, ano de fabricação e estado de funcionamento ao Carro, o método da própria subclasse atribui a ele uma

Categoria. E daí temos um carro completo!

Isso acontece porque quando criamos um Carro, estamos criando um Veículo ao mesmo tempo. Mas não em dois objetos diferentes, e sim em um só. Se Carro é uma subclasse de Veículo, e se X é um Carro, então ao mesmo tempo X é um Veículo.

É como dizer que temos uma superclasse Pessoas e uma subclasse Programador. Se eu digo que João é um Programador, por extensão ele é uma Pessoa.

Depois, para continuar nosso exemplo, eu vou criar uma outra subclasse de "veículo", e esta será chamada de Tanque, para produzir tanques de guerra.

```
public class Tanque extends veiculo{
	int calibre_canhao;
	Tanque (String m, int a, int cal){
		super(m, a);
		this.calibre_canhao = cal;
	}
	void levar_tiro() {
		this.estado_funcionamento = "Meio avariado";
		System.out.println ("Haha! Otário! Nem doeu!");
	}
}
```

Aqui, eu fiz quase o mesmo que tinha feito na classe Carro. A classe Tanque extends Veículo, e por isso tem um Modelo, um Ano_fabricacao e um Estado_funcionamento. Eu adicionei mais um atributo, que é o Calibre_canhao, usado para guardar a informação sobre a arma de ataque que o tanque de guerra possui.

Note que a outra subclasse de Veículo, a classe Carro, não possui um calibre de canhão. E que a própria superclasse Veículo não possui um calibre também. Isso é um dado específico, que só os tanques de guerra possuem.

Eu fiz um método construtor, assim como havia feito na classe Carro.

Só que depois disso eu fiz algo novo. Eu criei um método chamado "levar_tiro", mudando sua função – ele agora não deixa o carro totalmente estragado. Apenas muda o estado de funcionamento do Tanque de "Perfeito" para "Meio avariado", e ainda exibe uma mensagem tirando sarro da cara do inimigo.

Bom. Agora, eu tenho o dilema: todo Veículo, ao ser atingido, fica "Estragado". Mas o Tanque diz que ele só fica "Meio Avariado".

Isso é o que chamamos de sobreposição de métodos, uma das características principais da Herança. Eu tenho uma superclasse Veículo que determina que todo veículo atingido por uma bala fica estragado. Mas, na subclasse Tanque, eu especifico outra coisa. O método da subclasse sobrepõe o da superclasse.

Então, se eu criar agora as subclasses Moto, Trator e Barco, todas elas herdando diretamente de Veículo, então nós teremos Carros, Motos, Tratores e Barcos que, ao serem atingidos por um tiro, ficarão "Estragados". Mas teremos Tanques que, atingidos, ficam apenas "Meio avariados".

É como dizer que no Brasil há uma superclasse Pessoa, e que ela possui um método "aposentar-se". Todas as subclasses da classe Pessoa herdam este método sem sobrepô-lo. A não ser a subclasse Político, que sobrepõe este método e tem um cálculo completamente diferente de aposentadoria do resto das pessoas.

Polimorfismo

O fenômeno conhecido como Polimorfismo é assim: quando eu crio uma classe e depois crio diversas sub-classes para ela, eu estou na verdade criando tipos específicos da coisa que é representada pela superclasse.

Então, por exemplo, eu crio a classe Pessoa, e depois as sub-classes Milico e Político, que são tipos específicos de pessoas.

O caso é que, depois, eu posso criar uma variável X que é declarada como sendo do tipo genérico Pessoa, e X é capaz de receber um milico ou um político (tanto faz), daí a palavra Polimorfismo (Poli = Muitos e Morfos = Forma).

Isso parece bastante inútil à primeira vista. Mas torna-se útil no momento em que eu crio um vetor declarado como sendo de Pessoas e então preencho as posições dele com milicos, políticos ou quaisquer objetos de qualquer sub-classe de Pessoa.

```
abstract class pessoa {
   int idade;
   String nome;
   pessoa (int i, String n) {
      this.idade = i;
      this.nome = n;
   }
   abstract void mostrar();
}
class milico extends pessoa {
   String patente;
   milico (int i, String n, String p) {
      super(i,n);
      this.patente = p;
   }
   void mostrar () {
      System.out.println(nome + " é um " + patente + " cuja idade é " + idade + " anos.");
   }
}
class politico extends pessoa {
   String partido;
   politico (int i, String n, String p) {
      super(i,n);
      this.partido = p;
   }
   void mostrar () {
      System.out.println(nome + " é um político de " + idade + " anos de idade, filiado ao " + partido);
   }
}
class prog {
   public static void main(String[]args) {
      pessoa[]m = new pessoa[3];
      m[0] = new milico (23, "Juca", "Cabo");
      m[1] = new politico (31, "Joao", "PMDB");
      m[2] = new milico (56, "Fernandinho", "PTB");
      m[0].mostrar();
      m[1].mostrar();
      m[2].mostrar();
   }
}
```

Modificadores

Modificadores são argumentos que são colocados antes do nome no ato de criação de métodos ou variáveis, dando a estes métodos ou variáveis características especiais, mudando seu escopo de atuação ou impedindo-os de fazer algum tipo de coisa que nós precisamos ter certeza de que não acontecerá.

Vejamos então os principais modificadores que existem na linguagem Java.

Static

Quando criamos uma classe, e queremos ativar seus métodos ou pegar o valor de um atributo a partir de outra classe, precisamos instanciar a primeira classe. Porque a classe em si não tem métodos funcionais e nem valor nos atributos. O objeto criado a partir desta classe é que os tem.

Mas se o atributo ou o método recebe o modificador “static”, então este atributo ou método passa a ser ligado diretamente à classe, não aos seus objetos.

Com isso, eu posso muito bem executar um método static sem criar objeto algum da classe onde ele foi escrito, bastando para isso colocar a forma “nomedaclasse.nomedometodo()”.

Uma outra coisa que ocorre é que, se eu crio a classe “carro”, com o atributo static “cor”, e eu mudo a cor de um carro, o mesmo valor no mesmo atributo ocorre em todos os outros carros. Assim, se eu deixar o carro que está na variável X com a cor “azul”, todos os outros carros, Y, Z, ou qualquer outro, ficarão azuis.

Public

Um atributo ou método PUBLIC pode ser acessado e visto a partir de qualquer classe, de qualquer ponto do programa, abertamente, através da fórmula “NomeDoObjeto.NomeDoAtributo” ou então “NomeDoObjeto.NomeDoMétodo(argumentos)”.

Private

Um método que é PRIVATE não poderá ser acessado através de nenhuma outra classe a não ser os objetos daquela classe na qual foi criado. Assim, se eu crio o método PRIVATE ACELERAR, dentro da classe Carro, eu não posso simplesmente criar o objeto Carro X e querer usar X.ACELERAR(). Eu precisarei de outro método, dentro de Carro, que ative ACELERAR “por dentro”.

Existem também atributos PRIVATE, e variáveis também. Estes, só podem ser acessados por dentro da classe ou método onde foram criados, e servem para guardar valores de uso exclusivo naquele escopo.

Os métodos ou atributos PRIVATE não são vistos por ninguém de fora da classe de origem, nem mesmo pelas sub-classes.

Final

O modificador Final, utilizado em alguns objetos, métodos e variáveis, faz com que estes não possam ser reutilizados ou sobrescritos por subclasses no futuro (ou seja, mais adiante no código). Algumas funcionalidades, como as do ActionListener, só podem ser associadas a um componente qualquer (no caso do ActionListener, botões e menus), quando existe a certeza absoluta de que estes componentes não irão mudar ou evaporar-se por ação do programador em algum momento do código. Por isso, exige-se o uso do modificador “final” nestes componentes.

Protected

O Protected funciona como uma espécie de PRIVATE, mas sem ser tão radical. Os atributos ou métodos daquilo que é PROTECTED só serão enxergados por classes que estejam no mesmo pacote (package). Um pequeno exemplo:

```
package pai;
public class classepai {
		protected int variavel = 5;
}
```

A classe “classepai”, que pertence ao pacote “pai”, tem um atributo que é PROTECTED. Bom, até aqui, nada de mais.

```
package filho;
public class classefilho extends classepai{
		public static void main() {
			classepai p = new classepai();
			System.out.println(p.variavel); // ERRO: teste has protected access in pai.Pai
		}
}
```

A classe “classefilho” pertence ao pacote “filho”. Esse programa não compila. Se eu trocasse o pacote para “pai”, aqui, o programa compilaria e rodaria sem nenhum problema.

Pacotes

O uso de PACKAGES (pacotes) tem como função organizar as classes criadas em Java de forma muito semelhante ao que fazemos com os Módulos no Modula. Mas para quem só conhece Java, aqui vão as utilizações deste mecanismo:

1 – Criar um JAR

Pode-se criar um JAR com apenas as classes que estão no mesmo pacote, e isso é especialmente fácil de fazer no Eclipse. Com isso, podemos disponibilizar nossos programas em pacotes para o usuário baixar apenas alguns. Ou então, disponibilizar nossas classes prontas como um pacote.

Esta última utilidade é a mais legal de todas. Porque se eu criar um package chamado “utilidades”, e dentro dele colocar classes com funções variadas, outro programador poderá apenas anexar meu pacote JAR ao seu projeto e utilizar a fórmula IMPORT para puxar uma de minhas classes.

Como é que ele anexa meu JAR ao seu programa? Usando o comando Build Path, do Eclipse, que veremos mais adiante e não vem ao caso agora.

Digamos que ele queira importar a classe “apetrechos”, ele executará:

```
import utilidades.apetrechos;
```

E se ele quiser importar tudo de uma vez só, usará:

```
import utilidades.*;
```

2 - Organizar o programa em pedaços

No Eclipse, os Packages organizam nossas classes dentro do Package Explorer, o que nos permite organizar as classes em grupo segundo sua utilidade ou por critérios que podemos escolher livremente.

Tratamento de erros e exceções

Tratar erros no Java é muito simples. Utilizamos a expressão TRY e rodamos, dentro do bloco de Try (palavra que significa “tentar” em inglês). Somos obrigados a envolver rotinas que podem dar erros ou que provavelmente darão erro em algum momento.

Bons exemplos de rotinas nas quais o uso de TRY é obrigatório são, por exemplo, o acesso a banco de dados em um servidor remoto (uma situação na qual, mesmo que o código esteja perfeito, o programa pode dar problemas por causa de uma queda da rede, por exemplo), ou o acesso a arquivos de qualquer tipo (porque, novamente, mesmo com o programa estando perfeito ainda podem ocorrer problemas como o extravio do arquivo por ação de algum usuário desavisado).

Depois do TRY, usamos os CATCH (palavra que significa algo próximo de “agarrar”), porque precisamos saber quais tipos de erro são esperados. O Catch pega um certo tipo de erro ocorrido e então coloca este erro dentro de uma variável, que deve ser declarada na própria chamada do CATCH.

Depois, no fim, temos o FINALLY – indicando uma rotina que será rodada sempre, com ou sem erro. Normalmente é utilizada para colocar o programa de volta “dentro dos eixos” após a execução de uma rotina onde haja perigo de erros.

```
class errada {
   public static void main (String[]args) {
      int[]x = new int[4];
      for (int c = 0; c<10; c++) {
         try {
            x[c] = c;
         }
         catch(ArrayIndexOutOfBoundsException e) {
            System.out.println (e);
         }
         finally {
            System.out.println ("Terminou um ciclo.");
         }
      }
   }
}
```

Throw – informado os erros à classe que instanciou o objeto

A palavra Throw em inglês significa jogar, atirar alguma coisa. E Throws, é o verbo arremessar atribuído como se fosse na frase “Ele joga a pedra longe”. Utiliza-se isso para “jogar” o erro “para cima”, fazendo-o atingir a rotina que chamou o procedimento errado, não a própria rotina onde ocorre o erro.

Para que tudo fique mais claro, imagine a seguinte situação: Eu acabo de criar um programa que tem uma classe centralizando todo o trabalho lógico e os menus. E criei uma pequena classe que não faz nada, a não ser abrir um banco de dados.

Eu vou reutilizar a classe que abre banco de dados em todos os meus programas, para manter o padrão. Mas as mensagens de erro devem ser emitidas pela classe principal do programa, para que cada programa possa ter suas próprias mensagens.

Eu, então, faço como? Simples: a classe que abre banco de dados não vai tratar os erros que ocorrerem. Ela vai simplesmente arremessar esses erros para cima, deixando nas mãos da classe principal o tratamento do erro. E ela que trate-o como quiser.

A declaração do THROWS vai na classe onde o erro pode ocorrer, obrigando a classe que está “acima” a tratar o erro com um TRY e um CATCH.

```
 class errada2 {
   void rodar() throws ArrayIndexOutOfBoundsException {
       int[]x = new int[4];
       for (int c = 0; c<10; c++) {
          x[c] = c;
       }
    }
}

class progerrado {
    public static void main (String[]args) {
       errada2 x = new errada2();
       try {
          x.rodar();
       }
       catch (ArrayIndexOutOfBoundsException e) {
          System.out.println("Deu um erro");
       }
    }
}
```

Tipos de Excessão

Nos nossos exemplos acima, temos apenas um tipo de erro, o ArrayIndexOutOfBoundsException, que é bastante comum e familiar (e fácil de entender – o vetor está recebendo dados que estão além do seu tamanho). Mas temos inúmeros outros tipos de erro possíveis, o IOException, o SQLException, e por aí vai.

Se nós realmente não sabemos que tipo de erro uma determinada rotina poderá apresentar, podemos ter duas grandes certezas:

1) Estamos em apuros;
2) Precisamos tratar o erro mesmo assim.

Pensando no ponto número 2 da lista, usamos o tipo de erro Exception, que é a superclasse de todas as que falam sobre erros possíveis, prováveis ou imagináveis em Java.

Assim: catch (Exception e) {…

Escrevendo dados em arquivos textos e lendo eles de volta

Agora, vamos viajar um pouquinho no tempo. Há muitos, muitos, muitos anos atrás, todo mundo utilizava arquivos-texto para manipular dados. Era simples. Bastava criar um arquivo-texto no qual cada linha tivesse um dado específico, ou no qual uma linha comportasse toda a ficha de um funcionário, bem patrimonial ou o que seja. Depois, era preciso ler este arquivo-texto, em ordem, e obter os dados.

Agora, vou dar-lhe uma má notícia: este sistema de organização de dados ainda existe no governo brasileiro (este texto é de 2008). Diversos órgãos (como a Receita) recebem dados em formato texto, fornecendo aos desenvolvedores os padrões de escrita destes dados para que estes possam criar seus programas geradores de declarações e outras coisas.

Então, nós vamos, sim, ver este tipo de coisa. E chega a ser divertido, porque poderemos depois criar nosso próprio Bloco de Notas "paraguaio".

É no pacote Java.io que encontramos quase tudo o que precisaremos para fazer a criação e a leitura de arquivos binários ou de caracteres (texto).

Temos dois tipos de classes para usar. Um trabalha em nível de byte (binário), e é o dos Streams. O outro trabalha com caracteres, e são os Reader/Writer. Quase todas geram, possivelmente, erros do tipo IOException, obrigando-nos a tratar este tipo de erro.

A quantidade de classes tratando deste tipo de tarefa é muito grande. Mas nós usaremos apenas as mais utilizadas de todas as que existem hoje em dia. Não chegam a ser muito complicadas, e com elas é possível montar perfeitamente uma emulação de banco de dados em arquivos-texto.

```
import java.io.*;
class escritor {
   public static void main(String[]args) {
      try {
         FileWriter gravador = new FileWriter("arquivo.txt");
```

A primeira coisa que fazemos é instanciar a classe FileWriter, e declarar um nome de arquivo para o objeto dela (no caso, é o "gravador", com o arquivo "arquivo.txt"). Com isso, o programa cria um novo arquivo com o nome especificado. Eu poderia ter especificado um local (o nome de arquivo é como uma URL), mas não o fiz, portanto, o arquivo surgirá na própria pasta onde roda o programa.

```
         BufferedWriter saida = new BufferedWriter(gravador);
```

A segunda coisa é crier o BufferedWriter. Esta pequena maravilha, evolução de quase tudo o que havia anteriormente sobre leitura e gravação, nos permite gravar textos inteiros no arquivo aberto pelo FileWriter. Antigamente, usava-se escrever caracter por caracter, mas não com o Buffered.

```
         saida.write("Olá mundo \n");
         saida.write("Viva o jogo dos Canibais! ");
         saida.flush();
```

Os três comandos acima fazem o trabalho de graver um texto no arquivo aberto. Os saída.write escrevem texto em uma espécie de "fila de gravação", daquilo que irá para o arquivo assim que eu executar o comando FLUSH, aliás, mostrado na última das três linhas.

Alguns poderão perguntar: por quê ele usou o a expressão "\n" no final da primeira linha? Este sinal, chamado de "caracter de escape" (o que é meio bizarro, pois na verdade são dois caracteres) equivale ao sinal de "nova linha". É isso o que fica no arquivo quando alguém usa o ENTER em um programa

como, por exemplo, o EDIT do DOS ou os editores de texto mais simples para Windows.

```
      }
      catch(IOException e) {
         System.out.println("Não deu para escrever o arquivo");
      }
   }
}
```

E agora faremos um leitor para o arquivo que acabei de escrever:

```
import java.io.*;
class leitor {
   public static void main(String[]args) {
      try {
         FileReader gravador = new FileReader("arquivo.txt");
```

O File Reader é a classe que faz a leitura de arquivo, uma espécie de irmã da FileWriter, que vimos há pouco.

```
         BufferedReader entrada = new BufferedReader(gravador);
```

A classe irmã de BufferedWriter faz a leitura de um arquivo-texto.

```
         String linha;
         while ((linha = entrada.readLine())!=null) {
```

Aqui, vemos um momento de maior complexidade no programa (mas não muita). Vamos observar a condição contida neste WHILE: eu tenho uma operação, que é o linha = entrada.readLine(), no qual eu digo que a variável “linha” recebe uma linha do texto do arquivo que está aberto em “entrada” – a cada vez que esta rotina for rodada, o programa vai ler uma nova linha. Como eu usei um WHILE, ele vai percorrer o arquivo da primeira à última linha.

Depois, eu peço que o clico se repita enquanto o resultado desta operação não for NULL – a operação em si retorna um valor que é sempre positivo, mas quando não dá mais para ler o arquivo (porque ele chegou ao fim), retorna NULL, encerrando o clico do WHILE.

```
            System.out.println(linha);
         }
      }
      catch(IOException e) {
         System.out.println("Não deu para ler o arquivo");
      }
   }
}
```

Tokenizer (caracteres separadores)

Para podermos criar uma espécie de banco de dados no arquivo-texto, precisamos de separadores para distinguir os dados entre si. Para isso, usamos a classe StringTokenizer, com a qual determinamos um "token" (que pode ser uma vírgula, um ponto, dois caracteres, o que eu quiser) e este token vai separar os campos de dados.

No caso, imaginemos que eu criei um arquivo-texto (no Bloco de Notas) contendo apenas o seguinte texto:

```
Vodka; Russia; Forte
Canha; Brasil; Forte
Vinho; Egito; Medio
Cerveja; Sumeria; Fraca
```

O nome do arquivo é "arquivo.txt". Agora, vou fazer um programinha que lê estes dados e os exibe na tela:

```
import java.io.*;
import java.util.*;
class leitor {
   public static void main(String[]args) {
      try {
         FileReader gravador = new FileReader("arquivo.txt");
         BufferedReader entrada = new BufferedReader(gravador);
         String linha;
         while ((linha = entrada.readLine())!=null) {
```

Até aqui, eu operei apenas com o que já vimos anteriormente: um Reader e nada mais.

```
            StringTokenizer tok = new StringTokenizer(linha, ";");
```

Agora, eu estou pegando a linha de texto que havia sido capturada na variável "linha" e estou quebrando ela em pedaços, usando o ponto-e-vírgula como sinal de quebra. Logo abaixo, eu vou usar o método "nextToken()" para fazer o meu sistema percorrer todos os tokens que existem nesta linha de texto.

```
            System.out.println("Bebida: " + tok.nextToken());
            System.out.println("Origem: " + tok.nextToken());
            System.out.println("Grau de alcool: " + tok.nextToken());
            System.out.println("----------------------");
         }
      }
      catch(Exception e) {
         System.out.println("Não deu para escrever o arquivo");
      }
   }
}
```

Métodos da classe StringTokenizer:

boolean hasMoreTokens() // retorna true se existe mais tokens

String nextToken() // retorna próximo token; lança um NoSuchElementException se não houver mais tokens.

String nextToken(String delim) // o novo grupo de delimitador é usado

int countTokens() // retorna o número de tokens

Tocando sons em Java

Para tocarmos sons em um programa em Java, precisamos fazer um processo um pouco longo. Mas o resultado vale a pena, e pode-se fazer o computador executar o som enquanto executa outras coisas, tornando a música ambiente.

A idéia é: primeiro, abrimos um arquivo como se fôssemos ler dele alguma informação (e vamos). Depois, existe uma classe chamada AudioStream, que pega este arquivo aberto e lê ele como arquivo de som, decodificando-o. Então, podemos fazer o som tocar normalmente.

```
import  sun.audio.*;
import  java.io.*;

public class Tocador{
```

Primeira coisa, importar as classes que comandam esta questão de som: a "sun.audio" e a velha "java.io", para a leitura do arquivo – pois o som é um arquivo.

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

A próxima coisa que fazemos é abrir o arquivo de som, e este processo é igual ao da abertura do arquivo de texto que fazíamos em uma aula anterior.

```
                try {
                        InputStream arq = new FileInputStream("aha.wav");

                        AudioStream som = new AudioStream(arq);
```

Depois, vem a novidade: abrir um objeto da classe AudioStream usando o arquivo.

```
                        AudioPlayer.player.start(som);
```

Daí, temos o comando AudioPlayer.player.start(som), usado para fazer o som começar a tocar.

```
                }
                catch(Exception e) {
                        System.out.println("Deu pau");
                }
        }
}
```

Podemos, depois, usar um comando para fazer o som parar de tocar:

```
AudioPlayer.player.stop(som);
```

Uma outra grande jogada deste sistema é que é possível tocar dois sons ao mesmo tempo. Basta criar dois objetos do tipo InputStream e dois AudioStream. Com isso, podemos colocar uma música e uma batida de fundo para tocarem juntas.

Ou ainda, criar:

Um conjunto de InputStream e AudioStream com uma música ambiente.

Um conjunto de InputStream e AudioStream com o barulho de espadas se batendo.

Um conjunto de InputStream e AudioStream com o grito de um monstro.

Um conjunto de InputStream e AudioStream com o som de uma flecha voando.

E no final, criar um jogo de RPG com música de fundo, espadas e flechas que fazem barulho, e monstros que rosnam.

Pronto!

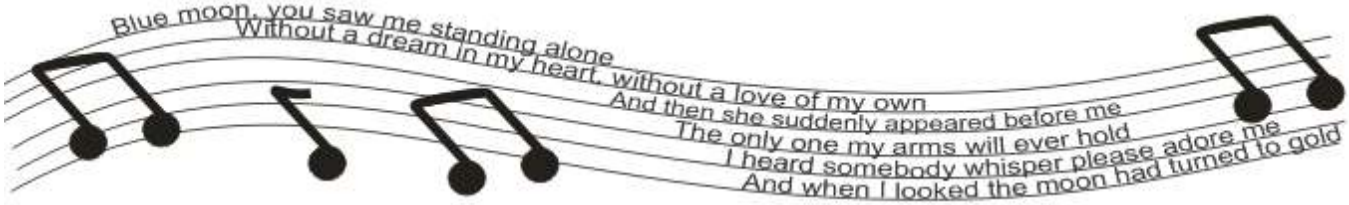
Agora você já pode aterrorizar os seus clientes, amigos e até familiares dando a eles um sistema que não apenas organiza seus dados de qualquer natureza, mas também os faz pensar na possibilidade do suicídio ao escutar uma versão modorrena de "La Vie em Rose", um experimento psicológico ainda não concluso, conduzido nas aulas de Java pelo lendário Bruno, no Senac!

Ou ainda...

Pode animar o trabalho da Contabilidade, que é muito monótono, fazendo o primeiro sistema contábil com trilha sonora tocada pelo **Sha Na Na**!

Ou melhor ainda...

Faça uma junção de músicas e coloque uma banda impensável para tocar: por quê escolher? Bote **juntos** Edith **Piaf** **COM** os caras do **Sha Na Na!**



A classe Console

Bom. Pessoal, eu sei que está ficando difícil de programar em Java, no modo texto, sem ter como digitar nada na tela.

