

Formação Java

Linguagem de Programação Java
Módulo III

Versão 1.0 Setembro 2005

Copyright Treinamento IBTA

Índice

INICIANDO	1
OBJETIVOS.....	1
HABILIDADES ADQUIRIDAS	1
1. REVISANDO A TECNOLOGIA E A PLATAFORMA JAVA	2
TIPOS DE PROGRAMAS JAVA	2
LISTA DAS PALAVRAS RESERVADAS DA LINGUAGEM JAVA:	3
A PORTABILIDADE DO JAVA.....	4
POR TRÁS DA JAVA RUNTIME ENVIROMNENT.....	5
DEFININDO NOMES (IDENTIFICADORES) PARA MÉTODOS, CLASSES E VARIÁVEIS:	6
ANATOMIA DE UMA CLASSE JAVA.	6
ENTENDENDO UMA APLICAÇÃO JAVA STAND-ALONE.	9
2. LINGUAGEM DE PROGRAMAÇÃO JAVA	10
OS TIPOS PRIMITIVOS DA LINGUAGEM JAVA.	11
USANDO OS TIPOS PRIMITIVOS.	12
USANDO VARIÁVEIS PRIMITIVAS COMO ATRIBUTOS DE OBJETOS.	13
“CASTING” E “PROMOTION” DE TIPOS PRIMITIVOS.	14
O TIPO REFERÊNCIA NA LINGUAGEM JAVA.	15
A NOTAÇÃO DO . (PONTO) E O USO DE UMA REFERÊNCIA.	15
ENDEREÇAMENTO DE MEMÓRIA DA JVM PARA UM OBJETO.	16
COMENTANDO O CÓDIGO.	17
VISUALIZANDO GRAFICAMENTE A MEMÓRIA	18
OPERADORES E EXPRESSÕES.....	19
OPERADORES ARITMÉTICOS.....	19
OPERADORES LÓGICOS	20
OPERADORES CONDICIONAIS	22
PRECEDÊNCIA DE OPERADORES	24
ESTRUTURAS DE REPETIÇÃO.	25
USANDO O FOR.	25
USANDO O WHILE.	27
USANDO O DO .. WHILE.	28
LAÇOS ANINHADOS.	29
ESTRUTURAS DE SELEÇÃO.	30
USANDO O IF.....	32
USANDO O SWITCH.	34
3. CRIANDO CLASSES EM JAVA.....	36
ENCAPSULAMENTO.....	38
DEFININDO VARIÁVEIS	41
ENTENDENDO A DIFERENÇA ENTRE “DE CLASSE” E “DE INSTÂNCIA”	42
ATRIBUTOS E MÉTODOS DE INSTÂNCIA (OBJETO).....	43
DEFININDO MÉTODOS	45
DEFININDO CONSTRUTORES	49
DEFININDO CONSTANTES	53
USANDO A REFERÊNCIA <i>THIS</i>	55
ATRIBUTOS E MÉTODOS DE CLASSE (STATIC).....	58
4. ARRAYS.....	60
JAVA ARRAYS.....	60
DEFININDO ARRAYS.....	62
CÓPIA DE ARRAYS.	66
DISCUSSÃO.	67
EXERCÍCIOS.	68
5. DESENHO AVANÇADO DE CLASSES	69

HERANÇA	69
GENERALIZAÇÃO.....	69
ESPECIALIZAÇÃO.....	69
REPRESENTAÇÃO NA UML.....	70
CODIFICANDO O USO DA HERANÇA EM JAVA.	71
OBJETOS A PARTIR DE SUBCLASSES.	71
DIAGRAMA UML DE CLASSES:	71
IMPLEMENTAÇÃO DAS CLASSES:.....	72
FORMAS DE HERANÇA	74
POLIMORFISMO.....	75
REESCRITA DE MÉTODO (OVERRIDING).....	77
SOBRECARGA DE MÉTODOS E CONSTRUTORES (OVERLOADING).....	80
REFINANDO A HIERARQUIA DE CLASSES.	81
DIAGRAMA UML DE CLASSES (REFINADO) DO MODELO DO BANCO.	83
CLASSES ABSTRATAS E MÉTODOS ABSTRATOS.	84
INTERFACES E MÉTODOS ABSTRATOS.	88
DISCUSSÃO.....	94
EXERCÍCIOS.....	95
6. MANIPULANDO ERROS E EXCEÇÕES	96
A HIERARQUIA DE EXCEÇÕES	96
EXCEÇÕES MAIS CONHECIDAS	97
CAPTURANDO E TRATANDO EXCEÇÕES.....	98
ESTRUTURA TRY-CATCH-FINALLY.....	98
USANDO O BLOCO TRY-CATCH.	99
CRIANDO EXCEÇÕES PARA NOSSAS APLICAÇÕES.	101
SEM O MECANISMO TRY-CATCH.....	101
DISCUSSÃO.....	106
EXERCÍCIOS.....	107
7. DESENVOLVIMENTO DE APLICAÇÕES EM JAVA	108
PARÂMETROS EM LINHA DE COMANDO.....	109
MANIPULANDO TEXTOS E CADEIAS DE CARACTERES.	110
ENTRADA E SAÍDA DE DADOS.	114
JAVA STREAMERS.....	115
HIERARQUIA DE CLASSES DO <i>INPUTSTREAM</i> :	115
HIERARQUIA DE CLASSES DO <i>OUTPUTSTREAM</i> :	116
JAVA READERS E WRITERS.....	117
CATEGORIAS E USO.	118
ARQUIVOS DE ACESSO RANDÔMICO.....	121
TRABALHANDO COM PROPRIEDADES.....	123
TRABALHANDO COM ARQUIVOS DE PROPRIEDADES.....	124
SERIALIZAÇÃO DE OBJETOS.	125
TRABALHANDO COM COLEÇÕES DE OBJETOS – COLLECTIONS APIS.	128
HIERARQUIA DO FRAMEWORK DAS COLLECTIONS:.....	129
ALTERANDO A CLASSE PESSOA, REESCREVENDO O EQUALS E O HASHCODE:.....	130
MANIPULANDO OBJETOS DO TIPO PESSOA DENTRO DE UM ARRAYLIST:.....	131
MANIPULANDO OBJETOS DO TIPO PESSOA DENTRO DE UM HASHMAP:.....	132
MANIPULANDO OBJETOS DO TIPO PESSOA DENTRO DE UM HASHSET:	133
WRAPPER CLASSES	134
DISCUSSÃO.....	135
EXERCÍCIOS.....	136
8. CONSTRUINDO GUIs (INTERFACES COM GRÁFICAS COM O USUÁRIO)	137
ENTENDENDO A HIERARQUIA DE CLASSES:	141
MODELO DE DESENVOLVIMENTO DE INTERFACES GRÁFICAS	142
MODELO DE DESENVOLVIMENTO DO SWING PARA GUIs	142
GERENCIADORES DE LAYOUT E POSICIONAMENTO.	144

ENTENDENDO O JAVA . AWT . FLOWLAYOUT	145
ENTENDENDO O JAVA . AWT . GRIDLAYOUT	146
ENTENDENDO O JAVA . AWT . BORDERLAYOUT	147
ENTENDENDO O JAVA . AWT . GRIDBAGLAYOUT	148
ENTENDENDO OS LAYOUT COMPOSTOS.....	150
MANIPULANDO ASPECTOS VISUAIS	152
TIPOS DE COMPONENTES VISUAIS	155
CRIANDO MENUS PARA O USUÁRIO.	156
TRABALHANDO COM CAIXA DE INFORMAÇÃO E DIÁLOGO.....	157
TRABALHANDO COM BOTÕES.	158
TRABALHANDO COM CAMPOS.....	159
EXEMPLO DE LEITURA DE SENHA:	161
EXIBINDO LISTAS, COMBOS E TABELAS.....	162
TRABALHANDO COM JLIST, LISTMODEL E LISTCELLRENDERER.....	163
TRABALHANDO COM JCOMBOBOX, COMBOBOXMODEL E COMBOBOXEDITOR.....	164
TRABALHANDO COM JTABLE, TABLEMODEL E TABLECELLRENDERER.	166
DISCUSSÃO	168
EXERCÍCIOS	169
9. TRATAMENTO DE EVENTOS PARA GUIS	170
MODELO DE DELEGAÇÃO PARA TRATAMENTO DE EVENTOS.	170
IMPLEMENTANDO O TRATAMENTO DE EVENTOS.	171
TRATANDO EVENTOS COMUNS.....	172
TRATANDO EVENTOS DE JANELAS.	173
TRATANDO EVENTOS DE BOTÕES E MENUS.	174
EXEMPLO DE EVENTOS EM MENUS:	175
TRATANDO EVENTOS DE TEXTOS.	178
TRATANDO EVENTOS DE COMBOS.....	181
DISCUSSÃO	184
EXERCÍCIOS	185
10. PROGRAMAÇÃO PARA REDES (NETWORK PROGRAMMING)	186
TRABALHANDO COM STREAM SOCKETS	187
CODIFICANDO UM SERVIDOR E UM CLIENTE SÍMPLES	188
DISCUSSÃO	189
EXERCÍCIOS	190
11. PROGRAMAÇÃO MULTI-TAREFA (MULTITHREADING PROGRAMMING)	191
JAVA THREAD API: CLASSES E INTERFACES.....	192
CONSTRUINDO THREADS	193
MANIPULANDO THREADS	195
MÁQUINA DE ESTADOS	196
CONTROLANDO A CONCORRÊNCIA.....	197
CONTROLANDO A EXECUÇÃO (ESTADOS DE ESPERA).	198
CONSTRUINDO E EXECUTANDO TIMERS	199
DISCUSSÃO	200
EXERCÍCIOS	201

Iniciando

Objetivos

O curso de Linguagem de Programação Java ensina aos alunos a sintaxe da linguagem de programação Java, a programação orientada a objeto com a linguagem de programação Java e a criação de interfaces gráficas do usuário (GUIs), exceções, entrada/saída (I/O) de arquivos, threads e operação em rede.

Programadores que estiverem familiarizados com conceitos de orientação a objeto poderão aprender como desenvolver uma aplicação Java.

O curso utiliza o Java 2 Software Development Kit (Java 2 SDK).

Habilidades Adquiridas

Após a conclusão deste curso, os alunos deverão estar aptos a:

- Criar aplicações sofisticadas em Java que tirem proveito das características de orientação a objeto da linguagem Java, tais como a herança e o polimorfismo.
- Utilizar bibliotecas entrada e saída de dados (I/O) para a leitura e gravação de dados e arquivos de texto.
- Criar e utilizar os componentes de GUI da tecnologia Java SWING: painéis, botões, títulos, campos de texto e áreas de texto.
- Criar programas multithread (multi-usuário).
- Criar um client simples de Transmission Control Protocol/Internet Protocol (TCP/IP) que se comunique por intermédio de sockets de rede (programação para redes).

1. Revisando a Tecnologia e a Plataforma Java

Sabemos que a Orientação a Objetos é uma nova maneira, mais natural e simples, de pensar, elaborar, documentar e construir aplicações e sistemas de informação.

Baseada nos conceitos de **abstração, objetos, classes, atributos (propriedades) e métodos (funções)**, temos um novo paradigma para a construção de aplicações e a tecnologia **Java** permite implementar todos esses conceitos, pois a linguagem é totalmente aderente as técnicas da orientação a objetos.

A partir deste capítulo veremos como desenvolver e implementar soluções completas usando a linguagem **Java** e suas tecnologias.

Tipos de programas Java

Todos os tipos de programas Java descritos abaixo, são escritos usando uma única linguagem, suas diferenças estão nas API's que eles usam.

Stand-Alone: aplicação baseada na J2SE, que tem total acesso aos recursos do sistema, memória, disco, rede, dispositivos, etc; Um servidor pode executar uma aplicação Stand-Alone, por exemplo, um WebServer. Uma estação de trabalho pode executar uma aplicação de Automação Comercial.

Java Applets™: pequenas aplicações, que não tem acesso aos recursos de hardware e depende de um navegador que suporte a J2SE para serem executados, geralmente usados para jogos, animações, teclados virtuais, etc.

Java Servlets™: programas escritos e preparados para serem executados dentro de servidores web baseados em J2EE, geralmente usados para gerar conteúdo dinâmico de web sites;

Java Midlets™: pequenas aplicações, extremamente seguras, e construídas para serem executadas dentro da J2ME, geralmente, celulares, Palm Tops, controladores eletrônicos, computadores de bordo, smart cards, tecnologia embarcada em veículos, etc;

JavaBeans™: pequenos programas, que seguem um padrão bem rígido de codificação, e que tem o propósito de serem reaproveitados em qualquer tipo de programa Java, sendo reaproveitados, e podendo ser chamados a partir de: stand-alone, applets, servlets e midlets.

Lista das palavras reservadas da linguagem Java:

O intuito não é explicar cada uma delas separadamente, e sim somente listá-las para podermos entender sua simplicidade.

Nenhuma das palavras abaixo podem ser usadas como nome de classes, objetos, atributos ou métodos.

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto*	private	transient
const*	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

*** reservadas, entretanto não são usadas, pois não representam instruções da JRE;**

A portabilidade do Java.

Como comentado anteriormente, o Java usa as duas formas computacionais de execução de software, compilação e interpretação.

O uso das duas formas faz com que o processo de desenvolvimento do software se torne mais acelerado, pois linguagens somente compiladas, necessitam de instruções adicionais da plataforma operacional para serem executados, e as linguagens somente interpretadas geralmente são muito lentas. Por isso o Java num primeiro momento é compilado (transformar código fonte) em instruções da Java Virtual Machine (bytecode).

Como o processo de execução é dividido em dois, temos a possibilidade de que o interpretador seja escrito para várias plataformas operacionais.

Ou seja, o mesmo bytecode Java não importando onde foi compilado, pode ser executado em qualquer JRE. Os fabricantes de hardware e sistemas operacionais, em conjunto com a JavaSoft, desenvolveram várias JREs para vários ambientes operacionais. Por isso, aplicações 100% Java são 100% portáveis.

Compilando...

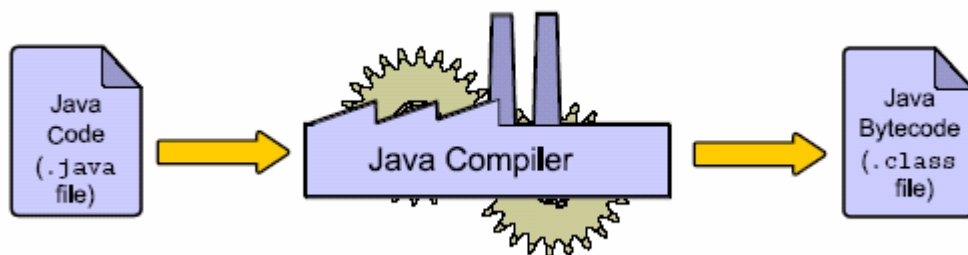


Figura 2

Executando...

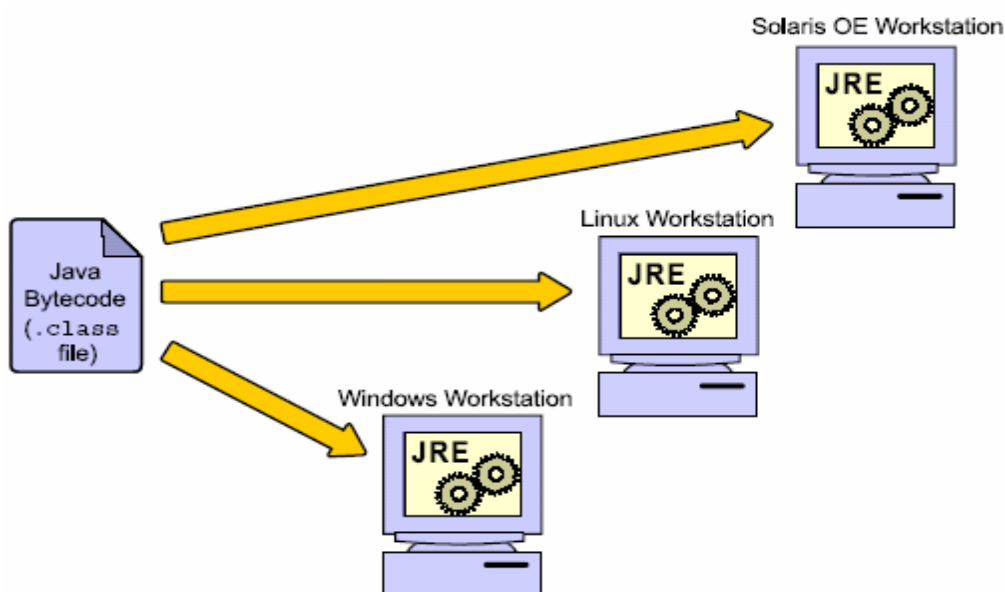


Figura 3

Por trás da Java Runtime Environment.

A JRE é um pacote de software, que é executado como um aplicativo do sistema operacional e que interpreta a execução de programas Java de forma escalável, performática e segura. Por isso é chamada de máquina virtual.

Veremos agora os principais componentes internos da JRE.

Java Runtime Environment

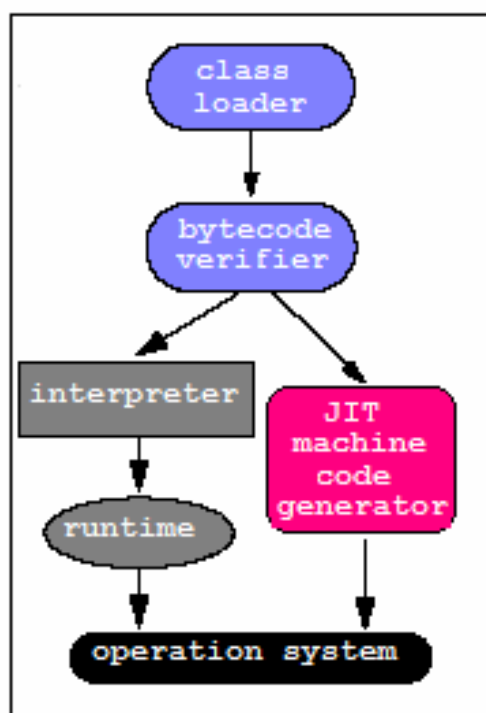


Figura 4

Classloader:

Responsável por carregar as classes Java (bytecode) que farão parte da execução dos aplicativos de um sistema.

Bytecode verifier:

Verifica se as classes carregadas são mesmo de fato classes Java, e validam se o código é seguro e não viola as especificações.

Interpreter e Runtime:

Interpretam uma porção do bytecode, transformando-o em chamadas do sistema operacional.

JIT (Just-In-Time Compiler):

Adquirido da Symatec em 1997, ele é responsável por compilar uma parte do bytecode Java diretamente em chamadas do sistema operacional.

Integrado a JRE, existe ainda um componente chamado **Garbage Collector**, que é responsável por coletar da memória objetos e dados que estão sendo mais usados.

O **Garbage Collector**, tem seu próprio algoritmo e ciclo de execução, por isso não temos controle sobre ele. Assim, em Java, o programador não precisa se preocupar em alocar e desalocar memória quando lida com objetos. No C ou C++, é de responsabilidade do programador o gerenciamento de memória.

Em Java, quando não precisamos mais de um objeto, basta anular sua referência elegendo-o a ser coletado, e o Garbage Collector fará o resto, garantindo a saúde da memória em que a aplicação está sendo executada.

Para cada conjunto, sistema operacional e hardware, existe uma JRE otimizada, capaz de garantir a execução de sistemas baseados em Java.

Definindo nomes (identificadores) para métodos, classes e variáveis:

Existem um documento formal, chamado Java Coding Conventions™, presente no site da JavaSoft que contém todas as regras de codificação e nomenclatura.

Anatomia de uma classe Java.

Todo programa Java será classificado em um dos tipos descritos anteriormente neste capítulo. O objetivo deste livro é mostrar a construção de aplicações stand-alone e JavaBeans™.

Todas as aplicações Java, independente do seu tipo, obrigatoriamente deve estar definida dentro de uma classe Java. E os objetos do nosso sistema, são representados em classes.

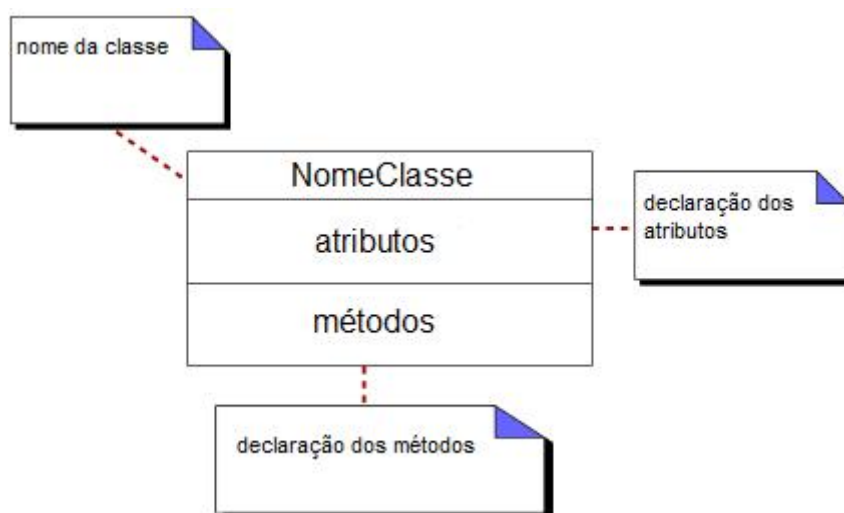


Figura 5

Nome da classe

Um identificador para a classe, que permite referenciá-la posteriormente.

Definindo uma classe

Uma classe define-se como abaixo e ela é composta por membros, atributos e/ou métodos.

```
[modificador] class [identificador] {  
  
    // conjunto de atributos  
  
    // conjunto de métodos  
}
```

Examinando o código abaixo podemos identificar os conceitos citados acima:

```
public class PrimeiroProgramaJava {  
  
    public static void main(String args[]) {  
        System.out.println("Bem vindo a Universidade Java!");  
    }  
}
```

Identificamos o seguinte:

public, static – modificadores;

PrimeiroProgramaJava, main – identificadores;

public static void main(String args[]) { } – método;

String args[] – argumento do método;

String – tipo de dados;

void – tipo de retorno do método;

class – tipo para classes;

System.out.println – chamada de um método de uma outra classe;

Definindo Atributos

O conjunto de propriedades da classe. Para cada propriedade, especifica-se:

```
[modificadores] tipo [nome] [= valor_inicial];
```

nome: um identificador para o atributo.

tipo: o tipo do atributo (inteiro, real, caráter, outra classe, etc.)

modificadores: opcionalmente, pode-se especificar o quão acessível é um atributo de um objeto a partir de outros objetos. Valores possíveis são:

- (privativo), nenhuma visibilidade externa;
- + (público), visibilidade externa total;
- # (protegido), visibilidade externa limitada.

Definindo Métodos

O conjunto de funcionalidades da classe. Para cada método, especifica-se sua **assinatura**, composta por:

```
[modificadores] tipo_de_retorno [nome] ([argumentos]) {  
    // corpo do método;  
}
```

nome: um identificador para o método.

tipo_de_retorno: quando o método tem um retorno, o tipo desse valor.

lista de argumentos: quando o método recebe parâmetros para sua execução, o tipo e um identificador para cada parâmetro.

modificadores: como para atributos, define o quão visível é um método a partir de objetos de outras classes.

Entendendo uma aplicação Java Stand-Alone.

Para que uma aplicação Java possa ser considerada como Stand-Alone e para que ela possa ser executada diretamente pelo interpretador, como vimos anteriormente, ela deve possuir esse método, da forma como está escrito.

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

É a partir dessa linha de código, que o programa Java começa a ser executado. O interpretador Java carrega a classe e chama o método **main(String args[])**, ele tem o papel de ser o start-point (iniciar execução) de uma aplicação Stand-Alone, e quando a execução chegar ao fim do método **main**, a aplicação é encerrada.

2. Linguagem de Programação Java

Uma linguagem de programação permite que façamos o mapeamento de uma determinada rotina ou processo existente no mundo real ou imaginário, para um programa de computador.

Para que o computador possa executar tal rotina, devemos usar uma linguagem que ele entenda. No caso de Java, a JRE entende bytecode, que é proveniente da compilação de programas java.

Devemos ter a seguinte percepção, um sistema construído usando a linguagem Java, será composto de muitas classes, cada uma com sua responsabilidade dentro do sistema, e deve ser escrito de acordo como a forma que ele será executado.

Um programa de computador deve refletir a execução de rotinas de cálculo, comparação e expressões matemáticas. Para entender quais tipos de expressões, rotinas e comparações que podemos executar, devemos primeiramente entender os tipos de dados que o Java possui para armazenar valores.

Dentro das linguagens de programação orientadas a objeto, a capacidade de se criar um elemento que representa um determinado tipo de dado, seja um atributo de um objeto ou um parâmetro de um método, devemos fazer uso de **variáveis**.

Na linguagem java, temos duas categorias de tipos de **variáveis**:

Tipos primitivos: capazes de armazenar valores simples, como números inteiros e ponto flutuante, caractere e booleano, e usamo-los **para executar cálculos e comparações**;

Tipos referência: capazes de armazenar as referencias dos objetos que vamos usar, ou seja, todo tipo de dado que não é primitivo, e usamo-los **para navegar entre os objetos de um sistema**;

Os tipos primitivos da linguagem Java.

Os tipos primitivos da linguagem Java foram inseridos para facilitar a codificação dos programas e otimizar o mecanismos de cálculo do interpretador.

Inteiros	Precisão	Range de valores
byte	8 bits	-2^7 até $2^7 - 1$
short	16 bits	-2^{15} até $2^{15} - 1$
char	16 bits	0 até $2^{15} - 1$
int *	32 bits	-2^{32} até $2^{32} - 1$
long	64 bits	-2^{64} até $2^{64} - 1$

boolean	8 bits	true ou false
---------	--------	---------------

Ponto Flutuante	Precisão	Range de valores
float	32 bits	Dependentes do Sistema Operacional
double *	64 bits	

* otimizados para a execução de cálculos dentro da JRE.

Usamos cada um desses tipos, de acordo com a necessidade:

byte: geralmente usado para valores reais muito pequenos, e manipulação de dados em arquivos;

short: também usado para valores pequenos;

char: usado para trabalhar com caracteres;

int: tipo de dados preferido para manipulação de valores inteiros;

long: usado quando o range de valor do int não é o suficiente;

float: usado como número real de precisão simples;

double: tipo numérico preferido para manipulação de valores de precisão dupla;

boolean: tipo booleano pra armazenar valores de verdadeiro (true) ou falso (false);

Usando os tipos primitivos.

Variáveis do tipo primitivo podem ser usadas como atributo de objeto, parâmetro de método, ou variável local de método.

Uma variável na sua definição clássica serve para armazenar um tipo de valor dentro de um algoritmo. E este valor pode variar dentro do programa, por isso ele se chama variável.

Outro uso comum de variáveis, é para identificarmos ao que ela se refere.

```
public class CalculoJurosSimples {  
    public static void main(String args[]) {  
        int prestacoes = 6;  
        double jurosMensal = 0.02;  
        double valorEmprestado = 1000.0;  
        double jurosPeriodo = prestacoes * jurosMensal;  
        double valorTotal = valorEmprestado * ( 1 + jurosPeriodo );  
        double valorJuros = valorTotal - valorEmprestado;  
        double valorParcela = valorTotal / prestacoes;  
  
        System.out.println("Valor total a pagar: "+ valorTotal);  
        System.out.println("Valor dos juros: "+ valorJuros);  
        System.out.println("Valor da parcela: "+ valorParcela);  
    }  
}
```

Usando variáveis primitivas como atributos de objetos.