Então, eu tenho aqui uma classe que nós sempre usamos dentro dos projetos e que se chama Console. Ela tem uns métodos estáticos que lidam com possibilidades de erro e tudo mais, permitindo uma integração bem legal do teclado com um programa.

```
/**
  An easy interface to read numbers and strings from
  standard input
  @version 1.10 10 Mar 1997
  @author Cay Horstmann
*/

public class Console
{

  public static void printPrompt(String prompt)
  { System.out.print(prompt + "Digita aí: ");
   System.out.flush();
  }

  public static String readLine()
  { int ch;
   String r = "";
   boolean done = false;
   while (!done)
   { try
     { ch = System.in.read();
      if (ch < 0 || (char)ch == ' ')
        done = true;
      else if ((char)ch != ' ') // weird--it used to do translation
        r = r + (char) ch;
     }
     catch(java.io.IOException e)
     { done = true;
     }
   }
   return r;
  }

  /**
   read a string from the console. The string is
   terminated by a newline

   @param prompt the prompt string to display
   @return the input string (without the newline)
  */

  public static String readLine(String prompt)
  { printPrompt(prompt);
   return readLine();
  }
```

```
 public static int readInt(String prompt)
 { while(true)
  { printPrompt(prompt);
    try
    { return Integer.valueOf
       (readLine().trim()).intValue();
    } catch(NumberFormatException e)
    { System.out.println
       ("Not an integer. Please try again!");
    }
  }
 }

  public static double readDouble(String prompt)
 { while(true)
  { printPrompt(prompt);
    try
    { return Double.parseDouble(readLine().trim());
    } catch(NumberFormatException e)
    { System.out.println
    ("Not a floating point number. Please try again!");
    }
  }
 }
}
```

Para usar esta classe no seu código, é fácil. Basta adicioná-la ao seu projeto e então usar seus métodos para carregar variáveis.

Exemplos:

String texto = Console.readLine(“Digite aqui:”);

Sendo que o argumento passado é o prompt que o usuário vê. Usando este método, seu programa em modo texto vai parar neste ponto, aguardando a digitação por parte do usuário.

De forma semelhante, use os métodos:

double x = Console.readDouble();

int x = Console.readInt();

E por aí vai.

Sobrepondo métodos – uma técnica essencial

Uma das coisas fundamentais sobre Java é que tudo o que existe na linguagem, ou quase tudo, são classes das quais criamos métodos. Não existe, portanto, nenhuma saída para quem não sabe trabalhar com classes.

Uma das coisas mais legais que há no Java é que nós podemos pegar uma classe que existe com certos métodos, e podemos criar objetos que não obedecem a estes métodos – mas os sobrepõe, não criando uma nova classe com a modificação, e sim criando um objeto que tem a modificação em si.

Eu vou dar um exemplo, Vou criar uma classe Pessoa que só vai possuir um método, para fins de exemplo.

```
class Pessoa {
	void gritar() {
		System.out.println(“Socorro!”);
	}
}
```

Agora, eu vou criar uma classe principal que vai criar uma Pessoa e então pedir que ela grite.

```
class prog {
	public static void main(String[]args) {
		Pessoa P1 = new Pessoa();
		P1.gritar();
	}
}
```

A pessoa P1 vai gritar “Socorro!”, e até aí morreu Neves, porque nós já vimos isso antes. Mas agora eu vou criar uma segunda pessoa, que terá seu próprio método “gritar()”, diferente do padrão das outras pessoas.

```
class prog {
	public static void main(String[]args) {
		Pessoa P1 = new Pessoa();
		P1.gritar();
		Pessoa P2 = new Pessoa () {
                                                        void gritar() {
                                                                System.out.println("Shoryuken!");
                                                        }
                                };
		P2.gritar();
	}
}
```

Interface gráfica em Java

Cansado de programar Java e só ver letrinhas pretas sobre um fundo branco?

Está com medo do seu futuro profissional por saber apenas desenvolver aplicações Java com interface de DOS?

Sente inveja do seu vizinho que faz programas em Java com telas cheias de campos e botões?

SEUS PROBLEMAS ACABARAM!

Vamos trabalhar agora a interface gráfica do seu programa em Java! Leia as próximas páginas e sua vida será iluminada!

Bibliotecas AWT e SWING

São as fontes das classes que utilizaremos a partir daqui. Nelas moram os componentes gráficos para Java, e por isso devemos importá-las no início de nossas classes gerenciadoras de telas:

```
import javax.swing.*;
import java.awt.*;
```

Nossa primeira tela

Pode parecer estranho para quem vem de outras linguagens, como o Delphi ou o VB, mas em Java, cada tipo de componente gráfico é uma classe com atributos e métodos próprios. E para utilizá-las, devemos instanciá-las.

A primeira e mais simples chama-se JFrame, e cria telas retangulares sem nada dentro, um espaço onde trabalharemos logo depois.

```
import javax.swing.*;
import java.awt.*;
class classetela {
   public static void main (String[]args) {
      JFrame tela = new JFrame("Primeira tela");
      tela.setBounds(100,100,600,400);
```

O método SetBounds trabalha com a localização e tamanho da tela, sendo a seguinte ordem: distância do canto esquerdo da tela, distância do topo da tela, largula e altura.

```
tela.getContentPane().setBackground(Color.RED);
```

Aqui, invocamos o método que verifica qual é o campo de Content do JFrame, ou seja, a região onde não está o menu nem as bordas, e muda sua cor a partir do método set Background. Eu usei uma biblioteca de cores, a Color, e defini a janela como vermelha; Poderia ter usado azul, cinza, verde, etc.

```
tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Aqui, eu determino qual a ação a executar quando este JFrame for fechado. No caso, ele fecha o programa todo, num efeito de BREAK. A instrução DISPOSE_ON_CLOSE apenas levará o JFrame a fechar sem derrubar o programa todo.

```
tela.setLayout(null);
```

Aqui, escolhi um Layout para o meu JFrame. Ou melhor, escolhi que seja nenhum. Existem Layouts prontos para JFrames, onde os botões ficam dispostos em forma de fileira, ou nos cantos, ou expandidos. A maioria tem um resultado grotesco. Se quisermos desenhar nossa tela com liberdade para fazê-la bonita, temos que escolher o NULL;

```
tela.setVisible(true);
```

Depois de preparado, o JFrame pode já ficar visível. Este passo vem no final de toda a sua montagem.

```
  }
}
```

JLabel

É um pequeno componente com o qual colocamos pedaços de texto sem nenhuma função, a não ser a de indicar para quê servem os campos de uma tela qualquer.

```
import javax.swing.*;
import java.awt.*;
class classetela {
   public static void main (String[]args) {
      JFrame tela = new JFrame("Primeira tela");
      JLabel texto = new JLabel("Frase indicativa");
      tela.setBounds(100,100,600,400);
      tela.getContentPane().setBackground(Color.RED);
      tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      texto.setBounds(100,100,200,30);
      tela.setLayout(null);
      tela.add(texto);
      tela.setVisible(true);
   }
}
```

O código acima não apresenta grandes novidades. Apenas instanciamos o JLabel, dando-lhe um texto qualquer para exibir. Depois, demos o mesmo setBounds nele, e em seguida fizemos a ação realmente revolucionária:

```
      tela.add(texto);
```

Adicionamos o objeto “texto” ao objeto “tela”. Note que o setBounds do “texto” passa a usar como bordas as bordas do JFrame. E fica contido nele.

O JLabel pode ter desenhos por dentro, mas veremos isso mais adiante, quando tratarmos do Icon, outro tipo de objeto Java.

JButton

O JButton é, finalmente, o botão. Colocamos ele nas telas com inúmeras funções. Estas funções serão tratadas mais adiante, quando chegarmos às aulas sobre Eventos. Por enquanto, basta-nos desenhar os botões.

O JButton poderá ser instanciado de três formas diferentes:

JButton() – aqui, teremos depois que rodar um método setText("algum texto") para dar-lhe um conteúdo (um conteúdo que aparece para o usuário).

JButton(String) – aqui, escrevemos algum texto para aparecer no botão e o usuário saber do que se trata.

JButton(icon) – neste caso, adicionamos uma figura ao botão para dar-lhe uma aparência personalizada.

Veja este código:

```
import javax.swing.*;
import java.awt.*;
class classetela {
   public static void main (String[]args) {
      JFrame tela = new JFrame("Primeira tela");
      JLabel texto = new JLabel("Frase indicativa");
      tela.setBounds(100,100,600,400);
      tela.getContentPane().setBackground(Color.YELLOW);
      tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      texto.setBounds(100,100,200,30);
      tela.setLayout(null);
      tela.add(texto);

      JButton botao= new JButton("Pressione aqui");
      botao.setBounds(100,200,200,30);
      botao.setBackground(Color.GREEN);
      botao.setForeground(Color.BLUE);
      botao.setToolTipText("Um exemplo de ToolTip");
      tela.add(botao);

      tela.setVisible(true);
   }
}
```

No exemplo acima, eu usei praticamente a mesma tela criada antes, mas adicionei as rotinas de criação de um botão. Primeiro, instanciei a classe, e usei os métodos já conhecidos setBounds e setBackground.

Em seguida, usei o setForeground, com o qual determino a cor da letra do texto do botão. E logo abaixo, determinei o texto da ToolTipText, ou seja, o texto que aparece na caixinha amarela de ajuda quando passamos o mouse sobre o botão.

Na seqüência, apenas usei o ADD para colocar o botão na "tela" e dei a tradicional ordem para que ela ficasse visível.

JTextField

O JTextField é uma caixa utilizada para que o usuário possa escrever um texto de uma linha só. Trata-se de um componente bastante simples, que tem basicamente as mesmas especificações dos que acabamos de ver: setBounds, setBackground, setForeground, e o ADD da tela sobre ele.

Um exemplo do código anterior, enriquecido com um JTextField:

```
import javax.swing.*;
import java.awt.*;
class classetela {
   public static void main (String[]args) {
      JFrame tela = new JFrame("Primeira tela");
      JLabel texto = new JLabel("Frase indicativa");
      tela.setBounds(100,100,600,400);
      tela.getContentPane().setBackground(Color.YELLOW);
      tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      texto.setBounds(100,100,200,30);
      tela.setLayout(null);
      tela.add(texto);

      JButton botao= new JButton("Pressione aqui");
      botao.setBounds(100,200,200,30);
      botao.setBackground(Color.GREEN);
      botao.setForeground(Color.BLUE);
      botao.setToolTipText("Um exemplo de ToolTip");
      tela.add(botao);

      JTextField caixa = new JTextField();
      caixa.setBounds(100,300, 200,30);
      caixa.setBackground(Color.RED);
      caixa.setForeground(Color.WHITE);
      tela.add(caixa);

      tela.setVisible(true);
   }
}
```

O tratamento do texto de um JTextField dar-se-á por meio de dois métodos bastante simples:

getText() – servirá para capturar o quê o usuário digitou no campo.

setText() – servirá para determinarmos que texto deve ir no campo. Muito útil quando se abre um registro de um banco de dados e o campo JTextField deve receber um campo de registro para alteração.

JFormattedTextField

São campos parecidos com a JTextField - o usuário pode escrever dentro normalmente. Só que agora o número de caracteres digitáveis é limitado, pode-se limitar o tipo de catacter digitado, e pode-se fazer o campo já esperar com alguns sinais prontos. É isso o que se usa nos campos onde uma pessoa deve digitar seu CPF e a caixinha de texto já vem com os pontos e o traço, por exemplo. Vamos ver como se faz, é bem fácil.

```
package Componentes;
import java.awt.event.*;
import javax.swing.*;

public class rodar {
	public static void min(String[]args) {
		JFrame tela = new JFrame();
		tela.setBounds(0,0, 800,600);

		try{
			txtCPF = new JFormattedTextField(new MaskFormatter("###.###.###-##"));
			txtData = new JFormattedTextField(new MaskFormatter("##/##/####"));
			txtCNPJ = new JFormattedTextField(new MaskFormatter("###.###.###/####-##"));
		}
		catch (Exception erro){
			JOptionPane.showMessageDialog(null,"Essa porcaria deu problema");
		}
```

Este programa começa normalmente com a criação da JFrame e tudo mais. E ele termina normalmente. Na verdade, seria mais um programa banal, desses que desenvolvemos todos os dias, não fosse pelo fato de eu ter adicionado uns objeto novos à tela. Esses objetos novos são como JTextFields. Só que com uma máscara.

A máscara, no caso é um MakFormatter, um objeto de uma classe criada apenas com o fim de associar-se a JFormattedTextFields e formar um campo com parâmetros para digitação.

No caso, o formato da Dat, por exemplo, já mostrará a TextField com as barras. Já os sinais de sustenido ficarão em branco, aguardando que o usuário digite números - EU DISSE NÚMEROS! NÃO LETRAS!

Se você quiser que o usuário possa digitar o que quiser em uma determinada posição, basta colocar um asterisco ao invés do sustenido.

Assim como a JTextField, sua variante Formatted precis ser adicionada à tela. Então, vamos acabar logo esse programa.

```
		tela.add(txtCPF);
		tela.add(txtData);
		tela.add(txtCNPJ);

		tela.setLayout(null);
		tela.setDefaultCloseOperation(JInternalFrame.DISPOSE_ON_CLOSE);
		tela.setVisible(true);
	}
}
```

JToggleButton

É uma variação bem básica do JButton. Trata-se de um botão que, uma vez pressionado, fica "afundado", demonstrando que tal opção está selecionada ou que tal funcionalidade está ativada.

```
import javax.swing.*;
import java.awt.*;
class classetela {
   public static void main (String[]args) {
      JFrame tela = new JFrame("Primeira tela");
      JLabel texto = new JLabel("Frase indicativa");
      tela.setBounds(100,100,600,400);
      tela.getContentPane().setBackground(Color.YELLOW);
      tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      texto.setBounds(100,100,200,30);
      tela.setLayout(null);
      tela.add(texto);

      JButton botao= new JButton("Pressione aqui");
      botao.setBounds(100,200,200,30);
      botao.setBackground(Color.GREEN);
      botao.setForeground(Color.BLUE);
      botao.setToolTipText("Um exemplo de ToolTip");
      tela.add(botao);

      JTextField caixa = new JTextField();
      caixa.setBounds(100,300, 200,30);
      caixa.setBackground(Color.RED);
      caixa.setForeground(Color.WHITE);
      tela.add(caixa);

      JToggleButton toglebotao= new JToggleButton("Toggle Button");
      toglebotao.setBounds(300,200,200,30);
      toglebotao.setBackground(Color.GREEN);
      toglebotao.setForeground(Color.BLUE);
      toglebotao.setToolTipText("Um exemplo de ToolTip");
      tela.add(toglebotao);


      tela.setVisible(true);
   }
}
```

O uso do JToggleButton torna-se possível por causa do método isSelected(), que retorna True ou False e nos diz se o botão está "afundado" ou não.

Icon

A classe Icon tem um uso bastante simples, mas muito importante. Com ela, podemos carregar uma imagem (devidamente criada em um arquivo JPG ou PNG), para dentro do programa, utilizando esta imagem como quisermos. O comando para isso é bastante simples. Basta instanciar a classe Icon criando uma ImageIcon que carregará o arquivo, cujo nome ou localização completa

Comando é assim:

```
Icon imagem = new ImageIcon("arquivo.jpg");
```

Existe ainda uma funcionalidade que muita gente desconhece: O formato de imagem PNG, que é capaz de armazenar instruções sobre transparências, quando lido, acaba numa imagem com transparências, pois o Java consegue respeitar bem estes parâmetros.

Depois de carregarmos uma imagem para dentro de uma variável, podemos atribuí-la a uma série de objetos através do método setIcon.

Exemplos:

```
JButton botao = new JButton();
botão.setIcon(imagem);

JLabel rotulo = new JLabel();
rotulo.setIcon(imagen);
```

Um detalhe importante a notar é que, quando atribuímos o Icon a um JButton, o botão ganha o desenho mas permanece com o aspecto "de botão", mudando a cor da borda quando colocamos o mouse sobre ele.

Já quando atribuímos o Icon a uma JLabel, a imagem fica parecendo solta no ar entro da tela. E aí, a transparência tem realmente sua função efetiva, uma vez que o fundo atrás da JLabel aparece plenamente.

É possível fazer botões redondos, triangulares ou com qualquer forma, usando JLabel, simplesmente desenhando o botão, colocando em uma JLabel e, na função do botão, incluir um comando para mudar o Icon da JLabel, trocando-o por uma figura do mesmo botão, mas com aparência de "pressionado".

JOptionPane

JOptionPane é o componente que cria as famosas caixinhas de mensagem para o usuário. Com ele, é possível fazer não apenas mensagens na tela mas também pedir dados, dar opções (do tipo Ok, Cancel), e fazer algumas outras operações.

Uma das facilidades de se usar o JOptionPane é que seus métodos são estáticos (static), então nem precisamos criar um objeto para executar nada. Usamos a classe, direto.

Um exemplo bem simples:

```
import javax.swing.*;
import java.awt.*;
class optiones {
   public static void main(String[]args) {
      JOptionPane.showMessageDialog(null, "Uma mensagem!", "Titulo", JOptionPane.ERROR_MESSAGE);
   }
}
```

No exemplo acima, chamamos o método showMessageDialog, que mostra apenas uma mensagem simples com um botão de OK embaixo. Os argumentos utilizados são:

NULL, colocado num campo onde devemos informar o objeto “pai” do JOptionPane. Este objeto “pai” (parent) ficará congelado embaixo da caixa de mensagem até que o usuário clique no OK.

Já o segundo argumento, fácil de entender, é a mensagem em si, uma String sem grande mistério.

O terceiro é o título da caixa de mensagem, aquele que vai no alto da caixa em sua barra superior, ao lado do botão de Fechar.

O último argumento é o tipo de mensagem que queremos exibir. Conforme o tipo, aparecerá um ícone ao lado do nosso texto.

JOptionPane com ícone próprio

Eu poderia ainda acrescentar um quinto argumento, colocando ali um objeto do tipo Icon. Daí, eu teria uma mensagem com minha própria figurinha de erro.

A sintaxe ficaria assim:

```
Icon figura = new ImageIcon ("foto.jpg");

JOptionPane.showMessageDialog(null, "Errou, seu imbecil!", "Erro", JOptionPane.PLAIN_MESSAGE, figura);
```

Tipos de mensagem:

JOptionPane.ERROR_MESSAGE	Mensagem de Erro, do tipo com o X na bolinha vermelha. Usando em caso de erros, normalmente, tratamento de exceções.
JOptionPane.INFORMATION_MESSAGE	Usado em mensagens do tipo Help, aparece com um pequeno I típico de placa da informações.
JOptionPane.WARNING_MESSAGE	Mensagens de aviso. Aparece um triângulo com um ponto de exclamação dentro.
JOptionPane.QUESTION_MESSAGE	Vem com um ponto de interrogação.
JOptionPane.PLAIN_MESSAGE	Aqui, não tem nada desenhado. Usado para casos nos quais queremos uma mensagem só com texto.

Além do método showMessageDialog, temos o showConfirmDialog, o showInputDialog e o showOptionDialog. Vejamos então cada um deles.

showConfirmDialog

Muito utilizado para operações do tipo “confirme ou cancele o que acabou de pedir”, é o melhor remédio contra usuários desavisados que clicam nas opções da tela a esmo.

```
import javax.swing.*;
import java.awt.*;
class optiones {
   public static void main(String[]args) {
         int opcao = JOptionPane.showConfirmDialog(null, "Você quer mesmo ler minha mensagem?", "Pergunta besta",
JOptionPane.YES_NO_CANCEL_OPTION);

      if (opcao == JOptionPane.OK_OPTION) {
         JOptionPane.showMessageDialog(null, "Você escolheu ler a mensagem!", "Parabéns!",
JOptionPane.INFORMATION_MESSAGE);
      }

      else if (opcao == JOptionPane.NO_OPTION) {
         JOptionPane.showMessageDialog(null, "Você é um imaturo!", "Porcaria!", JOptionPane.ERROR_MESSAGE);
      }

      else if (opcao == JOptionPane.CANCEL_OPTION) {
         JOptionPane.showMessageDialog(null, "Você ficou indeciso e resolveu recuar! É um fraco?", "Covarde!",
JOptionPane.QUESTION_MESSAGE);

      }
   }
}
```

No código acima, nossa pergunta é feita através de um JOptionPane.showConfirmDialog, cujos argumentos mais uma vez são o objeto “pai”, a mensagem, o título e o tipo de mensagem.

Só que o tipo de mensagem aqui é JOptionPane. YES_NO_CANCEL_OPTION. Com isso, eu obtive três botões, o Yes, o No, e o Cancel. Além do YES_NO_CANCEL, temos outras modalidades de pergunta:

YES_NO_CANCEL_OPTION	Botões Yes, No e Cancel
YES_NO_OPTION	Botões Yes e No
OK_CANCEL_OPTION	Botões OK e Cancel
PLAN_MESSAGE	Botão OK apenas, deixando igual ao showMessageDialog

showOptionDialog

Este método é, sem sombra de dúvidas, o mais legal de todos da coleção JOptionPane. Com ele, podemos fazer nossos próprios botões, em número indeterminado.

```
import javax.swing.*;

import java.awt.*;
class optiones {
   public static void main(String[]args) {

                        String[]botoes = {"Abort", "Retry", "Fail", "CTRL+ALT+DEL"};

                        int opcao = JOptionPane.showOptionDialog(null, "Escolha uma opção", "Sem saída",
JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE, null, botoes, botoes[0]);

            JOptionPane.showMessageDialog(null, botoes[opcao], "Opção escolhida", JOptionPane.WARNING_MESSAGE);


   }
}
```

No código acima, temos o seguinte:

Primeiro, eu criei um vetor de Strings chamado “botoes”. Nele, guardei uma série de textos para usar dentro dos botões da minha caixa de opções.

Depois, eu declarei uma variável int chamada “opcao”, onde ficará guardada a escolha que o usuário fizer. A esta variável, atribuí diretamente o valor da operação de JOptionPane.showOptionDialog. Veja que ele tem um montão de argumentos. Na ordem, temos: o objeto “pai”, a mensagem, o título, o tipo de opção (coloca-se o Default porque este atributo será ignorado), depois vem o tipo de mensagem (no caso, uma QUESTION_MESSAGE), em seguida vem um ícone (eu marque Null, mas poderia ter colocar um objeto do tipo Icon aqui), e logo depois vem o nome do vetor de onde tiramos os botões (no caso, o vetor “botoes" que eu acabara de criar), e em seguida, a opção que vem destacada como “default” (no caso, coloquei a primeira).

Na linha final do programa, eu simplesmente mandei exibir uma JOptionPane.showMessageDialog com os seguintes argumentos:

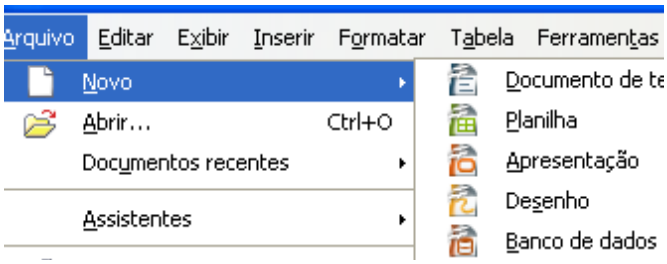
Componente “pai”: null.

Mensagem: o texto do vetor “botoes", no índice que corresponde à opção clicada (ou seja, vai aparecer o texto do botão clicado).

Título: “Opção escolhida”

Tipo de mensagem: “WARNING_MESSAGE”

showInputDialog



Este método é simplíssimo. Ele exibe na tela uma caixinha de mensagem com um campo de texto para o usuário colocar alguma coisa. Esta texto será carregado em uma String, que deve ser logo colocada como variável para a qual vamos atribuir o resultado.

Um código de exemplo:

```
import javax.swing.*;

import java.awt.*;
class optiones {
   public static void main(String[]args) {

                        String texto = JOptionPane.showInputDialog(null, "Escreva seu nome aí", "Pegadinha",
JOptionPane.QUESTION_MESSAGE);

            JOptionPane.showMessageDialog(null, texto, "O otário do ano", JOptionPane.INFORMATION_MESSAGE);


   }
}
```

Este código simplíssimo apenas pega o nome do usuário e depois o exibe como “o otário do ano”, numa premiação de mérito duvidoso.

JMenuBar, JMenu e JMenuItem

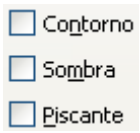
Estas classes são utilizadas sempre em conjunto, para criar um menu na parte superior do JFrame, e deixar seu software com aquela facilidade de uso que os usuários modernos exigem.

Aqui, começamos a trabalhar uma modalidade de programação que costumo chamar informalmente de "empilhamento de objetos", pois os objetos só funcionam quando colocamos um dentro do outro em uma espécie de pirâmide.

O objeto que serve de base é o JMenuBar, adicionado diretamente ao JFrame, e sobre ele vão todas as coisas. Ele é a barra clara no alto do JFrame, dentro da qual os menus ficarão. Depois, temos os JMenu, que ficam dentro da MenuBar e podem, inclusive, ficar um dentro do outro. Dentro dos JMenu, ficam os JMenuItem, que têm uma função cada um. Para ativar suas funções, utilizaremos os Listeners, mais adiante. Deixemos isso para outro capítulo.

```
import java.awt.*;
import javax.swing.*;
class menus {
	public static void main(String[]args) {
		JFrame tela = new JFrame();
		tela.setBounds(100,100,800,600);

		JMenuBar menuzao = new JMenuBar();

		JMenu menuarquivo = new JMenu("Arquivo");
		JMenu menuhelp = new JMenu("Help");
		JMenu menudentro = new JMenu("Mais opções");


		JMenuItem itemnovo = new JMenuItem("Novo");
		JMenuItem itemsalvar = new JMenuItem("Salvar");
		JMenuItem itemajuda = new JMenuItem("Ajuda");
		JMenuItem itemsobre = new JMenuItem("Sobre");
		JMenuItem itempanico = new JMenuItem("CTRL+ALT+DEL");
		JMenuItem itemterror = new JMenuItem("Tela Azul da Morte");

		menuarquivo.add(itemnovo);
		menuarquivo.add(itemsalvar);
		menuhelp.add(itemajuda);
		menuhelp.add(itemsobre);

		menudentro.add(itempanico);
		menudentro.add(itemterror);

		menuzao.add(menuarquivo);
		menuzao.add(menuhelp);

		menuarquivo.add(menudentro);

		tela.setJMenuBar(menuzao);
		tela.setVisible(true);
	}
}
```

JCheckBox



Uma JCheckBox é um daqueles componentes em que vemos uma caixinha quadrada com uma frase ao lado, e podemos marcar ou desmarcar a caixinha quadrada. As CheckBoxes não são usadas para fazer escolhas com apenas uma opção válida, podendo ser todas marcadas ou todas desmarcadas livremente, desde que o programador não coloque restrições no próprio código.

```
import java.awt.event.*;
import javax.swing.*;
public class checa {
        public static void main(String[]args) {
                JFrame tela = new JFrame("Checks!");
                tela.setBounds(10,10,800,600);
                tela.setLayout(null);

                final JCheckBox pizza = new JCheckBox("Quero uma Pizza.");
                final JCheckBox refri = new JCheckBox("Quero um refri.");

                pizza.setBounds(100,30,200,35);
                refri.setBounds(100,80,200,35);

                pizza.setSelected(true);

                tela.add(pizza);
                tela.add(refri);
                tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                tela.setVisible(true);
        }

}
```

Nesta parte inicial nada de novo: criamos um JFrame comum para receber os novos componentes.

Aqui, criamos os JCheckBoxes, dando-lhes, no ato de instanciá-los, já o texto que terão. E logo depois, usamos o setBounds para dar-lhes localização e tamanho.
O método setSelected, cujo argumento é um boolean, diz se a caixinha vem marcada ou não.

Depois, para termos um retorno do valor de cada JCheckBox, utilizamos o método isSelected(), que retorna um valor boolean. Assim:

```
boolean x = pizza.isSelected();
```

Ou ainda:

```
If (pizza.isSelected()) {
        - Funções a serem executadas.
}
```

Na página seguinte, temos um código de exemplo sobre como utilizar a CheckBox e ter seu retorno através de um botão ao qual foi adicionado um ActionListener.

Você vai ver ActionListeners ao longo dos nossos exemplos, então, para não ficar confuso demais, por favor dê uma lida no capítulo que se chama "Eventos (Listeners)".

Exemplo de código com uma JCheckBox retornando valor

```
import java.awt.event.*;
import javax.swing.*;

public class radio {
	public static void main(String[]args) {
		JFrame tela = new JFrame("Checks!");
		tela.setBounds(10,10,800,600);
		tela.setLayout(null);p

		final JCheckBox pizza = new JCheckBox("Quero uma Pizza.");
		final JCheckBox refri = new JCheckBox("Quero um refri.");

		pizza.setBounds(100,30,200,35);
		refri.setBounds(100,80,200,35);

		pizza.setSelected(true);

		JButton botao = new JButton("Fazer pedido");
		botao.setBounds(100,150,200,35);
		botao.addActionListener(new ActionListener() {

			public void actionPerformed(ActionEvent e) {
				if (pizza.isSelected()) {
					JOptionPane.showMessageDialog(null, "O sujeito quer uma pizza.", "Pedido",
JOptionPane.INFORMATION_MESSAGE);
				}
				if (refri.isSelected()) {
					JOptionPane.showMessageDialog(null, "O sujeito quer um refrigerante.", "Pedido",
JOptionPane.INFORMATION_MESSAGE);
				}
			}

		});

		tela.add(botao);

		tela.add(pizza);
		tela.add(refri);
		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		tela.setVisible(true);
	}
}
```

Voltamos a proceder com a criação da tela e dos JCheckBoxes como no exemplo anterior.

Aqui, criamos um JButton, em um esquema igual ao que já vimos anteriormente na parte dos Eventos (ActionListener, MouseListener, etc.). Note que, quando o JCheckBox "pizza" retornar TRUE no seu método isSelected(), o programa exibe uma mensagem.

Eu utilizei dois IFs, independentes, sem ELSE, para que quando o usuário marcar apenas uma opção, apareça só aquela opção, e quando ele marcar as duas, "pizza" e "refri", me apareçam duas mensagens consecutivas. É claro que isso é só um exemplo de programa. Na realidade, devemos concatenar duas Strings para formar a lista do pedido, ou gravar os dados em um Banco de Dados, ou fazer alguma outra coisa mais útil e funcional.

JRadioButton

O JRadioButton funciona apenas quando pensamos nele inserido em um grupo. Eu posso colocar quantas opções quiser, formando um menu, e se todos os meus JRadioButtons estiverem no mesmo grupo, o usuário só poderá marcar um deles. Ao marcar um, os outros perdem a marcação.

```
import java.awt.event.*;
import javax.swing.*;

public class radio {
	public static void main(String[]args) {
		JFrame tela = new JFrame("Radios!");
		tela.setBounds(10,10,800,600);
		tela.setLayout(null);

		JRadioButton designer = new JRadioButton("Eu sou WebDesigner!");
		JRadioButton programador = new JRadioButton("Eu sou programador Java!");
		designer.setBounds(20,20,250,35);
		programador.setBounds(20,60,250,35);

		ButtonGroup grupo = new ButtonGroup();
		grupo.add(designer);
		grupo.add(programador);

		tela.add(designer);
		tela.add(programador);

		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		tela.setVisible(true);
	}

}
```

Eu começo como quem não quer nada: criando um JFrame sem nenhuma novidade. Nenhuma, mesmo.

Logo depois, instancio os JRadioButton, já determinando seu texto de apresentação. Abaixo, uso o tradicional setBounds para posicionar e dimensionar cada um deles.

Mais abaixo, vem a verdadeira novidade: eu sou obrigado a criar um ButtonGroup para que os JRadioButtons saibam que são “irmãos” e comportem-se como tal, tendo sempre somente um deles selecionado.
Este ButtonGroup tem fins apenas lógicos, e não gráficos. Portanto, eu ainda preciso dar um ADD em cada botão para dentro da “tela”.

Na próxima página, teremos um exemplo de como funciona o retorno dos valores (novamente, assim como o JCheckBox, o JRadioButton retorna um BOOLEAN através do método isSelected()).

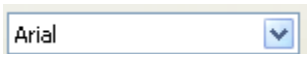
Exemplo funcional de uso do JRadioButton

```
import java.awt.event.*;

import javax.swing.*;

public class radio {
	public static void main(String[]args) {
		JFrame tela = new JFrame("Radios!");
		tela.setBounds(10,10,800,600);
		tela.setLayout(null);

		final JRadioButton designer = new JRadioButton("Eu sou WebDesigner!");
		final JRadioButton programador = new JRadioButton("Eu sou programador Java!");
		designer.setBounds(20,20,250,35);
		programador.setBounds(20,60,250,35);

		ButtonGroup grupo = new ButtonGroup();
		grupo.add(designer);
		grupo.add(programador);

		programador.setSelected(true);


		JButton botao = new JButton("Teste!");
		botao.setBounds(20,150,250,35);
		botao.addActionListener(new ActionListener() {

			public void actionPerformed(ActionEvent e) {
				if (designer.isSelected()) {
					JOptionPane.showMessageDialog(null, "Eu faço site colorido com logo cor
lilás.","Homem de rosa, o que é que você faz?",JOptionPane.INFORMATION_MESSAGE);
				}
				if (programador.isSelected()) {
					JOptionPane.showMessageDialog(null, "Escrevo código-fonte que assusta o
satanás.","Homem de preto, o que é que você faz?",JOptionPane.INFORMATION_MESSAGE);
				}

			}
		});

		tela.add(botao);

		tela.add(designer);
		tela.add(programador);

		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		tela.setVisible(true);
	}

}
```

Nesta primeira parte, eu faço tudo o que fiz no exemplo anterior: criar a tela e criar os JRadioButtons. Só que eu usei o método setSelected para dizer que a opção "programador" já vem marcada como TRUE desde o começo, ficando como opção default.

Aqui, usei nosso já conhecido ActionListener, para descobrir qual a caixa que está marcada. Note que usei novamente IFs independentes: o próprio ButtonGroup encarregar-se-á de manter apenas um deles marcado.

JComboBox



A JComboBox desenha uma caixinha parecida com a JTextField, só que ao invés de termos um texto a escrever, temos uma seta com a qual abrimos uma fileira de possibilidades de preenchimento para a caixa.

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;

public class combo {
	public static void main(String[]args){
		JFrame tela = new JFrame("Combo");
		tela.setBounds(10,10,800,600);

		JComboBox comb = new JComboBox();
		comb.setBounds(100,100,150,35);
		comb.addItem("Gaúcho");
		comb.addItem("Baiano");
		comb.addItem("Paulista");
		comb.addItem("Carioca");

		comb.setEditable(false);

		comb.setSelectedIndex(0);

		tela.setLayout(null);
		tela.add(comb);
		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		tela.setVisible(true);


	}

}
```

Eu começo como sempre: criando uma JFrame para abrigar nosso JComboBox.

Na hora de criar as JComboBoxes, eu começo instanciado-a na variável COMB. Depois, uso o método addItem("xxx") para escrever as opções que aparecerão ao usuário.
O método setEditable(boolean) diz se o usuário terá a opção de escrever na caixa (ignorando as opções disponíveis) ou não.
O método setSelectedIndex diz qual o item que aparecerá como default ao usuário. Usa-se aqui a mesma idéia dos índices de vetor para as opções.

Para obter o retorno das informações, temos uma série de métodos que nos dizem o que, afinal, o usuário escolheu. Os mais usados são os seguintes:

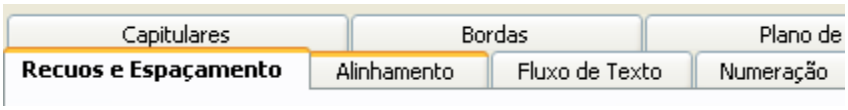
```
comb.getSelectedItem()
```

Este retorna o texto que está na caixa do Combo. Poderá ser usado especialmente para quando colocamos o setEditable em TRUE, ou seja, quando o usuário pode escrever na caixa. O retorno deste método é uma String, com o texto da caixa.

```
comb.getSelectedIndex();
```

Este método retorna o índice do item selecionado (como se as opções fossem um vetor). Caso o usuário possa escrever algo na caixa de texto, e escreva alguma coisa que não consta das opções fornecidas, então o índice que retorna é o -1. Portanto, só use se o setEditable for FALSE.

JTabbedPane



Este componente é utilizado para criarmos um daqueles menus separados em abas, as quais podem ser alternadas na parte superior da tela.

Uma TabbedPane funciona de maneira bem simples: primeiro, criamos nossas JTextFields, JLabels, Combos, Radios, Checks e o que mais quisermos. Depois, colocamos tudo isso dentro de JPanels. Em seguida, dizemos ao Java que cada JPanel é uma das "abas" da JTabbedPane.

```
import javax.swing.*;

import java.awt.*;

public class table {

	public static void main(String[]args) {
		JFrame tela = new JFrame("Telex");
		JTabbedPane tabs = new JTabbedPane(JTabbedPane.TOP);
		tela.setBounds(100,100,800,600);
		tabs.setBounds(10,10,770,550);

		JPanel painel1 = new JPanel();
		painel1.setBackground(Color.RED);

		JPanel painel2 = new JPanel();
		painel2.setBackground(Color.BLUE);

		tabs.add("Primeiro",painel1);
		tabs.add("Segundo",painel2);

		tela.add(tabs);
		tela.setLayout(null);
		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		tela.setVisible(true);

	}
}
```

No início, nada de novo: criamos os objetos e só. Mas note que o argumento da criação da JTabbedPane é JTabbedPane.TOP – no meu tabbed de exemplo, teremos os nomes das abas no topo. Pode-se usar ainda LEFT, RIGHT e BOTTOM.

Depois de criar os JPanels, inserimos eles com o método ADD da JTabbedPane, especificando qual a palavra que os representará no menu das abas.

É possível iniciar o programa com uma aba específica já selecionada, através do método:

tabs.setSelectedIndex(X) – onde X é o índice da tab que queremos ver selecionada – os índices atribuem-se por ordem de adição das abas à JTabbedPane, comportando-se como um vetor.

É possível também atribuir ícones às Tabs, usando o método:

tabs.seiIconAt(I) – onde I é um objeto do tipo Icon, já com uma imagem carregada.

JScrollPane

Uma JScrollPane é apenas um objeto que, associado a outras coisas (caixas de texto, tabelas, etc.) confere-lhes o "poder" de ter uma barra de Scroll.

```
import java.awt.event.*;
import javax.swing.*;

import java.awt.*;
public class scroler {
        public static void main(String[]args) {

                JTextField tex = new JTextField();
                tex.setBounds(0,0,300,700);

                JFrame tela = new JFrame();
                tela.setBounds(0,0,800,600);

                JScrollPane scrots = new JScrollPane(tex);
                scrots.setBounds(100,100,300,300);

                tela.add(scrots);


                tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                tela.setLayout(null);
                tela.setVisible(true);


        }
}
```

Este programa em si não tem quase nada de novo: criamos uma JFrame, um JTextField, e damos setBounds em tudo.

A única novidade é que criamos um JScrollPane associado à JTextField. E depois o colocamos na tela. Note que a JTextField vai parar "dentro" do JScrollPane. Tente agora escrever um texto longo na caixa de texto e veja o que acontece.

JTables

As tabelas que usamos para exibir dados no Java são o componente mais complexo que temos na área da interface gráfica.

Em primeiro lugar, uma tabela serve para demonstrar os dados de uma forma estática, sendo depois muito difícil de mudar seus dados durante a execução do programa. Por isso, usamos um modelo (DefaultTableModel), que é maleável, e atribuímos à JTable este modelo depois de modificarmos seus dados.

O exemplo abaixo é um código no qual nossa tabela não tem barras de rolagem. Se tivermos mais dados do que espaço para exibi-los no espaço da tabela, não veremos os de baixo.

```
import java.awt.event.*;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

import java.awt.*;
public class tabelas {
	public static void main(String[]args) {
		JFrame tela = new JFrame("Tabelas");
		tela.setBounds(0,0,1024,768);

		JTable tabela = new JTable();
		tabela.setBounds(10,10,200,400);
		tela.add(tabela);

		String[] colunas = {"Nome", "Salário"};

		DefaultTableModel modelo = (DefaultTableModel) (new DefaultTableModel(){
			public boolean isCellEditable(int row, int column) {
				return false;
			}
		});

		modelo.setColumnIdentifiers(colunas);

		modelo.setRowCount(0);

		Object[] objetos = new Object[2];
```

De início, são apenas as coisas tradicionais: imports, criação da JFrame, etc.

Aqui, eu crio a JTable e dou-lhe uma dimensão, adicionando-a à "tela" logo em seguida.

Crio um vetor de Strings para serem os cabeçalhos das minhas colunas.

Aqui, eu criei "modelo", um objeto da classe DefaultTableModel, e tive a preocupação de modificar o retorno do seu método isCellEditable, tornando seu retorno "false", para que o usuário não possa escrever diretamente na tabela. Este objeto "modelo", uma vez preenchido com os dados, deverá ser "colado" na JTable.

Aqui, eu transformei o meu vetor de Strings em cabeçalhos de colunas através do método setColumnsIdentifiers().

Com o método setRowCount, dizemos quantas linhas da tabela deverão ficar (o resto some). Eu disse que são ZERO, ou seja, zerei a tabela. Este procedimento deve ser feito sempre no começo do processo de carregar dados na tabela.

Aqui, criamos um vetor de objetos do tipo Object – a base de todas as coisas existentes em Java. Dentro deste vetor, poderemos gravar imagens (Icon), Strings, o que quisermos.

```
                objetos[0] = "Zé das Couves";
                objetos[1] = "1000";
                modelo.addRow(objetos);

                objetos[0] = "Silvio Santos";
                objetos[1] = "50000";
                modelo.addRow(objetos);

                objetos[0] = "Lula-lá";
                objetos[1] = "12000";
                modelo.addRow(objetos);

                tabela.setModel(modelo);

                tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                tela.setLayout(null);
                tela.setVisible(true);
        }
}
```

O próximo passo consiste em carregar este vetor "objetos" – no caso, com Strings. Em seguida, uso o método addRows do DefaultTableModel para adicionar o vetor "objetos" como uma linha de dados no modelo de tabela.

Note que estou reutilizando o vetor "objetos" para carregar novas linhas no "modelo". Eu vou repetir esta operação quantas vezes forem necessárias. Poderia ter feito esta operação repetitiva usando um FOR ou um WHILE. Isso será muito útil depois de vermos a parte de Banco de Dados.

Agora, meu objeto "modelo" está pronto. Ele é uma espécie de uma tabela "virtual", que não é visualizável. Para podermos vê-lo graficamente na tela, temos que usar um objeto JTable. E por isso mesmo criamos o "tabela" (lembra?). Agora, usamos o método setModel para unir os dois.

O resto é apenas a perfumaria de sempre: tirar o Layout, especificar o DefaultCloreOperation, e dar um setVisible na tela.

Depois, eu posso conseguir o retorno de dados da tabela, por exemplo, quando o usuário clicar duas vezes na minha tabela. Para isso, bastaria adicionar após o "tela.setVisible(true)" um MouseListener

```
tabela.addMouseListener(new MouseListener() {

                public void mouseClicked(MouseEvent e) {
                        if (e.getClickCount() == 2) {

                                int linha = tabela.getSelectedRow();
                                String mensagem = (tabela.getValueAt(linha,0).toString()) + " ganha R$ " +
(tabela.getValueAt(linha,1).toString());
```

Aqui, apenas especifiquei que a ação ocorrerá quando o usuário der um duplo clique. Nada novo.

A instrução acima é a única novidade deste código: temos um método chamado getSelectedRow, que vai descobrir qual é o índice da linha selecionada. Depois, temos o método tabela.getValueAt, no qual passamos como argumento uma coordenada parecida com as que usávamos para as matrizes. Assim, no caso, eu coloquei o número da linha, vírgula, o índice da coluna (zero e um, porque só temos essas duas).

```
JOptionPane.showMessageDialog(null, mensagem, "Dados", JOptionPane.INFORMATION_MESSAGE );
                        }
                }
                public void mouseEntered(MouseEvent e) {
                }
                public void mouseExited(MouseEvent e) {
```

```
            }
            public void mousePressed(MouseEvent e) {
            }
            public void mouseReleased(MouseEvent e) {
            }
    });
```

JTables e a interação com outros objetos - Uma JTable é o objeto perfeito para associar a um JScrollPane.Também podemos criar JTables carregadas com imagens.

JFileChooser

O JFileChooser é o componente que nos exibe uma caixinha para selecionar um arquivo no sistema de pastas do Windows ou do Linux (ou, de seja lá o que estivermos usando).

```
import javax.swing.*;
import sun.audio.*;
import com.sun.java.util.*;
import java.awt.*;
import java.awt.Event.*;
import java.io.*;
public class filechus {
        public static void main(String[]args) {

                JFrame tela = new JFrame("Telex");
                JFileChooser filex = new JFileChooser();

                int opcao = filex.showOpenDialog(tela);

                if (opcao==JFileChooser.APPROVE_OPTION) {
                        File nomearquivo = filex.getSelectedFile();

                        try {
                                InputStream arq = new FileInputStream(nomearquivo);
                                AudioStream som = new AudioStream(arq);
                                AudioPlayer.player.start(som);
                                System.out.println("Tocando");
                        }
                        catch(Exception e) {
                                System.out.println("Deu merda");
                        }

                }
                tela.setBounds(10,10,800,600);
                tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                tela.setVisible(true);
        }
}
```

criação de objetos, no caso, do JFrame e do JFileChooser.

O método showOpenDialog, que tem como argumento algum objeto ao qual a caixa de seleção estará associada. Este método retorna um valor INT, que é a resposta do usuário (se ele selecionou um arquivo ou se desistiu). Outro método, showSaveDialog(tela) dá uma caixinha apropriada para SALVAR o arquivo.

Se o retorno da opção anterior for afirmativo (esta é a questão no IF), eu carrego o arquivo selecionado em uma variável do tipo FILE, que guarda vários dados sobre o arquivo, e que podem ser extraídos por diversos métodos que veremos logo em seguida

O resto do código não faz parte realmente do JFileChooser – trata-se do código já visto para tocar música. O usuário deverá selecionar um arquivo WAV para tocar.

O resto do código são nossos conhecidos comandos de exibir o JFrame (quando o JFrame for fechado, o programa cairá fora do ar e a música cessará).

Métodos de retorno de dados mais usados do JFileChooser:

Boolean x = filex.getSelectedFile().canRead(); - retorna se o arquivo pode ser lido.

Boolean x = filex.getSelectedFile().canWrite(); - retorna se o arquivo pode ser sobrescrito.

String x = filex.getSelectedFile().getName(); - retorna o nome do arquivo.

String x = filex.getSelectedFile().getPath(); - retorna o local do arquivo.

boolean x = filex.getSelectedFile().exists(); - retorna se o arquivo existe ou não.

JInternalFrame

Finalmente, estamos chegando ao ponto no qual podemos criar uma aplicação funcional. Agora, criaremos as famosas "telinhas internas" do programa. Você já deve ter observado que, quando usamos o Microsoft Word, por exemplo, o programa em si tem uma grande tela com uma área dentro da qual fica apenas o "JFrame" com o texto que estamos trabalhando.

Aqui, faremos isso e não é nada muito complicado. A lógica é a seguinte: primeiro, precisamos ter um JDesktopPane, que é uma área, dentro de uma grande JFrame. Dentro desta JDesktopPane, teremos nossas JInternalFrame, que nada mais são do que pequenas telas internas.

```
import javax.swing.*;
import sun.audio.*;
import com.sun.java.util.*;
import java.awt.*;
import java.awt.Event.*;
import java.awt.event.*;
import java.io.*;
public class filechus {
        public static void main(String[]args) {

                JFrame tela = new JFrame("Telex");
                JDesktopPane deska = new JDesktopPane();

                final JInternalFrame telinha1 = new JInternalFrame("Tela Interna 1", true, true, true, true);
                final JInternalFrame telinha2 = new JInternalFrame("Tela Interna 2", true, true, true, true);

                telinha1.setBounds(0,0,300,400);
                telinha2.setBounds(0,0,300,400);

                telinha1.hide();
                telinha2.hide();

                telinha1.setDefaultCloseOperation(JInternalFrame.HIDE_ON_CLOSE);
                telinha2.setDefaultCloseOperation(JInternalFrame.HIDE_ON_CLOSE);
```

Começamos o programa como sempre: imports e criação da JFrame. Mas note que eu instanciei a classe JDesktopPane. Ela servirá apenas para fazer a acomodação dos JInternalFrame dentro da JFrame maior.

Na hora de instanciar as duas JInternalFrame (telinha1 e telinha2), note que eu coloquei como argumentos o título da janelinha e quatro TRUEs. Estes TRUEs (que bem poderiam ser FALSEs), são respectivamente:
A telinha é redimensionável?
A telinha tem o botão FECHAR em forma de X no canto?
A telinha pode ser maximizada?
A telinha pode ser minimizada?

Aqui, eu usei o setBounds normalmente nas JInternalFrames, para logo em seguida dar-lhes a instrução HIDE. Elas ficarão escondidas até que sejam chamadas. Elas estão "vivas", na memória, mas escondidas. "Hide" em inglês significa "esconder".

Note que eu escolhi, como DefaultCloseOperation, o HIDE_ON_CLOSE. Isso quer dizer que, quando fechadas, minhas telinhas internas não serão descarregadas da memória (o que aconteceria se fosse DISPOSE_ON_CLOSE) e nem o programa sairá do ar (o que aconteceria com EXIT_ON_CLOSE). As telinhas vão simplesmente ser escondidas, ficando invisíveis.

```
JMenuItem t1= new JMenuItem ("Tela 1");
JMenuItem t2 = new JMenuItem ("Tela 2");
JMenuBar barra = new JMenuBar();
JMenu menu = new JMenu("Opções");
```

Os passos seguintes não constituem novidade alguma. Aqui, criamos um menu para que o usuário possa "chamar" as duas telinhas de teste.

```
barra.add(menu);
menu.add(t1);
menu.add(t2);
```

```
t1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                telinha1.show();
        }
});

t2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                telinha2.show();
        }
});
```

Aqui, eu simplesmente adicionei ActionListeners aos meus itens de menu. A ação de cada um deles é simplesmente ativar o método "show()" de uma das telinhas, o que as fará ficar visíveis.

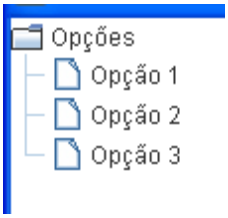
```
deska.add(telinha1);
deska.add(telinha2);

tela.add(deska);

tela.setJMenuBar(barra);
```

Agora, aqui é importante notar como as coisas são estruturadas umas dentro das outras. Eu tenho minhas JInternalFrames ("telinha'" e "telinha2") presas dentro de um JDesktopPane ("deska"). E o JDesktopPane preso dentro da JFrame ("tela").

As instruções que vêm logo em seguida são antigas para nós: a inserção da JMenuBar dentro da JFrame, e as instruções finais de exibição da tela.

```
tela.setBounds(10,10,800,600);
tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
tela.setVisible(true);


        }
}
```

Tree



Vamos ver agora um componente que dá um certo estilão a qualquer projeto. Trata-se do Tree. Com ele, podemos criar aqueles menus em que as opções ficam umas dentro das outras e vão se abrindo à medida que o usuário clica nelas.