Toda vez que definimos uma variável, fora de um método, ela se torna uma variável de objeto, ou de instância.

Usamos esse tipo de variável para armazenar valores dentro dos objetos, usando todo o poder da orientação a objetos.

Tomando como exemplo, se fossemos analisar um determinado conjunto de objetos dentro do sistema do restaurante, podemos identificar o objeto Conta. O que uma conta tem como atributos (características)?

Retomando os conceitos de classe, vimos que uma classe serve para criarmos objetos de uma determinada categoria ou tipo.

Poderíamos identificar para a conta atributos básicos como: número e saldo. Assim, poderíamos criar uma classe, que me permitisse criar várias contas que tivessem esses atributos.

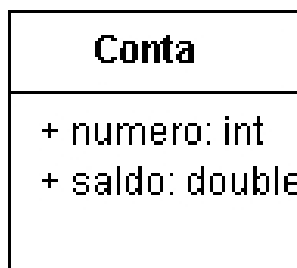


Figura 6

Vemos aqui, que podemos traduzir facilmente o diagrama UML para um classe Java.

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

* o sinal de (+) na UML significa modificador public para atributos ou métodos.

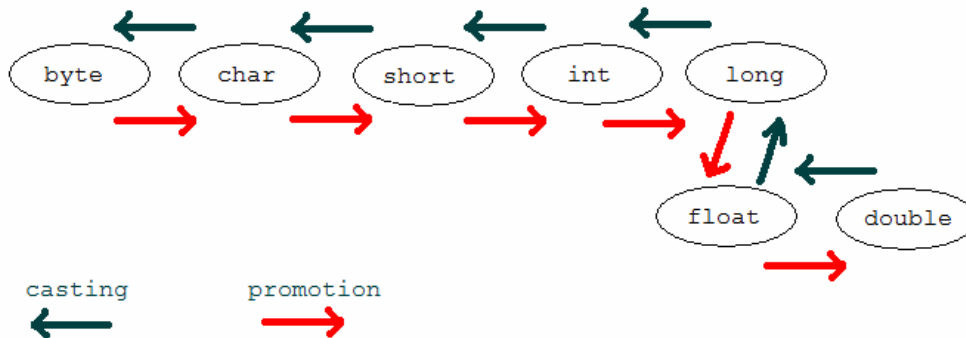
Com essa classe Conta, poderíamos manipular qualquer Conta que pudesse conter um número e um saldo.

Antes de manipularmos os atributos de um objeto, vamos aprender a manipular os objetos.

Na linguagem Java, para se manipular objetos, usamos as variáveis do tipo referência. E para não esquecermos, toda variável que não for do tipo primitivo, é do tipo referência.

“Casting” e “Promotion” de tipos primitivos.

A característica de Casting e Promotion, ou conversão de tipos, esta muito presente dentro da linguagem java, e é será sempre necessário quando estivermos usando tipos diferentes.

**Figura 7**

Vemos pelo desenho que se quisermos diminuir a precisão de uma variável, devemos fazer um **type-cast**, e quando houver o aumento da precisão precisamos fazer o **type-promotion**.

Em java, o compilador não nos obriga a fazer programaticamente o **type-promotion**, pois o **compilador java faz o auto-promotion** dos tipos primitivos.

Auto-promotion:

```
tipo-menor identificador2 = valorLiteral;  
tipo-maior identificador = identificador2;
```

A precisão do tipo menor é aumentada para um tipo maior, sem problemas.

Entretando, o **type-cast** deve ser feito programáticamente, garantido as precisões.

```
tipo-maior identificador = valorLiteral;  
tipo-menor identidicador2 = (tipo-menor) identificador;
```

Exemplos:

```
int x = 200;  
byte b = x; // falha na compilação  
byte c = (byte) x; // type-cast  
float f = x; // auto-promotion  
  
// lembre-se que os tipos preferidos são int e double.  
int z = b + c;  
double d = f + b;
```

O tipo referência na linguagem Java.

Os tipos referência foram inseridos na linguagem para facilitar a construção de código, gerência de memória pelo **Garbage Collector**, e prover a navegação entre os objetos do sistema.

Usamos os tipos referência para podermos manipular facilmente os objetos em memória.

Atualmente, com as capacidades computacionais crescendo, as linguagens orientadas a objetos, mesmo consumindo grandes capacidades de memória, são muito mais atrativas que as linguagens procedurais.

Usamos uma variável referência para ter acesso á uma instância de um objeto em memória. Então podemos criar dentro um programa um ou mais objetos a partir de uma classe.

A notação do . (ponto) e o uso de uma referência.

Para acessarmos o número da conta 1, precisamos usar a sua referência, `conta1`, e nessa referência, usando a notação do ponto (.) temos acesso aos membros do objeto daquela referência:

```
Conta conta1 = new Conta();  
conta1.numero = 10;  
conta1.saldo = 500;
```

Endereçamento de memória da JVM para um objeto.

Os endereços de memória, impressos com @, são resultantes das linhas:

```
System.out.println( conta1 );
System.out.println( conta2 );
```

Estes endereços de memória, dos objetos Java, não nos interessa, ele interessa apenas ao **Garbage Collector**.

Um objeto pode ter como atributo, não somente tipos primitivos, mas também tipos referência.

Vejamos a classe Produto abaixo:

Produto	
+ codigo: String	public class Produto {
+ descricao: String	public String codigo;
+ preco: double	public String descricao;
+ quantidade: int	public double preco;
	public int quantidade;
	}

Figura 8

Essa classe possui atributos que permite manipular vários tipos de produtos. Ou seja, analisando um supermercado, veremos que todos os seus produtos terão esses atributos.

Dentro desta classe vemos quais são os atributos não primitivos:

```
String codigo;
String descricao;
```

Vemos também dois atributos primitivos:

```
double preco;
int quantidade;
```

O tipo String permite armazenar cadeias de caractere, ou seja, palavras, frases, etc.

Podemos usar uma variável do tipo String de duas formas:

```
livro2.descricao = "Dossie Hacker!";
```

ou

```
livro2.descricao = new String("Dossie Hacker!");
```

A primeira forma é a mais intuitiva, e preferida por programadores experientes. A segunda forma, é mais clássica e fácil de entender, não resta dúvida que a variável descrição da classe Produto é do tipo String.

Comentando o código.

É muito comum escrever linhas de comentários dentro do código de um programa, com o propósito de explicar algoritmos e documentar a forma como o programador entendeu o que deve ser feito.

Um código bem comentado permite que a manutenção do programa, aplicação e do sistema, seja mais simples e mais precisa.

Tipos de comentários.

```
// comentário de uma única linha

/*
    Bloco de comentário
*/

/**
    Comentário em formato Java Doc
*/
```

A J2SDK possui um utilitário muito interessante que converte os comentários escritos em Java Doc em páginas HTML com navegação e organização padronizados internacionalmente pela JavaSoft.

Veja na figura abaixo uma imagem exemplo da tela da documentação Java Doc.

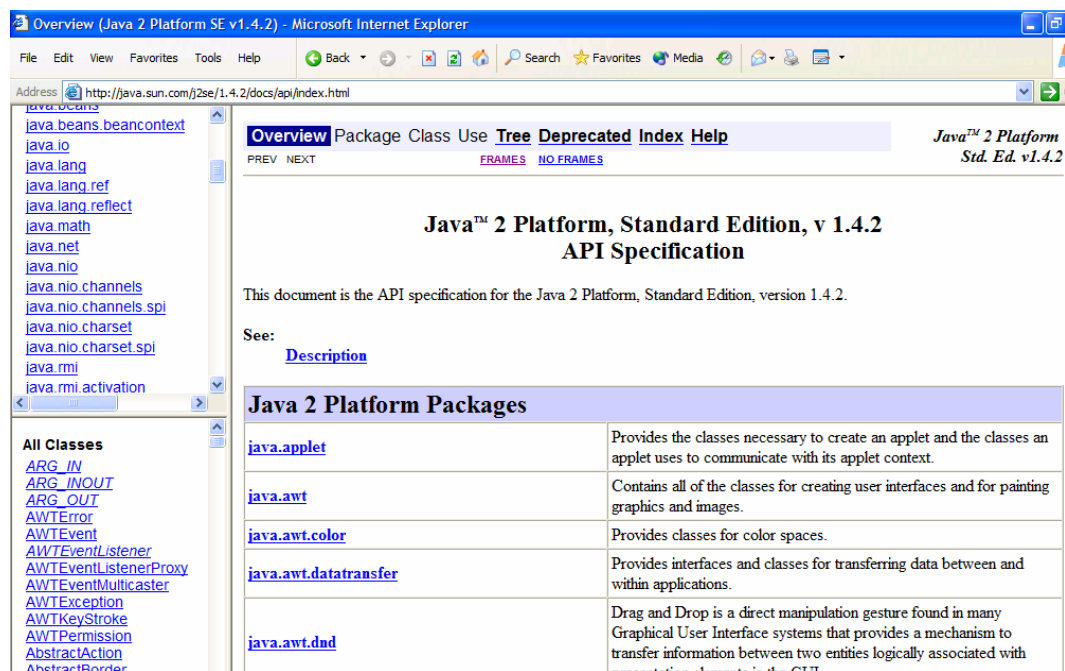


Figura 9

Veja aqui o Java Doc das classes que estão disponíveis dentro da J2SE 1.4.2
<http://java.sun.com/j2se/1.4/docs/api/index.html>

Veja aqui o tutorial de como utilizar o Java Doc:
<http://java.sun.com/j2se/1.4.2/docs/tool docs/windows/javadoc.html>

Visualizando graficamente a memória

Podemos examinar um snapshot de memória e identificar que o `TesteProduto2` inicializou um objeto `livro1` do tipo `Produto`.

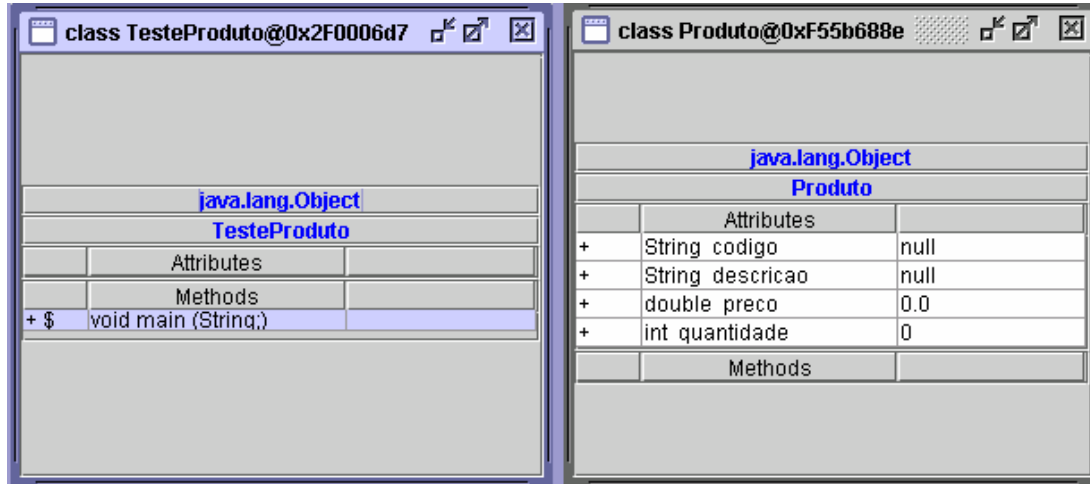


Figura 10

Vemos também que o `TesteProduto` possui duas referências para os objetos `livro1` e `livro2` do tipo `Produto`, bem como `livro1` e `livro2` possuem código e descrição na sua estrutura, que são referências para objetos do tipo `String`;

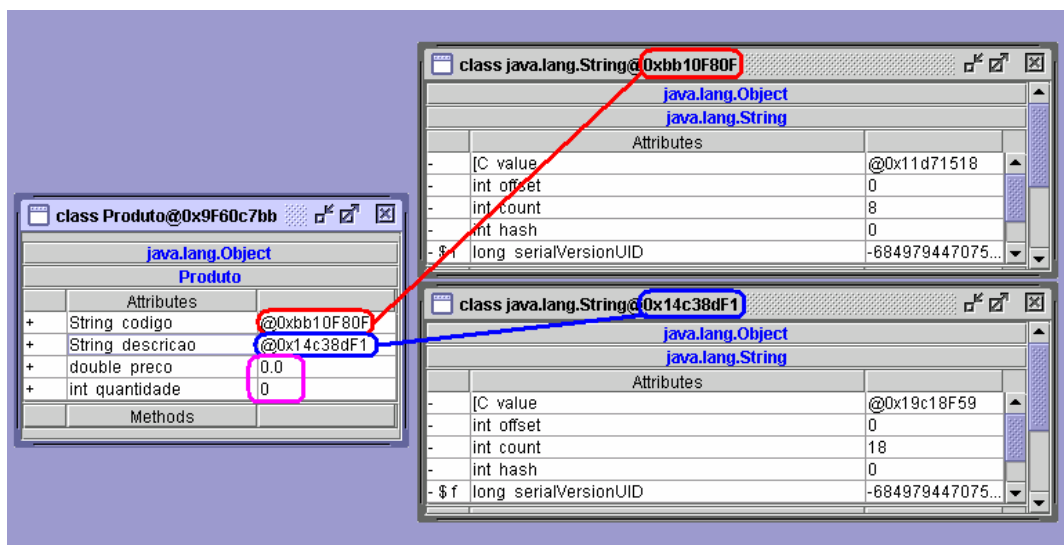


Figura 11

Em Java, quando uma variável não é do tipo primitivo, ela será do tipo referência, que por sua vez, é capaz de permitir a navegação entre os objetos. Vemos que `livro1.codigo` aponta para uma `String` através de uma referência.

Os tipos primitivos são armazenados diretamente em áreas de memória dentro dos objetos que os contêm.

Operadores e expressões.

Vimos que uma linguagem de programação tem a finalidade de permitir a criação de aplicações visando automatizar processos manuais ou mentais, facilitando a vida das pessoas.

O processo de trabalho comum das pessoas é composto de etapas, e essas etapas podem requerer a execução de cálculos simples ou complexos.

A técnica computacional que nos permite executar tais cálculos é chamada de **expressão**.

Uma **expressão** é composta de variáveis e **operadores**. O bom entendimento e uso das expressões e dos operadores permitem com que os algoritmos do software sejam bem escritos e que sejam executados da forma que foram idealizados.

Operadores aritméticos

- soma (+);
- subtração (-);
- multiplicação (*);
- divisão (/);
- resto da divisão (%), apenas para operandos inteiros;
- incremento (++), operador unário definido apenas para operandos inteiros, podendo ocorrer antes da variável (pré-incremento) ou após a variável (pós-incremento); e
- decremento (--), operador unário definido apenas para operandos inteiros, podendo ocorrer antes da variável (pré-decremento) ou após a variável (pós-decremento).

Servem para executar cálculos usando os tipos primitivos inteiros ou de ponto flutuante.

Cada uma das linhas de cálculo exibida no código abaixo são consideradas expressões. Usamos as expressões para realizar contas aritméticas, comparações, etc.

Operadores lógicos

Operações lógicas sobre valores inteiros atuam sobre a representação binária do valor armazenado, operando internamente bit a bit. Operadores desse tipo são:

- complemento (~), operador unário que reverte os valores dos bits na representação interna;
- OR bit-a-bit (|), operador binário que resulta em um bit 1 se pelo menos um dos bits na posição era 1;
- AND bit-a-bit (&), operador binário que resulta em um bit 0 se pelo menos um dos bits na posição era 0;
- XOR bit-a-bit (^), operador binário que resulta em um bit 1 se os bits na posição eram diferentes;
- deslocamento à esquerda (<<), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à esquerda;
- deslocamento à direita com extensão de sinal (>>), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à direita. Os bits inseridos à esquerda terão o mesmo valor do bit mais significativo da representação interna;
- deslocamento à direita com extensão 0 (>>>), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à direita. Os bits inseridos à esquerda terão o valor 0.

Os operadores binários são muito usados em algoritmos de criptografia, e muito pouco usados no dia-a-dia, por serem uma forma muito interessante de se trabalhar com algoritmos bit-a-bit.

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TesteBinarios {  
    public static void main(String a[]) {  
        int i = 2 & 10;  
        System.out.println("Somando bit a bit 2 com 10 = " + i);  
        int j = 2 | 10;  
        System.out.println("Multilicando bit a bit 2 com 10 = "+ j);  
        int k = 2 ^ 10;  
        System.out.println("O Complemento de 10 para 2 = " + k);  
        int x = 1000;  
        int div = x >> 2;  
        System.out.println("Rotacionando a direita "+ x +" >> 2 = "+ div );  
        int mult = x << 2;  
        System.out.println("Rotacionando a esquerda "+ x +" << 2 = "+ mult  
    );  
    }  
}
```

Operadores condicionais

Os operadores condicionais permitem realizar testes baseados nas comparações entre valores numéricos e/ou variáveis, permitindo uma tomada de decisão dentro dos aplicativos. Eles são:

- E lógico (`&&`), retorna `true` se as duas expressões booleanas forem verdadeiras;
- OU lógico (`||`), retorna `true` se uma das duas expressões forem verdadeiras;
- maior (`>`), retorna `true` se o primeiro valor for exclusivamente maior que o segundo;
- maior ou igual (`>=`), retorna `true` se o primeiro valor for maior que ou igual ao segundo;
- menor (`<`), retorna `true` se o primeiro valor for exclusivamente menor que o segundo;
- menor ou igual (`<=`), retorna `true` se o primeiro valor for menor que ou igual ao segundo;
- igual (`==`), retorna `true` se o primeiro valor for igual ao segundo;
- diferente (`!=`), retorna `true` se o primeiro valor não for igual ao segundo.
- operador condicional ternário. Na expressão `b ? s1 : s2`, `b` é booleano (variável ou expressão). O resultado da expressão será o resultado da expressão (ou variável) `s1` se `b` for verdadeiro, ou o resultado da expressão (ou variável) `s2` se `b` for falso.

Os operadores condicionais, na maioria dos casos são usado para testes booleanos, de verdadeiro ou falso, dentro das estruturas de desvio condicional.

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TesteCondicionais {  
  
    public static void main(String a[]) {  
  
        int i = 2;  
        int j = 2;  
        int k = 10;  
        int z = 0;  
        boolean b = true;  
        boolean c = true;  
        boolean f = false;  
        System.out.println( i+" igual a "+ j + " : "+ (i == j) );  
        System.out.println( i+" maior que "+ k + " : "+ (i > k) );  
        System.out.println( i+" menor que "+ k + " : "+ (i < k) );  
        System.out.println( i+" menor que "+ z + " : "+ (i < z) );  
        System.out.println( i+" maior ou igual a "+j+" : "+(i >= j) );  
        System.out.println( i+" menor ou igual a "+j+" : "+(i <= j) );  
        System.out.println( (i<3) ? i +" menor que 3" : i +" maior que 3"  
);  
  
        System.out.println( b+" && "+c+ " = "+ ( b && c ) );  
        System.out.println( b+" && "+f+ " = "+ ( b && f ) );  
        System.out.println( b+" || "+f+ " = "+ ( b || f ) );  
    }  
}
```

Precedência de operadores

Agora que conhecemos os operadores e expressões, precisamos entender a precedência deles, para evitarmos que nossos programas façam os cálculos errados.

Regras de precedência:

1. Operações entre parênteses;
2. Incremento e Decremento;
3. Multiplicação e divisão, executados da esquerda para a direita;
4. Adição e subtração, executados da esquerda para a direita;

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TestePrecedencia {  
  
    public static void main(String args[]) {  
        int x = 2 + 3 / 5 * 7 - 6;  
        System.out.println(" 2 + 3 / 5 * 7 - 6 = " + x );  
  
        int j = (2 + 3) / (5*7) - 7;  
        System.out.println("(2 + 3) / (5*7) - 6 = " + j );  
  
        int z = 2 + 3 / 5 * ( 7 - 6 );  
        System.out.println("2 + 3 / 5 * ( 7 - 6 ) = " + z );  
    }  
}
```

Vemos que a ordem de como tratamos os valores e variáveis dentro de uma expressão, podem interferir no resultado final.

Por isso é extremamente interessante que se use os parênteses dentro das expressões de forma a garantir a ordem de como os cálculos serão feitos.

Estruturas de repetição.

Como todas as aplicações e sistemas são baseados em execução de tarefas, e em alguns casos, com repetição de certas partes delas, todas as linguagens de programação fornecem estruturas para repetição, comumente chamamos de laço.

Em Java existem três estruturas que são comumente usadas para executar blocos de programas em ciclos de execução.

Imaginem um programa que tivesse de executar repetidamente uma determinada rotina, como faríamos?

Escrevemos infinitas linhas de código ou usando as estruturas de laço??

Um exemplo simples. Imagine um bloco de código que imprimisse os números da sena:

```
public class Sena {  
  
    public static void main(String args[]) {  
        System.out.println("1,2,3,4,5,6,7,8,9,10,16...,60");  
    }  
}
```

Ou ficaria melhor usando uma estrutura de repetição? Entendo essa situação, veremos o uso do **for**, **while**, **do .. while**.

Usando o for.

O for, foi projeto para testar uma condição inicial antes de iniciar o laço, executar uma declaração, executar o bloco de código dentro do laço, e no fim de uma repetição executar uma expressão.

Para usá-lo devemos entender sua sintaxe:

```
for ( declaracao; condição booleana; expressao ) {  
  
    // antes de iniciar o laço, e executada a  
    // declaração  
  
    // bloco de código que sera repetido  
    // ate que a condição booleana seja false  
  
    // antes da execução iniciar o proximo laço  
    // sera a executada a expressao  
  
}
```

Então, vamos converter o programa da Sena para fazermos o uso do for. Sabemos que os números da Sena variam de 1 a 60, e nessa ordem devem ser impressos.

```
public class Sena {  
  
    public static void main(String args[]) {  
        System.out.println("Numeros da Sena");  
        for ( int i=1; i <= 60; i++ ) {  
            System.out.print(i+",");  
        }  
        System.out.println();  
    }  
}
```

A linha que contém o for, diz ao interpretador: “Execute o que estiver compreendido no meu bloco de código ate que o *i* seja menor ou igual a 60, e antes de começar o próximo laço, incremente 1 em *i*”.

Usando o while.

Existe dentro do Java, uma estrutura de repetição mais simples, o **while**. O **while** funciona muito parecido com o **for**, entretanto ele tem menos parâmetros para a sua execução.

For e **while**, testam se podem começar a executar ou se podem executar aquele laço, antes de começarem.

Para usá-lo devemos entender sua sintaxe:

```
while (condição booleana) {  
  
    // bloco de código que será repetido  
    // até que a condição booleana seja false  
}
```

Com uma estrutura mais simples, o controle das repetições fica mais complexo.

Usando o **while** para escrever o programa da Sena:

```
public class SenaWhile {  
  
    public static void main(String args[]) {  
        System.out.println("Numeros da Sena");  
        int i = 1;  
        while( i <= 60 ) {  
            System.out.print(i+", ");  
            i++;  
        }  
        System.out.println();  
    }  
}
```

Devemos nos atentar, pois dentro do **while**, incrementamos um contador (**i++**), o qual é verificado a cada laço. Quando esse valor de **i**, chegar a 61, a execução é desviada para o fim do laço.

Usando o do .. while.

Assim como **while**, o **do .. while** fará um teste para verificar se ele continuar repetindo a execução as linhas de código internas ao laço.

A diferença, é que o **do .. while** testa no fim da execução e não antes dela. Ou seja, pelo menos uma execução acontece, diferente do **while** que testa no início.

Para usá-lo devemos entender sua sintaxe:

```
do {  
  
    // bloco de código que sera repetido  
    // ate que a condição booleana seja false  
} while ( condicao );
```

Usando o do .. while para escrever o programa da Sena:

```
public class SenaDoWhile {  
  
    public static void main(String args[]) {  
        System.out.println("Numeros da Sena");  
        int i = 1;  
        do {  
            System.out.print(i+", ");  
            i++;  
        } while( i <= 60 );  
        System.out.println();  
    }  
}
```

Vemos que neste caso, o while e do .. while, tiveram o mesmo comportamento.

Se quiséssemos imprimir a tabuada de um determinado número? Qual o algoritmo a seguir?

Devemos imprimir o número, e um valor quando multiplicado de 1, e sucessivamente até chegar ao 10.

Então podemos escrever o seguinte código:

```
public class Tabuada {  
  
    public static void main(String args[]) {  
        int numero = 5;  
        System.out.println("Tabuada do :"+ numero);  
        for ( int i=1; i<=10; i++) {  
            int resultado = numero * i;  
            System.out.println( numero + " * " + i + " = " + resultado );  
        }  
    }  
}
```

Laços Aninhados.

Podemos ainda combinar o uso das estruturas de repetição sem problemas, e com o intuito de executar rotinas complexas.

Veja o código abaixo, e veja como podemos aninhar as estruturas:

```
public class Aninhados {  
  
    public static void main(String args[]) {  
        int i = 1;  
        while ( i <= 9 ) {  
            for ( int j = 1; j <= 10; j++ ) {  
                System.out.print(i+"*"+j+",");  
            }  
            System.out.println();  
            i++;  
        }  
    }  
}
```

No próximo módulo exploraremos mais os laços de repetição e suas combinações.

Estruturas de seleção.

Como a finalidade dos sistemas de computação é mapear processos físicos ou mentais, facilitando a vida dos usuários, em todos eles exigem a necessidade de tomada de decisões dentro da execução de uma rotina. Essas tomadas de decisões são implementadas usando as estruturas de seleção.

Uma estrutura de seleção apresenta um desvio no processamento das informações, permitindo a criação de fluxos alternativos ao fluxo principal daquela rotina ou função.

Aliados as estruturas de repetição, poderemos criar algoritmos completos de execução de funções ou rotinas, para resolver problemas computacionais simples ou complexos.

As estruturas de seleção são usadas de forma comparativa, ou seja, se uma determinada variável possui um determinado valor, o programa segue um fluxo, caso contrário o programa segue outro fluxo.

Podemos representar isso graficamente.

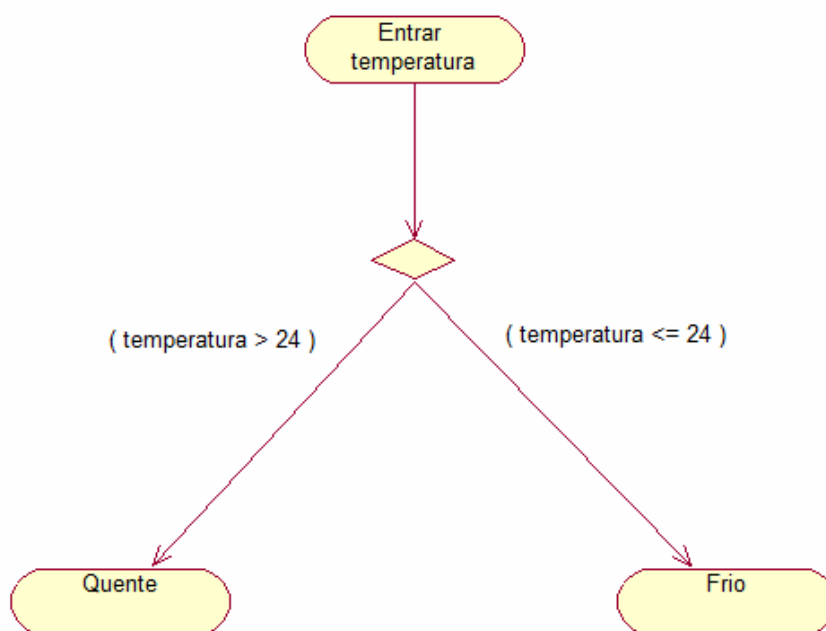


Figura 12

Podemos também aninhar as estruturas de decisão, para implementar algoritmos mais complexos.

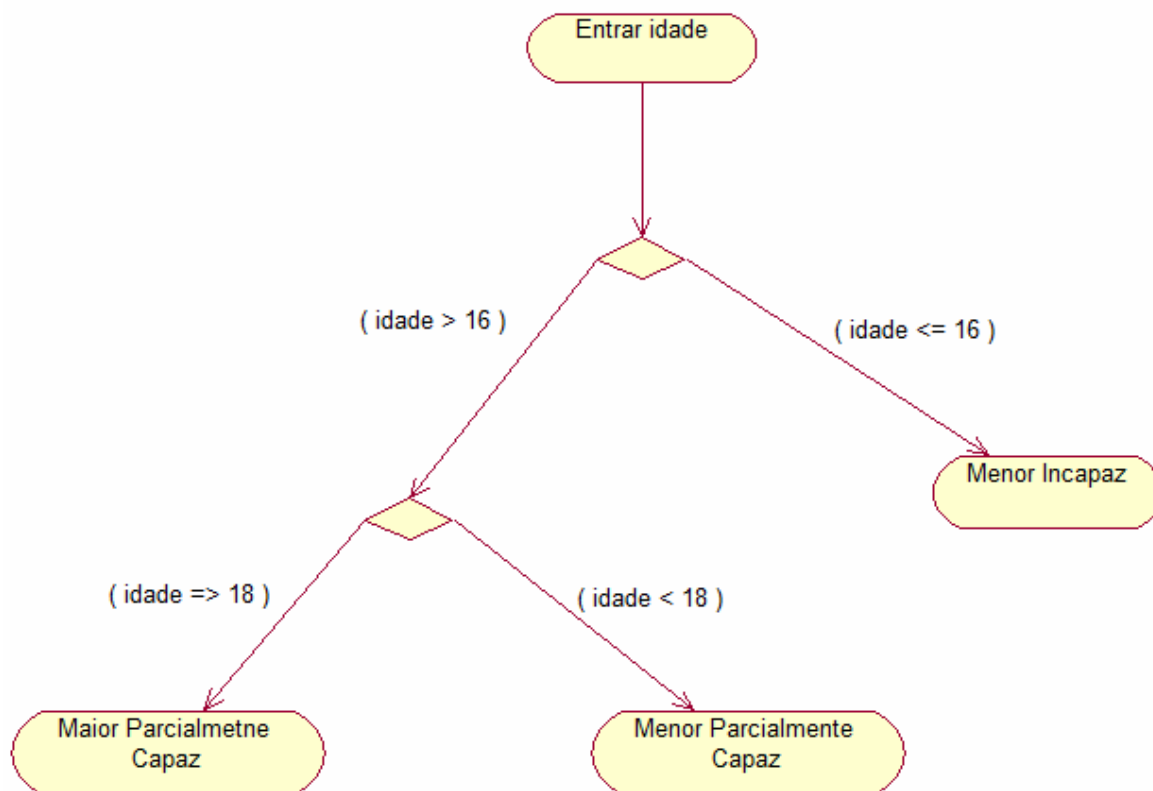


Figura 13

Em Java, existem duas estruturas de seleção:

```
if;  
switch;
```

Usando o if.

O if (tradução SE), é usado para comparar expressões, variáveis e condições booleanas.

Sintaxe:

```
if ( expressão booleana ) {  
    // bloco de código que será executado  
    // se a expressão booleana tiver valor true.  
}
```

Ainda dentro da estrutura do if podemos controlar os desvios de fluxo usando o else.
O uso do else é opcional.

```
if ( expressão booleana ) {  
    // bloco de código que será executado  
    // se a expressão booleana tiver valor true.  
} else {  
    // bloco de código que será executado  
    // se a expressão booleana tiver valor false.  
}
```

No caso de comparar múltiplas condições, podemos usar o else if.

```
if ( expressão booleana ) {  
    // bloco de código que será executado  
    // se a expressão booleana tiver valor true.  
} else if ( expressão booleana2 ) {  
    // bloco de código que será executado  
    // se a expressão booleana2 tiver valor true.  
}  
else {  
    // bloco de código que será executado  
    // se a expressão booleana e a expressao booleana2  
    //tiverem valor false.  
}
```

Dentro da expressão booleana, podemos usar os operadores lógicos OR (||) e AND (&&) para testar condições compostas facilitando a construção da estrutura `if`.

Estes operadores são chamados tecnicamente de “**short-circuit operators**”.

Exemplos:

```
if ( idade >= 0 && idade < 16 ) {  
    // executara este bloco de código se  
    // a idade for maior ou igual a zero E  
    // a idade for menor que 16  
} else if ( idade >= 17 && idade < 18 ) {  
    // executara este bloco de código se  
    // a idade for maior ou igual a 17 E  
    // a idade for menor que 18  
} else {  
    // executara este bloco de código se  
    // a idade for maior ou igual a 18  
}
```

...

```
if ( x == 0 || y == 0 ) {  
    // executara este bloco de código  
    // se x OU y forem igual a zero  
} else {  
    // executara este bloco de código  
    // se x E y forem diferentes de zero  
}
```

Usando o switch.

O `switch` funciona com uma estrutura para tratamento de casos específicos. Ele funciona com um único teste, ou seja, a estrutura testa uma condição numérica (não booleana), somente uma vez. E partir daquela condição ele executará todos os casos.

Sintaxe do `switch`:

```
switch ( int ) {  
  
    case (inteiro):  
        // código  
    case (inteiro):  
        // código  
    case (inteiro):  
        // código  
    default:  
        // código  
}
```

O `default` é opcional.

Nota: obrigatoriamente, o parâmetro que será comparado **somente pode ser um byte ou short ou char ou int**.

O caso de comparação (`case (inteiro):`) obrigatoriamente tem que ser um literal inteiro ou uma constante.

Exemplo de uso do `switch`.

```
switch ( x ) {  
  
    case 1:  
        // código A  
  
    case 10:  
        // código B  
  
    case 100:  
        // código C  
  
    default:  
        // código D  
}
```

Simulando:

quando `x = 1`, linhas executadas: A,B,C e D;
quando `x = 10`, linhas executadas: B,C e D;
quando `x = 100`, linhas executadas: C e D;
qualquer valor de `x`, diferente de 1, 10 e 100, D.

Podemos controlar o fluxo de execução do **switch**, ou de qualquer bloco de seleção ou repetição usando o **break**; Ele interrompe a execução saindo do bloco.

```
switch ( x ) {  
    case 1:  
        // código A  
        break;  
  
    case 10:  
        // código B  
  
    case 100:  
        // código C  
        break;  
  
    default:  
        // código D  
}
```

Simulando:

quando x = 1, linhas executadas: A;
quando x = 10, linhas executadas: B e C;
quando x = 100, linhas executadas: C;
qualquer valor de x, diferente de 1, 10 e 100, D.

3. Criando Classes em Java

Até agora vimos as principais estruturas da linguagem java. A partir deste ponto, estaremos definindo e implementando os conceitos iniciais da Programação Orientada a Objetos.

E quando falamos de implementação de sistemas orientados a objetos, devemos dominar o desenho e criação de classes.

Vamos revisar o conceito de classe:

“Uma classe é um modelo para a criação de objetos de um determinado tipo, a partir de um sistema real ou imaginário. Ela pode conter características de dados (atributos) e de funcionalidades (métodos). A partir de uma classe, podemos criar várias instância de objetos com características em comum.”

“Para se definir uma classe, analisamos uma categoria de objetos dentro de um contexto real ou imaginário, abstraindo suas características de dados e de funcionalidade. A partir desta classe, podemos instanciar um objeto dando-lhe uma referência (nome). Podemos usar a referência de um objeto para acessar seus atributos e utilizar seus métodos.”

Vemos então que uma classe é composta de atributos (dados) e métodos (funções).



conta

Figura 14

atributos: saldo,
limite, juros,
vencimento, etc.

funcionalidades:
debito, credito,
consultar saldo, etc.

A partir desta classe Conta, podemos criar vários tipos de conta que possua saldo, limite, juros e vencimento, e ainda podemos executar operações de débito, crédito e consultar saldo.

```
Conta cartaoCredito = new Conta();
```

```
Conta banco = new Conta();
```

A cada vez que o operador **new** é executado, cria-se uma instância em memória capaz de armazenar dados e executar funções. A essa instância em memória, damos o nome de objeto. Neste caso, temos duas instâncias em memória, do tipo Conta. Uma chama-se **cartaoCredito**, e a outra, **banco**.

Através de uma dessas referências, podemos acessar seus atributos e métodos.

Para facilitar o entendimento, a documentação e o desenho de um sistema orientado a objetos devemos lembrar da UML (Unified Modeling Language), já citada anteriormente.

deste capítulo abordaremos somente o Diagrama de Classes da UML, que nos auxiliará no processo de aprendizado sobre classes e objetos.

Como vimos anteriormente, a programação orientada a objetos é baseada em três fundamentos:

Encapsulamento;
Herança;
Polimorfismo;

Cada um deles contém um conjunto de conceitos, os quais devemos dominar.

Todos esse conceitos nos trarão muitos benefícios durante o desenvolvimento de software, e garantem que a evolução, adicionado melhorias e novas funcionalidades, sejam mais produtivas e rápidas.

Neste módulo, veremos os conceitos de encapsulamento, herança e polimorfismo.

Encapsulamento

Definição clássica oriunda da Análise e Desenho Orientado a Objetos: **“Disponibilizar uma interface pública, com granularidade controlada, para manipular os estados e executar operações de um objeto”.**

Resumindo, devemos esconder os atributos dos objetos provendo uma interface pública de métodos, impedindo o acesso a eles diretamente por outros objetos.

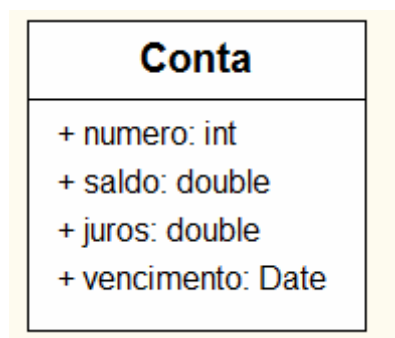
A linguagem Java, disponibiliza um conjunto de modificadores de acesso para os membros de classe, que nos permite definir vários níveis de encapsulamento.

Modificadores de acesso para os membros de classe:

Modificador	Mesma Classe	Mesmo Pacote	Subclasses	Universo
private (-)	*			
<none>	*	*		
protected (#)	*	*	*	
public (+)	*	*	*	*

* acesso permitido.

Analisando os modificadores de acesso, para encapsular um atributo de um objeto, devemos demarcá-lo como **private**, fazendo com que somente as referências de objetos provenientes desta classe tenham acesso a ele.



```
public class Conta {

    public int numero;
    public double saldo;
    public double juros;
    public Date vencimento;

}
```

Figura 15

(+) significa que o membro tem modificador público

Com os atributos públicos, podemos escrever código assim:

```
Conta contaBanco = new Conta();
contaBanco.numero = 1;
contaBanco.saldo = 100;
contaBanco.juros = 0.089;
```

Para fazer operações:

```
contaBanco.saldo += 300; // depósito de 300;
contaBanco.saldo -= 200; // saque de 200;
contaBanco.saldo = 1000; // alteracao direta de saldo
```

Analisando o mundo real de um banco, não podemos fazer tais operações sem haver um mínimo de validação de saldo, de valores a serem depositados, etc.

Para garantir que tais regras serão executadas, devemos usar o encapsulamento, assim alteraremos o modelo da classe Conta, para ter atributos privados e métodos públicos para manipulá-los.

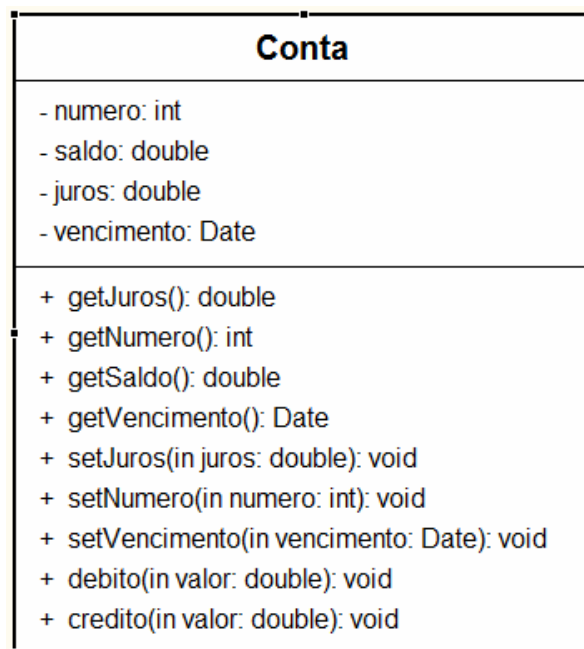


Figura 16

Legenda:
+ público
- privado

Notas:

1 - para cada atributo que pode ser consultado criamos um método `getNomeAtributo ()` (tecnicamente chamando de **accessor method**);

2 - para cada atributo que pode ser alterado, criamos um método `setNomeAtributo()` (tecnicamente chamando de **mutator method**);

3 – para os métodos que executam operações ou funções dos objetos, damos um nome que corresponde aquela função ou operação, facilitando seu entendimento. (tecnicamente chamando de **worker method**);

4- vemos que todos os atributos estão marcados como `private` e todos os métodos como `public`, esta é a forma de fornecer o encapsulamento em Java.

Examinemos o código abaixo para entendermos como os modificadores serão usados para determinar os atributos privados e métodos públicos de uma classe Java que nos permitirá construir objetos encapsulados.

Neste instante, acredite que o código está correto e funcional, o seu total entendimento será possível até o fim deste capítulo.

```
public class Conta {  
  
    private int numero;  
    private double saldo;  
    private double juros;  
    private java.util.Date vencimento;  
  
    //accessor methods  
    public double getJuros() {  
        return juros;  
    }  
  
    public int getNumero() {  
        return numero;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public java.util.Date getVencimento() {  
        return vencimento;  
    }  
  
    //mutator methods  
    public void setJuros(double juros) {  
        this.juros = juros;  
    }  
  
    public void setNumero(int numero) {  
        this.numero = numero;  
    }  
  
    public void setVencimento (java.util.Date vencimento) {  
        this.vencimento = vencimento;  
    }  
  
    //worker methods  
    public void debito(double valor) {  
        this.saldo -= valor;  
    }  
  
    public void credito(double valor) {  
        this.saldo += valor;  
    }  
  
}
```

Passaremos agora a entender a utilização e como escrever esses tipos de métodos, para garantir o encapsulamento dos objetos.

Definindo variáveis

Vimos anteriormente que sempre que desejarmos manipular determinado dado, ele deve ter um tipo, e estar associado a uma variável. Essa variável deveria ter um nome de fácil entendimento, este nome é o seu identificador dentro do código, e usamos este identificador para acessar o conteúdo de dados desta variável.

Na programação orientada a objetos, existem quatro tipos de variáveis, definidas pela forma como o interpretador Java vai armazená-las. Por isso, devemos entender como o interpretador Java tratará cada um dos tipos, para podermos escrever nossos programas de maneira correta e garantir o comportamento desejado deles.

Tipos de variáveis:

- De instância (atributos de objeto);**
- De classe;**
- Locais (variáveis e parâmetros de método);**
- Constantes;**

Para se manusear esses tipos de variáveis, devemos escrever métodos e seguir as práticas de encapsulamento. Por isso, existem três tipos de métodos para manipularmos esses tipos de variáveis.

Tipos de métodos:

- De instância (métodos do objeto);**
- De classe (métodos da classe);**
- Construtores (método especial);**

Relembrando, para se definir uma variável, devemos escolher um tipo de acordo com sua abrangência de tamanho e valor, um modificador quando for necessário, um nome de fácil entendimento e um valor inicial, podendo ser do tipo primitivo ou referência, seguindo a notação:

Formas:

```
[modificador] tipo identificador;
```

```
[modificador] tipo identificador = [valor_inicial];
```

Exemplos:

```
int idade = 18;
float temperatura = 31.2;
double salario = 1456.3;
char letraA = 'A';
byte base = 2;
    long distancia = 2000;

String faculdade = new String("IBTA!");
```

Entendendo a diferença entre “de classe” e “de instância”

Uma variável é considerada de instância quando ela estiver definida no corpo da classe, ou seja, fora dos métodos e não possuir o modificador **static**.

Quando uma variável for definida como parâmetro de método, ou dentro de um método, ela é chamada de local.

Quando uma variável for definida fora dos métodos e possuir o modificador **static**, ela será considerada como atributo de classe.

Quando uma variável for definida fora dos métodos e possuir o modificador **final**, ela será considerada uma constante.

Quando um método não possuir o modificador **static**, ele será considerado como de instância (**non-static**). Seu efeito restringe-se somente ao objeto chamado pela sua referência.

Quando um método possuir o modificador **static**, ele será considerado como de classe(**static**). Seu efeito abrange todos os objetos oriundos daquela classe.

Um método de instância (**non-static**), mesmo não sendo indicado, pode manipular atributos ou métodos de classe(**static**),.

Um método de classe (**static**) não pode manipular diretamente atributos ou métodos instância, para isso ele necessitará de uma referência para o objeto.

Atributos e métodos de classe (**static**), são raramente utilizados na programação orientada a objetos, pois eles não estão ligados aos objetos, e sim a uma classe.

Atributos e métodos de instância (objeto)

Toda vez que quisermos armazenar um dado dentro de um objeto, e poder acessá-lo através da referência deste objeto, definirão estes dados como variável de instância, ou seja, o seu valor pode ser alterado ou consultado usando uma referência de um objeto.

Um atributo de instância é visível á todos os métodos daquela instância. Vimos que a instância de um objeto corresponde a uma área de memória criada para armazenar a estrutura daquele objeto, que pode conter variáveis do tipo primitivo e do tipo referência.

Analisando um trecho do código da classe Conta:

```
public class Conta {  
  
    private int numero;  
    private double saldo;  
    private double juros;  
    private java.util.Date vencimento;  
  
    ..  
}
```

Vemos que ela possui 4 atributos de instância, 3 do tipo primitivo (numero, saldo e juros) e um do tipo referência java.util.Date (vencimento).

Podemos criar a partir da classe Conta, vários objetos do tipo Conta com conteúdo diferente e que pode ser manipulados através de uma referência.

No código abaixo vemos duas instâncias de objetos a partir da classe Conta, nomeadas como `conta1` e `conta2`, executando o programa `TesteConta`, vemos que elas tem valores diferentes, ou seja, são objetos diferentes, que podem ter dados diferentes.

```
public class TesteConta {  
  
    public static void main(String args[]) {  
        Conta conta1 = new Conta();  
        System.out.println("Ref. conta1: " + conta1 );  
  
        Conta conta2 = new Conta();  
        System.out.println("Ref. conta2: " + conta2 );  
    }  
}
```

No código abaixo, vemos que podemos usar os métodos definidos na classe conta (getNumero(), getSaldo(), credito() e débito), e que seu uso refletem somente na referência daquela instância:

```
public class TesteConta2 {  
  
    public static void main(String args[]) {  
        Conta contal = new Conta();  
        System.out.println("Ref. contal: " + contal );  
        contal.setNumero(1);  
        contal.credito(100);  
        contal.debito(20);  
        System.out.println("contal: Numero: " + contal.getNumero() );  
        System.out.println("contal: Saldo: " + contal.getSaldo() );  
        System.out.println();  
  
        Conta conta2 = new Conta();  
        System.out.println("Ref. conta2: " + conta2 );  
        conta2.setNumero(2);  
        conta2.credito(200);  
        conta2.debito(40);  
        System.out.println("conta2: Numero: " + conta2.getNumero() );  
        System.out.println("conta2: Saldo: " + conta2.getSaldo() );  
    }  
}
```

A programação orientada a objetos, consiste em abstrair os comportamentos de objetos comuns, categorizando-os em classes.

Ao definirmos uma classe, vamos determinar seus atributos e métodos usando o encapsulamento. A partir daí, podemos usar essa classe em qualquer tipo de programa java. No caso acima, usamos a classe Conta, para criarmos dois objetos dentro um programa stand-alone.

Para completar os mecanismos de encapsulamento, veremos como definir e escrever métodos para nossas classes.

Definindo Métodos

Usamos os métodos para executar operações, funções, ações, etc. dentro das aplicações. Devemos entender a sua utilidade, pois eles são as ferramentas mais importantes durante o desenvolvimento de aplicações, pois eles representam a parte funcional dos sistemas.

Para se escrever métodos eficientemente, devemos entender primeiro os tipos de métodos que podemos escrever numa classe:

Categorias de métodos baseados na programação orientada a objetos:

accessor : (de acesso), permite ler o valor de um atributo;
mutator: (de alteração), permite alterar o valor de um atributo;
worker: (de trabalho), permite executar uma determinada tarefa;
constructor: (de inicialização), permite criar um objeto e inicializá-lo;

Analisando a classe Conta listada acima, vemos que ela possui três tipos de métodos:

accessors: (todo os métodos que começam com get);
mutators: (todos que começam com set) e;
workers: (representam operações que podem ser feitas com uma Conta).

Usando este conceito então, se tivermos uma classe Pessoa, que deva ter atributos do tipo caracter, como nome e cpf, poderemos defini-la inicialmente assim:

```
public class Pessoa {  
  
    private String nome;  
    private String cpf;  
  
}
```

Como devemos seguir a prática do encapsulamento, devemos então criar ao menos, um conjunto de métodos (set e get) para cada um dos atributos:

Para definirmos um método, ele deve seguir a seguinte sintaxe:

```
[modificadores] tipo_retorno identificador( [parametros] ) {  
    // corpo do método  
}
```

Criando os métodos `get` para os atributos de instância da classe `Pessoa`:

```
public String getNome() {  
    // nome é de instância, do tipo String  
    return nome;  
}  
  
public String getCPF() {  
    // cpf é de instância, do tipo String  
    return cpf;  
}
```

Criando os métodos `set` para os atributos de instância da classe `Pessoa`:

```
public void setNome(String n) {  
    // n é uma variavel local do tipo String, nome é de instância  
    nome = n;  
}  
  
public void setCPF(String c) {  
    // c é uma variavel local do tipo String, cpf é de instância  
    cpf = c;  
}
```

O objetivo do método `set` é alterar o valor do atributo de instância, e do método `get`, retornar o valor do atributo de instância.

Devemos lembrar que somente podemos associar variáveis do tipo referência quando forem do mesmo tipo, e para as variáveis do tipo primitivo, devemos obedecer as regras de **casting** vistas anteriormente.

Todo método que possuir um tipo de retorno diferente de `void` devemos obrigatoriamente especificar um tipo primitivo ou um tipo referência como tipo de retorno, e no corpo do método retornar um valor ou variável do tipo do retorno usando a instrução **`return`**.

Podemos também criar um método de trabalho (worker), para a validação do CPF:

```
public boolean validaCPF() {  
    // variavel local  
    boolean validade = false;  
  
    // este algoritmo somente verifica se o cpf não e nulo.  
    if ( cpf != null ) {  
        validade = true;  
    } else {  
        validade = false;  
    }  
  
    return validade;  
}
```

A classe Pessoa fica assim então:

```
public class Pessoa {  
  
    private String nome;  
    private String cpf;  
  
    public String getNome() {  
        // nome é de instância, do tipo String  
        return nome;  
    }  
  
    public String getCPF() {  
        // cpf é de instância, do tipo String  
        return cpf;  
    }  
  
    public void setNome(String n) {  
        // n é uma variavel local do tipo String, nome é de instância  
        nome = n;  
    }  
  
    public void setCPF(String c) {  
        // c é uma variavel local do tipo String, cpf é de instância  
        cpf = c;  
    }  
  
    public boolean validaCPF() {  
        // variavel local  
        boolean validade = false;  
  
        // este algoritmo somente verifica se o cpf não é nulo.  
        if ( cpf != null ) {  
            validade = true;  
        } else {  
            validade = false;  
        }  
        return validade;  
    }  
}
```

Vemos que a variável `validade` criada dentro do método `validaCPF()` é uma variável local, ou seja, ela existe somente dentro daquele bloco de código, dentro do método. Fora do método, esta variável não existe.

Entretanto, a variável `cpf` que foi definida fora dos métodos, é considerada como de instância, e está visível dentro dos métodos `setCPF()`, `getCPF()` e `validaCPF()`.

Uma variável local existe somente durante a execução daquele método. Uma variável de instância existe enquanto o objeto estiver em memória e não estar eleito para ser coletado pelo **Garbage Collector**.

Podemos testar a classe Pessoa usando o programa TestePessoa:

```
public class TestePessoa {  
  
    public static void main(String args[]) {  
  
        Pessoa pessoal = new Pessoa();  
        pessoal.setNome("Jose Antonio");  
        pessoal.setCPF("123456789-00");  
        System.out.println("Nome: " + pessoal.getNome());  
        System.out.println("CPF: " + pessoal.getCPF());  
        System.out.println("CPF Valido: " + pessoal.validaCPF());  
        System.out.println("");  
  
        Pessoa pessoa2 = new Pessoa();  
        pessoa2.setNome("Antonio Jose");  
        System.out.println("Nome: " + pessoa2.getNome());  
        System.out.println("CPF: " + pessoa2.getCPF());  
        System.out.println("CPF Valido: " + pessoa2.validaCPF());  
    }  
}
```

Definindo Construtores

Como exposto anteriormente, podemos definir métodos para a inicialização dos objetos, um construtor é um método especial, por não possui tipo de retorno e levar o nome da Classe que o contém.

Um construtor pode conter parâmetros, os quais usamos para inicializar os nossos objetos de forma mais aderente ao modelo que estamos mapeando.

Para se definir um construtor seguimos a seguinte sintaxe:

```
[modificador] NomeClasse ( [parametros] ) {  
    // corpo do construtor.  
}
```

Toda classe em Java deve possuir ao menos um construtor. Opa! Mas até agora não escrevemos nenhum construtor em nossas classes, e todas elas compilaram e executaram, porque?

O compilador javac, se ele não encontrar nenhum construtor definido no código da classe, ele coloca para nós o construtor padrão:

```
public NomeClasse () {  
}
```

O compilador Java vai inserir exatamente o código acima quando compilar uma classe Java que não possui nenhum construtor definido.

Para a classe Conta, o construtor padrão fica:

```
public Conta () {  
}
```

Para a classe Pessoa, o construtor padrão fica:

```
public Pessoa () {  
}
```

É por isso que nos programas de Teste, quando são chamadas as linhas que possuem o operador **new** e um construtor, funcionam. Pois o compilador insere o construtor padrão se não nos lembrarmos dele.

O construtor de uma classe, é executado somente uma vez no ciclo de vida de um objeto.

Podemos escrever para uma classe vários construtores, com parâmetros diferentes para podermos inicializar nossos objetos de forma diferente.

Analizando a classe Conta, podemos inicializá-la com um saldo padrão, com um saldo inicial, com juros iniciais, e data.