A aplicação de um Tree segue um esquema parecido com o das Jtables – nós temos uma modelagem toda feita em um objeto que não é visível, e no final temos a "encarnação" deste modelo em um objeto que é passível de adição na tela.

```
import java.awt.event.*;
import java.sql.*;
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;

public class principal {
	static Connection con;
	public static void main(String[]args) {
		JFrame tela = new JFrame();
		tela.setBounds(10,10,600,400);

		Object[] opcoes = {
				"Opções",
					"Opção 1",
					"Opção 2",
					"Opção 3"
		};
		DefaultMutableTreeNode nos = new DefaultMutableTreeNode(opcoes[0]);
		DefaultMutableTreeNode child;
	for(int cont = 1; cont < opcoes.length; cont++){
		Object obj = opcoes[cont];
		child = new DefaultMutableTreeNode(obj);
		nos.add(child);
	}

	JTree tri = new JTree(nos);

	tri.setBounds(0,0,200,400);

	tela.add(tri);

		tela.setLayout(null);
		tela.setVisible(true);
		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
}
```

Eu começo meu código de uma forma bem banal: criando uma JFrame.

Notem que logo em seguida eu crio um Array de Object. Este Array contém as informações para um nó da minha Tree. Vejam que eu criei um objeto DefaultMutableTreeNode (um nó da Tree, que será atribuído a ela depois).

Este primeiro nó, eu criei importando opcoes[0], que é a palavra "Opções" - este nó será "paii" de outros que eu criarei depois.

Depois de criar o nó "nos", eu criei o "child" - este guardará os filhotes do "nos". Eu usarei um único objeto do tipo nó, o "child", que será reutilizado várias vezes.

Veja que eu fiz um FOR. Ele roda todos os elementos do Array "opcoes", menos o de índice zero (que é a palavra "Opções", e que eu já usei para fazer o nó "pai").

A cada rodada do FOR, eu carrego a palavra do índice em que a variável CONT está (e ela corre do 1 ao, no caso, 3). E a cada rodada, ela é carregada no nó "child".

Depois, eu adiciono "child" dentro do nó "no", e o FOR faz outro giro, reutilizando "child" para adicionar mais um filho a "no".

Depois de adicionar todos os nós "filhotes", a variável "child" torna-se inútil. Mas a variável "nos" agora guarda o nó que tem a palavra "Opções" e mais todos os sub-nós adicionados no processo que rodamos no FOR. Assim, temos o menu completinho.

Eventos (Listeners)

ActionListener

Hoje, vamos ver algo muito interessante: o ActionListener. Trata-se de uma classe que detecta se houve uma ação no objeto a que ela está associada (um botão, por exemplo), e que, dependendo da ação executada, rodará uma função qualquer. Vejamos um exemplo de código usando o ActionListener:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class mongol {
	public static void main(String[]args) {
		JFrame tela = new JFrame("Teste");
		JButton botao = new JButton("Aperte aqui");
		tela.setBounds(10,10,600,400);
		botao.setBounds(100,100,150,35);
```

Até aqui, não fizemos nada de diferente. Criamos a JFrame e um botão.

```
		ActionListener ac = new ActionListener() {
```

Aqui, instanciamos a classe ActionListener, mas modificamos um de seus métodos, no caso, o ActionPerformed, que tem o poder de fazer alguma coisa quando o usuário clicar no objeto que tem ActionListener.

```
			public void actionPerformed(ActionEvent e) {
```

A declaração acima é um método, o mesmo que citamos antes. O argumento dele é um evento ocorrido que fica em uma lista de eventos possíveis determinados pelo usuário, como veremos mais adiante.

```
				if ("botao1".equals(e.getActionCommand())) {

				  JOptionPane.showMessageDialog(null, "Apertou o botão!", "Olá!",
JOptionPane.INFORMATION_MESSAGE);
				}
```

Aqui, a coisa é simples: caso o ActionCommand do evento “e” seja igual a “botao1”, uma caixa com uma mensagem aparecerá na tela.

```
			}
		};
```

Quando alteramos um método da classe no momento de sua instanciação, terminamos com chaves e ponto-e-vírgula; Agora, vem o final, montar a tela e colocar este ActionListener para funcionar.

```
		botao.setActionCommand("botao1");
		botao.addActionListener(ac);
```

Acima, eu determinei qual é o ActionCommand do botão (no caso “botao1”), e depois adicionei o ActionListener ao botão, fazendo com que ele passe a “detectar” quando o usuário clica no botão, acionando o método ActionPerformed.

```
		tela.setLayout(null);
		tela.add(botao);
		tela.setVisible(true);
	}
}
```

FocusListener

O FocusListener se parece com o ActionListener em seu modo de construção, mas serve para detectar quando o foco (ou seja, o cursor, a seleção, não o ponteiro do mouse) está sobre um objeto ou não.

Ele possui dois métodos que precisam ser implementados: o focusGained e o focusLost, um para quando o objeto ganha foco e outro para quando perde.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class mongol {
	public static void main(String[]args) {
		final JFrame tela = new JFrame("Teste");
		final JTextField tex = new JTextField();
		final JTextField tex2 = new JTextField();
		tela.setBounds(20,20,800,600);
		tex.setBounds(50,50,300,35);
		tex2.setBounds(50,150,300,35);

		FocusListener foc = new FocusListener() {

			public void focusGained(FocusEvent e) {
				tex.setBackground(Color.RED);
			}

			public void focusLost(FocusEvent e) {
				tex.setBackground(Color.BLUE);
			}
		};
		tex.addFocusListener(foc);
		tela.setLayout(null);
		tela.add(tex);
		tela.add(tex2);
		tela.setVisible(true);
	}
}
```

KeyEvent

O KeyEvent serve para que eu possa coordenar as ações do usuário com o teclado quando ele está escrevendo em algum espaço ou utilizando alguma função. Posso atribuir KeyListener a qualquer objeto gráfico do Java – QUALQUER! – e detectar o que, afinal, o meu usuário está apertando no teclado.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class mongol {
	public static void main(String[]args) {
		final JFrame tela = new JFrame("Teste");
		final JTextField tex = new JTextField();
		final JTextField tex2 = new JTextField();
		tela.setBounds(20,20,800,600);
		tex.setBounds(50,50,300,35);
		tex2.setBounds(50,150,300,35);

		KeyListener key = new KeyListener() {
			public void keyPressed(KeyEvent e) {
				tex.setBackground(Color.BLUE);
				tex.setForeground(Color.WHITE);
			}
```

Acima, defini a ação do método keyPressed, ou seja, quando o usuário enfia o dedo no botão.

```
			public void keyReleased(KeyEvent e) {
				tex.setBackground(Color.BLACK);
				tex.setForeground(Color.YELLOW);
			}
```

Acima, defini a ação a executar para quando o usuário tira o dedo do botão que havia pressionado.

```
			public void keyTyped(KeyEvent e) {
				JOptionPane.showMessageDialog(null, "Typou rapaz", "Obviedades",
JOptionPane.INFORMATION_MESSAGE);
			}
```

O método keyTyped funciona quando o apertão da tecla faz efeito. Isso quer dizer que se eu botar o dedo numa letra do teclado, e não tirar mais, e esta letra ficar se repetindo, o evento keyTyped também ficará.

```
		};



		tex.addKeyListener(key);

		tela.setLayout(null);
		tela.add(tex);
		tela.add(tex2);
		tela.setVisible(true);
	}
}
```

NOTA: Pode-se detectar qual o botão apertado pelo usuário explorando a variável “KeyEvent e”, que retornará o código da tecla apertada.

MouseListener

O MouseListener, como seu nome obviamente revela, serve para tratarmos eventos relacionados ao uso do mouse.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class mongol {
	public static void main(String[]args) {
		final JFrame tela = new JFrame("Teste");
		final JTextField tex = new JTextField();
		final JTextField tex2 = new JTextField();
		tela.setBounds(20,20,800,600);
		tex.setBounds(50,50,300,35);
		tex2.setBounds(50,150,300,35);


		MouseListener mickey = new MouseListener() {

			public void mouseClicked(MouseEvent e) {
				int contador = e.getClickCount();
				if (contador == 1) {
					tex.setText("Você deu um clique!");
				}
				else if (contador == 2) {
					tex.setText("Você deu um duplo clique!");
				}
			}
```

No método acima, o mouseClicked, ou detecto que o usuário deu um ou mais cliques no meu componente que possui o MouseListener. Bom. Uma vez definido que isto ocorre, eu resolvi dificultar um pouco as coisas, colocando em prática o método getClickCount, que detecta quantas clicadas rápidas o usuário deu sobre o objeto. Com isso, posso definir ações para o clique, o duplo clique, o triplo clique, até o milhonésimo clique (os usuários é que ficariam cansados, mas tudo bem).

```
			public void mouseEntered(MouseEvent e) {
				tex.setText("O mouse entrou!");
			}
```

O método mouseEntered ocorre quando o mouse adentra, quando o ponteiro passar por cima.

```
			public void mouseExited(MouseEvent e) {
				tex.setText("O mouse saiu!");
			}
```

Obviamente, o mouseExited ocorre quando o ponteiro sair de cima do objeto.

```
			public void mousePressed(MouseEvent e) {
				tex.setBackground(Color.YELLOW);
			}
```

mousePressed é um evento que continua em ação enquanto o usuário não tirar o dedo de cima do botão do mouse, ficando com ele apertado.

```
                public void mouseReleased(MouseEvent e) {
                        tex.setBackground(Color.WHITE);
                }
```

O evento mouseReleased acontece quando o usuário finalmente larga o botão.

```
                };

                tex.addMouseListener(mickey);


                tela.setLayout(null);
                tela.add(tex);
                tela.add(tex2);
                tela.setVisible(true);
        }
}
```

Nós podemos saber qual é a localização do ponteiro do mouse utilizando dois métodos:

getX() – retorna a posição do mouse no eixo X (horizontal) dentro do objeto a que estamos atribuindo o MouseListener.

getY() – Faz o mesmo, só que com a posição do eixo Y (vertical).

Uma coleção de pequenos comandos soltos

Leitura de dimensões de componentes gráficos e da própria tela

```
int x = qualquer componente gráfico.getHeight();
```

Pega a altura de qualquer componente gráfico.

```
int x = qualquer componente gráfico.getWidth();
```

Pega a largura de qualquer componente gráfico.

```
double alt = Toolkit.getDefaultToolkit().getScreenSize().getHeight();
double larg = Toolkit.getDefaultToolkit().getScreenSize().getWidth();
```

Estes dois commandos pegam, respectivamente, a altura e a largura da TELA, não de um componente ou da JFrame. Com isso, é possível centralizar nossa JFrame no vídeo do usuário, mesmo que não saibamos a definição (em pixels) utilizada por ele. A fórmula para isso é simples:

Posição da JFrame no eixo X é igual a: Largura da tela, dividida por dois, menos a largura do JFrame, dividida por dois. E o mesmo repete-se em relação ao eixo Y, mas associado à altura da tela.

Conversão de dados

Transformar Strings em int

```
int qualquercoisa = Integer.parseInt(stringqualquer);
```

Transformar int em String

```
stringqualquer = Integer.toString(intzinho);
```

Transformar String em double

```
double numero = Double.ValueOf(stringzinha).doubleValue();
```

Transformar double em String

```
stringzinha = Double.toString(doublequalquer);
```

Transformar double em int

```
int x = (int) numerodouble;
```

Fazendo uma JFrame sumir, morrer e derrubar o programa

Já estudamos a JFrame em diversos exemplos, mas vale a pena dar uma olhada na relação que existe entre o setDefaultCloseOperation e os modos de fechamento da JFrame.

Existem três formas de uma JFrame desaparecer da tela:

HIDE – Apenas esconde a JFrame, mas o objeto continua na memória, podendo ser recuperada a qualquer momento pelo método SHOW (tela.hide() e tela.show()).

DISPOSE – O objeto JFrame é destruído completamente. Para fazê-lo reaparecer na tela, é preciso usar o NEW e criar um objeto novo, re-declarar a variável, e tudo mais. Este é o método mais utilizado para operar com JInternalFrames, onde utilizar-se-á a sintaxe "tela.dispose()".

EXIT – Na verdade, não é uma operação associada à JFrame, e sim, uma ordem de fechamento do programa inteiro.

Relações com o setDefaultCloseOperation

O setDefaultCloseOperation serve para dizermos ao programa o que queremos que o sistema faça quando fechamos uma JFrame qualquer.

HIDE_ON_CLOSE – O fechamento leva a janela a esconder-se, embora ela ainda exista.

DISPOSE_ON_CLOSE – O fechamento leva a tela a se destruir, e para fazê-la reaparecer, precisamos usar o NEW, re-declarar a variável, etc.

EXIT_ON_CLOSE – Faz com que, ao fechar uma JFrame, o programa inteiro seja descarregado da memória do computador. Utilize esta definição na JFrame principal do programa.

Sintaxe de exemplo:

```
tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Ringo não minimiza!

MATA!

Ordenando a uma JFrame que abra em modo maximizado

Existe um método, o setExtendedState, no qual podemos colocar como valor MAXIMIZED_BOTH, o que nos trará a JFrame maximizada. Se for um JInternalFrame, ele virá maximizado dentro do JDesktopPane ao qual pertence.

Sintaxe:

```
tela.setExtendedState(JFrame.MAXIMIZED_BOTH);
```

Existe uma grande diferença entre maximizar e simplesmente descobrir o tamanho da tela, redimensionando o JFrame para o tamanho da tela. Quando usamos redimensionamento com setBounds ou setSize, o programa continua tendo o botão de MAXIMIZAR, o que não acontece com o caso explicado acima.

Redimensionando imagens para caberem dentro da JLabel

Para realizar esta tarefa, vamos explorar mais a fundo a classe ImageIcon, que até agora utilizamos apenas de forma secundária, para carregar arquivos dentro de Icons, conforme todos devem já ter testado algumas vezes. Esta classe possui uma função bem interessante, na qual as imagens podem ser lidas e redimensionadas (na tela, não no seu respectivo arquivo JPG ou PNG, claro).

```
ImageIcon imagem = new ImageIcon("foto.jpg");
```

O primeiro comando é o conhecido carregamento da imagem de arquivo para dentro da variável.

```
Image redimensionadora = imagem.getImage().getScaledInstance(largura, altura, Image.SCALE_DEFAULT);
```

Este segundo comando é mais interessante. Nele, eu instancio a classe Image, que guardará uma imagem "abstrata", invisível, mais ou menos na mesma lógica do DefaultTableModel (a tabela invisível, lembram?). Dentro deste objeto criado, carregarei o que resultar da seguinte operação:

De dentro de "imagem", retiramos só a imagem sem dados adicionais (método getImage). E neste objeto mesmo rodamos o método getScaledInstance, que recebe como argumentos a largura que a imagem deve ter, a altura, e o tipo de redimensionamento desejado. Usei o SLACE_DEFAULT, que é bem equilibrado. Poderia ter usado SCALE_FAST, no qual a imagem pode ficar meio pixelada, mas o processamento é mais rápido, ou SCALE_SMOOTH, que fica sempre bem feito, mas é mais lento.

```
imagem = new ImageIcon(redimensionadora);
```

No final, carrego a imagem "abstrata", invisível, em um formato visível: o nosso tradicional ImageIcon imagem, que será utilizado de forma prática.

Colocando uma imagem de plano de fundo dentro do JDesktopPane

Na verdade, colocar uma imagem de fundo no JDesktopPane é igualzinho a colocar uma imagem em qualquer lugar: usamos o JLabel, e colocamos ele dentro do JDesktopPane. Porém, ficam no ar algumas questões, como o dimensionamento desta JLabel. Ora, acabamos de ver comandos que descobrem a largura da tela, comandos de conversão de dados em tipos diferentes, comandos de redimensionamento de imagem. Chegou a hora de usar tudo isso ao mesmo tempo.

Um código de exemplo:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class dentro {
	public static void main(String[]args) {
		JFrame tela = new JFrame("Programa");
		final JDesktopPane deska = new JDesktopPane();
		JMenuBar barra = new JMenuBar();
		JMenu opcoes = new JMenu("Opções");
		JMenuItem abreinterna = new JMenuItem("Abrir telinha interna");

		double alt = Toolkit.getDefaultToolkit().getScreenSize().getHeight();
		double larg = Toolkit.getDefaultToolkit().getScreenSize().getWidth();

		int altura = (int) alt;
		int largura = (int) larg;


		ImageIcon imagem = new ImageIcon("foto.jpg");
		Image redimensionadora = imagem.getImage().getScaledInstance(largura, altura, Image.SCALE_DEFAULT);
		imagem = new ImageIcon(redimensionadora);

		JLabel lab = new JLabel();
		lab.setIcon(imagem);

		lab.setBounds(0,0,largura,altura);

		deska.add(lab);


		abreinterna.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				JInternalFrame interna = new JInternalFrame("Tela menor", true, true, true, true);
				interna.setBounds(10,10,400,400);
				interna.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
				deska.add(interna);
				interna.setVisible(true);
			}});


		opcoes.add(abreinterna);
		barra.add(opcoes);
		tela.setJMenuBar(barra);


		tela.getContentPane().add(deska);
```

Primeiro passo, instanciar os objetos que usaremos. Vamos montar um menu com uma opção só.

Nestas 4 linhas, eu descobri o tamanho da tela e converti suas dimensões em INT.

Após redimensionar a imagem dentro do ImageIcon, criei a Label com a imagem dentro e a adicionei ao JDesktopPane. Note que eu dimensionei a JLabel e portanto a imagem a partir do tamanho da tela. Assim, criei uma imagem de fundo para o programa.

Dentro do ActionListener, eu coloquei toda a rotina de criação da JInternalFrame. Inclusive, note que seu método ao fechar é o DISPOSE_ON_CLOSE. Ou seja, a telinha interna será criada para uso, e destruída (liberando memória) quando não se quiser usar.

```
        tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        tela.setExtendedState(JFrame.MAXIMIZED_BOTH);
        tela.setVisible(true);
    }
}
```

Os comandos finais são a adição do JDesktopPane dentro da tela, e os tradicionais comandos que colocam a tela no ar. Ou seja, nosso habitual final de código de todos os exemplos. Só que aqui a tela abre maximizada.

Atenção!

Não passe deste ponto sem ler algum material sobre SQL!

Você deve conhecer SQL e é bem provável que, em breve, haja também uma obra do mesmo autor sobre SQL.

Se você não manja nada de SQL e nem sabe o que é um banco de dados, então não prossiga! Procure informar-se primeiro.

Se você não sabe como funciona um Banco de Dados,

NÃO PROSSIGA!

Eu avisei! Os incrédulos irão pagar!!!! há há há ha!!

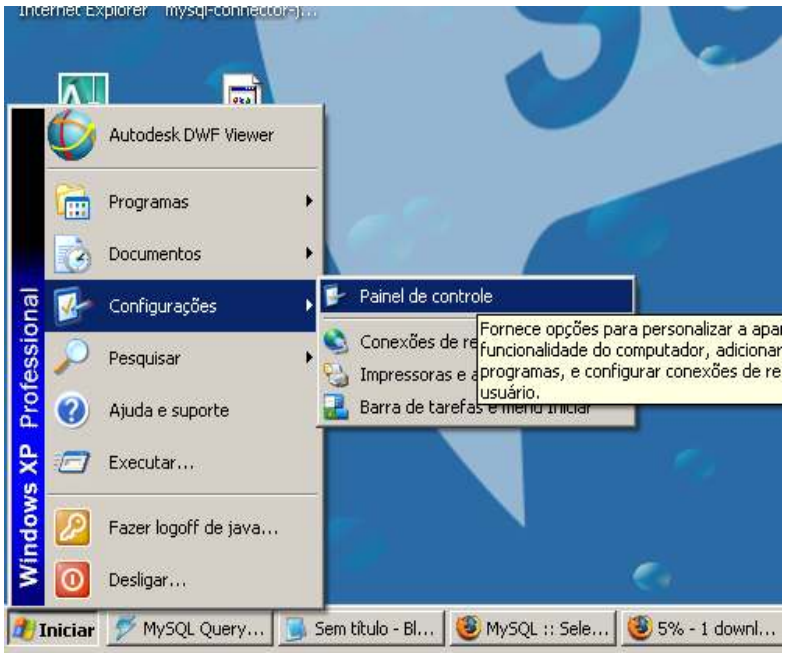
Conexão Java + Banco de Dados

Hoje começaremos a conexão do nosso programa com um banco de dados. De início, vamos usar a ODBC para conectar com uma base Access.

O que é a ODBC? ODBC é um dispositivo existente no Windows com o qual podemos listar as bases de dados configuradas neste computador ou pela rede e na qual as bases de dados recebem um nome (um Alias). Uma vez criada a conexão ODBC, podemos acessar o banco de dados usando este Alias.

Como se configura o ODBC? Fácil! Primeiro, eu crio uma base de dados, digamos, no Access.

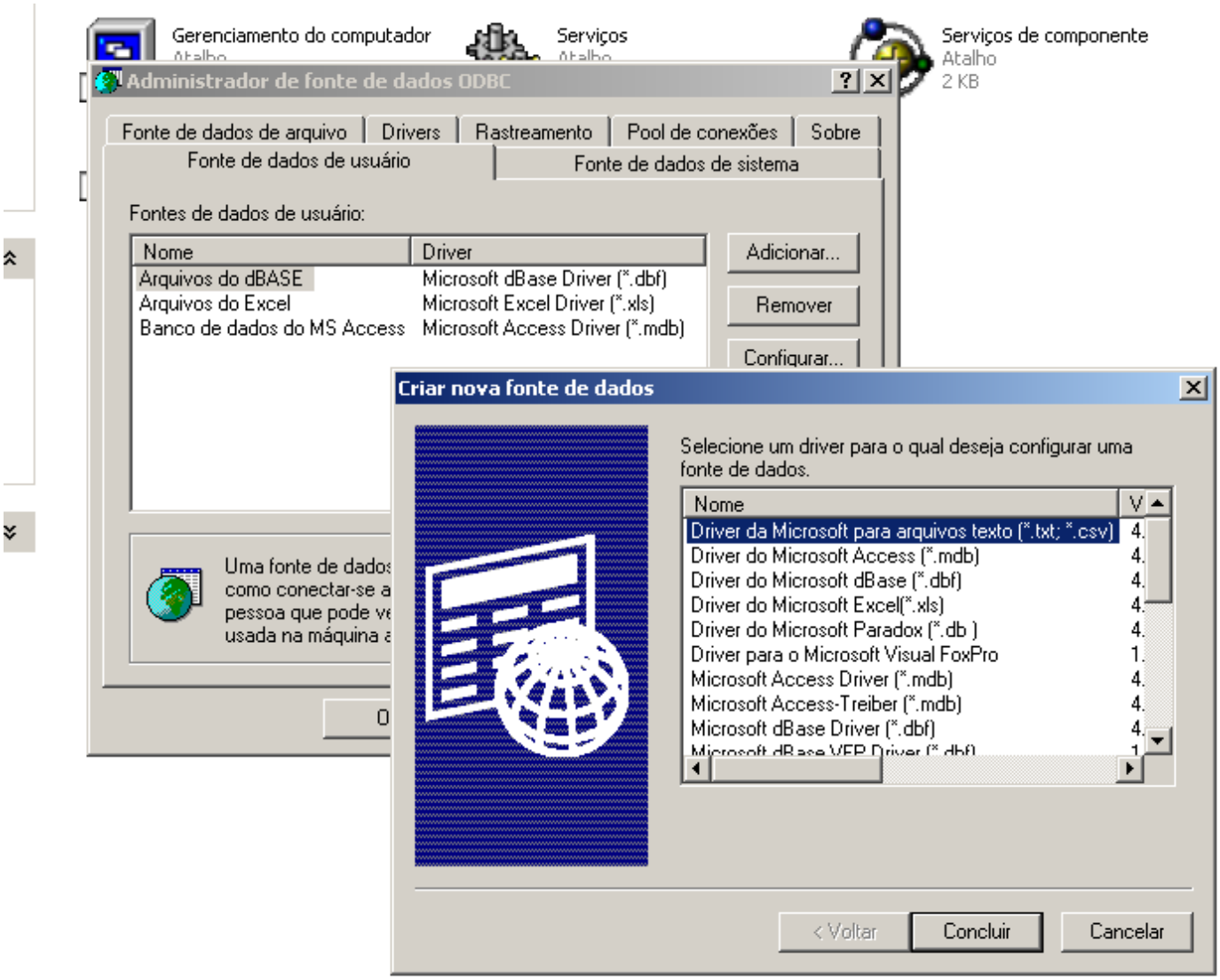
Depois, vá ao Painel de Controle do Windows.



Escolha as Ferramentas Administrativas.