```
public class Conta {

    private int numero;
    private double saldo;
    private double juros;
    private java.util.Date vencimento;
    ..

    // construtores
    // padrão
    public Conta() {
        numero = 0;
        // saldo padrao é ZERO.
        saldo = 0;
        juros = 0;
        // data atual do sistema
        vencimento = new java.util.Date();
    }

    // com parametros
    public Conta(int numeroConta, double saldoInicial) {
        numero = numeroConta;
        saldo = saldoInicial;
        juros = 0;
        // data atual do sistema
        vencimento = new java.util.Date();
    }

    public Conta(int numeroConta, double saldoInicial,
        double jurosInicial) {

        numero = numeroConta;
        saldo = saldoInicial;
        juros = jurosInicial;
        // data atual do sistema
        vencimento = new java.util.Date();
    }

    public Conta(int numeroConta, double saldoInicial,
        double jurosInicial, java.util.Date dataVencimento) {

        numero = numeroConta;
        saldo = saldoInicial;
        juros = jurosInicial;
        vencimento = dataVencimento;
    }

    ...
}
```


Analisando a classe Pessoa, poderemos inicializá-la com um nome e com um CPF.

```
public class Pessoa {  
  
    private String nome;  
    private String cpf;  
  
    public Pessoa() {  
    }  
  
    public Pessoa(String nomePessoa) {  
        nome = nomePessoa;  
    }  
  
    public Pessoa(String nomePessoa, String cpfPessoa) {  
        nome = nomePessoa;  
        cpf = cpfPessoa;  
    }  
    ...  
}
```

O código dos construtores de uma classe se originará a partir da análise de como eles devem ser inicializados durante a execução dos aplicativos.

O construtor é usado para passagem de parâmetros de inicialização á uma instância de um objeto no ato de sua construção, quando chamamos usando a instrução `new`.

A função do construtor, além de inicializar a instância de um objeto a partir de uma classe, é retornar o endereço de memória para a referência que a instrução `new` está sendo associada.

```
Pessoa pessoal = new Pessoa("Jose Antonio", "123456789-00");
```

A variável `pessoal`, do tipo referência, recebe o endereço de memória em que o objeto foi alocado. E partir desta referência, temos acesso aos valores passados ao objeto através de seus atributos e métodos de instância.

Executando a TesteConta3:

```
public class TesteConta3 {

    public static void main(String args[]) {
        Conta contal = new Conta(1, 100, 0);
        System.out.println("Ref. contal: " + contal );
        contal.debito(20);
        System.out.println("contal: Numero: " + contal.getNumero() );
        System.out.println("contal: Saldo: " + contal.getSaldo() );
        System.out.println("contal: Juros: " + contal.getJuros() );
        System.out.println("contal: Vencimento: " + contal.getVencimento()
    );

        System.out.println();

        java.util.Date vencimentoConta2 = new java.util.Date(99, 2, 30);
        Conta conta2 = new Conta(2, 200, 0.03, vencimentoConta2 );
        System.out.println("Ref. conta2: " + conta2 );
        conta2.debito(40);
        System.out.println("conta2: Numero: " + conta2.getNumero() );
        System.out.println("conta2: Saldo: " + conta2.getSaldo() );
        System.out.println("conta2: Juros: " + conta2.getJuros() );
        System.out.println("conta3: Vencimento: " + conta2.getVencimento()
    );
    }
}
```

Definindo Constantes

Vimos anteriormente como definir variáveis. Uma constante, é uma variável especial, que uma vez inicializada não aceita ter seu valor alterado.

Em Java para se definir uma constante, usamos o modificador `final`, e no identificador usamos as letras todas maiúsculas seguindo a convenção de codificação.

Geralmente para uma Constante, definimos com modificador de acesso `public`, pois permite que acessemos tal valor a partir de qualquer classe `static`, que permite ser acessado diretamente pelo nome da classe sem haver uma instância do Objeto.

Temos a seguinte sintaxe:

```
[modificador acesso] static final tipo identificador = valor;
```

Exemplo:

```
public static final double PI = 3.141516;
```

As constantes são de grande valia dentro das técnicas de programação, pois permitem que o código fonte fique mais claro e legível quando testamos valores da aplicação com parâmetros de sistema, configuração ou de aplicação.

Exemplo de uso de constante:

```
// Endereco.java
public class Endereco {

    public static final int COMERCIAL = 0;
    public static final int RESIDENCIAL = 1;

    private int tipo;
    private String logradouro;

    public Endereco() {
        tipo = Endereco.COMERCIAL;
    }

    public Endereco(int tipoEnd, String logradouroEnd) {
        tipo = tipoEnd;
        logradouro = logradouroEnd;
    }

    public int getTipo() {
        return tipo;
    }

    public void setLogradouro(String logradouroEnd) {
        logradouro = logradouroEnd;
    }

    public String getLogradouro() {
        String local = "";
        switch( tipo ) {
            case Endereco.COMERCIAL:
                local += "COM: ";
                break;
            default:
                local += "RES: ";
        }
        local += logradouro;
        return local;
    }
}

// TesteEndereco.java
public class TesteEndereco {

    public static void main(String args[]) {
        Endereco end1 = new Endereco();
        end1.setLogradouro("Rua Universidade Java!");
        System.out.println( end1.getTipo() + " - " + end1.getLogradouro() );

        Endereco end2 = new Endereco(Endereco.COMERCIAL, "Rua Digerati");
        System.out.println( end2.getTipo() + " - " + end2.getLogradouro() );
    }
}
```

Usando a referência *this*.

Esta keyword do Java tem um significado e um propósito muito interessante, pois ela é a referência do próprio objeto. Ela representa a localização em memória de onde o objeto foi instanciado.

Usamos a referência `this` para indicar ao interpretador que o membro (atributo ou método) associado a ele faz parte da instância corrente.

Usamos o `this` também para diferenciar os atributos de instância dos parâmetros de métodos, fazendo com que possamos usar o mesmo nome em ambos e ter um código mais legível.

O uso do **this**, é muito comum dentro de métodos set (**mutator**) e de construtores.

O que é mais fácil de entender?

```
public void setLogradouro(String logradouroEnd) {  
    // nomes diferentes para as variáveis  
    logradouro = logradouroEnd;  
}
```

Ou?

```
public void setLogradouro(String logradouro) {  
    // this.logradouro refere-se ao atributo de instancia  
    // chamado logradouro  
  
    // o identificador se refere a parâmetro de método logradouro.  
    this.logradouro = logradouro;  
}
```

Seguindo a conversão de código da JavaSoft, devemos fazer uso do **this**, principalmente nos métodos set. Nos métodos get não há relevância, pois o retorno refere-se a um tipo de instância, e o uso do **this** se faz desnecessário.

```
public int getTipo() {  
    // tipo é um atributo de instância,  
    // num método get não usamos colocar o this.  
    return tipo;  
}
```

Reescrevendo a classe Endereço

```
// Endereco.java
public class Endereco {

    public static final int COMERCIAL = 0;
    public static final int RESIDENCIAL = 1;

    private int tipo;
    private String logradouro;

    public Endereco() {
        this.tipo = Endereco.COMERCIAL;
    }

    public Endereco(int tipo, String logradouro) {
        this.tipo = tipo;
        // chamando o método abaixo desta instância
        this.setLogradouro( logradouro );
    }

    public void setLogradouro(String logradouro) {
        this.logradouro = logradouro;
    }

    public int getTipo() {
        return tipo;
    }

    public String getLogradouro() {
        String local = "";
        switch( this.tipo ) {
            case Endereco.COMERCIAL:
                local += "COM: ";
                break;
            default:
                local += "RES: ";
        }
        local += logradouro;
        return local;
    }
}
```

O uso do **this**. facilita a leitura de código por trazer o significado implícito de atributo de instância. O **this**. somente estará presente dentro de uma instância, ou seja, dentro de um método não-estático.

Podemos usar o **this** também para chamar construtores dentro da própria classe, permitindo um maior aproveitamento de código.

Sintaxe: **this([parametros]);**

Exemplo:

```
public Endereco() {  
    // procura um construtor na classe Endereco que possua  
    // parametros int, String  
    this( Endereco.COMERCIAL, "" );  
}
```

Reescrevendo a classe Endereço

```
// Endereco.java  
public class Endereco {  
    // constantes  
    public static final int COMERCIAL = 0;  
    public static final int RESIDENCIAL = 1;  
    // atributos de instancia  
    private int tipo;  
    private String logradouro;  
    // construtor  
    public Endereco() {  
        // procura um construtor na classe Endereco que possua  
        // parametros int, String. Veja o construtor abaixo.  
        this( Endereco.COMERCIAL, "" );  
    }  
    // construtor  
    public Endereco(int tipo, String logradouro) {  
        this.tipo = tipo;  
        // chamando o método abaixo desta instância  
        this.setLogradouro( logradouro );  
    }  
    // mutator ( set )  
    public void setLogradouro(String logradouro) {  
        this.logradouro = logradouro;  
    }  
    // accessor( get )  
    public int getTipo() {  
        return tipo;  
    }  
    // accessor( get )  
    public String getLogradouro() {  
        String local = "";  
        switch( this.tipo ) {  
            case Endereco.COMERCIAL:  
                local += "COM: ";  
                break;  
            default:  
                local += "RES: ";  
        }  
        local += logradouro;  
        return local;  
    }  
}
```

Atributos e métodos de classe (static)

Vimos até agora conceitos voltados à instância de uma classe, que é uma área de memória, qual chamamos de objeto, que é capaz de armazenar dados e permitir a manipulação deles.

Entretanto, da mesma forma que podemos definir atributos para os objetos, podemos defini-los para as classes.

Definir um atributo de classe significa que ele estará definido fora de métodos e obrigatoriamente deve ter o modificador **static**.

A melhor maneira de inicializar atributos de classe, ou seja, **static**, é usar o bloco de inicialização estático:

```
static {  
    // manipulação de atributos de classe, static.  
}
```

Ou seja, é interessante usá-lo principalmente como variáveis de configuração que devem ser inicializadas quando carregamos a aplicação.

Exemplo de uso:

Se no modelo do nosso restaurante, queremos ter uma taxa básica de serviço, que geralmente é de 10%, poderíamos defini-la como sendo uma variável de uma classe de configuração.

```
public class RestauranteConfig {  
  
    private static double taxaServico;  
  
    static {  
        taxaServico = 10.0;  
    }  
}
```


Seguindo a idéia de encapsulamento, devemos escrever métodos para manipular esses atributos de classe (**static**). Um método que vai manipular esse tipo de atributo pode ser de classe (**static**) ou de instância (**non-static**). Damos preferência para os métodos de classe (**static**)

Exemplo de uso:

```
public class RestauranteConfig {

    private static double taxaServico;

    static {
        taxaServico = 10.0;
    }

    public static double getTaxaServico() {
        //note que não usamos o this,
        // e sim o nome da classe como referência
        // para os atributos de classe.
        return RestauranteConfig.taxaServico;
    }

    public static void setTaxaServico(double taxaServico) {
        //note que não usamos o this,
        // e sim o nome da classe como referência
        // para os atributos de classe.
        RestauranteConfig.taxaServico = taxaServico;
    }
}

public class TesteRestauranteConfig {

    public static void main(String arg[]) {
        System.out.println("Taxa de serviço: "+
        RestauranteConfig.getTaxaServico() );
        RestauranteConfig.setTaxaServico( 11.0 );
        System.out.println("Nova Taxa de serviço: "+
        RestauranteConfig.getTaxaServico() );
    }
}
```

Temos de tomar cuidados especiais com atributo de classe, pois ele é um endereço único, para todos os objetos provenientes daquela classe.

Alterando seu valor, será refletido em todas as instâncias, por isso é usado para configuração ou parâmetros globais de aplicação.

4. Arrays

As técnicas computacionais tem problemas clássicos para manipular conjuntos de dados em memória, essa manipulação geralmente é feita usando-se vetores (array de uma dimensão) ou matrizes (array de múltiplas dimensões).

Geralmente usamos arrays para colecionar objetos, podendo ordená-los, ou pesquisá-los, ou manuseá-los de forma mais simples e robusta.

Imagine se tivéssemos de gerar e armazenar em memória 6 números gerados a partir dos números da Mega Sena.

```
public class GeraNumero {  
  
    public static void main(String args[]) {  
        int numero1 = ...  
        ...  
        int numero6 = ...  
    }  
}
```

Para que a computação de dados em massa, não se algo inviável, nas linguagens de programação nasceram os Arrays, que permitem manusear elementos em memória.

Java Arrays

Na linguagem Java, os arrays são uma ferramenta poderosa para manipulação de coleções de tipos referência (objetos) ou de tipos primitivos. Todos os arrays em java, também são considerados objetos, então eles são do tipo referência.

Para se definir um array de uma dimensão, ou vetor, podemos usar algumas das notações abaixo:

```
tipo idenitificador[] = new tipo[tamanho];
```

Veja como definir um array com capacidade de armazenar 10 números inteiros:

```
int inteiros[] = new int[10];
```

Os arrays quando inicializados e não preenchidos são inicializados com valores default:

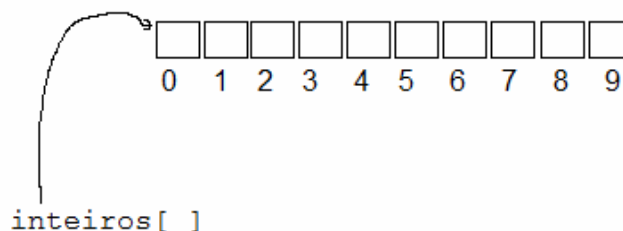
Tipos inteiros: 0;

Tipos ponto flutuante: 0.0;

Tipo booleano: false;

Tipo referência: null;

Em memória vemos que a referência `inteiros[]` aponta para um array :



```
// este código exibe o endereço de memória do array.
System.out.println( inteiros );
```

Para ter-se uma manipulação facilitada, os arrays em java, possuem um campo de somente leitura que devolve o seu tamanho (numero de elementos que ele pode armazenar) em memória:

```
int tamanhoInteiros = inteiros.length; // 10
```

O índice inicial dos arrays é o ZERO. Ou seja, para varrer um array, devemos sempre considerar o intervalo ZERO → TAMANHO-1;

Podemos também usar arrays para trabalhar com matrizes, ou seja, arrays multi-dimensionais.

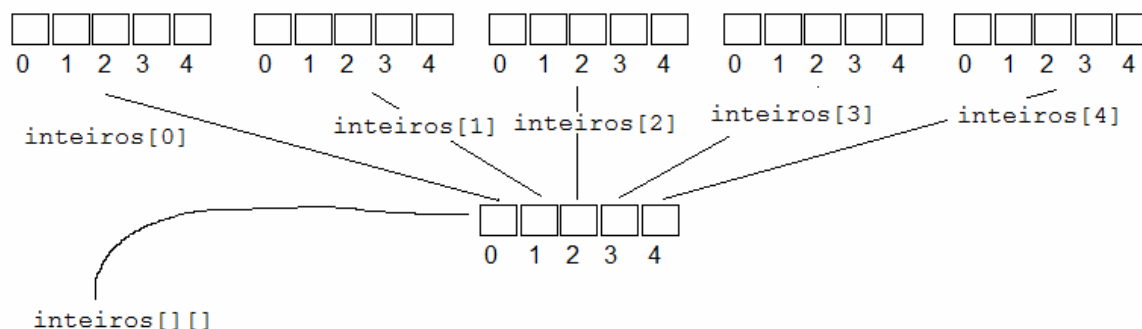
```
tipo identificador[][] = new tipo[linhas][colunas];
```

Veja como definir um array mult-dimensional de inteiros:

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

Criamos assim, uma matriz em memória capaz de armazenar 25 elementos, divididos em 5 linhas e 5 colunas.

Vizualizando na memória:



Vemos então, que na verdade um array vai guardar a referência de outro array, e assim sucessivamente, até chegar ao elemento propriamente dito. Nesse caso, os elementos estão nas posições `inteiros[0][0]` até `inteiros[4][4]`.

Definindo arrays.

Em java, existem várias sintaxes válidas para se definir arrays.

```
int vetor[] = new int[10]; // definição
int[] vetor= new int[10];
int vetor[] = { 1,2,3,4,5,6,7,8,9,10 }; // definição e
preenchimento
```

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

Para consultar o tamanho de um vetor:

```
int tamanhoVetor = vetor.length; // 10
```

Para consultar os tamanhos da matriz:

```
int matrizLinhas = matriz.length; // 3
int matrizColunas = matriz[0].length; // 5
```

Para definir os valores de um vetor:

```
vetor[0] = 0;
...
vetor[9] = 9;
```

Para consultar os tamanhos da matriz:

```
int matrizLinhas = matriz.length; // 3
int matrizColunas = matriz[0].length; // 5
```

Para definir os valores em uma matriz:

```
matriz[0][0] = 0;
...
matriz[2][4] = 14;
```

Manipulando arrays.

Por isso, para facilitar a manipulação de vetores e matrizes usamos as estrutura de repetição e seleção.

```
// preenchendo um vetor
for (int i=0; i < vetor.length; i++) {
    vetor[i] = i;
}
```

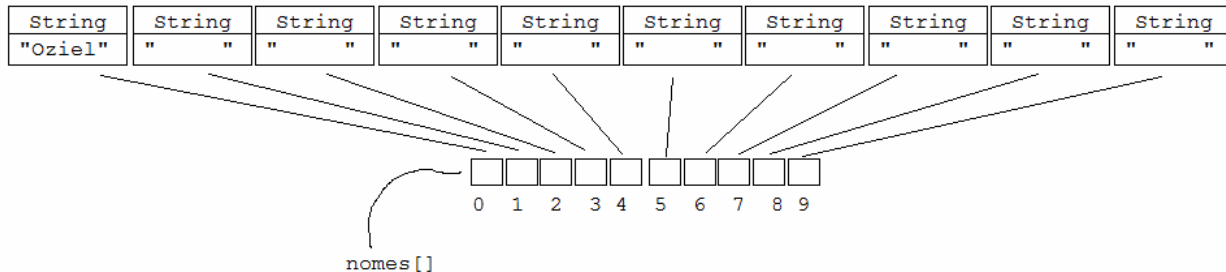
```
// preenchendo uma matriz
for (int i=0; i < matriz.length; i++) {
    for ( int j=0; j < matriz[i].length; j++ ) {
        matriz[i][j] = i*j;
    }
}
```

```
// imprimindo um vetor
for (int i=0; i < vetor.length; i++) {
    System.out.println( vetor[i] );
}
```

```
// imprimindo uma matriz
for (int i=0; i < matriz.length; i++) {
    for ( int j=0; j < matriz[i].length; j++ ) {
        System.out.println( matriz[i][j] );
    }
}
```

Definindo e manipulando arrays de objetos.

Em java, podemos criar arrays de tipos referência. Entretanto, eles não armazenam o valor diretamente como os tipos primitivos, eles armazenam referências dos objetos.



Os arrays de objetos também serão usados como forma de implementar uma relação de multiplicidade, ou seja, quando um objeto se relaciona com dois ou mais objetos. Exemplo: Um restaurante, possui várias contas, ou um Hotel, possui vários quartos.

A sintaxe para se criar arrays de objetos é a mesma usada nos tipos primitivos:

```
String nomes[] = new String[10]; // armazena 10 objetos String
```

```
String pessoas[][] = new String[10][10];
```

```
// preenchendo um vetor de Strings
```

```
for (int i=0; i < nomes.length; i++) {
    nomes[i] = new String("Valor: "+i);
}
```

```
// preenchendo uma matriz
```

```
for (int i=0; i < pessoas.length; i++) {
    for ( int j=0; j < matriz[i].length; j++ ) {
        pessoas [i][j] = new String("Pessoa: "+ i * j);
    }
}
```

```
// imprimindo um vetor de Strings
```

```
for (int i=0; i < nomes.length; i++) {
    System.out.print( nomes[i]+ " " );
}
```

```
// imprimindo uma matriz
```

```
for (int i=0; i < pessoas.length; i++) {
    for ( int j=0; j < pessoas [i].length; j++ ) {
        System.out.print( pessoas[i][j]+ " " );
    }
}
```

Exemplo de manipulação de objetos em arrays:

```
// Ponto.java
public class Ponto {

    public int x;
    public int y;
}

// TestaPontos.java
public class TestaPontos {

    public static void main(String args[]) {
        Ponto[] pontos = new Ponto[10];
        // criando pontos
        for (int i=0; i< pontos.length; i++) {
            Ponto p = new Ponto(); // cria um objeto ponto
            p.x = i;
            p.y = i * 2;
            pontos[i] = p;
        }

        for (int i=0; i< pontos.length; i++) {
            Ponto p = pontos[i];
            System.out.println("Ponto: X:" + p.x + " Y: " + p.y);
        }
    }
}
```

Cópia de arrays.

Ao invés de que precisemos escrever várias linhas de código, a cada vez que se precise copiar um array para um outro, a linguagem java possui uma classe chamada `java.lang.System` que possui uma rotina que já realiza essa tarefa.

```
public class TestaCopia {  
  
    public static void main(String args[]) {  
        int original[] = { 1, 2, 3, 4, 5, 6 };  
        int copia[] = new int[10];  
        System.arraycopy(original, 0, copia, 0, original.length);  
        System.out.print("Original: ");  
        for (int i=0; i < original.length; i++) {  
            System.out.print( original[i]+" ");  
        }  
        System.out.println();  
        System.out.print("Copia: ");  
        for (int i=0; i < copia.length; i++) {  
            System.out.print( copia[i]+" ");  
        }  
    }  
}
```


Discussão.

Para que servem os Arrays nas linguagens de programação?

Vetores e matrizes são elementos complexos de manipular, não existe nada mais fácil em Java para se manipular objetos?

Quando um objeto tiver uma relação múltipla com outro, vamos definir usando arrays?

Exercícios.

5. Desenho Avançado de Classes

No capítulo anterior vimos como construir classes a partir de um modelo aplicamos o conceito de encapsulamento e manipulamos o estado de alguns objetos usando métodos de instância e vimos como trabalhar com atributos e métodos de classe.

Neste capítulo vamos conhecer as técnicas avançadas da programação orientada a objetos e sua implementação em java para os conceitos:

Herança, Polimorfismo e utilização de Pacotes;

Herança

O conceito de encapsular estrutura e comportamento em um tipo não são exclusivos da orientação a objetos; particularmente, a programação por tipos abstratos de dados segue esse mesmo conceito.

O que torna a orientação a objetos única é o conceito de herança.

Herança é um mecanismo que permite que características comuns a diversas classes com comportamentos comuns, sejam abstraídas e centralizadas em uma classe base, ou superclasse.

A partir de uma superclasse outras classes podem ser especificadas. Cada classe derivada ou subclasse recebe as características (atributos e métodos) da superclasse, e ainda podemos acrescentar a uma subclasse elementos particulares a ela. Os construtores não são herdados.

Sendo uma linguagem de programação orientada a objetos, a linguagem Java oferece mecanismos para definir subclasses a partir de superclasses. Uma subclasse somente pode herdar uma única superclasse diretamente.

É fundamental que se tenha uma boa compreensão sobre como objetos a partir de subclasses são criados e manipulados, assim como das restrições de acesso aos membros das superclasses.

Herança é sempre utilizada em Java, mesmo que não explicitamente. Quando uma classe é criada e não há nenhuma referência à uma superclasse, implicitamente o compilador insere na classe criada uma relação de herança direta com a classe `java.lang.Object`, é por esse motivo que todos os objetos podem invocar os métodos da classe `java.lang.Object`, tais como `equals()` e `toString()`.

Generalização

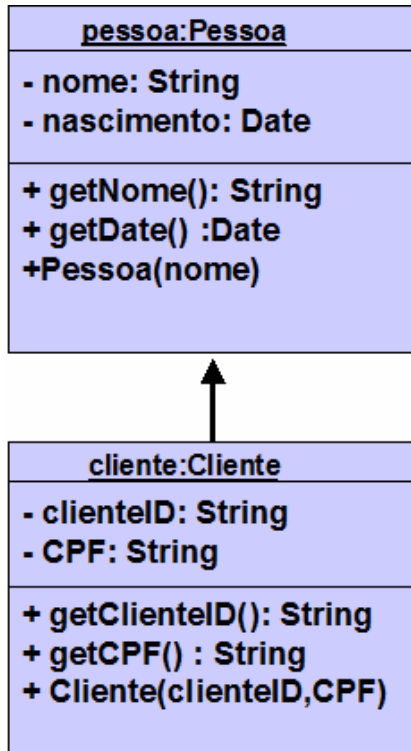
Processo pelo qual identificamos atributos comuns em classes diferentes que são colocados numa superclasse, menos específica e mais genérica.

Especialização

Processo pelo qual identificamos atributos incomuns em classes diferentes que são colocados nas subclasses de uma superclasse, tornando-a mais específica e menos genérica.

Representação na UML

A UML possui uma notação especificar para representar que uma classe estende o comportamento de outra.



Vemos então que a classe “cliente” é filha da classe “pessoa”, e herda os atributos e métodos do pai.

Para o objeto pessoa, posso recuperar o nome e a data de nascimento.

Para o objeto cliente, posso recuperar o nome, a data de nascimento, o CPF e o código do cliente.

A capacidade de herança é o mecanismo que garante a linguagem Java altos índices de reaproveitamento de código, permitindo uma melhor componentização.

Os construtores não são herdados, por isso devemos na subclasse especificar construtores que chamem os construtores da superclasse, para isso usamos a keyword **super**.

Codificando o uso da herança em Java.

Para especificarmos que uma classe estende outra, usa-se na definição de classe uma keyword **extends** que relaciona uma subclasse com a sua superclasse.

Uma subclasse pode estender somente uma superclasse diretamente. A herança em Java é simples.

```
public class Pessoa {  
    // definição de classe  
  
}  
  
public class Cliente extends Pessoa {  
    // definição de classe  
  
}
```

Objetos a partir de subclasses.

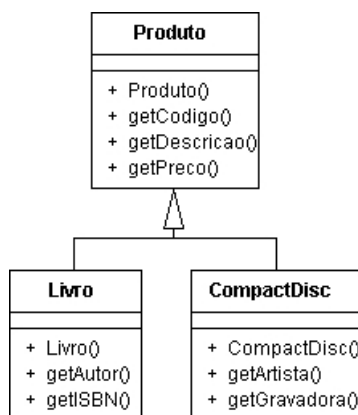
Quando instanciamos um objeto a partir de uma subclasse, ele “ganha” da superclasse os membros nela definidos, entretanto, o modelo de acesso aos membros continua valendo. Membros demarcados como **private** não podem ser acessados nas subclasses, membros **public** ou **protected** podem ser acessados usando a keyword **super**.

Assim, para executar um construtor de uma superclasse e garantir o encapsulamento e reaproveitamento de código, usamos `super([argumentos do construtor])`, e que deve ser a primeira instrução do construtor da subclasse.

Ex: Uma loja que vende livros e cds precisa catalogá-los com código, descrição, preço, autor, artista, número isbn e gravadora.

Usando os conceitos de generalização podemos criar uma superclasse **Produto** que conterá código, descrição e preço. Enquanto a subclasse **Livro** possuirá autor e isbn, e a classe **CompactDisc** possuirá artista e gravadora.

Diagrama UML de classes:



Implementação das classes:

```
public class Produto extends Object {

    private int codigo;
    private double preco;
    private String descricao;

    public Produto(int codigo, double preco, String descricao) {
        this.codigo = codigo;
        this.preco = preco;
        this.descricao = descricao;
    }

    public int getCodigo() {
        return codigo;
    }

    public double getPreco() {
        return preco;
    }

    public String getDescricao() {
        return descricao;
    }
}

public class Livro extends Produto {

    private String autor;
    private String isbn;

    public Livro (int codigo, double preco, String descricao,
                  String autor, String isbn) {
        // chama um construtor da super classe.
        // a instrucao super(), deve ser a 1a linha do construtor
        super( codigo, preco, descricao );
        // seta os atributos desta classe.
        this.autor = autor;
        this.isbn = isbn;
    }

    public String getAutor() {
        return autor;
    }

    public String getISBN() {
        return isbn;
    }
}
```

```
public class CompactDisc extends Produto {

    private String artista;
    private String gravadora;

    public CompactDisc (int codigo, double preco, String descricao,
                        String artista, String gravadora) {
        // chama um construtor da super classe.
        // a instrucao super(), deve ser a 1a linha do construtor
        super( codigo, preco, descricao );
        // seta os atributos desta classe.
        this.artista = artista;
        this.gravadora = gravadora;
    }

    public String getArtista() {
        return artista;
    }

    public String getGravadora() {
        return gravadora;
    }
}
```

Testando o uso destas subclasses:

```
public class TestaProdutos {

    public static void main(String arg[]) {
        Livro livro = new Livro(1, 55.0, "Universidade Java!", "Oziel
Moreira Neto", "01010101010");
        System.out.println("Cod:" + livro.getCodigo() );
        System.out.println("Desc:" + livro.getDescricao() );
        System.out.println("Preco:" + livro.getPreco() );
        System.out.println("Autor:" + livro.getAutor() );
        System.out.println("ISBN:" + livro.getISBN() );
        System.out.println();

        CompactDisc cd = new CompactDisc(2, 25.0, "Rock Brasileiro!",
"Brasileiros", "BrasilRecords");
        System.out.println("Cod:" + cd.getCodigo() );
        System.out.println("Desc:" + cd.getDescricao() );
        System.out.println("Preco:" + cd.getPreco() );
        System.out.println("Artista:" + cd.getArtista() );
        System.out.println("Gravadora:" + cd.getGravadora() );
        System.out.println();
    }
}
```

Podemos dizer então que Livro e CompactDisc são subclasses de Produto, ou seja, especializações de produto, ou ainda tipos de produto.

Formas de herança

Há várias formas de relacionamentos em herança:

Extensão: uma subclasse estende uma superclasse, acrescentando novos membros (atributos e/ou métodos) que definem novos comportamentos que são somados aos já existentes na superclasse. A superclasse permanece inalterada, motivo pelo qual este tipo de relacionamento é tecnicamente referenciado como **herança estrita**.

Especificação: a superclasse abstrata especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Ou seja, especifica-se que apenas a *interface* (conjunto de especificação dos métodos públicos) da superclasse é herdada pela subclasse. Dando a subclasse a responsabilidade de transformar os conceitos abstratos da superclasse em implementações concreta.

Combinação de extensão e especificação: a subclasse herda a interface de uma classe abstrata, e uma implementação padrão dos métodos concretos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse, ou ter que oferecer alguma implementação para métodos que a superclasse tenha declarado como abstratos. Normalmente, este tipo de relacionamento é denominado **herança polimórfica**.

Polimorfismo.

Polimorfismo é o princípio pelo usamos objetos a partir de uma subclasse através de referências do tipo de uma superclasse da hierarquia.

A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de introspecção da Java Virtual Machine.

No caso de polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos. Esse mecanismo é chamado de **overriding**, reescrita de método.

O uso do polimorfismo introduz os conceitos relacionados de *herança* e a motivação para a definição de métodos abstratos, melhorando a modelagem para garantir a evolução das aplicações.

É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução, este processo é chamado de invocação virtual de métodos.

Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode ser não-intuitivo, pois acontece em tempo de execução.

Usando o exemplo anterior, da hierarquia de produtos, podemos:

```
Produto produto1 = new Livro(1, 55.0, "Universidade Java!", "Oziel Moreira Neto", "01010101010");
```

```
Produto produto2 = new CompactDisc(2, 25.0, "Rock Brasileiro!", "Brasileiros", "BrasilRecords");
```

Entretanto, nas referências da superclasse Produto produto1 e produto2, temos acesso somente aos métodos da classe Produto (getCodigo, getPreco e getDescricao), entretanto os objetos em memória são do tipo Livro (produto1) e CompactDisc (produto2), pois foi a partir destas classes que os objetos foram criados.

Podemos acessar novamente os membros das subclasses fazendo um Type Cast:

```
Livro livro1 = (Livro) produto1;
```

```
CompactDisc cd1 = (CompactDisc) produto2;
```

Devemos lembrar que todo CompactDisc é um produto, todo Livro é um produto, entretanto, nem todo Produto será um CompactDisc ou um Livro.

Isto significa que podemos criar um método numa classe que receba um objeto do tipo do Produto. Analisando por esse lado, vemos que neste método poderíamos passar objetos originados da classe Produto e das suas subclasses.

```
public class Catalogo {  
  
    public void adicionaProduto(Produto produto) {  
        ...  
    }  
}  
  
public class TesteCatalogo {  
  
    ...  
  
    Catalogo catalogo = new Catalogo();  
  
    Produto produto1 = new Livro(1, 55.0, "Universidade Java!", "Oziel  
Moreira Neto", "01010101010");  
  
    Produto produto2 = new CompactDisc(2, 25.0, "Rock Brasileiro!",  
"Brasileiros", "BrasilRecords");  
  
    catalogo.adicionaProduto ( produto1 );  
    catalogo.adicionaProduto ( produto2 );  
  
    ...  
}
```

O Polimorfismo permite que manipulemos hierarquias inteiras de objetos usando como tipo na definição de interface de métodos o tipo de uma superclasse.

Isso significa que ao vermos um método que receba como parametro o tipo `java.lang.Object`, podemos passar para este método qualquer tipo de objeto, pois em java, todos os objetos são filho da classe Object?

Reescrita de método (Overriding)

Uma outra característica que o polimorfismo permite, é a sobreescrita de métodos (**overriding**), que consiste em alterar o comportamento de um método numa subclasse, mantendo a mesma assinatura da superclasse. Ou seja, damos comportamentos diferentes á um mesmo conceito.

Um caso clássico de overriding, é reescrever o método `toString()` nas classes filhas de `java.lang.Object`.

```
public class Object {  
    public String toString() {  
        ...  
    }  
}  
  
// devemos lembrar que toda classe Java é filha de Object.  
public class Produto extends Object {  
    public String toString() {  
        return "Produto: ";  
    }  
}  
  
public class TesteProduto {  
    Object obj = new Produto();  
  
    // o objeto em memoria referenciado por obj  
    // é do tipo Produto, portanto, quando  
    // for executado obj.toString(), sera chamado  
    // o método definido na classe Produto.  
    System.out.println( obj.toString() );  
}
```

Outro caso clássico, quando temos dois tipos de contas diferentes dentro do sistema do banco, Poupanca e ContaEspecial:

```
public class Conta {

    private double saldo;

    public Conta(double saldo) {
        this.saldo = saldo;
    }

    public void debito(double valor) {
        saldo -= valor;
    }

    public void credito(double valor) {
        saldo += valor;
    }

    public double getSaldo() {
        return saldo;
    }

}

public class ContaEspecial extends Conta {

    private double limite;

    public ContaEspecial(double saldo, double limite) {
        super(saldo);
        this.limite = limite;
    }

    // overriding do metodo Conta.debito(double)
    public void debito(double valor) {
        // chama o metodo da Produto.getSaldo() pois o
        // o atributo saldo e private
        if ( super.getSaldo() > 0 ) {
            // chama o metodo da Produto.debito(double) pois o
            // o atributo saldo e private
            super.debito( valor );
        } else {
            limite -= valor;
        }
    }

    // overriding do metodo Conta.getSaldo()
    public double getSaldo() {
        // chama o metodo da Produto.getSaldo() pois o
        // o atributo saldo e private
        return super.getSaldo() + limite;
    }

}
```

```
public class Poupanca extends Conta {

    private double juros;

    public Poupanca (double saldo, double juros) {
        super(saldo);
        this.juros = juros;
    }

    // overriding do metodo Conta.credito(double)
    public void credito(double valor) {
        // multiplica o valor do saldo * juros
        // e soma o valor depositado como credito.
        double valorCredito = ( super.getSaldo() * juros ) + valor;
        // como o atributo saldo e private da superclasse,
        // chamamos um metodo da superclasse para alterar
        // o estado do atributo.
        super.credito( valorCredito );
    }

}

public class TesteContas {

    public static void main(String args[]) {

        Conta c1 = new Poupanca(100.00, 0.1);
        c1.debito( 50 );
        c1.credito( 15 );
        System.out.println("Saldo Poupanca: " + c1.getSaldo() );
        System.out.println();

        Conta c2 = new ContaEspecial(100.00, 200.00);
        c2.debito( 75 );
        c2.credito( 15 );
        System.out.println("Saldo Especial: " + c2.getSaldo() );
        System.out.println();

    }

}
```

Quando definimos para subclasses, métodos com o mesmo nome, mesmos tipos e número de parametros e mesmas exceções na cláusula throws, dizemos que é a reescrita de método. Pois a intenção é manter a mesma assinatura de método que a superclasse, mas com implementação diferente.

Sobrecarga de métodos e construtores (Overloading)

Uma outra característica que o polimorfismo permite, é a sobrecarga de métodos ou construtores.

Quando definimos para uma mesma classe, vários métodos com o mesmo nome, mas com parametros diferentes, dizemos que é uma sobrecarga de métodos.

Quando definimos para uma mesma classe, vários construtores com o mesmo nome, mas com parametros diferentes, dizemos que é uma sobrecarga de construtores.

Vendo a classe Conta abaixo:

```
public class Conta {

    private int numero;
    private double saldo;
    private double juros;
    private java.util.Date vencimento;
    ..

    // construtores
    // construtor padrão sobrecarregado
    public Conta() {
        numero = 0;
        // saldo padrao é ZERO.
        saldo = 0;
        juros = 0;
        // data atual do sistema
        vencimento = new java.util.Date();
    }

    // construtor com parametros sobrecarregado
    public Conta(int numeroConta, double saldoInicial) {
        numero = numeroConta;
        saldo = saldoInicial;
        juros = 0;
        // data atual do sistema
        vencimento = new java.util.Date();
    }
    ...

    // metodo sobrecarregado
    public void debito(double valor) {
        ...
    }

    // metodo sobrecarregado
    public void debito(long valor) {
        ...
    }
}
```

Refinando a hierarquia de classes.

Durante o processo de generalização e especialização de classes numa árvore de herança, há a necessidade de delimitação desta árvore. Onde ela começa e onde ela deve terminar para garantir que os comportamentos definidos serão mantidos na evolução do sistema.

Para isso, a linguagem java traz os conceitos de classe abstrata e interface, métodos concretos e abstratos, classes concretas, e classes que não podem ser mais especializadas, limitando a herança.

Até agora, vimos somente classes concretas com métodos concretos, capazes de gerar objetos e que podem ser especializadas. Entretanto, alguns modelos de negócio exigirão um controle mais apurado da herança. Para isso usaremos dois conceitos:

“Final Classes”: usando o modificador final na declaração de classe, impedimos que esta classe seja especializada, ou seja, ela não pode mais dar herança.

No modelo do banco, a classe ContaEspecial pode ser marcada como uma **“Final Class”**, para evitar que ela seja especializada.

```
public final class ContaEspecial extends Conta {  
    ...  
}
```

Assim, poderíamos dizer que a classe ContaEspecial é uma forma final de uma Conta.

“Final Methods”: usando o modificador final na declaração de método, impedimos que este método seja especializado, ou seja, ela não pode mais ser reescrito na herança.

Seguindo o mesmo princípio, poderíamos transformar um método da classe Poupanca em uma **“Final Method”** também:

```
public class Poupanca extends Conta {  
    ...  
    // overriding do metodo Conta.credito(double)  
    public final void credito(double valor) {  
        // multiplica o valor do saldo * juros  
        // e soma o valor depositado como credito.  
        double valorCredito = ( super.getSaldo() * juros ) + valor;  
        // como o atributo saldo e private da superclasse,  
        // chamamos um metodo da superclasse para alterar  
        // o estado do atributo.  
        super.credito( valorCredito );  
    }  
    ..  
}
```

Assim, poderíamos dizer que a classe Poupanca, tem um método que não pode ser especializado, este método tem sua forma final definido na classe Poupanca.

Qualquer classe, pode ter métodos concretos e/ou métodos final, que são aqueles na herança não pode ser especializados.

“**Final Variables**”: usando o modificador final na declaração de um atributo, impedimos que este atributo seja alterado depois de inicializado, ou seja, ele se torna uma constante.

```
public class Conta {  
    ...  
    public static final double CPMF = 0.0038;  
}
```

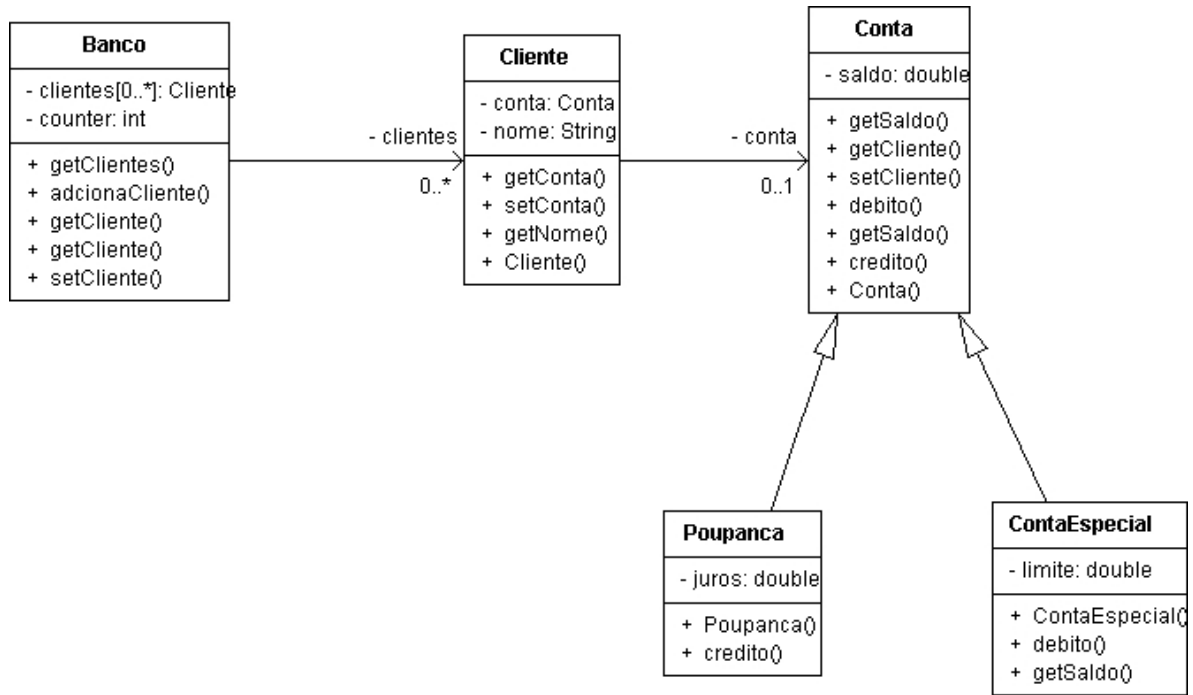
Obs: **Constantes**, geralmente são definidas como **public static final**, para poderem ser acessadas por qualquer outra classe.

Resumindo:

Quando um atributo é demarcado com final, ele se torna uma constante. Quando um método é demarcado como final, ele não pode ser reescrito na herança. E quando uma classe é demarcada como **final**, ela não pode mais ser especializada.

Diagrama UML de Classes (Refinado) do Modelo do Banco.

Aqui está o diagrama completo de classe do modelo do banco.



Através do diagrama, podemos ver que o Banco tem vários Clientes e que cada Cliente tem uma Conta, podendo ser do tipo Poupanca ou tipo ContaEspecial.

Classes abstratas e métodos abstratos.

Da mesma forma que limitamos os ramos finais de uma hierarquia de classes, devemos especificar quais são as superclasses mais genéricas do modelo, e assim garantir que este conjunto de classes foi bem definido nos processos de generalização e especialização.

Classes muito genéricas, geralmente não terão associadas a elas objetos, justamente por serem muito genéricas. No caso do modelo da loja virtual que trabalha com Produtos, a loja não vende Produtos, e sim Livros, CDs, e outros produtos que podem aparecer, que são específicos. Portanto faria mais sentido que a classe Produto fosse demarcada como abstrata, não podendo gerar objetos e se tornando como uma superclasse genérica deste modelo.

Uma classe abstrata pode conter métodos com comportamento bem definidos (concretos) e/ou métodos abstratos, que servem somente como conceito para um determinado comportamento.

Quando uma subclasse especializa uma classe abstrata, ela é obrigada a reescrever todos os métodos abstratos da superclasse, garantido que eles tenham uma implementação concreta.

No caso do modelo do Produto, podemos definir a classe Produto como abstrata que vai conter um método abstrato `getDescricaoCompleta()` fazendo com que suas subclasses reescrevam este método e deem uma implementação específica para cada tipo de produto.

Para se definir uma classe abstrata usamos a sintaxe:

```
[modificador] abstract class identificador [extends OutraClasse] {  
  
    // os métodos abstratos não possuem corpo,  
    // somente definição.  
    [modificador] abstract tipo identificador ([argumentos]);  
  
}
```

Alterando a classe Produto.

```
public abstract class Produto extends Object {

    private int codigo;
    private double preco;
    private String descricao;

    public Produto(int codigo, double preco, String descricao) {
        this.codigo = codigo;
        this.preco = preco;
        this.descricao = descricao;
    }

    public int getCodigo() {
        return codigo;
    }

    public double getPreco() {
        return preco;
    }

    public String getDescricao() {
        return descricao;
    }

    // metodo abstrato não possui corpo de implementacao.
    public abstract String getDescricaoCompleta();
}
```

Revisitando os códigos fontes das classes Livro, CompactDisc e TestaProdutos, devemos reescrever o método `public abstract String getDescricaoCompleta();`

```
public class Livro extends Produto {

    ...

    public String getDescricaoCompleta() {
        return "Livro: " + super.getDescricao() + " Autor: " + autor + " ISBN: " +
isbn;
    }
}

public class CompactDisc extends Produto {

    ...

    public String getDescricaoCompleta() {
        return "CompactDisc: " + super.getDescricao() + " Artista: " + artista
+" Gravadora: "+gravadora;
    }
}
```

Alterando a classe TestaProdutos.

```
public class TestaProdutos {

    public static void main(String arg[]) {
        Livro livro = new Livro(1, 55.0, "Universidade Java!", "Oziel
Moreira Neto", "01010101010");
        System.out.println("Cod:" + livro.getCodigo() );
        System.out.println("Desc:" + livro.getDescricao() );
        System.out.println("Preco:" + livro.getPreco() );
        System.out.println("Autor:" + livro.getAutor() );
        System.out.println("ISBN:" + livro.getISBN() );
        System.out.println("Descricao: "+livro.getDescricaoCompleta() );
        System.out.println();

        CompactDisc cd = new CompactDisc(2, 25.0, "Rock Brasileiro!",
"Brasileiros", "BrasilRecords");
        System.out.println("Cod:" + cd.getCodigo() );
        System.out.println("Desc:" + cd.getDescricao() );
        System.out.println("Preco:" + cd.getPreco() );
        System.out.println("Artista:" + cd.getArtista() );
        System.out.println("Gravadora:" + cd.getGravadora() );
        System.out.println("Descricao: "+ cd.getDescricaoCompleta() );

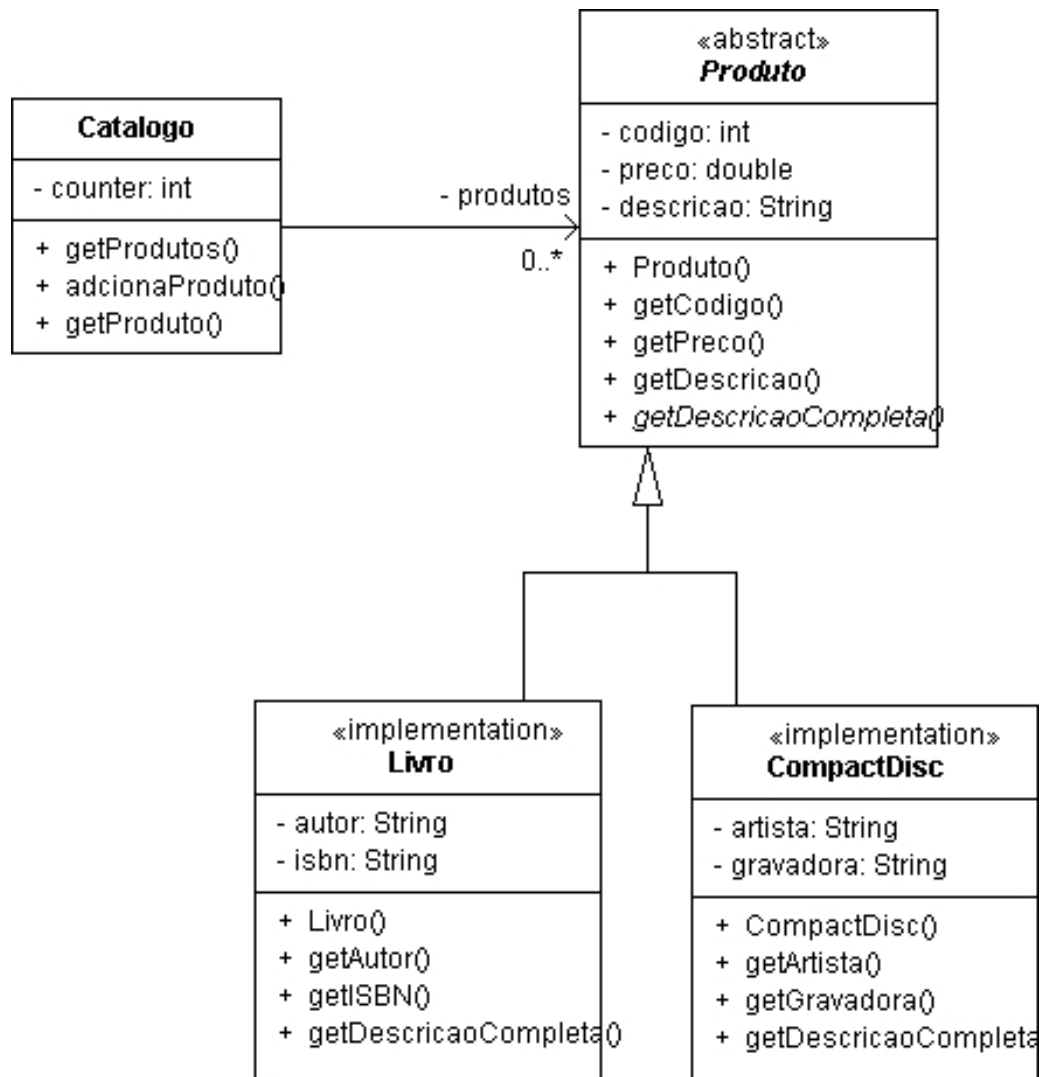
        System.out.println();

    }
}
```

Dessa forma qualquer objeto do tipo Produto, obrigatoriamente terá o método `getDescricaoCompleta()`.

Na UML, os métodos abstratos aparecem em itálico, e a classe recebe um esteriótipo <<abstract>>, as subclasses que especializam uma superclasse abstrata, recebem o esteriótipo de <<implementation>>.

Veja a UML completa do modelo de Produto.



Através do diagrama, podemos ver que o Catálogo tem vários Produtos e podendo ser do tipo Livro ou tipo CompactDisc.

Interfaces e métodos abstratos.

Seguindo o mesmo conceito das classes abstratas, as Interfaces em Java tem o propósito de forçar que uma subclasse reescreva todos os métodos abstratos definindo um comportamento para eles.

As interfaces são consideradas como os maiores níveis de abstração existente dentro da orientação a objetos, pois não possuem conceitos concretos, somente abstratos.

A intenção de se definir uma interface dentro de um modelo, é para garantir que o conjunto de classes que implementam esta interface, pelo menos terá aquele comportamento definido pelo conjunto de métodos que a interface descreve.

Diferentemente das classes Abstratas que podem ter métodos concretos, uma Interface Java somente pode conter métodos abstratos, não pode conter construtores e somente atributos estáticos ou constantes.

Um interface pode estender outra interface, dessa forma, quando uma subclasse implementar uma interface que estende outra, ela deve implementar todos os métodos abstratos definidos nas duas interfaces.

Diferentemente de quando uma classe Abstrata estende outra classe, pois em Java a herança é simples, uma subclasse pode implementar múltiplas interfaces.

Para se definir uma interface usamos a sintaxe:

```
public interface identificador [extends OutraInterface] {  
  
    // os métodos de uma interface não possuem corpo,  
    // somente definição.  
    [modificador] tipoRetorno identificador ([argumentos]);  
  
}
```

Poderíamos assim definir uma interface para o modelo de Produtos, chamada Vendavel, que possui um método abstrato `getValorVenda()`.