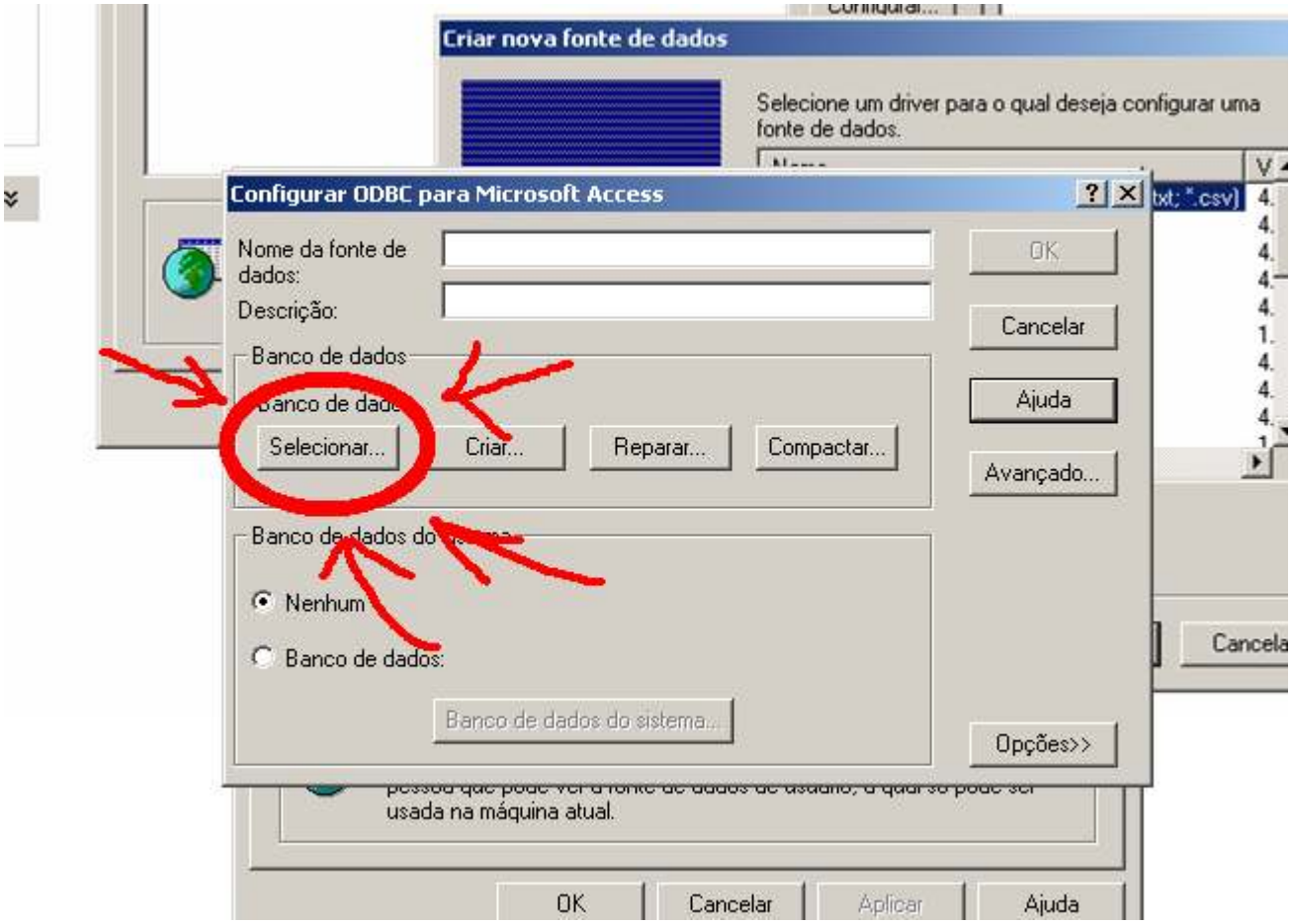
Depois, escolha as Fontes de Dados (ODBC).

Depois, vá ao botão “Adicionar” e escolha o tipo de base de dados. No caso, eu escolho neste exemplo a do Access.



Daí, aparecerá a caixinha abaixo. Você precisará encontrar o arquivo do banco de dados Access, que

você gravou ao cria-lo. Para isso, use o botão Selecionar e vá navegando pelas pastas até achar seu arquivo.



No campo “Descrição”, escreva um nome (um Alias) para o seu banco de dados. Nada muito complicado. Opte por nomes curtos. Coisas como “bd_compras” ou algo assim.

O uso de ODBC para conectar base de dados serve para a maior parte dos SGBD (Sistemas Gerenciadores de Banco de Dados) existentes na face da Terra. Você pode implementar nos seus clientes se quiser. Só que há outras maneiras de ligar um Banco de Dados a um programa.

Em seguida, você deverá criar sua aplicação Java com interação com o Banco de Dados. Vamos em frente então.

Criando a aplicação integrada

Eu vou começar construindo uma classe bem simples cuja única função é criar uma conexão com o banco de dados, que será "armazenada" no objeto CON, que recebeu este nome por preferência minha (ele poderia se chamar CONECTA, CHUMBINHO ou MARIOLA, tanto faz).

```
import java.sql.*;
public class conecta{
        static Connection con;
static void getConexao () {
        try {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
catch(ClassNotFoundException e) {
                System.out.println("Classe do driver nao encontrada:");
                 System.out.println(e.getMessage());
                 System.exit(1);
}
        try {
        String url = "jdbc:odbc:dbase";
                        con = DriverManager.getConnection(url, "username", "senha");
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                }
}
}
```

BONITO, NÉ? Vamos agora esmiuçar este código linha por linha:

```
import java.sql.*;
```

Importa a biblioteca de classes para trabalhar com banco de dados.

```
public class conecta{
        static Connection con;
```

Aqui eu instanciei a classe Connection, que armazenará a conexão com o BD.

```
   static void getConexao () {
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }
```

Aqui, dei início ao meu método getConexao, que recebeu este nome também de forma casual – o método que conecta pode ter qualquer nome. Aqui, usei um comando que talvez seja novidade para alguns: o Class.forName. O Java tem dessas: aqui eu mando ele tentar achar essa classe pelo nome dela e carregar. Na hora que eu fiz isso, comecei a ter acesso a umas coisas novas, como o DriverManager, associada a esse driver que eu acabei de carregar (no caso, o JDBC-ODBC). É interessante notar que, quando usamos a classe SUN.JDBC.ODBC, dificilmente teremos como resposta um erro de ausência da classe, a não ser que o usuário esteja usando uma JRE do tempo das Grandes Navegações. Só que esta verificação será ainda muito útil, quando importarmos uma classe que faz, por exemplo, o trabalho de conexão do Java com uma base de dados MySQL – esta classe deverá ser "anexada" ao projeto, de modo que pode ou não estar disponível – e se não estiver, gerará um erro, visto logo no Catch.

```
      catch(ClassNotFoundException e) {
         System.out.println("Classe do driver nao encontrada!");
```

```
        System.exit(1);
    }
```

Bom, aqui eu tratei o erro, caso o programa não ache a classe a partir do forName. O negócio aqui é derrubar o programa. No caso dos programas que fazemos na prática, do dia a dia das empresas, a idéia nunca será derrubar o programa, e sim avisar ao usuário que deu algum problema e mandá-lo entrar em contato com o fabricante.

Um erro desses jamais acontece em condições normais, porque a classe que tem o driver sempre será compactada dentro do JAR junto com o programa. Se o usuário tiver esta surpresa desagradável, de o programa não achar a classe, então o software não deveria nem ter saído da fase de testes, porque o defeito vem desde a mesa do desenvolvedor e por alguma razão jamais foi detectado.

```
    try {
        String url = "jdbc:odbc:banco";
        con = DriverManager.getConnection(url, "username", "senha");
```

Agora, olhem que legal: a classe DriverManager tem um método que é o getConnection, e ele "retorna" uma conexão com banco de dados – não uma String ou coisa assim, mas a conexão. E joga ela para a variável CON, que já é uma Connection. Notem que eu usei ali três argumentos: COM, USERNAME e SENHA. O Con é necessariamente sempre uma Connection que já tenha uma conexão a banco de dados guardada. Já no lugar de USERNAME e SENHA, colocamos o nome de usuário e a senha para aquela determinada base de dados.

```
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                }
}
}
```

Depois, tem o Catch do carregamento da conexão. Aqui, não tem segredo – se o banco falhar, ele tem que fazer alguma coisa. Eu não coloquei nada, deixei só o código que o Eclipse gera. Mas vocês vão colocar uma mensagem de erro ou coisa assim. Erros de conexão a banco de dados normalmente ocorrem por queda da rede (caso o BD fique em um servidor) ou porque algum ser quase humano, de inteligência rara e formidável, desligou o servidor. Ou ainda porque a filha do dono da empresa resolveu limpar o servidor, "removendo os vírus" e acabou com a base de dados.

Consultando o Banco de Dados

Para carregar os dados, você vai precisar usar o Connection em associação com um Statement, que é um tipo de "ordem" para a Connection retornar alguma coisa.

Depois de mandar a classe CONECTA rodar seu método getConexao, eu faço assim:

```
Statement comandoSQL = conecta.con.createStatement();
```

Aqui, criei o Statement chamado comandoSQL ligado á conexão CON da classe CONECTA. Lembrando que estes nomes não são fixos, e eu poderia ter usado outro nome de objeto ao invés de COM – ELKEMARAVILHA, por exemplo. A classe Conecta também, poderia chamar-se CHURROS e não faria a menor diferença.

```
String sql = "Select * from tabela;";
ResultSet resultado = comandoSQL.executeQuery(sql);
```

Aqui, logo depois de criar o Statement, eu crio uma String armazenando um comando SQL qualquer. Depois, eu crio uma variável do tipo RESULTSET, que é como um vetor de dados tipo OBJECT, e que armazena os dados que retornam uma consulta SQL qualquer chamada pelo método executeQuery. Ele recebe como argumento uma String onde há o comando SQL. Eu bem poderia escrever a String ali mesmo, dentro do argumento do ResultSet, mas decidi fazer isso antes porque fica mais fácil de entender o código mais tarde, quando eu fizer manutenção no sistema.

```
modelo.setRowCount(0);
while(resultado.next()){
Object[] objetos = new Object[3];
        for(int i = 0; i < 3; i++) {
                objetos[i] = resultado3.getObject(i+2);
}
        modelo.addRow(objetos);
}
```

Bom, aqui eu usei os dados da tabela para carregar um DefaultTableModel, como nós já vimos lá na parte das JTables. No caso, o comando resultado.next faz "rodar" uma linha do ResultSet, e cada linha dessas é um pequeno vetor com os campos do registro. Daí saiu a idéia de usar um FOR e tudo mais. Mas essa coisa de carregar um vetor de Objects e atirar para dentro do TableModel é matéria velha. Nós também já vimos como se faz a leitura "linha a linha" de uma base de dados quando trabalhamos com o BufferedReader, naquela parte da matéria na qual escrevemos e lemos arquivos-texto.

Gravando no BD (Insert e Update)

Para gravar e dar um update ou um delete no banco de dados, enfim, para fazer qualquer coisa que não seja um Select, nós fazemos assim: Começamos por criar uma String com o comando SQL desejado.

```
String sql = "insert into pessoas values (" + codigo + ", '" + name + "', " + idade + ", '" + salario +"');";
```

Tá, nenhuma novidade.

```
Statement comandoSQL = con.createStatement();
int resultado = comandoSQL.executeUpdate(sql);
comandoSQL.close();
```

Aqui temos as novidades do momento: Nós criamos o Statement igualzinho ao da consulta, mas depois, ao invés de darmos um executeQuery, damos um executeUpdate, o que não significa que utilizamos isso apenas para dar UPDATE, mas também INSERT e tudo o mais que eu já expliquei antes. A idéia aqui é que ele vai executar, mas não vai retornar um ResultSet, tanto que o retorno disso é um INT – esse INT nos diz quantas linhas foram afetadas pela nossa instrução. Ou seja, se retornar 0, podes crer que é quase sinônimo de erro. Ele vai retornar alguma coisa, isso é certo. Daí temos que tratar o retorno de alguma forma.

Viram? Não tem muito segredo. Colocar o banco de dados em MySQL exige umas pequenas mudanças e a importação de umas classes diferentes, mas nada além disso. Vejamos a seguir como se faz:

Java + MySQL

A primeira coisa que devemos fazer é um download: vamos baixar o "MySQL Connector/J", um pequeno arquivo JAR com tudo o que precisamos. Ele faz uma conversão do protocolo usado pelo JDBC do Java para o de rede usado pelo MySQL.

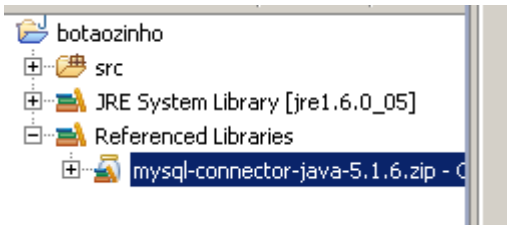
Preste muita atenção porque você poderá incorporar Bancos de Dados Oracle, SQLServer, ou qualquer outro que apareça por aí usando o mesmo caminho. Cada empresa que desenvolve um banco de dados próprio automaticamente cria um JAR com o conector para Java, para que os desenvolvedores Java possam usar seus produtos. Então, fique tranqüilo!

O endereço para download é: http://dev.mysql.com/downloads/connector/j/ - Este endereço poderá mudar no futuro. Caso isso aconteça, recorra ao Google mesmo.

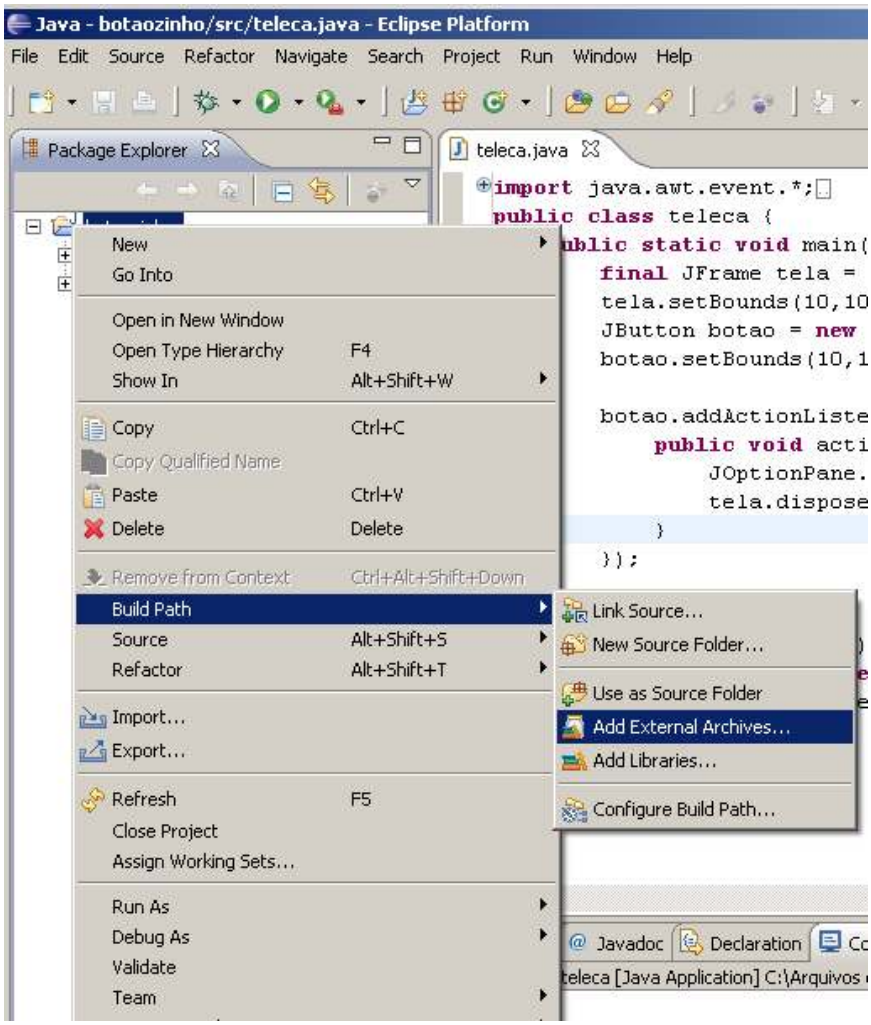
Depois de baixar o componente para alguma pasta qualquer. Eu vou ao meu projeto e clico nele com o botão direito. Então, escolho a opção Build Path e então entro em Add External Archive.

Então, através do sistema de pastas do Windows, ache o arquivo JAR do Connector. Ele será adicionado ao Path do programa. Mais tarde veremos como se faz para "compilar" o conector MySQL junto com o JAR todo.

Veja como fica o projeto com o JAR adicionado:



Agora, só falta construir a aplicação.



A aplicação Java com MySQL integrado

```
import java.awt.event.*;
import java.sql.*;
import javax.swing.*;

public class teleca {
	static Connection con;
	public static void main(String[]args) {
		final JFrame tela = new JFrame();
		tela.setBounds(10,10,600,400);
		JButton botao = new JButton("Aperte");
		botao.setBounds(10,10,170,30);

		botao.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent arg0) {
				JOptionPane.showMessageDialog(null, "Opa! Pidi-pra-pará-parou!");
				conector();
				tela.dispose();
			}
		});

		tela.add(botao);
		tela.setLayout(null);
		tela.setVisible(true);
		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}


	static void conector () {
		try {
			Class.forName("com.mysql.jdbc.Driver");
	}
	catch(ClassNotFoundException err) {
		System.out.println("Classe do driver nao encontrada:");
		System.out.println(err.getMessage());
		System.exit(1);
	}
```

Nesta parte inicial eu estou criando uma aplicação normal como já conhecemos há muito tempo, com um botão que ativa o método "conector()" antes de fechar o programa. A única novidade são os imports novos e a instanciação da classe Connection. Mas isso já vimos no exemplo anterior.

A primeira parte é o uso do forName que nós já usamos antes, para colocar em operação a classe que lida com o MySQL e a JDBC – veja que eu mudei essa linha em relação á aplicação com ODBC.

A seguir, eu vou construir um método que realmente abre a base de dados e faz alguma coisa com ela:

A primeira coisa é carregar alguma coisa para a Connection que aqui tem o nome de "con". Eu vou passar como parâmetros uma URL formada pelo IP do servidor, barra, nome da base de dados. E vou passar o username e a senha.

```
	try {
		String url = "jdbc:mysql://192.168.86.25/fabio";
		con = DriverManager.getConnection(url, "root", "churros");

		Statement comandoSQL = con.createStatement();
```

```
String sql = "Select nome from personagens;";
ResultSet resultado = comandoSQL.executeQuery(sql);
```

Aí, eu criei o Statement, algo que nós já fizemos no exemplo com ODBC. Abaixo, temos uma rotina bem simples que apenas roda um WHILE que vai rodar todo o ResultSet até o final e exibir os dados levantados na consulta SQL da primeira à última linha. Ele vai exibir esses dados numa JOptionPane, o que deixa tudo mais espalhafatoso, mas para fins de exemplo está bom.

```
while(resultado.next()){

        JOptionPane.showMessageDialog(null, resultado.getObject(1).toString());
}


}
catch (SQLException err) {
        //      TODO Auto-generated catch block
}
}
}
```

Se eu quisesse gravar informações, apagar ou dar um UPDATE, eu usaria as rotinas SQL dentro do Statement como eu usei no exemplo da base de dados no ODBC, pois isso nunca muda.

Na verdade, o que muda é o conector importado para o projeto e também a classe que usamos no "forName". Ah, e também os parâmetros necessários para a conexão (o MySQL exige o IP, já o ODBC não, etc.). O resto do sistema continua igualzinho.

Nota sobre componentes de conexão baixados da Web

Eu já disse que os componentes de conexão dos bancos de dados com o Java podem ser baixados da Internet. Pois os manuais e a documentação deles também podem. Normalmente, os fabricantes dos BDs criam os conectores e fazem um pequeno manual. Como o manual é quase sempre em inglês, não tenha vergonha de pesquisar nos Fóruns.

Antigamente, pesquisar uma técnica qualquer na Internet era considerado coisa de amador, que bons programadores conseguiam se virar sozinhos sem ficar olhando os Fóruns. Hoje em dia, isso é um pensamento ultrapassado. As pessoas criam classes e as distribuem pela Web com a intenção simples de tornar a vida dos outros mais fácil (e fazer fama, claro). Então, aproveite!

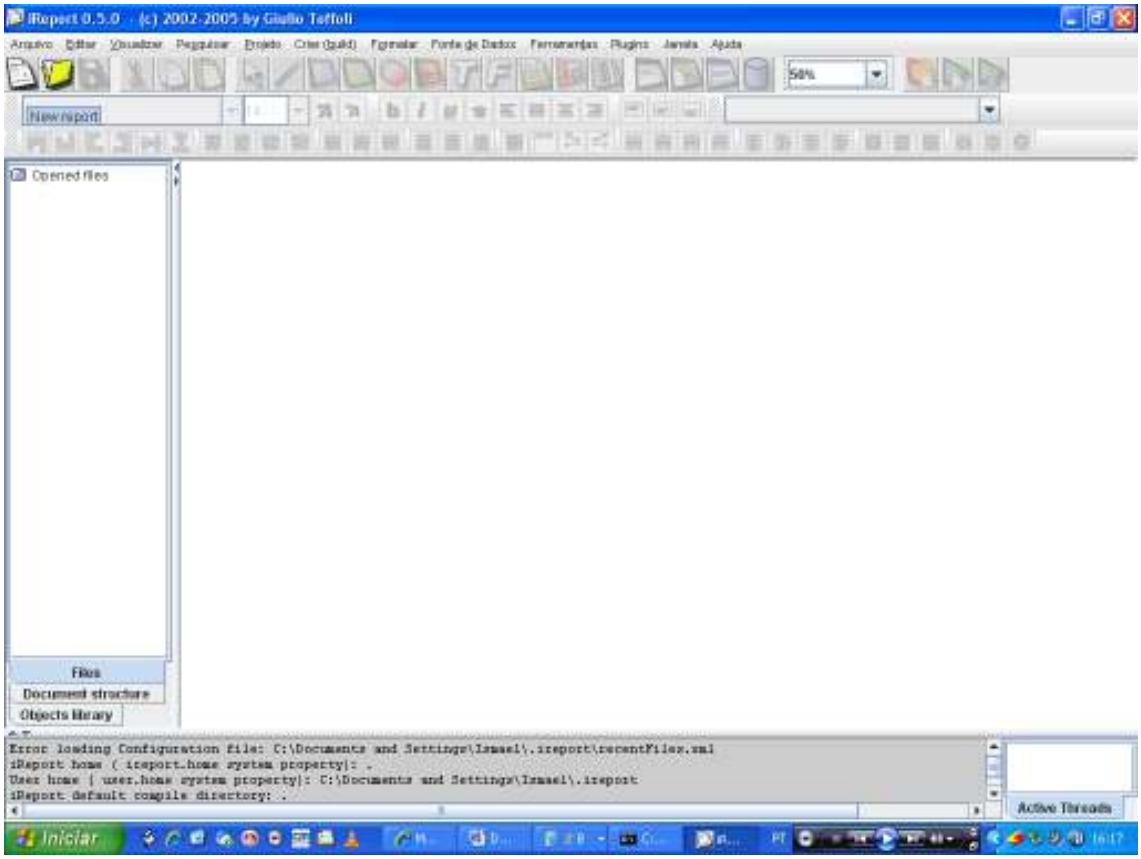
Se o seu SGBD preferido não está listado nesta pequena apostila e se você não encontrou documentação oficial sobre ele em parte alguma, não tenha medo de usar o Google e catar uma classe, mesmo que seja feita por algum outro programador. Você não será considerado burro por não saber, magicamente, o nome de todos os conectores e as classes deles para usar o "forName". Você será considerado burro se tentar reinventar a roda. Faça o seu sistema, venda ele, e fature dinheiro. Sem culpa e sem frescuras. Ninguém é burro por querer facilitar a vida. Burro é quem não completa o projeto no prazo.

Criando formulários de impressão com IReport

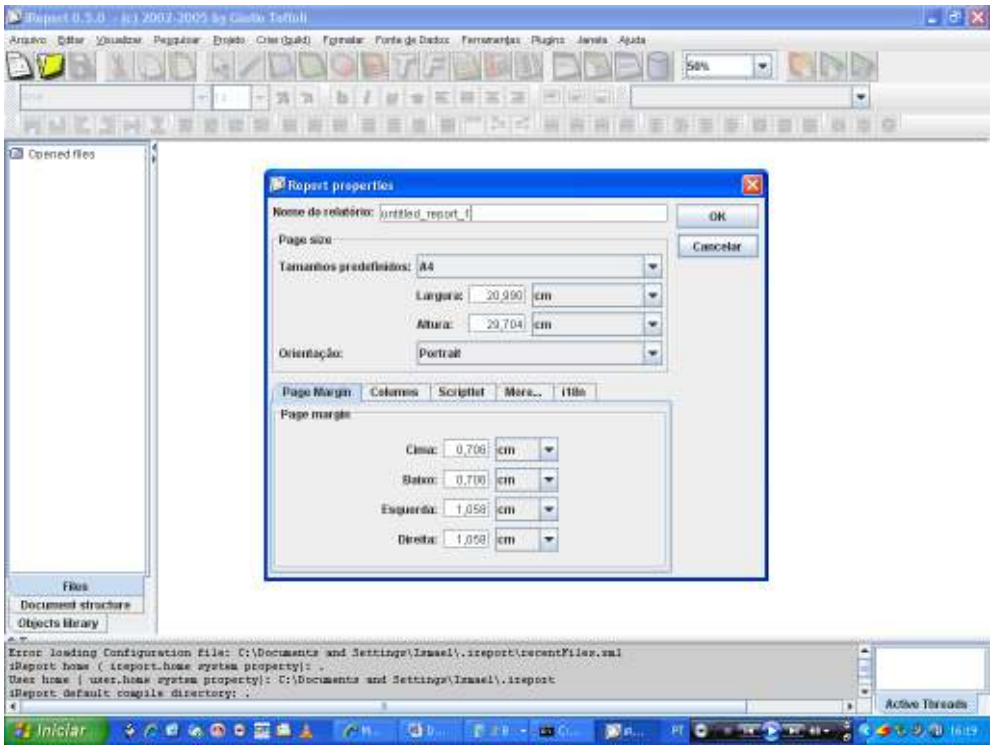
Existem inúmeros editores de relatórios na Internet. Mas nós vamos ter que escolher um deles para trabalhar. Eu escolhi, para este livro, usar o padrão JasperReports, que é gratuito. Um JasperReport é um arquivo XML com instruções sobre a montagem de um relatório utilizando os dados de um banco de dados qualquer.