```
public interface Vendavel {  
  
    public double getValorVenda();  
  
}
```

Dessa forma, todas os objetos do modelo que quiserem ser vendavel, implementam a interface Vendavel, dando um comportamento para o método `getValorVenda()` de acordo com a necessidade de cada objeto.

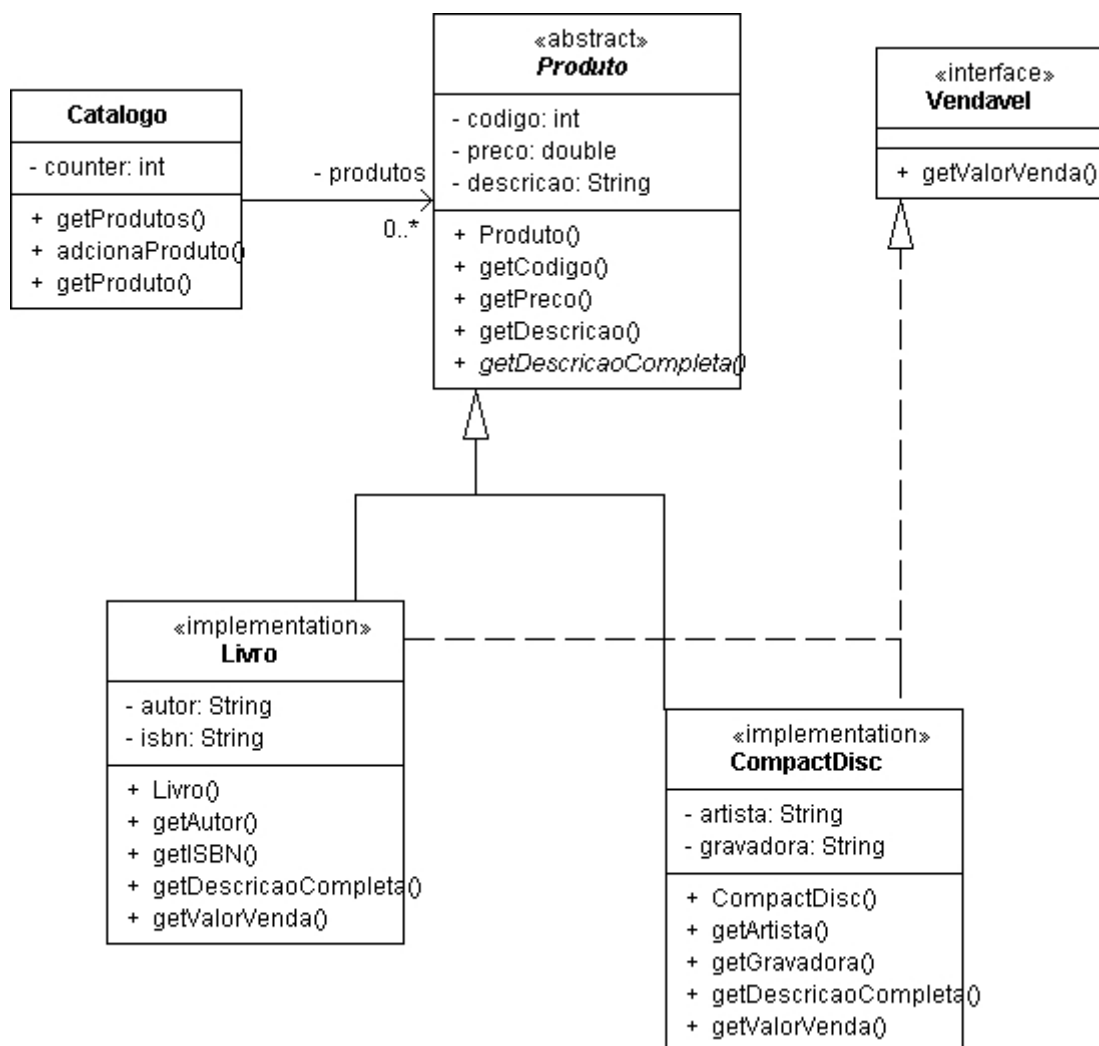
Entendemos que o CompactDisc e o Livro são vendáveis, e assim alteraremos essas classes.

Para que uma subclasse implemente uma interface, devemos alterar a definição de classe.

```
public class Livro extends Produto implements Vendavel {
    ...
    public double getValorVenda() {
        return getPreco() * 1.2;
    }
}

public class CompactDisc extends Produto implements Vendavel {
    ...
    public double getValorVenda() {
        return getPreco() * 1.1;
    }
}
```

Na UML fica assim:

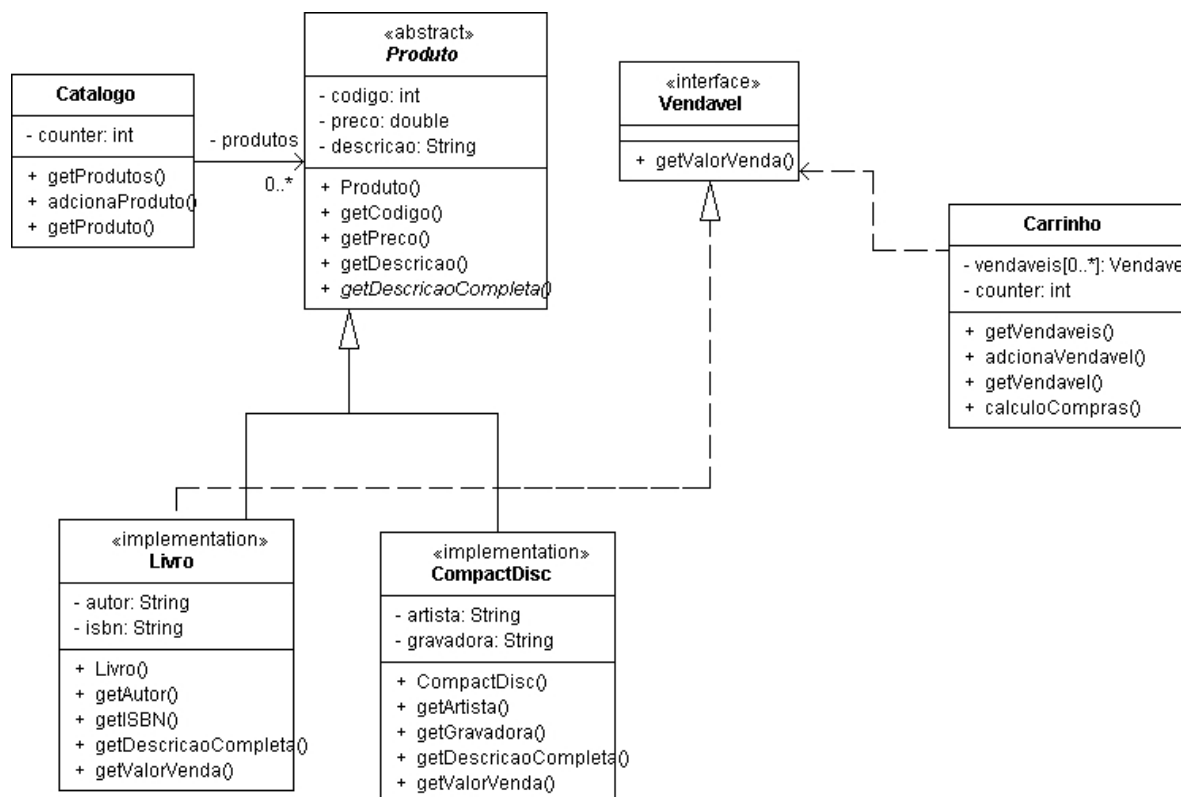


Como uma interface define um tipo, podemos usar esse tipo como referência de objetos. Assim podemos usar o tipo de uma interface como parametro de métodos ou construtores, usando o conceito do polimorfismo.

Podemos assim criar um carrinho de compras, que armazenara produtos vendaveis e calcula o valor das compras usando preco de venda.

```
public class Carrinho {  
  
    private Vendavel[] vendaveis;  
    private int counter;  
  
    public Carrinho() {  
        vendaveis = new Vendavel[10];  
        counter = 0;  
    }  
  
    public Vendavel[] getVendaveis() {  
        return vendaveis;  
    }  
  
    public int adicionaVendavel(Vendavel vendavel) {  
        vendaveis[ counter++ ] = vendavel;  
        return counter;  
    }  
  
    public Vendavel getVendavel(int index) {  
        return vendaveis[ index ];  
    }  
  
    public double calculoCompras() {  
        double valor = 0;  
        for ( int i =0; i<vendaveis.length; i++ ) {  
            valor += vendaveis[i].getValorVenda();  
        }  
        return valor;  
    }  
}
```


O modelo final da UML fica assim:



```

public class TestaCarrinho {

    public static void main(String arg[]) {
        Livro livro = new Livro(1, 55.0, "Universidade Java!", "Oziel
Moreira Neto", "01010101010");
        CompactDisc cd = new CompactDisc(2, 25.0, "Rock Brasileiro!",
"Brasileiros", "BrasilRecords");

        Carrinho carrinho = new Carrinho();
        carrinho.adicionaVendavel( livro );
        carrinho.adicionaVendavel( cd );
        System.out.println("Valor das compras: " + carrinho.calculoCompras()
);
    }
}
    
```

Organizando as classes em pacotes.

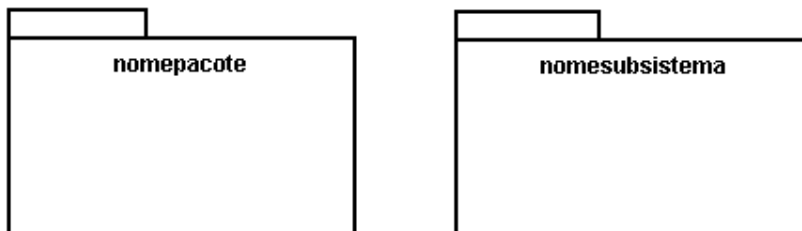
Para facilitar a organização de nossas classes, a tecnologia Java nos fornece a capacidade de classificarmos nossas classes e armazená-las em diretórios de classes que podem ser referenciados dentro do código. Ao diretório de classe damos o nome de pacote.

O programador deve separar em pacotes as classes por responsabilidades, a própria API da JavaSoft foi dividida assim:

java.io – pacote que contém classes específicas para manipula a entrada e saída de dados;
java.lang – pacote que contém as classes core da J2SE e não precisa ser importado;
java.util – pacote com classes utilitárias, coleções de objetos, etc.
java.text – pacote para a formatação textual, de números, data, etc.
java.awt – pacote que contém as classes necessárias para construção de GUIs simples;
javax.swing – pacote que contém as classes necessárias para construção de GUIs complexas;

Exemplo de notação para pacotes que podem ser agrupados por tecnologia ou por grupo de funcionalidades (Sistema ou SubSistema)

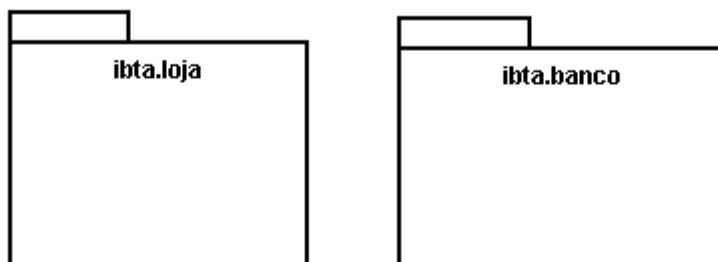
Na notação o nome de um pacote é sempre minúsculo!



Para colocarmos nossas classes em pacotes, usamos as keyword `package`.

Para que possamos usar uma classe de um pacote, dentro de uma outra classe de outro pacote, usamos a keyword `import`.

Pacotes podem ter subpacotes, separados por `.` (ponto).



Devemos alterar os códigos fontes referenciando qual será o pacote de cada um das classes:

<code>package ibta.banco;</code>	<code>package ibta.banco;</code>
<code>public class Banco {</code>	<code>public class Cliente {</code>
<code>..</code>	<code>..</code>
<code>}</code>	<code>}</code>
<code>package ibta.banco;</code>	<code>package ibta.banco;</code>
<code>public class Conta {</code>	<code>public final class ContaEspecial</code>
<code>..</code>	<code>extends Conta {</code>
<code>}</code>	<code>..</code>
<code>package ibta.banco;</code>	<code>}</code>
<code>public class Poupanca</code>	
<code>extends Conta {</code>	
<code>..</code>	
<code>}</code>	
<code>package ibta.loja;</code>	<code>package ibta.loja;</code>
<code>public class Carrinho {</code>	<code>public class Catalogo {</code>
<code>..</code>	<code>..</code>
<code>}</code>	<code>}</code>
<code>package ibta.loja;</code>	<code>package ibta.loja;</code>
<code>public class CompactDisc</code>	<code>public class Livro</code>
<code>extends Produto</code>	<code>extends Produto</code>
<code>implements Vendavel {</code>	<code>implements Vendavel {</code>
<code>..</code>	<code>..</code>
<code>}</code>	<code>}</code>
<code>package ibta.loja;</code>	<code>package ibta.loja;</code>
<code>public abstract class Produto {</code>	<code>public interface Vendavel {</code>
<code>..</code>	<code>..</code>
<code>}</code>	<code>}</code>

Dessa forma, as classes deverão ser salvas numa estrutura de diretórios que segue o nome do pacote, onde o . (ponto) significa subdiretório, e o caminho de diretórios antes do primeiro nome do pacote chamamos de CLASSPATH (caminho para as classes):

Alterando o código para usarmos pacotes em nossas classes, melhorando a separação delas, devemos alterar as classes de Testes para importar os pacotes de classes:

<code>import ibta.loja.*;</code>	<code>import ibta.loja.*;</code>
<code>public class TestaProdutos {</code>	<code>public class TestaCarrinho {</code>
<code>..</code>	<code>..</code>
<code>}</code>	<code>}</code>
<code>Import ibta.banco.*;</code>	
<code>public class TesteContas {</code>	
<code>..</code>	
<code>}</code>	

A execução das classes de teste não fica alterada.

Discussão.

Resumindo. O que se entende por Herança?

Quando devemos usar especialização e generalização?

Sabemos que o polimorfismo nos fornece uma ferramenta poderosa na adição de tipos de objetos, então quer dizer que ele nos permite ter sistemas mais flexíveis e que tem menos impacto nas mudanças?

Qual a necessidade de usar Classes Abstratas?

Qual a maior diferença de Classes Abstratas e Interfaces?

Exercícios.

6. Manipulando Erros e Exceções

Uma **exceção** é um objeto gerado com o propósito de indicar que algum tipo de condição excepcional ocorreu durante a execução de uma chamada de método ou construtor.

Assim, as exceções estão associadas a condições de erro que não tinham como ser verificadas durante a compilação do programa e que acontecerão somente durante a sua execução.

As duas atividades associadas à manipulação de uma exceção são: *geração* e *captura*

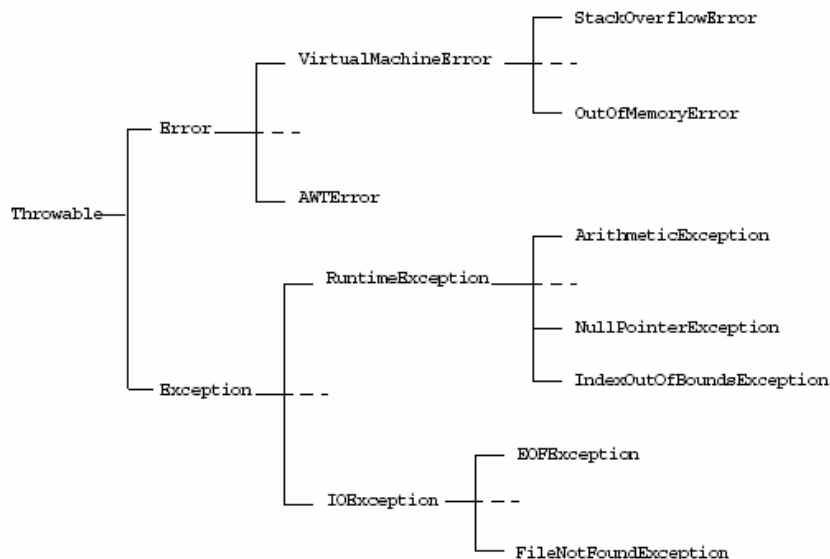
A manipulação (tratamento) de uma situação excepcional significa definir ações necessárias para a recuperação da situação de falha e dar continuidade a execução do programa.

Para cada exceção que pode ocorrer durante a execução de um código java, um bloco de ações de tratamento (um *exception handler*) deverá ser especificado.

É um mecanismo adequado à manipulação de erros síncronos, para situações onde a recuperação da falha é possível.

A hierarquia de exceções

Uma exceção em Java é um objeto. A classe raiz de todas as exceções é a `java.lang.Throwable`. Apenas objetos dessa classe ou de suas subclasses podem ser gerados, propagados e capturados através do mecanismo de tratamento de exceções.



A classe `java.lang.Throwable` tem duas subclasses:

`java.lang.Exception` : É a superclasse das classes derivadas de `java.lang.Throwable` que indica situações que a aplicação pode gerar e realizar um tratamento apropriado permitindo prosseguir com o processamento. Exemplo: Quando acessamos um arquivo, deveremos nos preocupar com as condições de erro de IO (entrada e saída), para isso existe a classe `java.io.IOException`;

`java.lang.Error` : É a superclasse das classes derivadas de `java.lang.Throwable` que indica situações que a aplicação não poderá tratar. Usualmente indica situações anormais, que não deveriam ocorrer dentro da JRE. Exemplo: Se por alguma condição anormal, ocorrer uma falha no sistema operacional que impeça da JRE alocar memória, acontecerá uma `java.lang.OutOfMemoryError`.

Exceções mais conhecidas

`java.lang.ArithmeticException` : indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0.

`java.lang.NumberFormatException` : indica que tentou-se a conversão de uma `String` para um formato numérico, mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de `java.lang.IllegalArgumentException`.

`java.lang.ArrayIndexOutOfBoundsException` : indica a tentativa de acesso a um elemento de um array fora de seus limites -- ou o índice era negativo ou era maior ou igual ao tamanho do array. É uma subclasse de `java.lang.IndexOutOfBoundsException`, assim como a classe `java.lang.StringIndexOutOfBoundsException`.

`java.lang.NullPointerException`: indica que a aplicação usou uma referência com valor null para invocar um método ou acessando um atributo de um objeto, por exemplo.

`java.lang.ClassNotFoundException`: indica que a aplicação tentou carregar uma classe mas não foi possível encontrá-la.

`java.io.IOException`: indica a ocorrência de algum tipo de erro em operações de entrada e saída.

Entre os erros mais conhecidos, subclasses de `java.lang.Error`, estão `java.lang.StackOverflowError` e `java.lang.OutOfMemoryError`. São situações onde não é possível uma correção a partir de um tratamento realizado pelo próprio programa que está executando.

Como as exceções em java são objetos, podem incluir atributos, métodos e construtores. A classe `java.lang.Throwable` inclui o registro do estado atual da pilha de chamadas de métodos e uma `String` associada a uma mensagem de erro, que pode ser obtida para qualquer exceção através do método `Throwable.getMessage()`.

Todas as exceções derivadas de `java.lang.RuntimeException`, não são obrigadas a serem tratadas, pois são condições excepcionais que acontecem por falha nos algoritmos que o programador desenvolveu durante a chamada daquele método. Estas são chamadas tecnicamente de “**Unchecked Exceptions**”.

Todas as exceções derivadas de `java.lang.Exception`, exceto as derivadas de `java.lang.RuntimeException`, são obrigadas a serem tratadas, pois são condições excepcionais esperadas e que podem acontecer durante a chamada daquele método. Estas são chamadas tecnicamente de “**Checked Exceptions**”

Todas as exceções derivadas de `java.lang.Error`, devem ser tratadas, entretanto, quando acontecerem interrompem a execução do aplicativo. Estas são chamadas tecnicamente de “**Errors**”

Capturando e tratando exceções.

A sinalização da exceção é propagada a partir do bloco de código onde ela ocorreu através de toda a pilha de invocações de métodos até que a exceção seja **capturada** por um bloco manipulador de exceção.

Eventualmente, se tal bloco não existir em nenhum ponto da pilha de invocações de métodos, a sinalização da exceção atinge o método `main()`, fazendo com que o interpretador Java apresente uma mensagem de erro e aborte sua execução.

Estrutura try-catch-finally.

A captura e o tratamento de exceções em Java se dá através da especificação de blocos `try`, `catch` e `finally`, definidos através destas mesmas palavras reservadas da linguagem. Um comando `try/catch/finally` obedece à seguinte sintaxe:

```
try {  
    // código que inclui comandos/invocações de métodos  
    // que podem gerar uma situação de exceção.  
  
} catch (Throwable t) {  
    // bloco de tratamento associado à condição de  
    // exceção derivada de Throwable ou a qualquer uma de suas  
    // subclasses, identificada aqui pelo objeto  
    // com referência t  
  
} finally {  
    // bloco de código que sempre será executado após  
    // o bloco try, independentemente de sua conclusão  
    // ter ocorrido normalmente ou ter sido interrompida  
    // este bloco é opcional  
}
```

Os blocos não podem ser separados por outros comandos -- um erro de sintaxe seria detectado pelo compilador Java neste caso. Cada bloco `try` pode ser seguido por zero ou mais blocos `catch`, onde cada bloco `catch` refere-se a uma única exceção.

O bloco `finally`, quando houver, é sempre executado independente se houve ou não uma exceção.

Em geral, ele inclui comandos que liberam recursos que eventualmente possam ter sido alocados durante o processamento do bloco try e que necessitam ser liberados, independentemente de a execução ter encerrado com sucesso ou ter sido interrompida por uma condição de exceção. A presença desse bloco é opcional.

Usando o bloco try-catch.

No programa abaixo faremos a leitura de um número via teclado, esta operação exige o uso de mecanismo de IO, que por sua vez podem gerar exceções que devem ser tratadas.

```
import java.io.*;

public class ConverteInteiro {

    public String leLinha() {
        byte[] lidos = new byte[20];
        System.in.read(lidos); // lê o buffer do teclado
        String texto = new String(lidos);
        return texto;
    }

    public int leInt() {
        String linha = leLinha();
        return Integer.parseInt(linha);
    }

    public static void main(String[] args) {
        ConverteInteiro ci = new ConverteInteiro ();
        System.out.print("Entre inteiro: ");
        int valor = ci.leInt();
        System.out.println("Valor lido foi: " + valor);
    }
}
```

Compilando o programa, vemos que o compilador informa que uma exceção esperada `java.io.IOException` que é uma “**Checked Exception**”, não foi devidamente tratada.

Lendo a documentação da classe `java.lang.System` da Java API, vemos que o método `read()` do atributo estático `in` que é do tipo `java.io.InputStream` foi definido assim:

Veja aqui a definição: [http://java.sun.com/j2se/1.4.1/docs/api/java/io/InputStream.html#read\(byte\[\]\)](http://java.sun.com/j2se/1.4.1/docs/api/java/io/InputStream.html#read(byte[]))

```
public int read(byte[] b) throws java.io.IOException
```

A keyword `throws`, indica que este método gera e arremessa a quem chamá-lo uma exceção do tipo `java.io.IOException` em uma determinada situação, e que se esta situação acontecer, o chamador deve estar apto a interpretá-la.

Corrigindo o programa, inserimos o bloco de tratamento de exceções para a exceção esperada que é gerada pelo método `System.in.read(byte[] b)` throws `IOException`.

```
import java.io.*;

public class ConverteInteiro2 {

    public String leLinha() {
        byte[] lidos = new byte[20];
        String texto = "";
        try {
            System.in.read(lidos); // le o buffer do teclado
            texto = new String(lidos);
            texto = texto.trim(); // retira os espacos em branco
        } catch (IOException ioException) {
            // se acontecer um erro na leitura
            // imprime na tela o erro
            ioException.printStackTrace();
        }
        return texto;
    }

    public int leInt() {
        String linha = leLinha();
        return Integer.parseInt(linha);
    }

    public static void main(String[] args) {
        ConverteInteiro2 ci = new ConverteInteiro2();
        System.out.print("Entre inteiro: ");
        int valor = ci.leInt();
        System.out.println("Valor lido foi: " + valor);
    }
}
```

Vemos que a compilação acontece com sucesso!.

Criando exceções customizadas.

Em muitos casos, para facilitar o mapeamento das falhas em nossas aplicações para mensagens amigáveis para o usuário criaremos exceções customizadas para essas funcionalidades.

Imaginando o problema de um banco. Quando manipulamos operações de débito e crédito, devemos seguir algumas regras. A seguir veremos dois códigos fontes, um que se utiliza do mecanismo de tratamento de exceções e outro que não.

Sem o mecanismo try-catch.

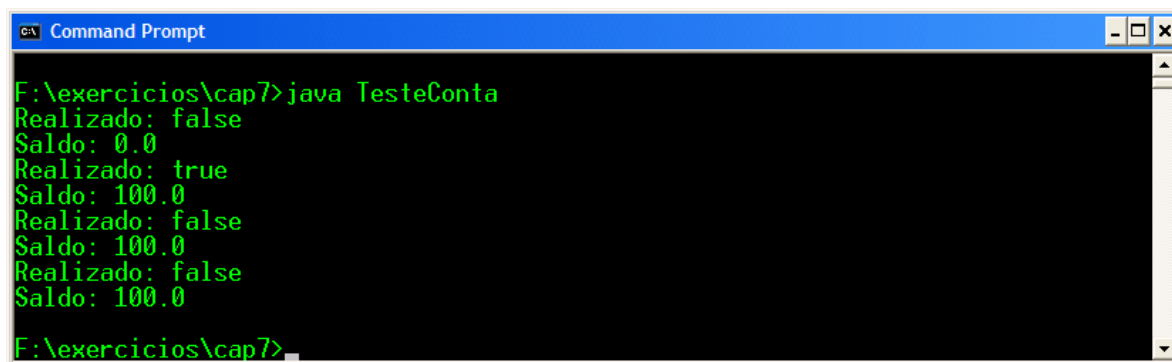
```
public class Conta {  
  
    private double saldo;  
  
    public boolean debito(double valor) {  
        if ( valor < 0 ) {  
            return false;  
        } else {  
            if ( saldo - valor < 0 ) {  
                return false;  
            } else {  
                saldo -= valor;  
                return true;  
            }  
        }  
    }  
  
    public boolean credito(double valor) {  
        if ( valor < 0 ) {  
            return false;  
        } else {  
            saldo += valor;  
            return true;  
        }  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}
```

```
public class TesteConta {

    public static void main(String args[]) {

        Conta c1 = new Conta();
        System.out.println("Realizado: " + c1.debito(100) );
        System.out.println("Saldo: " + c1.getSaldo() );
        System.out.println("Realizado: " + c1.credito(100) );
        System.out.println("Saldo: " + c1.getSaldo() );
        System.out.println("Realizado: " + c1.credito(-100) );
        System.out.println("Saldo: " + c1.getSaldo() );
        System.out.println("Realizado: " + c1.debito(-100) );
        System.out.println("Saldo: " + c1.getSaldo() );

    }
}
```



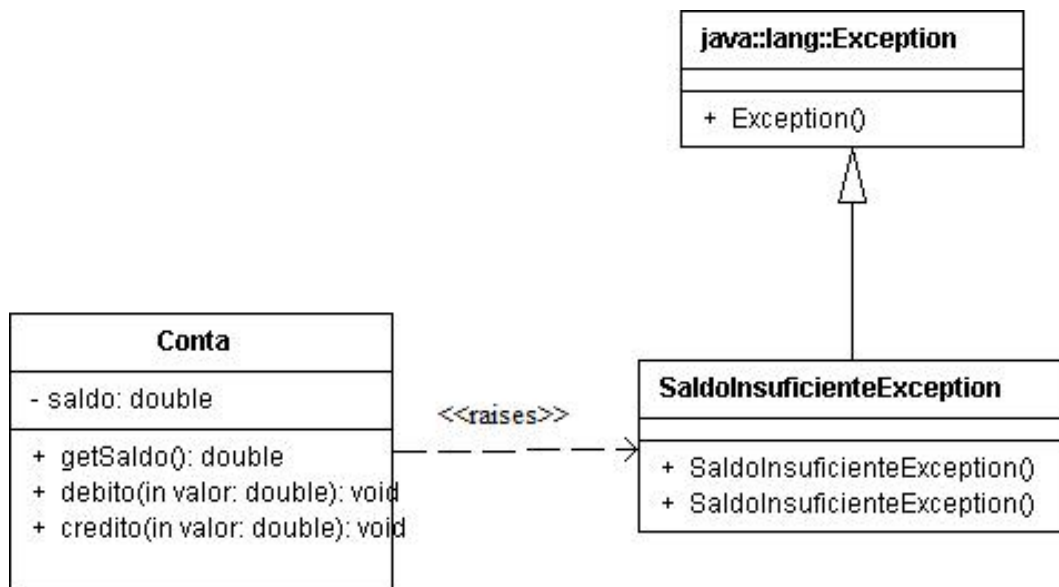
```

C:\> Command Prompt
F:\exercicios\cap7>java TesteConta
Realizado: false
Saldo: 0.0
Realizado: true
Saldo: 100.0
Realizado: false
Saldo: 100.0
Realizado: false
Saldo: 100.0
F:\exercicios\cap7>
  
```

Com o mecanismo try-catch.

Neste caso, criaremos uma nova classe exceção que represente saldo insuficiente: *SaldoInsuficienteException*, como vimos, todas as exceções que devem ser tratadas devem ser subclasses de *java.lang.Exception*.

Na UML, quando uma classe arremessa exceção vemos o seguinte desenho:



O nome de uma exceção deve representar por si só o seu significado e geralmente esta associado á uma ação ou estado do objeto.

```
public class SaldoInsuficienteException extends Exception {

    public SaldoInsuficienteException () {
        super("Falta de saldo para esta operação!");
    }

    public SaldoInsuficienteException (String mensagem) {
        super(mensagem);
    }

}
```

Alteraremos a classe Conta, para quando o saldo for insuficiente, seja gerada e arremessada uma exceção com uma mensagem amigável.

```
public class Conta2 {  
  
    private double saldo;  
  
    public void debito(double valor)  
        throws SaldoInsuficienteException {  
        if ( valor < 0 ) {  
            // unchecked exception não precisa ser declarada na clausula thorws  
            throw new IllegalArgumentException  
                ("Valor de debito deve ser maior que zero!");  
        } else {  
            if ( saldo - valor < 0 ) {  
                // checked exceptio, deve ser declara na clausula throws  
                throw new SaldoInsuficienteException  
                    ("Saldo insuficiente! Atual: "+saldo);  
            } else {  
                saldo -= valor;  
            }  
        }  
    }  
  
    public void credito(double valor) {  
        if ( valor < 0 ) {  
            // unchecked exception não precisa ser declarada na clausula thorws  
            throw new IllegalArgumentException  
                ("Valor de credito deve ser maior que zero!");  
        } else {  
            saldo += valor;  
        }  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
}
```

Alteraremos a classe TesteConta, para usar o mecanismo try-catch-finally e tratar a execucao criada quando ela acontecer.

```
public class TesteConta2 {  
    public static void main(String args[]) {  
        Conta2 c2 = new Conta2();  
        try {  
            c2.credito(100);  
            System.out.println("Saldo: "+ c2.getSaldo() );  
            c2.debito(50);  
            System.out.println("Saldo: "+ c2.getSaldo() );  
            c2.debito(500);  
            System.out.println("Saldo: "+ c2.getSaldo() );  
        } catch (SaldoInsuficienteException e) {  
            // se acontecer uma SaldoInsuficienteException  
            // caira neste bloco  
            e.printStackTrace();  
        } catch (Exception e) {  
            // se acontecer qualquer outra Exception  
            // caira neste bloco  
            e.printStackTrace();  
        } finally {  
            System.out.println("Saldo: "+ c2.getSaldo() );  
        }  
    }  
}
```

O uso do mecanismo de exceções para estes casos evita a codificação de estruturas de controle complexo na execução de chamadas de métodos consecutivos e dependentes, facilitando a construção e o entendimento do fluxo do programa.

Discussão.

Já que as exceções em Java são objetos, então temos classes para representá-las?

Como identificar que um método lança uma exceção seu chamador?

Devemos então sempre definir exceções customizadas para nossas aplicações?

Qual a principal diferença entre CHECKED e UNCHECKED Exceptions?

Exercícios

7. Desenvolvimento de Aplicações em Java

Vimos nos capítulos anteriores como programar em Java, agora vamos desenvolver o estudo sobre os principais componentes usados na construção de aplicações stand-alone em Java, usando a Java2 Standard Edition.

Assim como em qualquer outra linguagem de programação, as funcionalidades principais de uma linguagem inclui:

- manipulação de textos;
- manipulação de estruturas de dados;
- parâmetros em linha de comando para inicializar aplicativos;
- manipulação de arquivos de texto;
- manipulação de arquivos randômicos;
- manipulação de arquivos de configuração;
- formatação de dados;

Essas funcionalidades vai nos permitir implementar uma boa parte das funcionalidades de qualquer tipo de sistema ou aplicação.

As aplicações stand-alone podem ser construídas como pequenos utilitários, sem interface gráfica para o usuário usando parâmetros em linhas de comando, ou serem sistemas complexos com interfaces de usuários bem simples, ou interfaces de usuários complexas.

A construção de interfaces gráficas para os usuários será vista na parte IV do livro, onde abordaremos os conceitos de usabilidade, desenho, construção e manipulação de eventos.

Em Java, seguindo os princípios da Orientação a Objetos, usaremos componentes já pré-escritos, testados, estáveis e distribuídos com a JRE, nos permitindo usufruir dessas funcionalidades dentro dos nossos aplicativos. Esses componentes fazer parte da J2SE API.

Os principais pacotes que usaremos para escrever aplicações stand-alone utilitárias sem interfaces gráficas são:

- `java.lang` – fornece as classes fundamentais de linguagem java;
- `java.io` – fornece classes para manipulação de arquivos;
- `java.util` – fornece as estruturas de dados e classes utilitárias;
- `java.text` – fornece as classes para formatação de dados;
- `java.math` – fornece as classes para manipular números grandes;

Parâmetros em linha de comando.

Usamos a passagem de parâmetros em linha de comando para permitir a construção de utilitários que podem ser configurados durante a sua execução, fazendo com que tenhamos aplicativos flexíveis que podem ser amplamente utilizados.

Esta funcionalidade é composta por dois elementos, a passagem dos parâmetros na linha de comando de execução do interpretador java, e a codificação do tratamento de tais parâmetros dentro do aplicativo.

Passando parâmetros para o aplicativo:

Os parâmetros são passados separados por espaços, e se quisermos passar um parâmetro único que contenha espaços ele deve estar entre aspas.

Formato: `java ClasseMain parametro1 parametro2 "parametro 3"`

Dentro da aplicação stand-alone a JRE chama o método `main(String args[])` que é responsável por receber os parâmetros e aloca-los dentro de um array de String.

Cada parâmetro é colocado neste array, ordenado na forma como foi passado na linha do interpretador java.

```
args[0] = "parametro1";  
args[1] = "parametro2";  
args[2] = "parametro 3";
```

Exemplo:

```
public class TesteParametros {  
  
    public static void main(String args[]) {  
        int tamanho = args.length;  
        System.out.println("Numero de Parametros: "+tamanho);  
        for(int i=0; i < tamanho; i++) {  
            System.out.println("Param: " + i + " Valor: "+ args[i] );  
        }  
    }  
}
```

Executando a classe acima com:

```
java TesteParametros parametro1 parametro2 "parametro 3"
```

Veremos uma saída com dados dos parâmetros e sua quantidade.

Manipulando textos e cadeias de caracteres.

Grande parte das funcionalidades de uma aplicação, utilitário ou componentes, envolve a manipulação de textos e conjuntos de caracteres.

Para manipular tais conjuntos de caracteres utilizamos basicamente três classes: `java.lang.String`, `java.lang.StringBuffer` e `java.lang.StringTokenizer`.

A classe `java.lang.String` tem muitas funcionalidades para a manipulação de textos, mas as principais são: comparação de caracteres, contagem de caracteres, conversão para maiúsculas e minúsculas, pesquisa de caracteres, eliminação de caracteres, inserção de caracteres e tratamento de conjunto de caracteres.

Para fornecer tais funcionalidades a classe `java.lang.String` fornece um grande número de métodos. Os principais são:

```
public boolean equals(Object) - compara se a String tem o mesmo conteúdo;
```

```
public int length() - retorna o tamanho da String;
```

```
public int indexOf( char c) - retorna a posição onde o caractere se encontra;
```

```
public char charAt( int index ) - retorna o caractere da posição;
```

```
public String[] split( String cadeia ) - retorna um array de String proveniente da divisão da String;
```

```
public String trim() - retorna uma nova String sem os espaços em branco do início e fim da cadeia;
```

```
public String toLowerCase() - converte os caracteres da String para minúsculas;
```

```
public String toUpperCase() - converte os caracteres da String para maiúsculas;
```

```
public String substring(int inicio, int fim) - retorna uma nova String contendo os caracteres do intervalo ( fim - inicio ).
```

```
public String replace(char antigo, char novo) - retorna uma nova String trocando os caracteres que forem iguais ao antigo pelo novo.
```

Existem muitos outros métodos na classe `java.lang.String` que não serão listados aqui, por isso sugiro o estudo via a documentação da Java API.

Exemplo de uso:

```
public class TesteString {  
  
    public static void main(String args[]) {  
        String texto = "IBTA Java!";  
        int tamanho = texto.length();  
        System.out.println("Texto: "+texto);  
        System.out.println("Tamanho: "+tamanho);  
        String texto2 = texto.toUpperCase();  
        System.out.println("Texto2: "+texto2);  
        String texto3 = texto.toLowerCase();  
        System.out.println("Texto3: "+texto3);  
        String texto4 = texto3.replace( 'u', 'U' );  
        texto4 = texto4.replace( 'j', 'J' );  
        System.out.println("Texto4: "+texto4);  
        if ( texto3.equals( texto ) ) {  
            System.out.println(" texto3 e texto são iguais! ");  
        } else {  
            System.out.println(" texto3 e texto são diferentes! ");  
        }  
  
        if ( texto4.equals( texto ) ){  
            System.out.println(" texto4 e texto são iguais! ");  
        } else {  
            System.out.println(" texto4 e texto são diferentes! ");  
        }  
    }  
}
```

Usamos a classe `java.lang.StringBuffer`, para trabalhar com buffers de Strings e poder fazer operações de pesquisa, substituição, inserção e exclusão de Strings dentro de um texto.

`public StringBuffer append(String string)` - insere a string no fim da do buffer;

`public StringBuffer insert(int index, String string)` - insere a string na posição informada no index;

`public int indexOf(String string)` - retorna a posição onde a string se encontra;

`public StringBuffer reverse()` - inverte todo o conteúdo do buffer;

`public StringBuffer delete(int inicio, int fim)` - exclui do buffer as string nas posições do intervalo (fim - inicio);

Existem muitos outro métodos na classe `java.lang.StringBuffer` que não serão listados aqui, por isso sugiro o estudo via a documentação da Java API.

```
import java.util.StringTokenizer;

public class TesteStringBuffer {

    public static void main(String args[]) {
        String texto = "IBTA Java! Java é aqui!";
        StringTokenizer tokenizer = new StringTokenizer( texto, "!");
        int numero = tokenizer.countTokens();
        String[] tokens = new String[ numero ];
        int count = 0;
        while ( tokenizer.hasMoreTokens() ) {
            tokens[count] = tokenizer.nextToken();
            count++;
        }
        System.out.println("Texto: "+texto);
        System.out.println("# Tokens: "+numero);
        System.out.print("Tokens: ");
        for ( int i = 0 ; i<tokens.length; i++) {
            System.out.print( tokens[i] +",");
        }
        System.out.println();
        StringBuffer buffer = new StringBuffer( texto );
        StringBuffer invertido = buffer.reverse();
        System.out.println("Texto: "+texto);
        System.out.println("Invertido: "+invertido );
    }
}
```

Exercícios

Entrada e saída de dados.

A entrada e saída de dados de um aplicativo ou sistema pode ser feita de várias formas. A entrada pode ser via teclado, arquivo ou interface gráfica, e saída pode ser para console de texto, arquivos ou componente gráfico. Como dito anteriormente, veremos a construção de interfaces gráficas mais adiante.

Para podermos fazer bom uso das classes que permitem a execução de operações de entrada e saída de dados devemos primeiramente entender como a Tecnologia Java trata essas operações de IO.

Todas as operações de IO da linguagem Java são feitas usando as classes do pacote `java.io`, esse pacote contém classes de ***Streamers, Readers e Writers***.

Os ***Streamers*** são separados por categorias de acordo com sua características, ou seja, existem Streamers que são usados para leitura de bytes de um recurso, gravação de bytes para um recurso, leitura de objetos de um recurso e gravação de objetos para um recurso. Não se pode usar o mesmo Streamer para leitura e escrita. Os Streamers existem desde a JDK 1.0.

Os ***Readers e Writers*** são mecanismos evoluídos dos Streamers. Os Readers permitem a leitura de caracteres de um recursos e os Writers a escrita de caracteres para um recurso. Quando usamos um Reader ou Writer, trabalhamos com `char` e não com `byte`. Os Readers e Writers foram incorporado na JDK 1.1

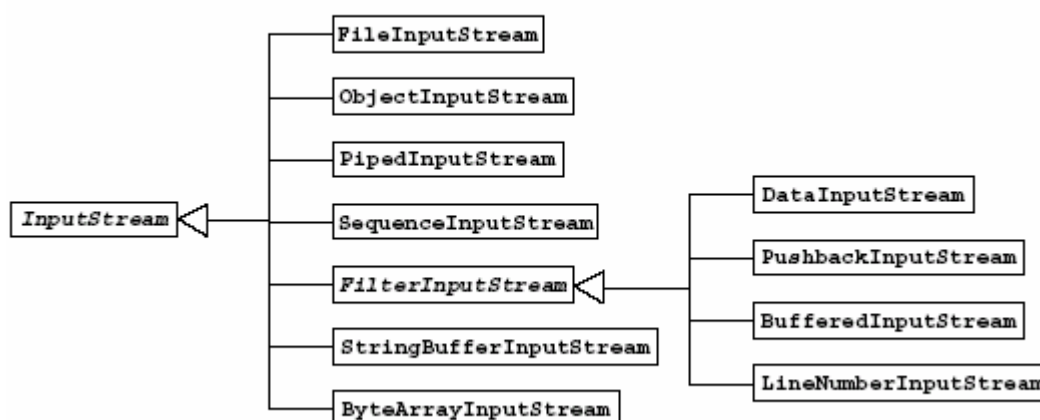
Em 90% dos aplicativos Java, usaremos Readers para leitura ou Writers para a escrita, em 10 % dos casos usaremos Streamers.

Java Streamers.

Como em Java existem Streamers para cada tipo de recurso, os Streamers podem ser do tipo **InputStream** (entrada) ou **OutputStream** (saída), veremos que todos os Streamers são subclasses da `java.io.InputStream` ou da `java.io.OutputStream` e manipulam bytes.

Quando optamos usar um Streamer, estaremos manipulando o tipo mais básico de informação, o byte, que todas as plataformas e tecnologias suportam.

Hierarquia de classes do **InputStream**:

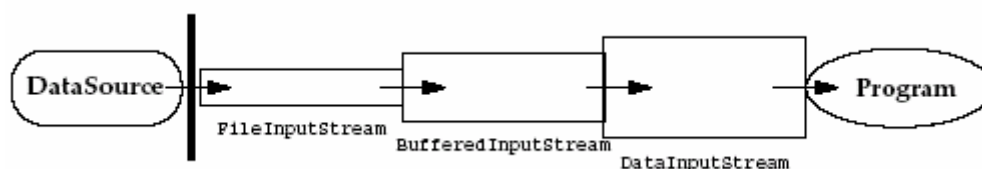


Usando os InputStreams:

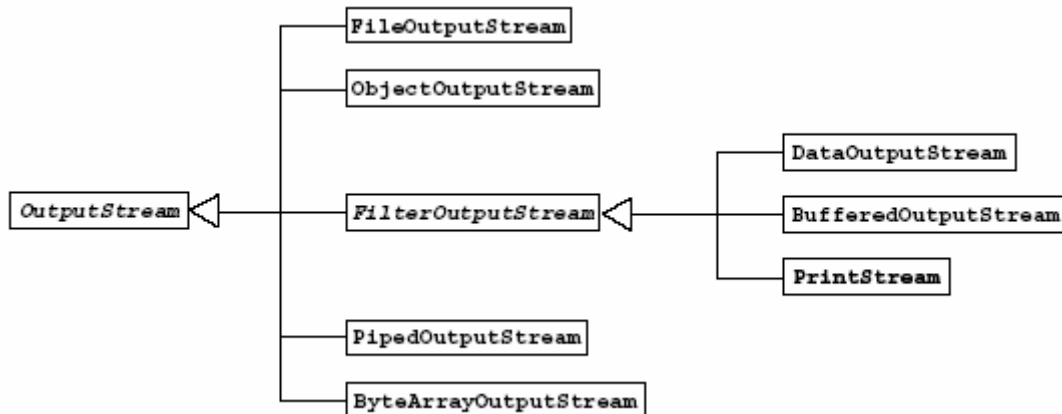
Toda vez que formos providenciar a entrada de dados de um recurso (**DataSource**) para uma aplicação (**Program**), usaremos ao menos dois Streams, isto porque os InputStreams lêem byte a byte, por isso algumas subclasses lêem linhas de bytes tornando a manipulação dessas informações mais fácil.

No caso abaixo, vemos uma cadeia de Streamers para a leitura buferizada de bytes (`DataInputStream` ligado ao `BufferedInputStream`) a partir de um arquivo (`FileInputStream`).

Nem sempre, usaremos a leitura Buferizada. Buferizar melhora a performance, mas consome mais memória.



Hierarquia de classes do *OutputStream*:

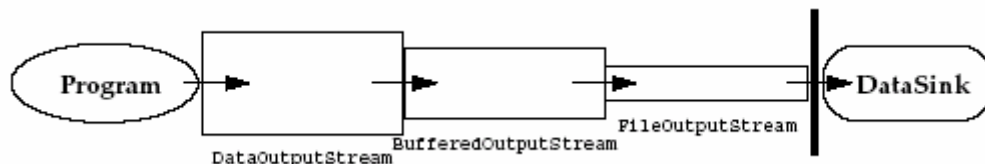


Usando os OutputStreams:

Toda vez que formos providenciar a saída de dados para um recurso (**DataSink**) de uma aplicação (**Program**), usaremos ao menos dois Streams, isto porque os InputStreams gravam byte a byte, por isso algumas subclasses gravam linhas de bytes tornando a manipulação dessas informações mais fácil.

No caso abaixo, vemos uma cadeia de Streamers para a gravação buferizada de bytes (DataOutputStream ligado ao BufferedOutputStream) para um arquivo (FileOutputStream).

Nem sempre, usaremos a gravação Buferizada. Buferizar melhora a performance, mas consome mais memória.



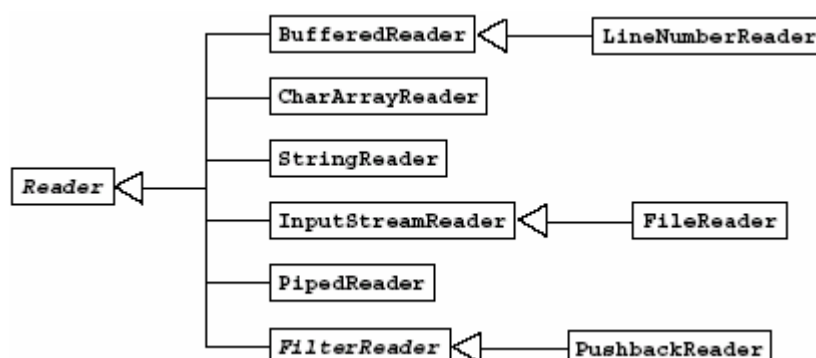
Quando os analista modelam uma aplicação, eles especificam a necessidade de buferização, e no caso disso acontecer, basta usarmos entre o recurso e a aplicação um dos Buffers diponíveis na J2SE API.

Java Readers e Writers.

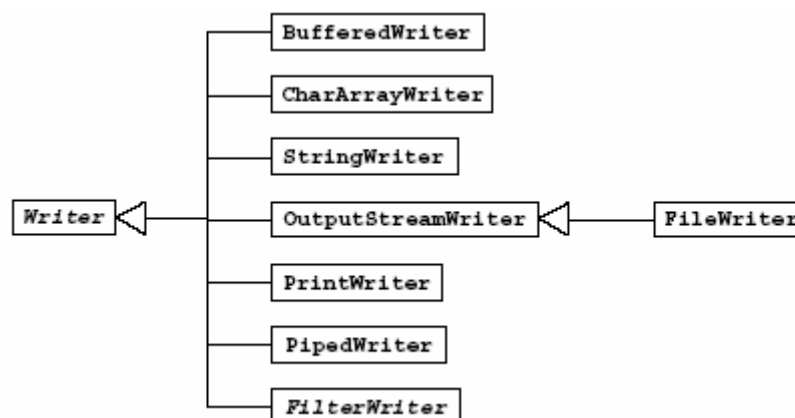
Para facilitar a construção de aplicativos, na JDK 1.1 foram inseridos os **Readers** e **Writers**, criados para facilitar a manipulação de caracteres e evitar o desconforto de manipular bytes. Como o próprio nome diz, os Readers servem para a leitura e os Writers para escrita.

Os Readers são subclasses da `java.io.Reader` e os Writers da `java.io.Writer`.

Hierarquia de classes dos **Readers**:



Hierarquia de classes dos **Writers**:



Categorias e Uso.

Para facilitar o entendimento do uso dos Streamers, Reader e Writere, foram compiladas algumas tabelas, que servirão como referência deste assunto:

Streamers que representam os recursos:

Tipo	Char Streamers	Byte Streamers
Arquivo	FileReader FileWriter	FileInputStream FileOutputStream
Memória: Array	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
Memória: String	StringReader StringWriter	
Pipe: IPC	PipedReader PipedWriter	PipedInputStream PipedOutputStream

IPC – Inter Process Communication – Troca de dados entre processos de execução de uma aplicação, muito utilizado em aplicações para servidores.

Streamers de processamento de dados:

Tipo	Char Streamers	Byte Streamers
Buferização	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtros	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Conversão de byte para char	InputStreamReader OuputStreamWriter	
Serialização		ObjectInputStream ObjectOutputStream
Conversão de dados		DataInputStream DataOutputStream
Contagem	LineNumberReader	LineNumberInputStream
Pico	PushbackReader	PushbackInputStream
Impressão	PrintWriter	PrintStream

Exemplo Reader e Writer:

```
import java.io.*;

public class Copy {
    public static void main(String[] args) {
        try {
            FileReader input = new FileReader(args[0]);
            FileWriter output = new FileWriter(args[1]);
            char[] buffer = new char[128];
            int charsRead = 0;
            charsRead = input.read(buffer);
            while ( charsRead != -1 ) {
                output.write(buffer, 0, charsRead);
                charsRead = input.read(buffer);
            }
            input.close();
            output.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exemplo para ler dados usando o teclado:

O objeto `java.io.InputStream` representado pela referência `System.in` é a entrada padrão de dados via console (teclado) para todos os ambientes operacionais.

Para poder ler dados usamos um Stream de Caracter ligado a ele, e para ler linha a linha usamos o `BufferedReader` como ponte.

```
import java.io.*;

public class LeTeclado {

    public static void main (String args[]) {
        String s = null;
        InputStreamReader ir = new InputStreamReader(System.in);
        BufferedReader in = new BufferedReader(ir);
        System.out.println("Tecle Ctrl-Z para sair!");
        try {
            s = in.readLine();
            while ( s != null ) {
                System.out.println("Lido: " + s);
                s = in.readLine();
            }
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Exemplo para gravar dados num arquivo lendo via teclado:

```
import java.io.*;

public class GravaArquivo {

    public static void main (String[] args) {
        File file = new File(args[0]);
        try {
            BufferedReader in = new BufferedReader( new InputStreamReader(
System.in ));
            PrintWriter out = new PrintWriter(new FileWriter(file));
            String texto = null;
            System.out.print("Digite o texto! ");
            System.out.println("Ctrl+Z para gravar!");
            while ( (texto = in.readLine() ) != null ) {
                out.println(texto );
            }
            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Arquivos de acesso randômico.

Como vimos, os Streams, Readers e Writers, servem para manipular a entrada e saída de dados, de um recurso para a aplicação. E geralmente lidam com conteúdos fixos de dados, arquivos inteiros, que podem estar buferizados ou não.

Entretanto, em muitas aplicações, as vezes se faz necessário controlar arquivos de dados dinâmicos que armazenarão informações podem ser alteradas sem a necessidade de reescrever o arquivo todo, garantindo uma melhor performance.

O arquivo de acesso randômico permite que se faça leitura ou gravação de um registro no arquivo, num modelo assíncrono. Ou seja, a aplicação pode ler ou gravar registros em qualquer posição do arquivo.

Para manipularmos os registros de um arquivo de acesso randômico, usamos a classe `java.io.RandomAccessFile` que está presente desde a JDK 1.0.

Esse tipo de arquivo, geralmente é usado como base de dados para as aplicações de pequeno porte. Sendo assim, é interessante que antes de se construir um programa que use um arquivo de acesso randômico que definamos os modelo de dados (registro) que será armazenado no arquivo.

Exemplo DataFile.java:

```
public class DataFile {

    public static final String LEITURA_ESCRITA = "rw";

    private RandomAccessFile arquivoDados;
    private File arquivoFisico;
    private String nomeArquivo;

    public DataFile(String nomeArquivo) {
        this.nomeArquivo = nomeArquivo;
    }

    public void inicializa() throws IOException {
        arquivoFisico = new File( nomeArquivo );
    }

    public void abre() throws IOException {
        abre( LEITURA );
    }

    public void abre(String modo) throws IOException {
        System.out.print("Abrindo arquivo dados: " + nomeArquivo);
        arquivoDados = new RandomAccessFile ( arquivoFisico, modo );
        arquivoDados.seek( 0 );
        System.out.println(" ... OK!");
    }

    public void fecha() throws IOException {
        System.out.print("Fechando arquivo dados: " + nomeArquivo);
        arquivoDados.close();
        System.out.println(" ... OK!");
    }

    ...
}
```

```

    public long insere(String registro) throws IOException {
        System.out.println("Inserindo registro! ");
        if ( registro == null || registro.length() > 100 ) {
            throw new IOException("Não é possível inserir registro nulo ou com tamanho
maior que 100 caracteres!");
        } else {
            registro.trim();
            int pad = registro.length();
            for ( int i=0; i < ( 100 - pad ); i++) {
                registro += " ";
            }
            long tamanho = arquivoDados.length();
            arquivoDados.seek( tamanho );
            arquivoDados.writeUTF( registro );
            tamanho = arquivoDados.length();
            long posicao = tamanho / 100;
            System.out.println(" ..OK! ");
            return posicao ;
        }
    }

    public String recupera(long numeroRegistro) throws IOException {
        System.out.print("Lendo registro! ");
        long posicao = numeroRegistro * 102;
        arquivoDados.seek( posicao );
        String registro = arquivoDados.readUTF();
        System.out.println(" ..OK! ");
        return registro;
    }
}

```

Exemplo TesteRandomico.java:

```

import java.io.*;

public class TesteRandomico {

    public static void main(String args[]) throws Exception {
        if ( args.length == 1 && args[0] != null ) {
            DataFile dataFile = new DataFile( args[0] );
            dataFile.inicializa();
            dataFile.abre( DataFile.LEITURA_ESCRITA );
            //long posicao = dataFile.insere("Univesidade Java");
            //System.out.println("Posicao: "+posicao);
            String registro = dataFile.recupera( 3 );
            System.out.println("Registro: "+ registro );
            dataFile.fecha();
        } else {
            System.out.println("uso: java TesteRandomico nomeArquivoDados.dat" );
        }
    }
}

```


Trabalhando com propriedades.

Dentro da tecnologia Java, as propriedades (properties) são usadas para configurar aplicações. Geralmente as aplicações tem configurações de diretório de execução, diretório temporário, nome do usuário corrente, diretório corrente do usuário, diretório de classes da JRE, etc.

Para trabalhar com as propriedades da JRE, usamos a classe `java.lang.System` que fornece métodos para manipularmos tais propriedades que são representadas pela classe `java.util.Properties`.