Dá para escrever um JasperReport "na unha", usando o Bloco de Notas. Mas é mais fácil fazer isso com o editor apropriado. No caso, estou falando do IReports, um programa de interface visual no qual podemos desenhar nosso relatório para impressão puxando os dados no lugar certo, sem grandes esforços.

Vamos então criar um relatório no IReports.

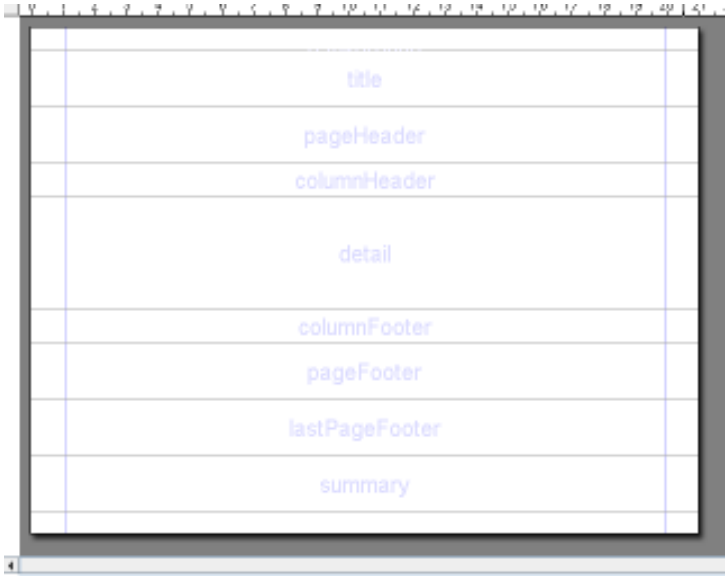


Esta aí é a interface do IReport. Vamos trabalhar, no começo, apertando o botão da extrema-esquerda para criar um NOVO documento.



Surge então uma caixinha com configurações de página. Aqui se escolhe o tipo de papel e as margens. Para quem conhece Word e outros derivados, esta tela não tem nenhuma novidade.

Ao iniciar o novo projeto, surge para nós uma tela com uma folha de papel em branco. Esta folha está separada em partes. Cada parte corresponde a um pedaço do documento que vai sair impresso.



TITLE – É o título do documento, e aparece apenas na primeira página. Serve para fazer capas e coisas assim.

PAGEHEADER – É o cabeçalho do documento. Aparecerá no topo de todas as páginas.

COLUMSNHEADER – São os cabeçalhos das colunas. Aparecerão sempre antes dos dados, mas nós podemos fazer o IReport repetir estes cabeçalhos dentro dos grupos (veremos depois o que são grupos de dados).

DETAIL – Aqui vão os dados em si, empilhados, correndo uma tabela do banco de dados qualquer.

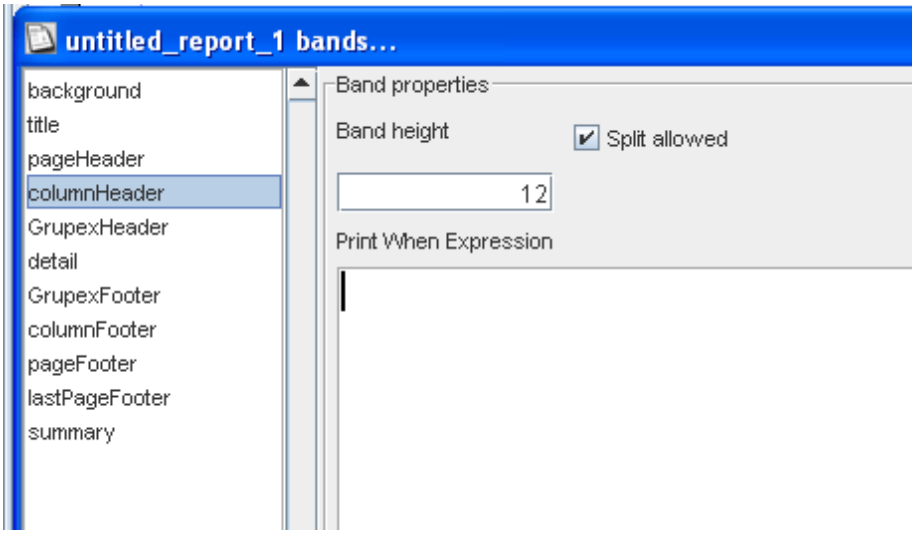
COLUMNFOOTER – É o rodapé do espaço iniciado com ColumnHeaders.

PAGEFOOTER – É o rodapé da página, igual em todas as páginas.

LASTPAGEFOOTER – É o rodapé da última página, uma espécie de antônimo para o Title.

SUMMARY – É um espaço no final do documento utilizado para colocarmos sumários ou resumos de encerramento.

Podemos editar as propriedades de tamanho e setar se é visível ou não uma banda usando o Gerenciador de Bandas. É muito fácil abrí-lo. Basta clicar sobre algum ponto vazio da folha com o botão direito e escolher “Properties”. Daí, veremos uma tela assim:



Aqui, não temos grande coisa. Vemos as bandas em uma lista na esquerda, e ao lado o Band Width. O Band Width representa a altura de cada banda. Se colocarmos um zero ali, a banda desaparece.

Já o campo “Split Alowed”, quando marcado, permite que uma determinada banda, ao não caber em uma única página, seja repartida. Se estiver desmarcada, ela fica presa à primeira página e sai cortada – e aí, azar!

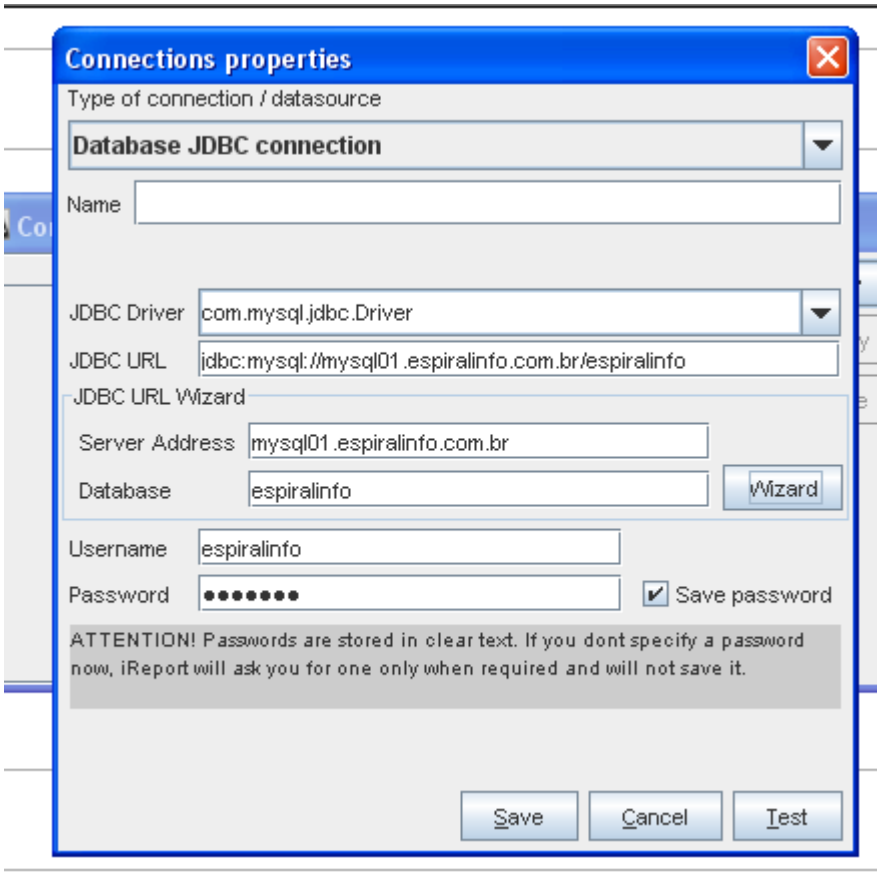
Colocando dados no Report

Após criarmos a página, nos falta ligar este relatório á base de dados. Para isso, usamos o menu Fonte de Dados.

Dentro de Conexões, nós podemos criar conexões com o BD. Trata-se de um exercício simples de preencher campos. Se você optar por escrever o endereço do servidor e o nome da base de dados nos campos designados para tal, nunca se esqueça de clicar em Wizard, para passar estes parâmetros para a string de conexão no segundo campo, de cima para baixo. É ele quem vale na hora de conectar.

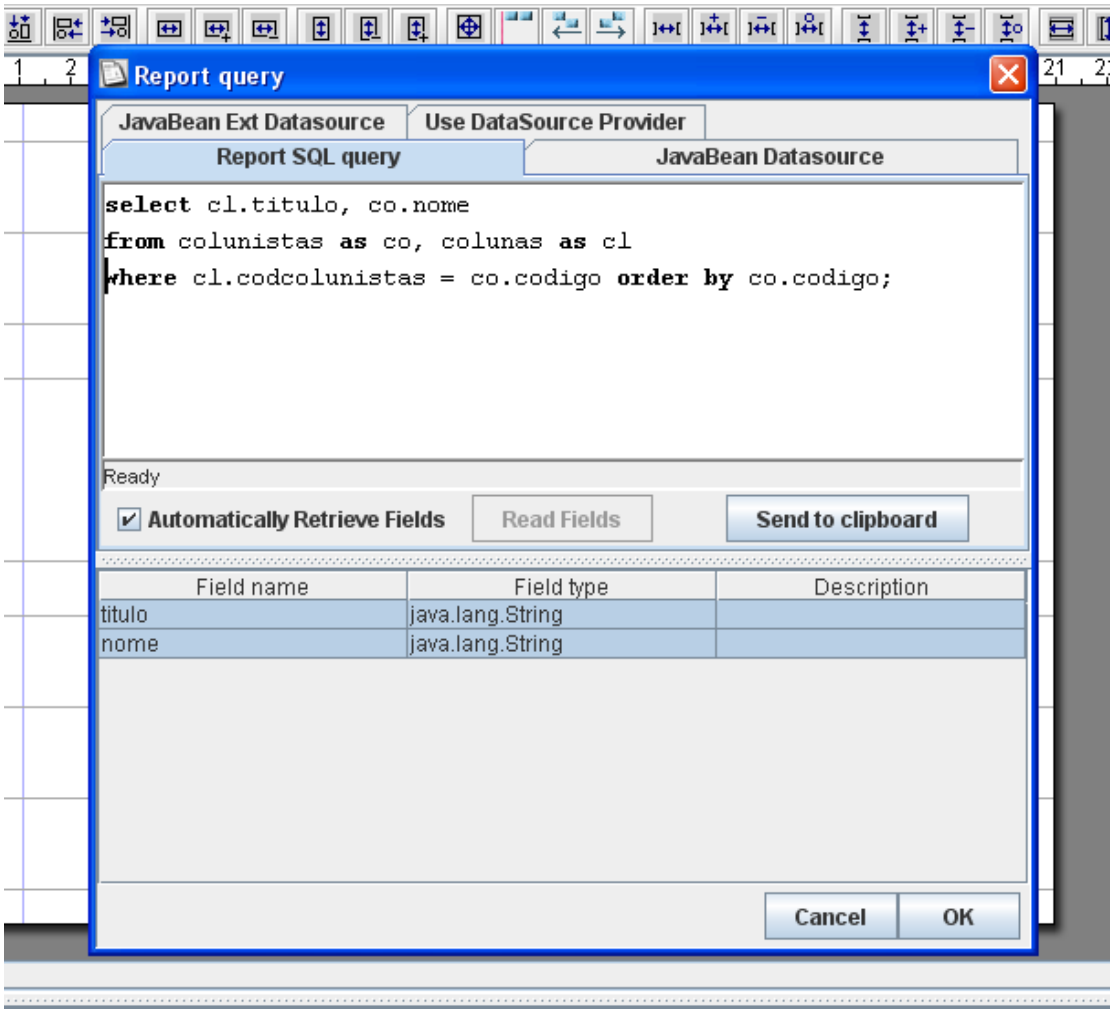
Não confunda: Server Address é um endereço IP ou domínio do servidor para encontra-lo em uma rede. Se o próprio PC onde está sendo gerado o IReport for o portador do BD usado, então será preciso escrever “localhost”.

Não se preocupe com o fato de estarmos usando uma conexão com IP e tudo para ligar nosso IReport a uma base de dados. Isso só vai ter validade para usarmos nos testes dos relatórios dos nossos sistemas. Na hora de rodar o programa nos computadores de um cliente, por exemplo, nós podemos dizer, através do programa em Java, qual é o IP do Banco de Dados. Então, crie seu Report com uma base de dados que te sirva para os testes do programa, e depois quando ele for implementado nos clientes, os Reports continuarão a funcionar numa boa.



Depois, no próprio menu “Fonte de Dados”, escolha a opção Consultas do Relatório. Aqui, você deve ir à Report SQL Query e escrever uma Query SQL que vai retornar dados, preenchendo o relatório com

informações a serem trabalhadas.



De volta à tela principal do IReports, chegou a hora de dar uma olhada no menu da parte inferior esquerda. Lá, vamos ativar a aba Objects Library, que contém os objetos que formam o nosso relatório.

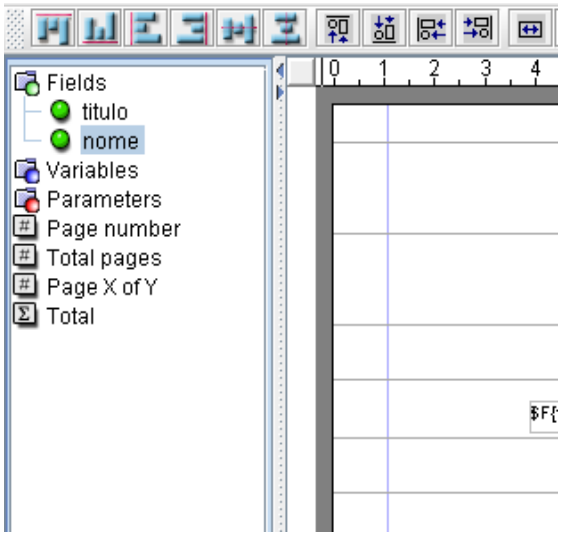
Os principais tipos de dados a colocar dentro do relatório são os FIELDS, os VARIABLES e os PARAMETERS.

FIELDS – São campos tirados da base de dados. Colocar um Field faz o Report imprimir os dados que aquele campo contém.

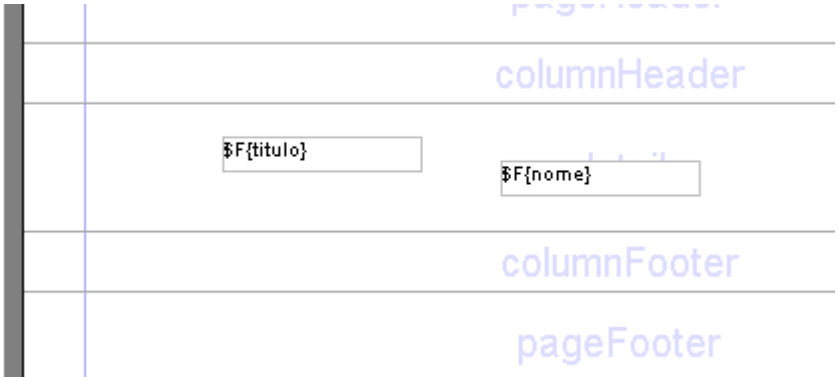
VARIABLES – São variáveis. Podemos cria-las para executar funções de soma ou outras coisas parecidas.

PARAMETERS – São utilizado para "importar" os valores passados pelo programa feito em Java. Estes parâmetros devem ser passados no momento de se chamar a impressão.

Eu posso criar variáveis e parâmetros, clicando com o botão direito sobre os menos da esquerda e escolhendo ADD, e então, o que eu quero adicionar.

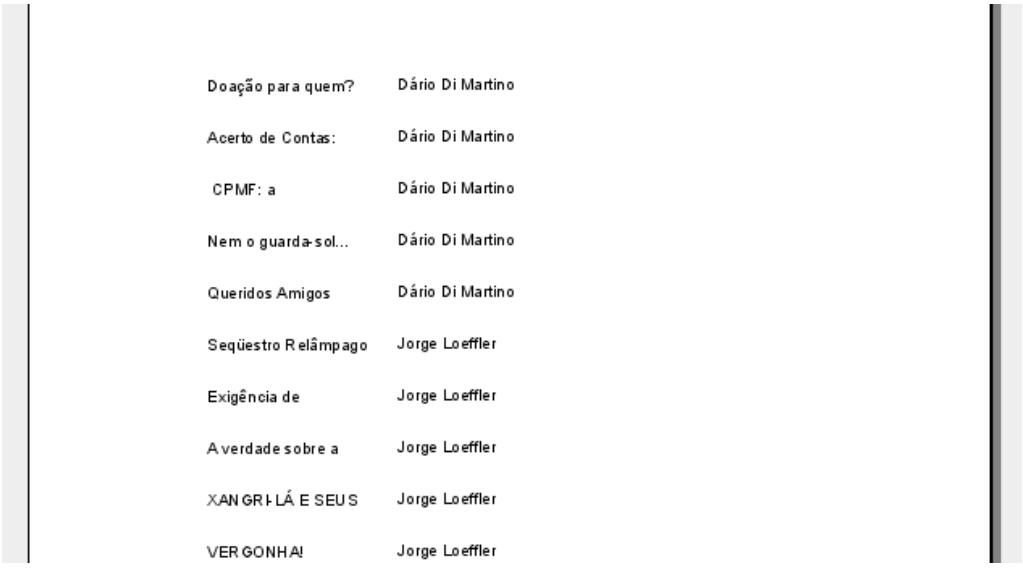


Para fazer os campos, variáveis e parâmetros aparecerem no Report, eu devo arrasta-los para dentro da folha branca, na sessão que eu quiser.



No meu exemplo, eu usei como banco de dados a base de dados do meu site, que aliás, você deveria visitar. Mas isso é outra história. O fato é que eu sei meu site, e com minha Query SQL, mandei retornar os títulos das colunas de opinião do site e o nome do autor de cada uma. Obviamente, aparecerá uma enorme lista de títulos com nomes repetidos, pois eu mandei ordenar pelo código do autor.

Então, o resultado será esse:



Doação para quem?	Dário Di Martino
Acerto de Contas:	Dário Di Martino
CPMF: a	Dário Di Martino
Nem o guarda-sol...	Dário Di Martino
Queridos Amigos	Dário Di Martino
Seqüestro Relâmpago	Jorge Loeffler
Exigência de	Jorge Loeffler
A verdade sobre a	Jorge Loeffler
XANGRI-LÁ E SEUS	Jorge Loeffler
VERGONHA!	Jorge Loeffler

Montando grupos

Não parece ser muito promissor. E não é. Mas aí, vamos ver uma coisa nova. Clique no botão Groups.

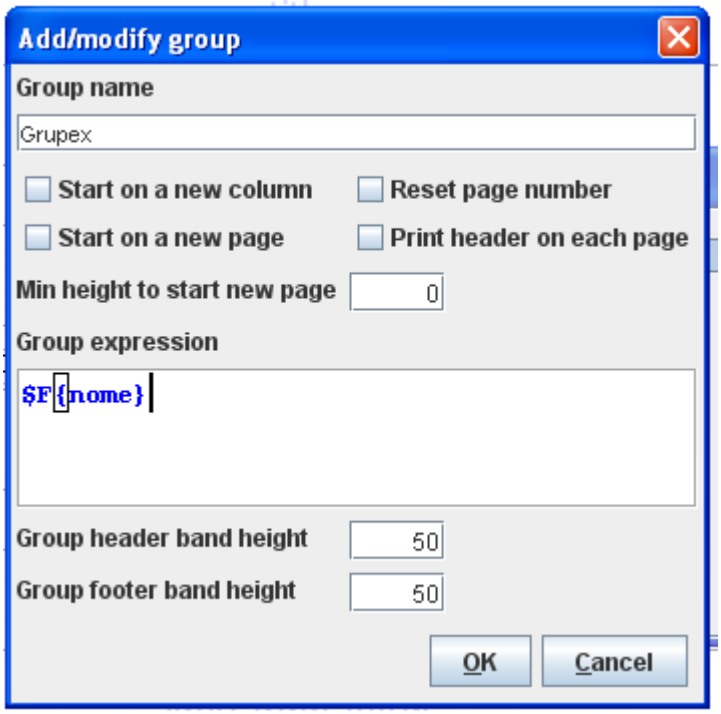
Aparecerão os nomes dos grupos que nós temos no Report, mas como não temos nenhum ainda, não aparecerá nada disponível – a não ser o botão NEW, que nós vamos pressionar, claro.

Uma vez dentro do quadro de criação de grupos, precisamos definir duas coisas: o nome do nosso grupo e qual será a expressão agrupadora dele. O nome é algo bem simples, basta escrever qualquer coisa ali dentro.

Eu vou escrever “Grupex”.

Já a expressão segue uma lógica mais complexa, mas não muito. É preciso dizer que tipo de coisa se está usando como parâmetro de agrupamento – se é um campo do banco de dados, se é um Parameter ou se é uma Variable. Para distinguir um do outro, usa-se uma letra (F para Field, campo; V para Variable; P para Parameter).

Eu, no caso, estou agrupando as colunas dos meus articulistas de opinião dentro de grupos formados pelas colunas de um mesmo autor. Então, vou aproveitar o campo NOME, escrevendo a expressão $F{nome}.



Existem umas caixinhas para eu marcar também. Vamos ver o que elas fazem.

Start on a New Column – repete os ColumnHeaders dentro do grupo.

Start on a New Page – Ele quebra a página a cada vez que um novo grupo vai se iniciar.

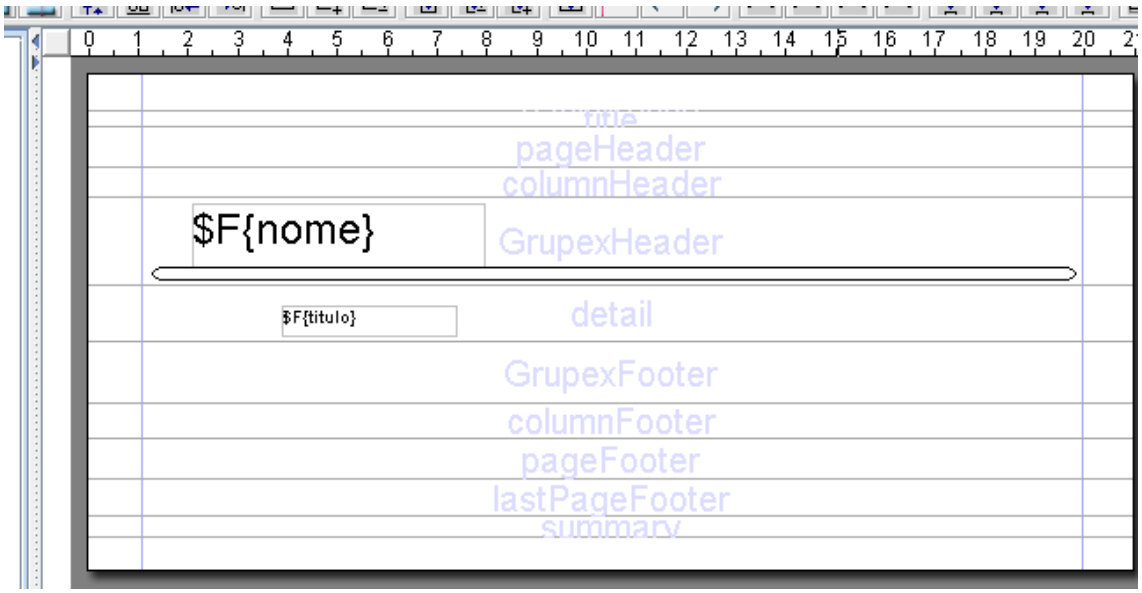
Reset Page Number – A primeira página de cada grupo será numerada como se fosse a primeira do relatório, caso usemos numeração de páginas (veremos isso adiante).

Print header on each Page – Caso o grupo extenda-se por mais de uma página, o GroupHeader será repetido em cada uma delas.

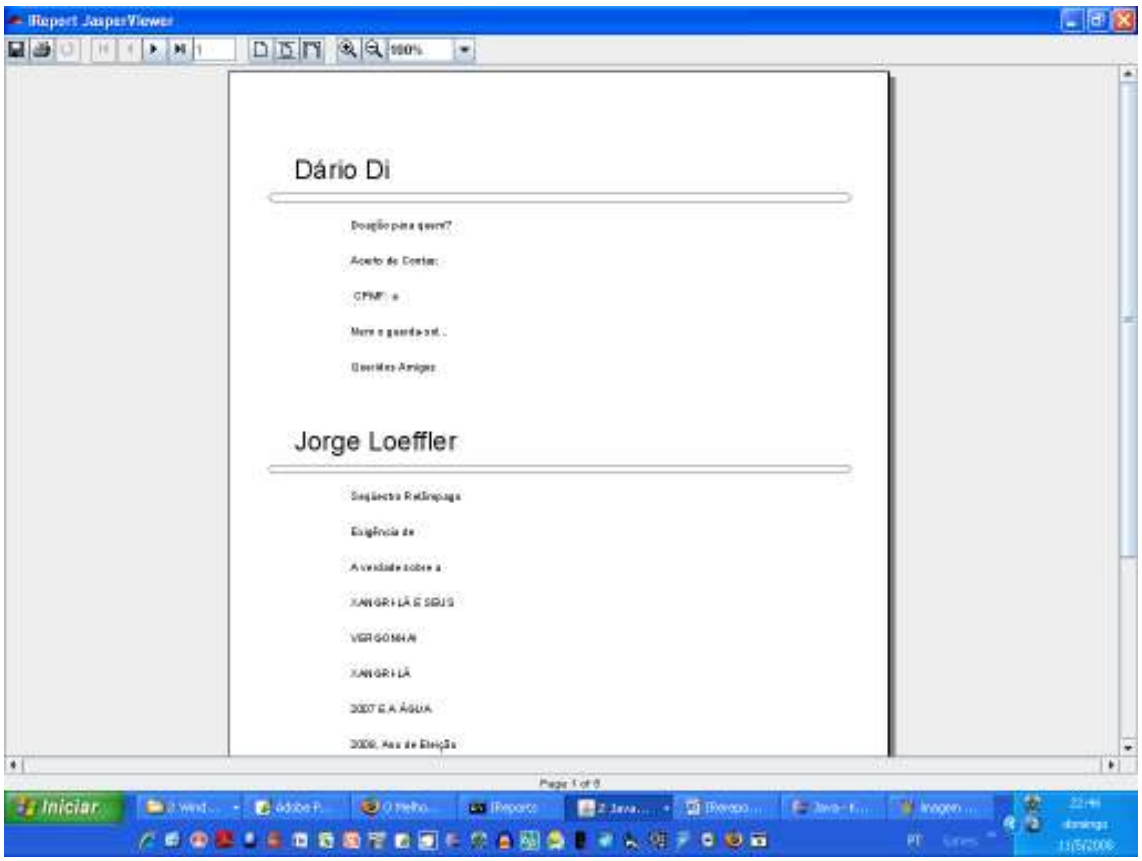
O que é o GroupHeader?

É um cabeçalho que existe para cada item de um determinado grupo. Seu nome não aparece como "GroupHeader", e sim como "<nome_do_grupo>Header". Assim, se eu criei o grupo "Grupex", eu terei um "GrupexHeader".

Não se preocupe. Ao criar um grupo, o próprio IReport coloca um Header para ele.



No meu exemplo, veja que eu usei o GrupexHeader para colocar o NOME do colunista cujos textos serão listados, e coloquei um retângulo meio cilíndrico (ou sei lá o que é aquilo ali). Esse desenho se repetirá como cabeçalho de cada grupo formado pelos textos de um mesmo autor. O resultado fica assim:



Claro, meu relatório está uma porcaria, muito feio e sem planejamento gráfico algum. Mas você vai criar relatórios bonitinho. Isso aqui é apenas um exemplo. Mas já deu para ver como se faz, certo?

Bem, e se eu quiser colocar um grupo dentro do outro? Sem problemas! Basta criar mais grupos lá no botãozinho Groups. Você poderá depois arrastá-los com o mouse para dentro e para fora uns dos outros, montando sua própria hierarquia.

Testando o Report

Quando eu crio um Report, como sei que ele está ficando legal? Através da Fé? Não! Eu uso botões exatamente criados para testes.

Este comando testa seu relatório com os dados do Banco de Dados, dando uma visão realista do trabalho como ficará no fim.

Este botão roda o relatório com uma base de dados vazia, apenas para conferirmos o visual que ficará. Isso evita que tenhamos que rodar consultas pesadas ao BD a cada pequeno teste.

Botões do IReport

Já que viemos até aqui, vamos ver então os botões do IReport e para que servem:

O botão com o T insere um campo de texto estático – algo que escrevemos aqui como se fosse uma Label apenas para indicar que dado vem a seguir ou para mostrar informações fixas.

O primeiro botão desta carreirinha é um gerenciador de Fields, Variables e Parameters. O segunto é o já explicado Groups. O terceiro gerencia as bandas do relatório (que nós já abrimos de outro jeito). E o quarto é o gerenciador de bases de dados. Depois deles, vem o Zoom.

Esses três botões do canto têm a mesma função que teriam no Word: criar um novo arquivo, abrir um existente e salvar o que estamos fazendo.

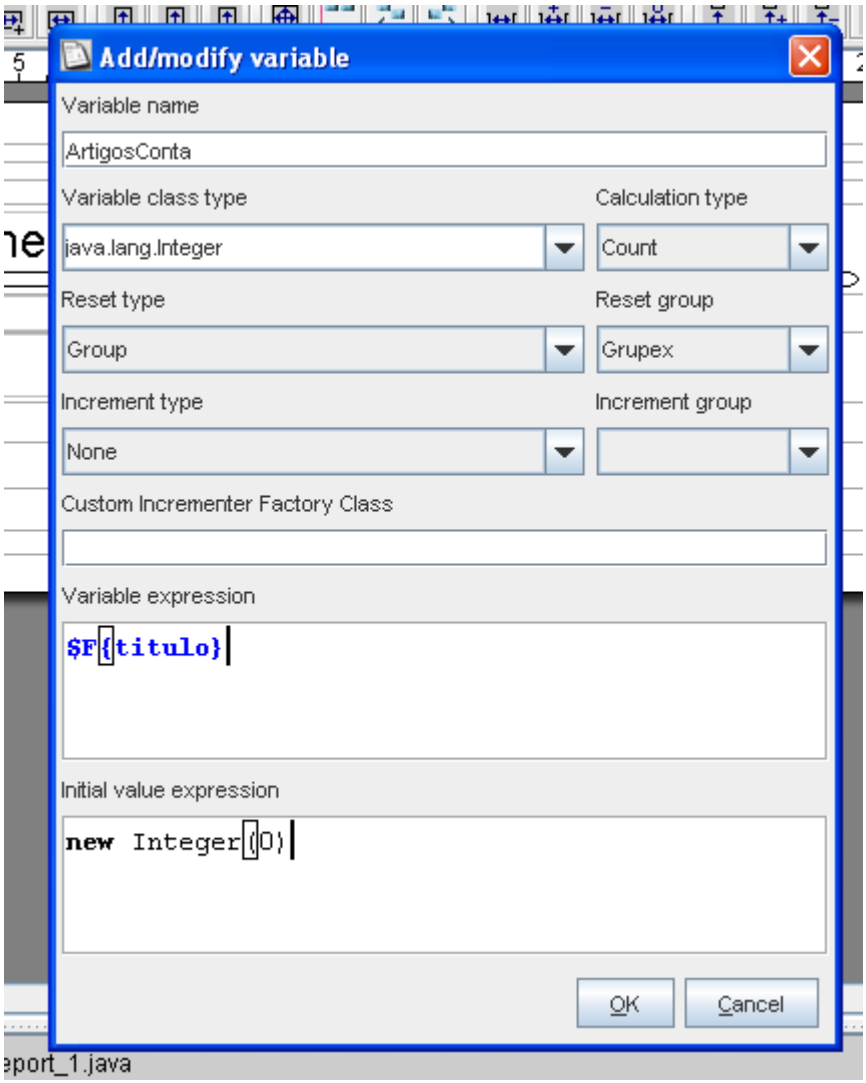
O botão com a seta faz o Ireport “des-selecionar” uma utra opção eventualmente clicada. A linha, bem... ela desenha linhas. Depois, temos as ferramentas para desenhar retângulo, círculo, retângulo com as quinas arredonadadas.
O botão laranja com um pôr-do-sol desenhado serve para importarmos fotos de arquivos externos como JPG ou BMP, GIF, essas coisas.

Esses botões aqui, todos, abrem caixinhas que nos ajudam a criar funções mais complexas no nosso projeto.
A primeira trata de sub-relatórios, a possibilidade de inserirmos um relatório dentro do outro.
A segunda, de gráficos (no melhor estilo Excel).
A terceira é uma das mais completas, e serve para inserirmos códigos de barra. Ela tem um campo chamado “Barcode Expression” onde botamos o valor a ser convertido em código de barras. Mais adiante vamos ver como manipulá-lo para obter um emissor de códigos de barras.

Uma das coisas mais legais, apesar de não parecerem, são os elementos gráficos como o retângulo e o círculo. Se você clicar duas vezes sobre um objeto, ele lhe mostrará suas características. E se você manipular estas características, poderá então fazer gráficos. Cada figura tem, por exemplo, Width e Height, o que nos permite setar sua altura e largura com base em dados da base de dados ou cálculos dentro do Report mesmo!

Trabalhando com variáveis

Quando queremos criar uma variável, usamos o seguinte método: clicamos com o botão direito sobre as Variables, Fields ou Parameters, e selecionamos ADD – VARIABLE. Aparecerá uma tela para nos ajudar a fazer os cálculos que queremos.



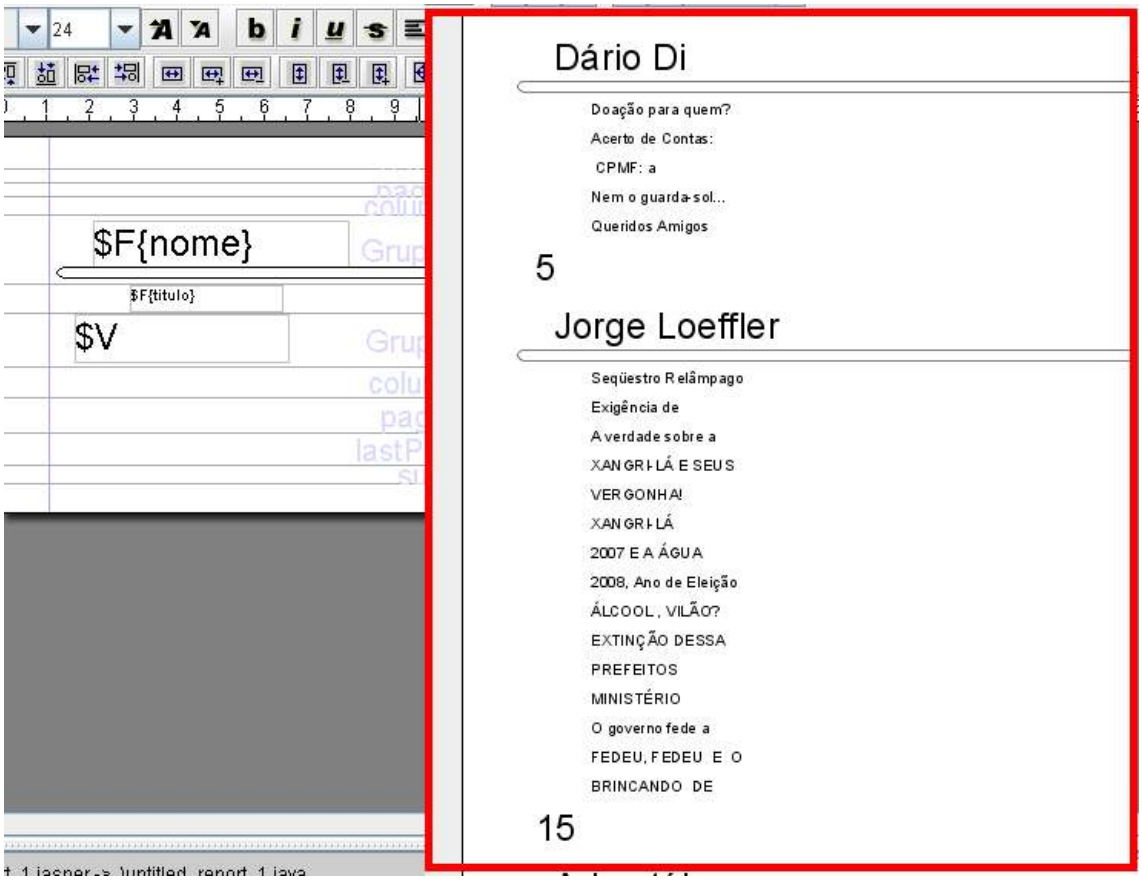
Só para fins de demonstração, veja que eu estou criando uma variável que vai calcular quantas colunas de opinião cada colunista escreveu. Eu defini o tipo dela como “Java.lang.Integer”, ou seja, um número inteiro. Coloquei o tipo de cálculo dela em COUNT, ou seja, contagem (ela vai ficar somando cada rodada de Detail que rolar, ou seja, vai contar quantos registros ocorreram no Detail).

Eu preciso dizer o que, afinal de contas, esta variável está calculando. Então, eu defini isso em Variable Expression, onde botei o campo “titulo” tirado da minha base de dados.

Veja que em Reset Type, eu marquei Group. Ou seja, a contagem será reiniciada a cada grupo. Eu preciso definir qual grupo, então marquei o Grupex na ComboBox ao lado.

Por fim, eu preciso inicializar esta variável com algum valor. Eu coloquei um New Integer, um novo número Integer, que é zero.

No final, eu vou colocar minha variável logo no final da lista de cada colunista (em GrupexFooter), e vejamos o resultado:



Depois, veremos como usar Parameters. Não vem ao caso agora. Eu vou finalizar o processo salvando o arquivo JRXML para dentro da pasta onde o meu programa Java está sendo desenvolvido. Mais tarde, este JRXML deverá acompanhar o resto do projeto também.

Como integrar o JasperReport ao programa Java

Para começar, vamos precisar fazer a importação de uma montanha de pequenos componentes que estão dentro da pasta de instalação do IReport, sub-pasta Lib. Eles são:

Commons-digester

Commons-collection

Commons-logging

Commons-logging-api

Commons-beanutils

Commons-itext (esse serve para quando formos fazer um PDF a partir de um relatório que deveria ser impresso).

Daí, construo um programa que vai, primeiro, compilar meu relatório para dentro de uma variável, criar uma conexão para dizer ao Report que eu estou usando aquela conexão, e gerar o relatório. A primeira coisa a fazer é importar as classes necessárias. A lista é enorme:

```
import java.io.IOException;
import java.net.URL;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.HashMap;
import net.sf.jasperreports.engine.*;
import net.sf.jasperreports.engine.design.JasperDesign;
import net.sf.jasperreports.engine.util.JRLoader;
import net.sf.jasperreports.view.JasperViewer;
import org.apache.commons.digester.*;
import org.apache.*;
import org.apache.commons.collections.*;
import org.apache.commons.logging.*;

public class imprimir {
	static Connection con;
	public static void main (String[]args) {
		new imprimir().imprime();
	}
```

Aí acima, eu criei o método MAIN que apenas instancia a própria classe à qual ele pertence e depois roda o método IMPRIME. Acontece que o método MAIN é Static. Toda variável ou objeto dentro de um método Static é também Static, então, eu não posso rodar direto a impressão dentro do MAIN porque diversos objetos utilizados neste processo não suportam ser Static. O jeito é criar outro método, não-Static, e chamá-lo a partir do MAIN.

```
	void imprime() {
	try {
		Class.forName("com.mysql.jdbc.Driver");
	}
```

Aí acima, temos o bom e velho Class.forName, que vai primeiro verificar a existência da classe de componentes que usamos para conextar à base de dados MySQL. Se houver um erro, o problema é naquela peça de hardware que fica entre o teclado e o encosto da cadeira. Você esqueceu de incluir no Build Path a classe certa.

```
	catch(ClassNotFoundException err) {
```

```
                System.out.println("Classe do driver nao encontrada:");
                System.out.println(err.getMessage());
                System.exit(1);
        }
        try {
                String url = "jdbc:mysql://192.168.86.13/javanoite";
                con = DriverManager.getConnection(url, "root", "123");
        }
        catch (SQLException err) {
                //      TODO Auto-generated catch block
        }
```

Bla bla bla… Acabamos de repetir o código que já usamos para incorporar o programa em Java a uma base de dados MySQL. Nada de novo. Se você ficou em dúvida sobre o código acima, por favor faça uma pergunta ao instrutor (se você não estiver estudando sozinho em casa, claro, mas neste caso mande sua dúvida ao autor da apostila). O código que acabamos de ver deveria ser conhecido, porque trata só da conexão a banco de dados. O JasperReport não tem como saber, dinamicamente, onde está o BD do nosso programa e nós não vamos criar um grupo de relatórios para cada cliente em separado. Então, vamos deixar o papel de "guia" sobre a localização do BD nas mãos do programa.

```
                try {
        JasperReport jr = JasperManager.loadReport("exemplo.jasper");
        JasperPrint jp = JasperManager.fillReport(jr, new HashMap(), con);
        JasperExportManager.exportReportToPdfFile(jp, "arquivo.pdf");
```

Os três commandos acima têm funções diferenciadas. O primeiro deles serve para instanciar um objeto JasperReport, "puxando" o arquivo exemplo.jasper (que eu criei antes, no IReport), e deixando-o pronto para eu fazer o que quiser dele. O arquivo externo na verdade é o meu modelo de relatório.

O segundo comando preenche o relatório com dados. Mas quais dados? Bom. Ele tem como argumentos o JasperReport "JR" (o nome foi escolha minha, pode ser qualquer coisa), depois tem um HashMap com argumentos que nós não usaremos agora, e finalmente a conexão que deverá ser usada pelo JasperReport (a boa e velha CON). Se você tiver feito direitinho o trabalho de criação das rotinas SQL no IReport, então tudo deverá dar certo, e a variável JP armazenará um relatório preenchido.

O terceiro comando lida com exportação do relatório preenchido. Eu não preciso usá-lo pois posso usar o relatório pronto (que está no objeto JP) para visualizar e imprimir. Só que, digamos, eu resolvi exportar para um PDF o relatório. O comando está aqui porque é interessante que vocês aprendam como se faz. Ele exporta o relatório prontinho que está em JP para um arquivo (no caso, "arquivo.pdf").

```
                }
                catch (JRException e) {
                        System.out.println("Deu pau");
                        e.printStackTrace();
                }
        }
}
```

Acima, temos apenas o tratamento do Catch daquele TRY no qual geramos o relatório. Porque, digamos que algum gênio pensador tenha deletado o arquivo JASPER, ou algo assim.

Depois de fazer tudo isso, podemos simplesmente visualizar nosso relatório na tela e dar ao usuário a possibilidade de imprimi-lo com o Jasper Report Viewer:

```
JRViewer jrv = new JRViewer(report);
viewer.getContentPane().add(jrv);
viewer.show();
```

Agora, se eu gerei um PDF e quero abrir o PDF, não tem problema, é so usar:

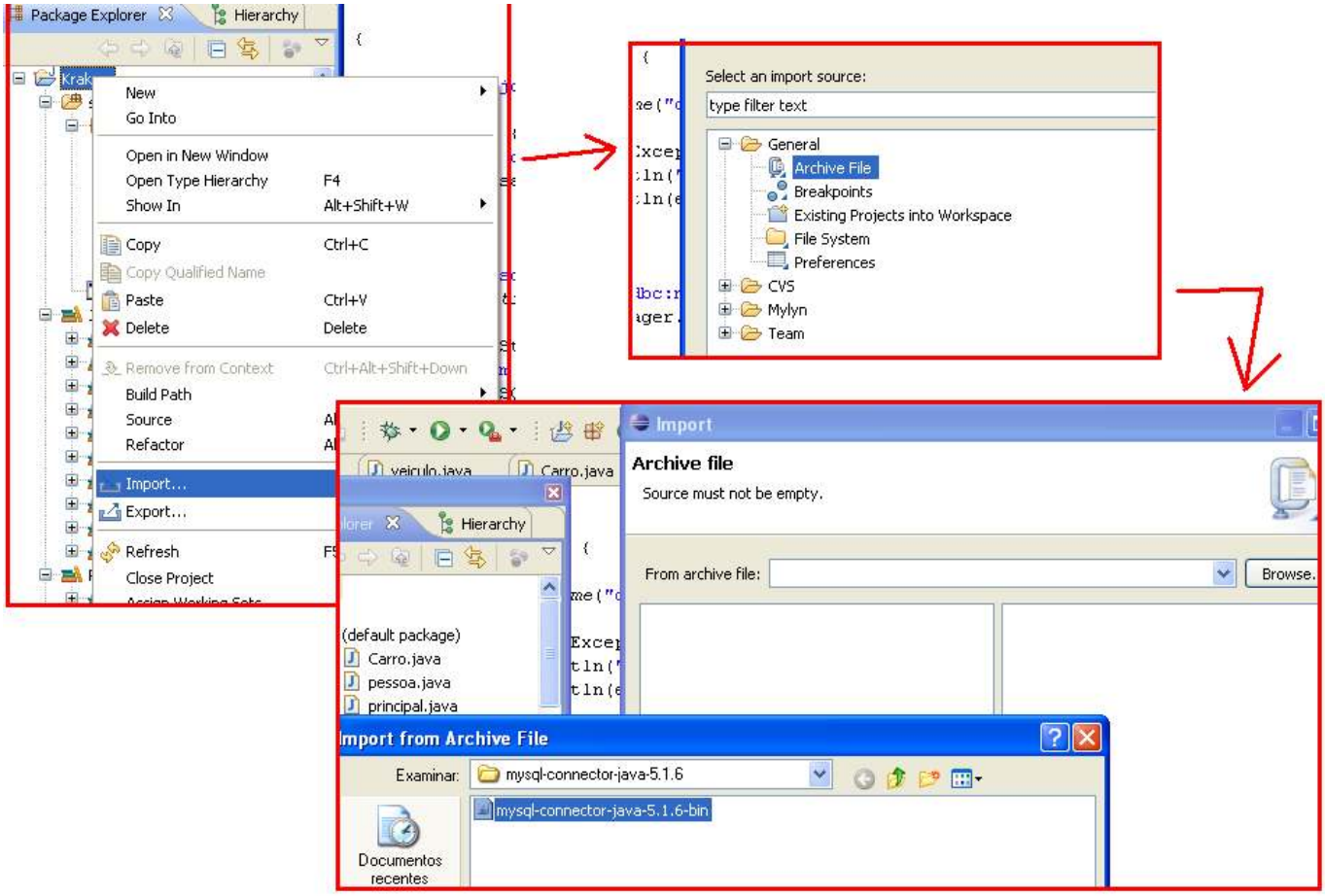
```
Runtime.getRuntime().exec("rundll32 url.dll,FileProtocolHandler " + "arquivo.pdf");
```

Note que na linha acima eu usei a classe Runtime. Esta classe é uma espécie de “genérico” significando “o sistema sobre o qual estou rodando seja qual for”. Então, dou um getRuntime() (o programa descobre onde estamos rodando). E manda executar um programa (bem evidente no método EXEC).

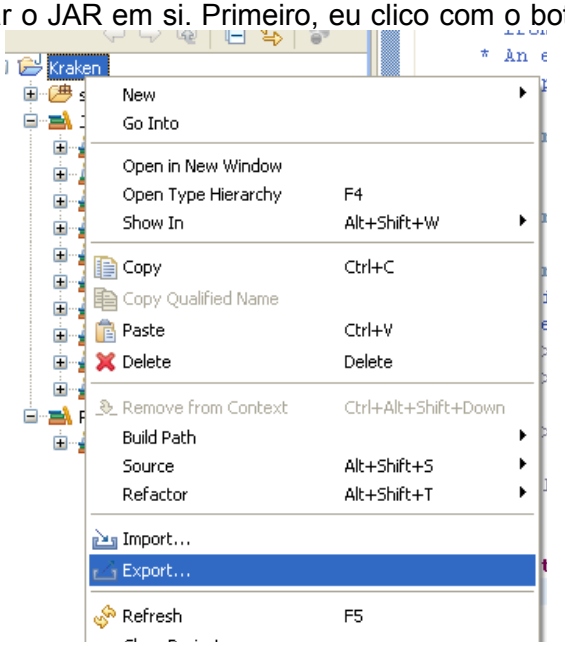
Depois, eu dou um comando que bem poderia ser comparado com um comando no MS-DOS. Aliás, é a mesma coisa. Eu mando ele rodar o componente que abre um arquivo descobrindo qual é o programa-padrão para aquele tipo de arquivos. No caso, como eu mandei ele rodar o arquivo ARQUIVO.PDF, o sistema vai certamente abrir o Adobe Reader e me mostrar o conteúdo do PDF.

Compilando o projeto depois de pronto

Para “compilar” seu projeto criando portanto um arquivo JAR executável, você deve seguir alguns passos. O primeiro deles é IMPORTAR todas as classes que você está usando e que não fazem parte do núcleo do seu projeto para dentro do projeto. Fazemos isso usando o IMPORT, e dando um BROWSE no arquivo. Abaixo, alguns screenshots deste processo.

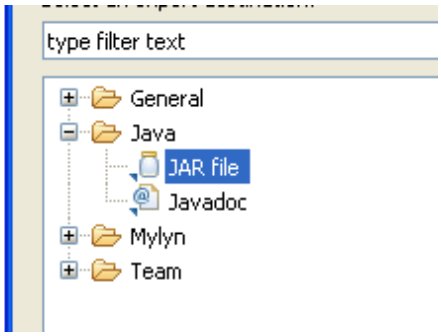


Agora, eu vou criar o JAR em si. Primeiro, eu clico com o botão direito sobre o nome do meu projeto e

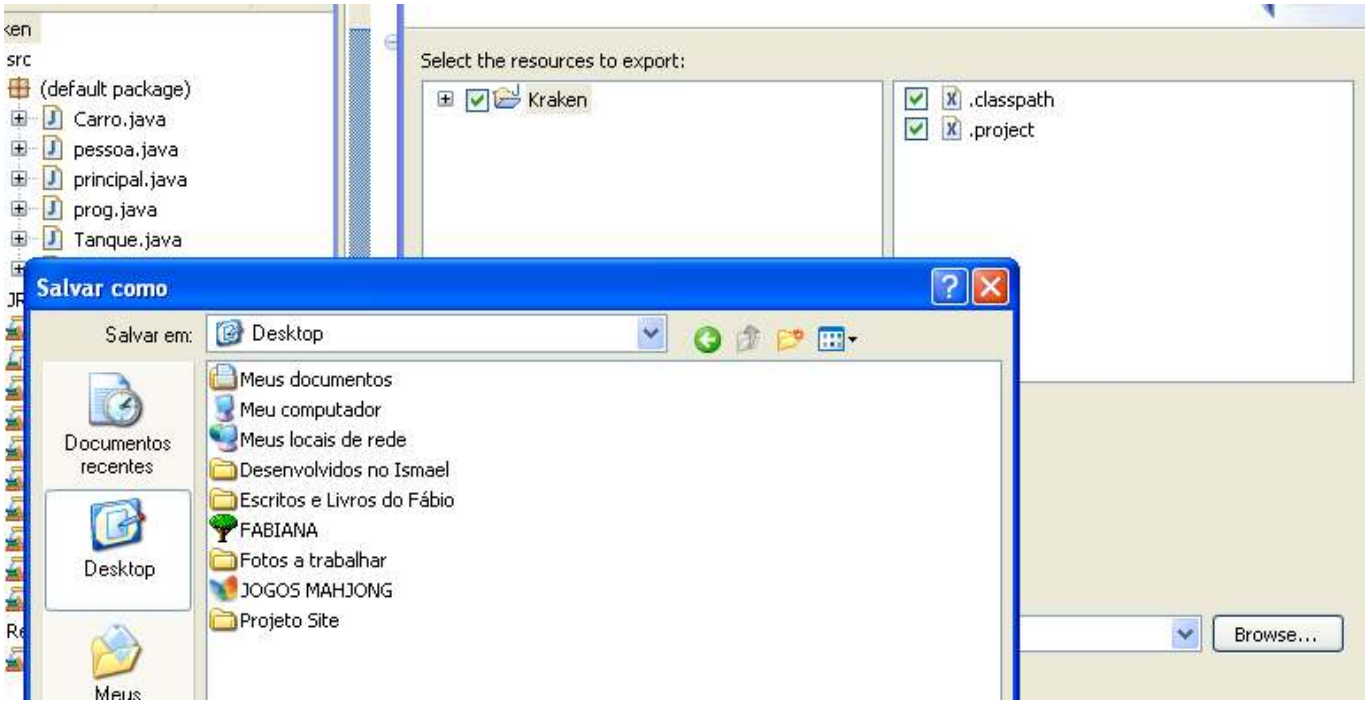


escolho EXPORT.

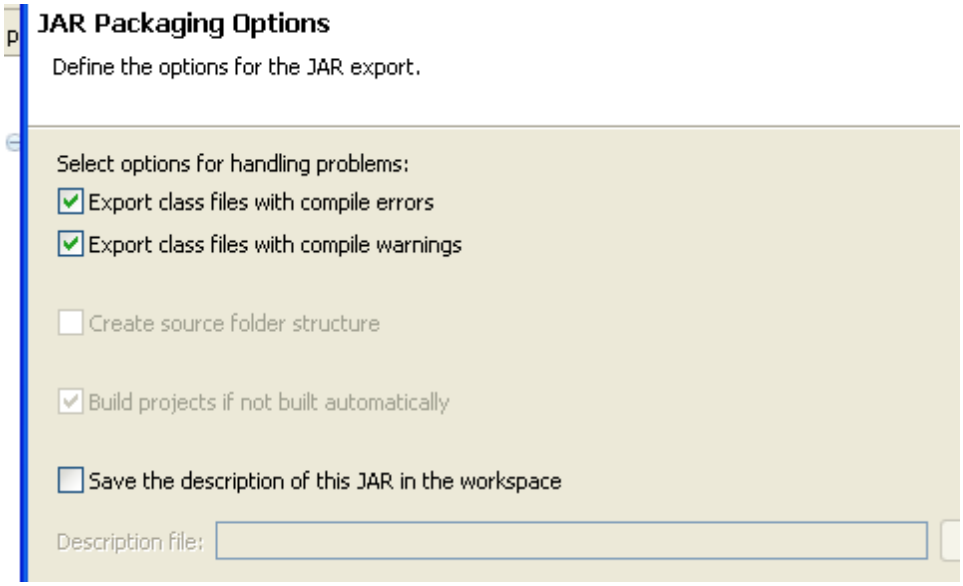
Em seguida, eu vou escolher JAR FILE, que é o que eu quero fazer mesmo.



Em seguida, eu entro em uma tela na qual devo dar um BROWSE e escolher a pasta e nome de arquivo para o meu projeto fechado. Vemos isso na figura abaixo.



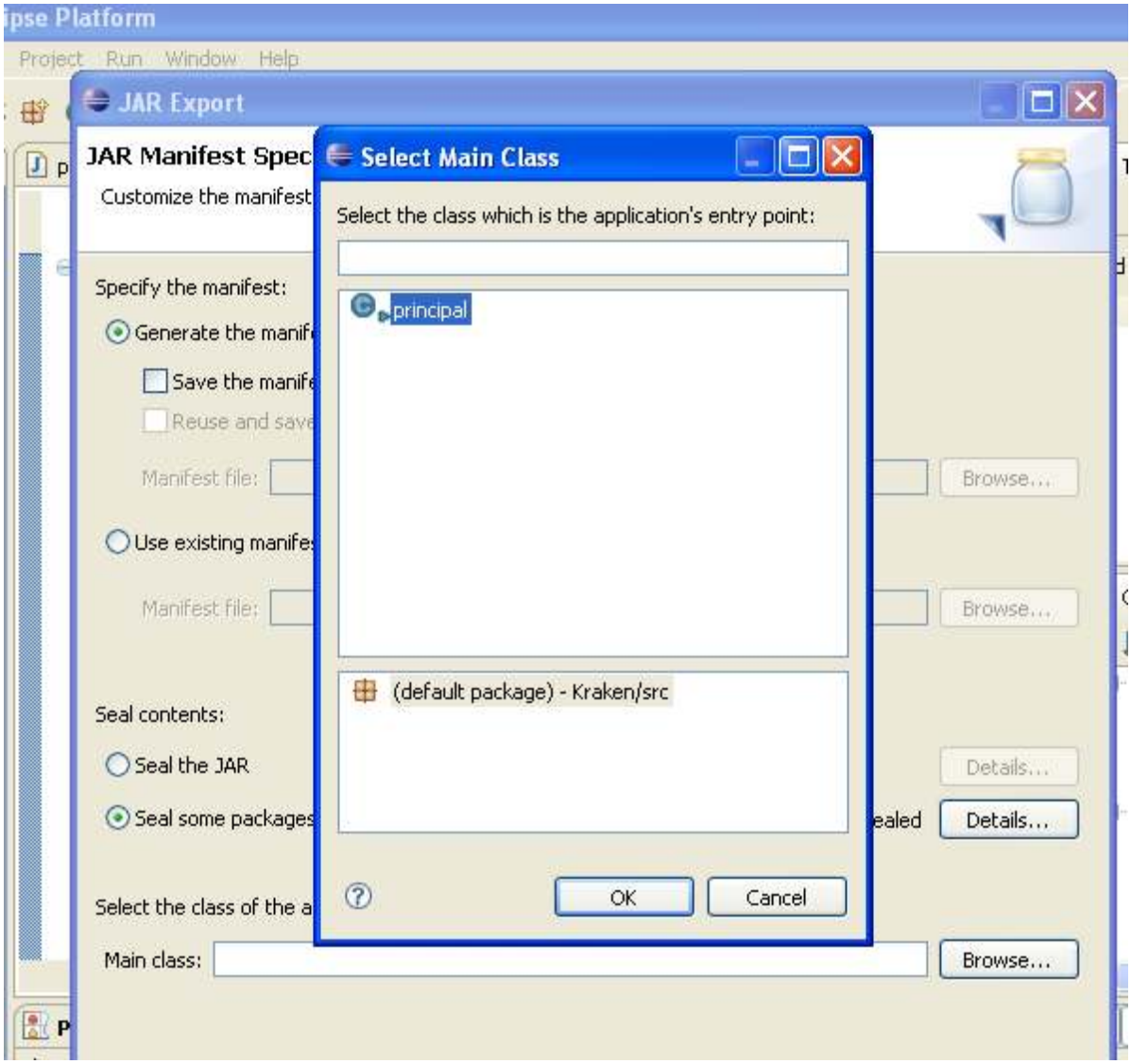
Depois de clicar NEXT, eu passo por uma tela na qual devo dizer se quero fechar no JAR até mesmo classes com erros ou Warnings. Warnings são avisos sobre criação de variáveis que não serão usadas e coisas do tipo. Nada que trave o programa.



Agora, cheguei à última tela. Ela tem escolhas bastante importantes. Eu posso escolher que o Eclipse gere para mim o MANIFEST, um arquivo que na verdade contém toda a informação que a Máquina Virtual precisa para saber como rodar este programa (tipo, saber qual classe contém o método MAIN a ser rodado no começo, etc.).

Eu escolhi Generate Manifest File, para o Eclipse gerar o arquivo automaticamente. Então, fui até o BROWSE da Main Class e escolhi qual a classe que possui o método principal MAIN.

Ao final, cliquei em FINISH (nesta mesma tela, mas mais embaixo), e o Eclipse me gerou o JAR executável. Agora é só levar até o PC do meu cliente. Se houverem problemas ao rodar o JAR, veja nos EXTRAS deste livro como solucioná-los.



Para que o seu programa funcione no computador do seu cliente, você vai ter que instalar nele a JRE, que pode ser baixada da Internet gratuitamente sem nenhum problema.

Agora, se o computador do cliente já tinha uma JRE, ou se está se confundindo na hora de abrir seus JARs, o procedimento para consertar isso é muito simples. Isso normalmente ocorre no Windows. Basta desassociar os JARs de qualquer programa, que o Windows mesmo achará a maneira certa de abrir o programa.

Para fazer isso, abra Meu Computador, vá ao menu Ferramentas, abra Opções da pasta, vá até a aba Tipos de Arquivo e EXCLUA o formato JAR. Pronto! Seu Windows perdeu as configurações que o estavam confundindo e agora obedecerá à JRE bem bonitinho, abrindo seu JAR e rodando o programa.

Extras

Ao acabar o corpo principal desta apostila, percebi que algumas funções realmente interessantes ficaram de fora. Por isso, resolvi fazer um apêndice com "extras" interessantes para os desenvolvedores novos em linguagem Java. Espero que alguma coisa disso seja útil aos leitores.

Como chamar um outro programa EXE

Em Java, podemos pedir que o computador execute um outro programa, um EXE, permitindo então que nossas aplicações tenham interação com outras já existentes e até com softwares nativos do sistema operacional que estamos usando.

Fazer isso é muito simples.

```
public class principal {
	public static void main(String[]args) {
		try		{
		 Runtime rt = Runtime.getRuntime() ;
		 Process p = rt.exec("hanoi.EXE") ;
		 p.destroy() ;
		}
		catch(Exception e){
			System.out.println("Deu pau");
		}
	}
}
```

Meu código acima consiste simplesmente de um TRY dentro do qual:

a) Eu crio um objeto Runtime, carregando nele a especificação de qual é o ambiente no qual meu programa está rodando.

b) Eu ordeno que, neste ambiente, seja chamada a execução do programa (e daí, eu coloco o nome do programa, que no meu caso é o joguinho Torre de Hanói).

Há uma maneira mais complexa, e completa, de fazer isso na qual nós usamos o InputStream para receber, do programa, especificações de erros que possam eventualmente travar seu funcionamento. Neste caso, o programa em Java é capaz de saber se houve alguym problema com a aplicação que ele chamou. O código então fica um pouco maior.

```
import java.io.*;
public class principal {
	public static void main(String[]args) {
		try		{
		 Runtime rt = Runtime.getRuntime() ;
		 Process p = rt.exec("hanoi.EXE") ;
		 InputStream in = p.getInputStream() ;
		 OutputStream out = p.getOutputStream ();
		 InputStream err = p.getErrorStream() ;
		 p.destroy() ;
		}
		catch(Exception e){
			System.out.println("Deu pau");
		}
	}
}
```

Usando impressoras matriciais diretamente

Imprimir coisas com o JasperReports é realmente uma maravilha. O sujeito chega lá, desenha um relatório cheio de enfeites gráficos, e fica esperando a folha sair parecendo um caderno de menininha de 12 anos, tudo floreado e coloridinho.

O problema é que no mundo real dos negócios, existem as impressoras matriciais e as impressoras fiscais. E estas impressoras são utilizadas em larga escala para imprimir dados em folhas padronizadas que já vêm prontas – é o caso dos contracheques, das notas fiscais, das Darfs, dos Docs bancários, e de inúmeros outros documentos que existem e são utilizados. Imprimir corretamente dados sobre uma folha que já vem com os campos impressos pode parecer um desafio absurdo. Mas não é.

Antes de existirem CrystalReports, IReports e outros softwares de criação de formulários padronizados, as pessoas imprimiam as coisas enviando o texto diretamente para a impressora. As impressoras tinham um tamanho-padrão e formato-padrão de letras, e o texto saía, corrido, naquele tamanho. As impressoras matriciais e fiscais têm um tamanho que é próprio delas para imprimir letras. Este tamanho pode ser alterado através de comandos de compressão, negrito e outros (dependendo da impressora), mas de uma forma geral todas imprimem do mesmo tamanho. Os documentos impressos por bancos, governo e empresas seguem os padrões de espaçamento de linha destes tamanhos-padrão das impressoras matriciais.

Então, nós precisamos saber como enviar um documento para a impressora sem usar o IReport. Fazer isso vai nos dar um pouco de trabalho, mas é por isso que eu trouxe hoje uma classe para a gente usar. Tenho ela aqui no meu cinto de utilidades.

```
import javax.print.*;
import java.io.ByteArrayInputStream;
import java.io.InputStream;
public class matricial {
   private static PrintService impressora = null;
   static void imprime(String texto) {
```

Eu estou criando o método como Static para que possamos chamá-lo de uma foma rápida a partir de qualquer ponto do sistema com o comando matricial.imprime(“Texto a imprimir”);

```
try {
   DocFlavor df = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
   PrintService[ ] ps = PrintServiceLookup.lookupPrintServices(df, null);
   for (PrintService p: ps) {
      System.out.println("Impressora encontrada: " + p.getName());
      if (p.getName().contains("Text") || p.getName().contains("Generic"))  {
         System.out.println("Impressora Selecionada: " + p.getName());
         impressora = p;
         break;
      }
   }
} catch (Exception e) {
   System.out.println("Já deu pau na hora de catar a impressora");
}
```

Este primeiro Try simplesmente abre os PrintServices (as impressoras disponíveis no Sistema Operacional) e tenta catar uma impressora. Ele roda um FOR que vai girar enquanto for possível carregar um PrintService para dentro da variável PS. Não se preocupe, este comando captura as impressoras configuradas tanto no Windows como no Linux.

Dentro do FOR, ele faz uma verificação (veja o IF ali dentro). Se o nome da impressora contiver (método CONTAINS) a palavra “Text” ou “Generic”, ele vai carregar a variável IMPRESSORA com o nome da impressora. Esta verificação existe porque as impressoras matriciais, hoje em dia, são normalmente cadastradas com nomes como “Generic 9 Pins Text Only”, e coisas parecidas. Mas se a sua impressora matricial tiver outro nome, tipo “Epson LX310”, então é bom alterar o IF desta

verificação.

Bem. Se ele não encontrar impressora alguma cadastrada no sistema operacional, então ele vai simplesmente pular o FOR.

O Catch só será disparado caso não haja um serviço de cadastro das impressoras disponíveis. Daí, o erro é no seu sistema operacional e só Deus pode te ajudar, porque eu não faço a mínima idéia do que possa estar acontecendo. Alguém com certeza apagou arquivos de sistema. O que fazer num caso desses? Não sei. Isso nunca aconteceu comigo ou com alguém que eu conheça. Imagino que seja preciso recuperar o sistema ou reinstalá-lo.

```
if (impressora == null) {
    System.out.println("Que merda! Nenhuma impressora matricial encontrada!");
} else {
```

Aqui, eu verifiquei se a "impressora", aquela variável que só seria carregada caso houvesse uma impressora matricial instalada, é NULL ou não. Se ela for NULL, então o IF dentro do FOR no procedimento anterior não encontrou nada que s possa usar.

No entanto, se ele encontrou, então vamos para o ELSE, onde a impressão realmente acontece.

```
                try {
                  DocPrintJob dpj = impressora.createPrintJob();
                  InputStream stream = new ByteArrayInputStream(texto.getBytes());
                  DocFlavor flavor = DocFlavor.INPUT_STREAM.AUTOSENSE;
                  Doc doc = new SimpleDoc(stream, flavor, null);
                  dpj.print(doc, null);
                } catch (PrintException e) {
                  e.printStackTrace();
                }
            }
        }
    }
```

Aqui está o núcleo deste código. Primeiro, eu uso a classe DocPrintJob, que vai armazenar um documento para a impressora. Este documento permanece vazio até que eu mande ele imprimir alguma coisa.

Depois, eu crio um InputStream para guardar um Array de bytes, formado pelo texto a ser impresso.

Agora, eu tenho um trabalho de impressão vazio e uma Array de bytes loucos para serem impressos. Só falta juntar tudo. E é daí que eu configuro o "flavor" do meu serviço de impressão, programando-o para auto-gerenciar os bytes que forem entrando nele.

Depois, eu instancio a classe Doc, criando um documento de impressão completo. Ele tem como parâmetros os dados que formam o documento, o Flavor (modo de trabalhar os dados) e os atributos especiais, que eu coloquei como NULL, porque não precisamos usar eles (na verdade, eu jamais vi um caso em que se usasse para alguma coisa, mas...).

No comando seguinte eu peço para o meu DocPrintJob fazer uma impressão, finalmente, recebendo os dados já tratados do Stream. Novamente, eu tenho como colocar atributos – que eu não vou colocar – no segundo argumento do comando.

Agora, o documento deverá sair na impressora matricial numa boa. Se você montou sua String Texto com todos os dados com espaçamentos corretos, então seu contracheque deverá sair bem impresso. Espero que o valor também dê prazer aos olhos no momento da leitura.

As quebras de linha deverão ser feitas na própria String Texto. Ao invés de usarmos o bom e velho caracter "\n", vamos usar outros. E há como enviar comandos de Negrito, Compressed e outras coisas diretamente para a impressora. O único problema é que os comandos que precisamos passar variam de impressora para impressora. Vale sempre consultar os manuais delas. A maioria das impressoras fiscais quebra a linha quando mandamos a expressão "\r\n".

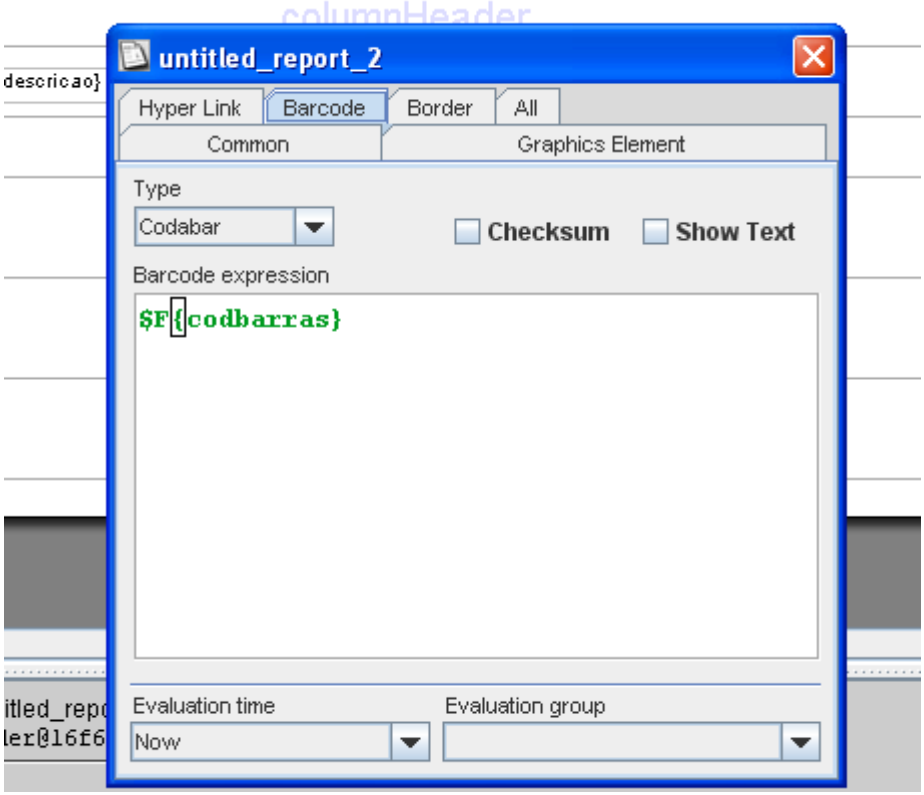
Gerando um relatório com Códigos de Barra

Eu vou começar criando uma tabela de produtos bem simples no meu banco de dados. Nesta tabela, eu tenho apenas descrições de produtos com seus códigos de barra respectivos. Assim:

cod	descricao	codbarras
1	Sabão	123456
2	Arroz	654321
3	Batata	612543

Agora, eu vou criar o Report consultando nesta tabela para procurar dados. Para isso, sigo o procedimento que já vimos – criando um Report perfeitamente normal.

Só que lá pelas tantas, eu vou inserir um componente Código de Barras ao lado do nome dos produtos na banda Detail. E vou clicar duas vezes neste componente criado, fazendo então uma edição de suas Properties.



Ao atribuir dentro da Barcode Expression o valor de um campo, eu faço com que o Ireport desenhe o código de barras para aquele número. Assim, fica fácil criar boletos, controles de patrimônio e outras utilidades que necessitem de código de barras.

O resultado final é o seguinte:

Sabão

Arroz

Batata

Como ler um código de barras

Bom, primeiro, você vai ter que acoplar um leitor de código de barras ao seu PC. Isso não é complicado, pois esses aparelhos funcionam ligados à USB do computador.

O fato é que quando temos uma JtextField com o cursor dentro dela, e passamos um código de barras pelo leitor, a reação do programa será de receber os números lidos no código, e de acionar o ActionListener do TextField, se houver algum.

O ActionListeneter pode ser adicionado a um JTextField e será chamado quando o usuário apertar ENTER, normalmente após digitar alguma coisa. Isso também ocorre quando passamos um código de barras através do leitor.

Então, basta construir uma tela com um JtextField equipado com ActionListener, que pegará o valor passado pelo leitor de código de barras e fará a comparação com um número que ele tenha armazenado como sende de algum produto.

Veja o código de exemplo, no qual o código 123456 está estampado no Sabão em Pó de um supermercado:

```
import java.awt.event.*;
import java.sql.*;
import javax.swing.*;

public class principal {
	static Connection con;
	public static void main(String[]args) {
		JFrame tela = new JFrame();
		tela.setBounds(10,10,600,400);

		final JTextField texto = new JTextField();
		texto.setBounds(10,10,150,30);

		texto.addActionListener(new ActionListener() {


			public void actionPerformed(ActionEvent e) {
				if (texto.getText().equals("123456")) {
					JOptionPane.showMessageDialog(null, "Sabão em pó");
				}

			}

		});

		tela.add(texto);
		tela.setLayout(null);
		tela.setVisible(true);
		tela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
}
```

Créditos

Cast (in order of appearance)

Friedrich Nietzsche	as Himself
Fábio	as Himself
Bruno	as Himself
Luis	as Himself
Sérgio	as Himself and Angry JOptionPane Face
Diego	as Himself
Daniel	as Sir Not Apeearing in this Film
Michel	as Himself
Mauricio	as Himself
Maicon	as Himself
Fernando Pozzebon	as Himself
Clauber Junio	as Himself
Dejair	as Himself
Platão	as Sócrates (quote by Renée Descartes)
Edith Piaf	as Herself
Jon Bauman	as Sha-na-na's "Bowzer"
The Sha na na Boys	as Sha-na-na Singers
Kenneth Branagh	as Hamlet, prince of Denmark
PacMan	as Himself
John Wayne	as Ringo
Lon Chaney	as London After Midnight Monster

Java By
Sun Microsystems

Soundtrack
"Blue Moon", by The Marcels
"La Vie en Rose", by Edith Piaf

Directed by
Fábio Burch Salvador

Produced by
Senac São Leopoldo

Printed by
ACCópias

Chief of Project Engeneering
Sergio Costi

Special Cleaning Specialist
Tia Rose

Chief Manager of Development Compilator Programator Memory-Gastator
Daniel Beckel

Sample Codes Inspirator
Angus McGyver

Food and Coke Supplies Manager
Saddam da Feitoria

```
        System.out.println("Fim do livro");

        }
}
```