Por exemplo, se quisermos listar as propriedades do sistema corrente, podemos usar o seguinte código:

```
import java.util.*;

public class TestePropriedades {

    public static void main(String args[]) {
        Properties props = System.getProperties();
        Enumeration prop_names = props.propertyNames();
        while ( prop_names.hasMoreElements() ) {
            String prop_name = (String) prop_names.nextElement();
            String property = props.getProperty(prop_name);
            System.out.println("property '" + prop_name + "' is '" +
property + "'");
        }
    }
}
```

Trabalhando com arquivos de propriedades.

Uma outra forma de se manipular propriedades é criar um arquivo contendo as propriedades e carregá-lo em tempo de execução:

Arquivo nome.properties:

```
#tipo=papel
#comentários
usuario=user
administrador=admin
convidado=guest
```

Código:

```
// class CarregaPropriedades.java
import java.io.*;
import java.util.*;

public class CarregaPropriedades {
    private Properties props;

    public CarregaPropriedades(String nomeArquivo) {
        try {
            FileInputStream fis = new FileInputStream( nomeArquivo );
            props = new Properties();
            props.load( fis );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public String getProperty(String nome) {
        return (String) props.get( nome );
    }
}

// class TesteCarregaPropriedades.java
public class TesteCarregaPropriedades {
    public static void main(String args[]) {
        CarregaPropriedades carregador = null;
        if ( args.length == 1 ) {
            carregador = new CarregaPropriedades ( args[0] );
            System.out.println("usuario="+ carregador.getProperty("usuario") );
            System.out.println("convidado="+ carregador.getProperty("convidado") );
            System.out.println("administrador="+ carregador.getProperty("administrador") );
        } else {
            System.out.println("Uso: java TesteCarregaPropriedades arquivo.properties ");
        }
    }
}
```

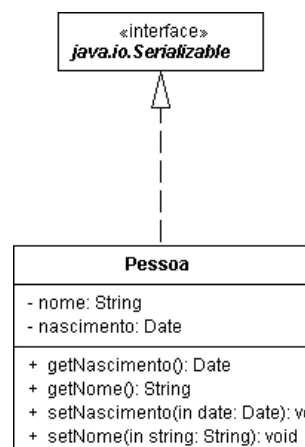
Serialização de objetos.

A tecnologia Java possui uma forma de armazenar em arquivos de dados ou transmitir via rede os objetos no estado exatamente como eles estavam na memória, preservando seu conteúdo e estado, esse mecanismo chama-se serialização.

O processo de gravação de um objeto é a serialização, e o de leitura deserialização. Durante estes processos somente poderão ser serializados ou deserializados atributos de instância. Atributos de classe (estáticos) não são serializados ou deserializados.

Quando o objeto que será serializado for proveniente de uma subclasse, todos os atributos de instância, mesmos os provenientes das superclasses serão serializados podendo ser deserializados.

Dessa forma usamos estes processos para armazenar ou transmitir os objetos garantindo todo seu conteúdo e estado.



Para que um objeto possa ser serializado é necessário que a classe que o originou ou que uma superclasse da sua hierarquia implemente a interface `java.io.Serializable`, que é uma interface de marcação. Este interface marca os objetos permitindo que a JRE possa serializa-los.

O objeto a ser serializado levará um atributo a mais quando for serializado, esse atributo é definido como: `static final long serialVersionUID = valor;`

Quando o objeto for deserializado, para garantir as versões de classes que geraram esse objeto, esse valores são recomputados e verificados, se não forem identicos será gerada uma `java.lang.ClassCastException`.

A J2SDK traz uma ferramenta que nos lista o valor do `serialVersionUID` da classe que desejamos saber se o objetos provenientes dela podem ser serializados, esta ferramenta é o `serialver`.

Uso: `serialver nomecompletodaclasse`

```
Command Prompt
c:\>serialver java.lang.String
java.lang.String: static final long serialVersionUID = -6849794470754667710L;
c:\>
```

The screenshot shows a Windows Command Prompt window. The user has entered the command `serialver java.lang.String`. The output of the command is `java.lang.String: static final long serialVersionUID = -6849794470754667710L;`. The prompt is currently at `c:\>`.

Para serializar e deserializar objetos usamos um conjunto de Streamers especiais criados para este propósito e existentes desde a JDK 1.0, `java.io.ObjectOutputStream` e `java.io.ObjectInputStream`.

Faremos agora um exemplo de Serialização e Deserialização de um objeto a partir da classe Pessoa desenhada acima.

```
// Pessoa.java
public class Pessoa implements java.io.Serializable {
    private String nome;
    private java.util.Date nascimento;

    public java.util.Date getNascimento() {
        return nascimento;
    }

    public String getNome() {
        return nome;
    }

    public void setNascimento(java.util.Date nascimento) {
        this.nascimento = nascimento;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
}

// SerializadorPessoa.java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializadorPessoa {

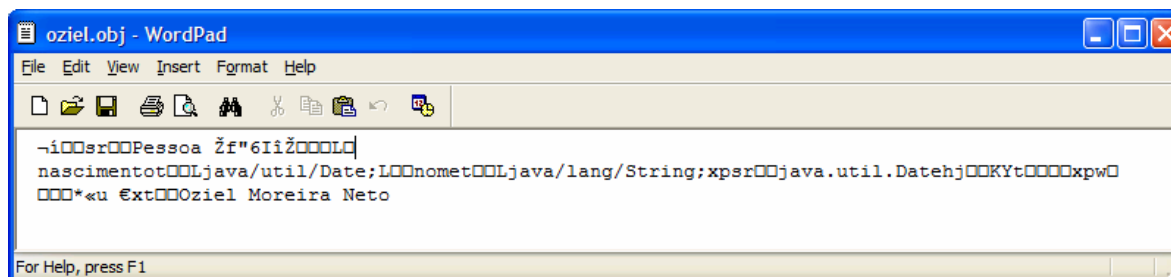
    public void gravaPessoa(String nomeArquivo, Pessoa pessoa)
        throws IOException {
        if ( pessoa != null && nomeArquivo != null ) {
            FileOutputStream fos = new FileOutputStream( nomeArquivo );
            ObjectOutputStream oos = new ObjectOutputStream ( fos );
            oos.writeObject( pessoa );
            oos.close();
        }
    }

    public Pessoa recuperaPessoa(String nomeArquivo)
        throws IOException, ClassNotFoundException {
        Pessoa pessoa = null;
        if ( nomeArquivo != null ){
            FileInputStream fis = new FileInputStream( nomeArquivo );
            ObjectInputStream ois = new ObjectInputStream( fis );
            pessoa = (Pessoa) ois.readObject();
            ois.close();
        }
        return pessoa;
    }
}
```

```
// classe TesteSerializadorPessoa.java
public class TesteSerializadorPessoa {

    public static void main(String args[]) {
        Pessoa pessoa = new Pessoa();
        pessoa.setNome("Oziel Moreira Neto");
        pessoa.setNascimento( new java.util.Date(75,9,23) );
        try {
            SerializadorPessoa serializador = new SerializadorPessoa();
            serializador.gravaPessoa("oziel.obj", pessoa);
            pessoa = null;
            pessoa = serializador.recuperaPessoa("oziel.obj");
            System.out.println( pessoa.getNome() );
            System.out.println( pessoa.getNascimento() );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Dados serializados:



Trabalhando com coleções de objetos – Collections APIs.

Dentro de todaas as aplicações e sistemas, se faz se necessário trabalhar com objetos múltiplos. Podemos trabalhar com objetos colecionados em estruturas de listas, conjuntos, mapas e árvores.

Usamos uma coleção de objetos para:

- suportar relacionamentos múltiplos entre objetos;
- substituir o uso e manipulação de arrays;
- trabalhar com estruturas de dados em memória para ordenação;

Na tecnologia Java existem dois tipos de coleções de objetos:

Legacy – estruturas criadas na JDK 1.0, sincronizadas e sem capacidade de ordenação, definidas pelas classes: `java.util.Vector` (lista) e `java.util.Hashtable` (mapa);

Collection – criadas na J2SE 1.2, não sincronizadas, com capacidade de ordenação e introduziu as estruturas de conjuntos e árvore.

Atualmente evitamos usar em nossos programas as classes `Vector` e `Hashtable`, por serem sincronizadas, podendo degradar a performance, dando preferência as classes do framework das collections que implementam a interface **`java.util.Collection`**.

As **Java Collections**™ fornecem uma interface de métodos simples que manipulam qualquer objeto java, pois seus métodos manipulam a classe `java.lang.Object`.

As classes concretas da API de collection são divididas de acordo com sua categoria de superclasse, e quando uma subclasse implementa uma das interfaces abaixo temos:

- `List` – lista de objetos não ordenados, que permite duplicados;
- `Set` – conjunto de objetos não ordenados, que não permite duplicados;
- `Map` – estrutura em memória que armazena os objetos de acordo com uma chave;

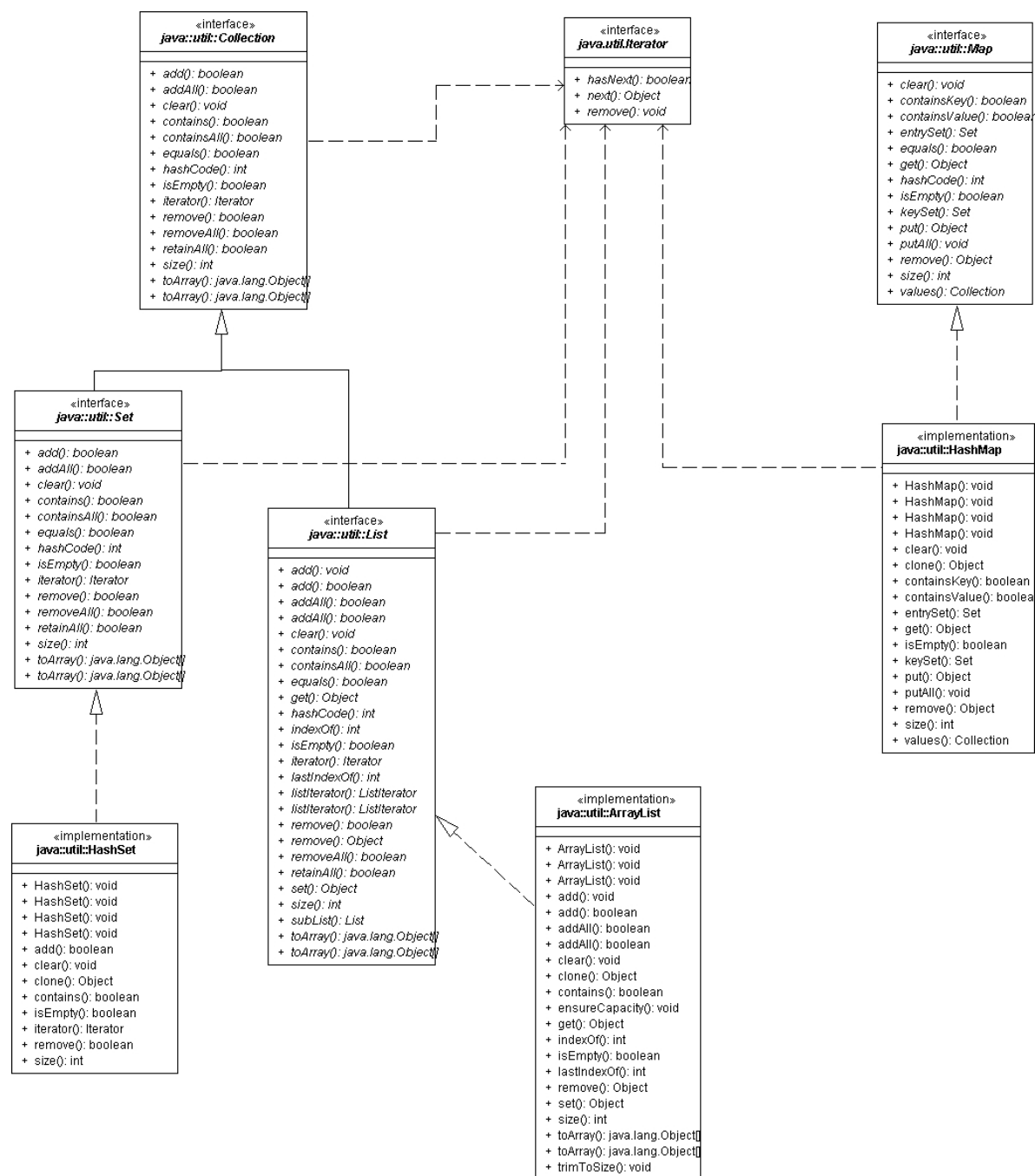
Como este framework é muito extenso, vamos estudar as três principais estruturas de manipulação de coleções de objetos, `java.util.ArrayList`, `java.util.HashMap` e `java.util.HashSet`.

Para o perfeito funcionamento das Collections, é imprescindível que o os nossos objetos reescrevam os seguintes métodos da `java.lang.Object`:

```
public boolean equals(Object obj);
```

```
public int hashCode();
```

Hierarquia do framework das Collections:



Alterando a classe Pessoa, reescrevendo o equals e o hashCode:

```
public class Pessoa implements java.io.Serializable {

    private String nome;
    private java.util.Date nascimento;

    public java.util.Date getNascimento() {
        return nascimento;
    }

    public String getNome() {
        return nome;
    }

    public void setNascimento(java.util.Date nascimento) {
        this.nascimento = nascimento;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public boolean equals(Object obj) {
        boolean flag = false;
        if ( obj instanceof Pessoa ) {
            Pessoa that = (Pessoa) obj;
            flag = that.nome.equals( this.nome ) && that.nascimento.equals(
this.nascimento );
        }
        return flag;
    }

    public int hashCode() {
        return ( nome != null && nascimento != null ) ? nome.hashCode() ^
nascimento.hahsCode() : 0;
    }
}
```

O método `public boolean equals(Object obj)` deve testar se o objeto passado (obj) tem mesmo conteúdo do objeto que esta sendo comparado.

Se o conteúdo dos dois objetos forem o mesmo, então devemos retornar `true`, caso contrário devemos retornar `false`.

Geralmente testamos atributos que servem como identidade do objeto, no caso da nossa classe Pessoa, testamos o nome e a data de nascimento, se eles forem iguais, então os objetos são iguais.

```
public boolean equals(Object obj) {
    boolean flag = false;
    if ( obj instanceof Pessoa ) {
        Pessoa that = (Pessoa) obj;
        flag=that.nome.equals(this.nome) && that.nascimento.equals(this.nascimento);
    }
    return flag;
}
```

O método `public int hashCode()` deve retornar um número inteiro, que é um valor único para aquele objeto. Geralmente fazemos um XOR dos atributos identidade.

```
public int hashCode() {
    return(nome!=null && nascimento!=null) ? nome.hashCode() ^ nascimento.hahsCode(): 0;
}
```


Manipulando objetos do tipo Pessoa dentro de um ArrayList:

```
import java.util.*;

public class ListaPessoas {
    private ArrayList pessoas;

    public ListaPessoas() {
        pessoas = new ArrayList( 10 );
    }

    public void adiciona(Pessoa pessoa) {
        // se pessoa nao for nulo e nas pessoas nao conter pessoa
        // para testar se pessoa nao esta nas pessoas,
        // sera usado o Pessoa.equals(obj).
        if ( pessoa != null && !pessoas.contains( pessoa ) ) {
            pessoas.add( pessoa );
            System.out.println("Objeto adicionado por na lista!" + pessoa);
        } else {
            System.out.println("Objeto nao adicionado por ja existir na lista ou por estar
nulo!");
        }
    }

    public Pessoa recupera(int index) {
        Pessoa pessoa = null;
        int tamanho = pessoas.size();
        if ( index < tamanho ) {
            pessoa = (Pessoa) pessoas.get( index );
        } else {
            System.out.println("Indice nao encontrado!");
        }
        return pessoa;
    }

    // como o arraylist nao possui metodos de busca, teremos de fazer
    // uma varredura completa nos elementos da lista.
    public Pessoa recupera(String nome) {
        Pessoa pessoa = null;
        // varrendo a lista pessoas do inicio ate o fim.
        Iterator it = pessoas.iterator();
        while( it.hasNext() ){
            Pessoa p = (Pessoa) it.next();
            // se o nome for igual a algum na lista
            if ( p.getNome().equals( nome ) ) {
                pessoa = p;
                break;
            }
        }
        return pessoa;
    }
}

public class TesteListaPessoas {

    public static void main(String args[]) {
        ListaPessoas listaPessoas = new ListaPessoas();
        Pessoa p1 = new Pessoa();
        p1.setNome("Oziel");
        p1.setNascimento( new java.util.Date(70,8,22) );
        listaPessoas.adiciona( p1 );
        Pessoa p2 = new Pessoa();
        p2.setNome("Jose");
        p2.setNascimento( new java.util.Date(85,4,24) );
        listaPessoas.adiciona( p2 );
        Pessoa p3 = listaPessoas.recupera("Oziel");
        System.out.println( p3 );
        Pessoa p4 = listaPessoas.recupera( 1 );
        System.out.println( p4 );
    }
}
```

Manipulando objetos do tipo Pessoa dentro de um HashMap:

```
// MapaPessoas.java
import java.util.*;

public class MapaPessoas {
    private HashMap pessoas;

    public MapaPessoas() {
        pessoas = new HashMap();
    }

    public void adiciona(Pessoa pessoa) {
        if ( pessoa != null ) {
            // cria um objeto Integer para a chave da pessoa
            String chave = pessoa.getNome();
            // se no mapa das pessoas nao contiver a chave
            if ( ! pessoas.containsKey( chave ) ) {
                pessoas.put( chave, pessoa );
            }
        }
    }

    public Pessoa recupera(String nome) {
        Pessoa pessoa = null;
        if ( nome != null ) {
            if ( pessoas.containsKey( nome ) ) {
                pessoa = (Pessoa) pessoas.get( nome );
            }
        }
        return pessoa;
    }
}

// class TesteMapaPessoas.java
public class TesteMapaPessoas {

    public static void main(String args[]) {
        MapaPessoas mapaPessoas = new MapaPessoas ();
        Pessoa p1 = new Pessoa();
        p1.setNome("Oziel");
        p1.setNascimento( new java.util.Date(70,8,22) );
        mapaPessoas.adiciona( p1 );
        Pessoa p2 = new Pessoa();
        p2.setNome("Jose");
        p2.setNascimento( new java.util.Date(85,4,24) );
        mapaPessoas.adiciona( p2 );

        Pessoa p3 = mapaPessoas.recupera("Oziel");
        System.out.println( p3 );
        Pessoa p4 = mapaPessoas.recupera("Jose");
        System.out.println( p4 );
    }
}
```

Manipulando objetos do tipo Pessoa dentro de um HashSet:

```
//class ConjuntoPessoas.java
import java.util.*;

public class ConjuntoPessoas {
    private HashSet pessoas;

    public ConjuntoPessoas () {
        pessoas = new HashSet();
    }

    public void adiciona(Pessoa pessoa) {
        if ( pessoa != null ) {
            // se no conjunto das pessoas nao contiver a pessoa
            //testando pelo hashCode
            if ( ! pessoas.contains( pessoa ) ) {
                pessoas.add( pessoa );
            }
        }
    }

    // como o set nao possui metodos de busca, ou pesquise por chave
    // teremos de fazer uma varredura completa nos elementos do conjunto
    public Pessoa recupera(String nome) {
        Pessoa pessoa = null;
        Iterator it = pessoas.iterator();
        while( it.hasNext() ){
            if ( p.getNome().equals( nome ) ) {
                pessoa = p;
                break;
            }
        }
        return pessoa;
    }
}

// class TesteConjuntoPessoas.java
public class TesteConjuntoPessoas {

    public static void main(String args[]) {
        ConjuntoPessoas conjuntoPessoas = new ConjuntoPessoas ();
        Pessoa p1 = new Pessoa();
        p1.setNome("Oziel");
        p1.setNascimento( new java.util.Date(70,8,22) );
        conjuntoPessoas.adiciona( p1 );
        Pessoa p2 = new Pessoa();
        p2.setNome("Jose");
        p2.setNascimento( new java.util.Date(85,4,24) );
        conjuntoPessoas.adiciona( p2 );
        Pessoa p3 = conjuntoPessoas.recupera("Oziel");
        System.out.println( p3 );
        Pessoa p4 = conjuntoPessoas.recupera("Jose");
        System.out.println( p4 );
    }
}
```

Wrapper Classes

Como em java, os tipos primitivos não são objetos, e os objetos são somente do tipo referência, existem um conjunto de classes dentro do pacote `java.lang` que nos permite converter os conteúdos dos tipos primitivos em tipos referência, e vice-versa. Essas classes são chamadas de Wrapper Classes.

Para cada tipo primitivo, existem um tipo referência que nos permite convertê-los em `String`, ou outro tipo primitivo.

Primitivo	Referência
<code>byte</code>	<code>java.lang.Byte</code>
<code>short</code>	<code>java.lang.Short</code>
<code>char</code>	<code>java.lang.Character</code>
<code>int</code>	<code>java.lang.Integer</code>
<code>long</code>	<code>java.lang.Long</code>
<code>float</code>	<code>java.lang.Float</code>
<code>double</code>	<code>java.lang.Double</code>

Em algumas APIs, existem métodos que manipulam as Wrapper classes para formatação e conversão. Podemos usar em nossas aplicações tais conversões.

Exemplo:

```
// convertendo um primitivo para String
int numero = 10;
Integer wrapperNumero = new Integer( numero );
String strNumero = wrapperNumero.toString();

// convertendo uma String para primitivo
String strNumero = "10";
Integer wrapperNumero = new Integer( strNumero );
int numero = wrapperNumero.intValue();
```

Todas classes das **wrapper classes**, possuem pelo menos um construtor que recebe uma `String` e um método que retorna seu valor do tipo primitivo respectivo de acordo com a tabela acima:

```
// convertendo uma String para primitivo
String strInt = "10";
int inteiro = Integer.parseInt( strInt );
double duplo = Double.parseDouble( strInt );
```

Se ocorrer uma falha na conversão, pois a `String` não deve conter caracteres, será gerada uma `java.lang.IllegalArgumentException`.

Veremos nos próximos capítulos usos mais reais das Wrapper Classes, pois sua funcionalidade é bem específica.

Discussão

Vimos nesse capítulo uma série de conceitos que são usados em aplicações de fato, ou seja, arquivos de configurações (properties), coleções de objetos, manipulação de arquivos texto e de arquivos de dados (Randômico).

Necessariamente todas as aplicações usam esses recursos?

Quais dos recursos nesse capítulo são mais fáceis de se implementar?

Podemos usar a serialização de objetos para armazenar nossos objetos temporariamente e indefinidamente?

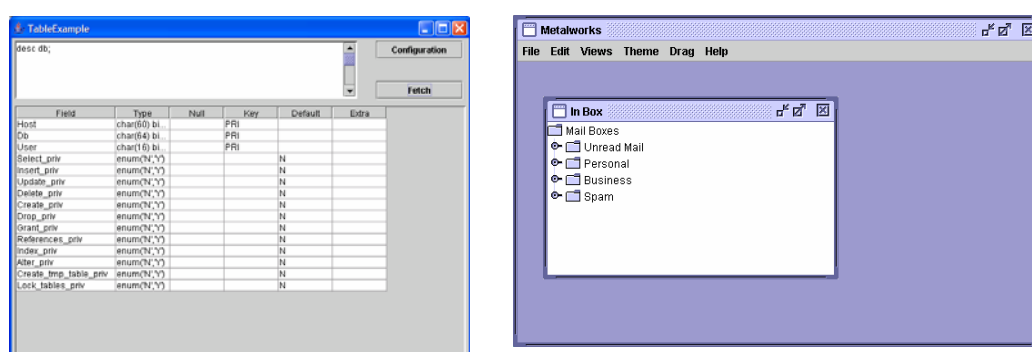
Exercícios

8. Construindo GUIs (Interfaces com Gráficas com o Usuário)

As Java Foundation Classes (JFC) foram criadas com o intuito de permitir aos programadores escreverem aplicativos para o usuário final com grande riqueza de interface e comunicação visual.

Composta de uma grande gama de classes com o intuito de permitir ao usuário uma experiência de uso bem intuitiva e confortável, e para os programadores facilidades de internacionalização e construção.

Exemplo de interfaces que pode ser construída usando os componentes da JFC/SWING:



Os pacotes de classes que compõe as JFC são:

AWT – Abstract Windowing Toolkit (`java.awt`) – componentes básicos para interfaces gráficas, exibidos nativamente pelo sistema operacional com resolução de 72DPI.

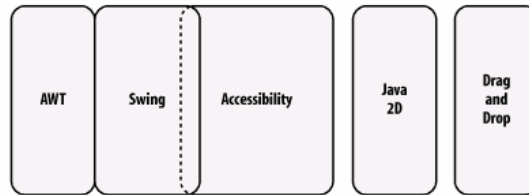
Accessibility – Dentro das classes do SWING, todos os componentes possuem ferramentas de acessibilidade que permitem aos deficientes visuais de menor grau usarem uma aplicação Java.

2D API – Conjuntos de classes que permite a construção de elementos 2D, como textos, ferramentas gráficas, mistura de cores, formas, preenchimento, padrões, animações simples, etc.

DnD – Drag and Drop – classes que nos permitem escrever ferramentas intuitivas de arrastar e soltar.

SWING – (`javax.swing`) – Componentes 100% Java criados com o intuito de oferecer um conjunto de elementos de interface gráfica com grande riqueza de detalhes e expansividade. Diferente o AWT que tem partes nativas da plataforma, o SWING pode ser executado em qualquer plataforma e manter o seu visual, tamanhos de fontes, cores e geometria (“look and feel”) com resolução de 300DPI.

Java Foundation Classes



As diferenças dos componentes do AWT em relação SWING são muito grandes, desde a sua construção até seu comportamento, exceto pelo controle de eventos e pelos gerenciadores de layout.

Dentro do AWT e do SWING, os gerenciadores de layout (layout managers) nos permite escrever uma interface gráfica complexa sem nos preocupar com as dimensões em pixels de cada componente, pois quando criamos um componente que usa um layout manager ele será dimensionado de acordo com algoritmos de posicionamento e redimensionamento com os quais não precisamos nos preocupar.

As JFC foram desenhadas com o intuito de facilitar a construção de aplicativos que necessitam de uma usabilidade facilitada para o usuário.

Dentro do SWING, os pacotes de classes são:

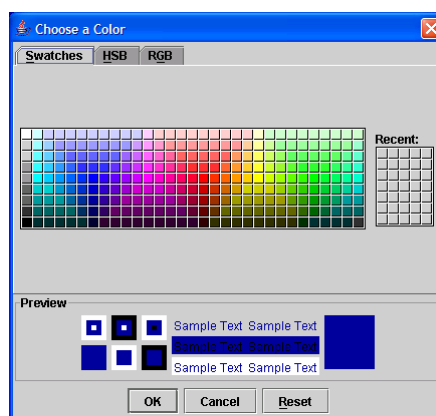
javax.accessibility: contém classes e interfaces que nos permitem criar elementos com acesso facilitado ao usuário que necessita de comportamento intuitivo;

javax.swing: contém os componentes principais do SWING que nos permite criar as aplicações gráficas com janelas, botões, listas, combos, etc.

javax.swing.border: contém as definições de elementos gráficos para a construção de bordas;

javax.swing.event e **java.awt.event:** contém as classes de suporte ao tratamento de eventos;

javax.swing.colorchooser: suporte ao componente JColorChooser (ferramenta para a escolha de cores);

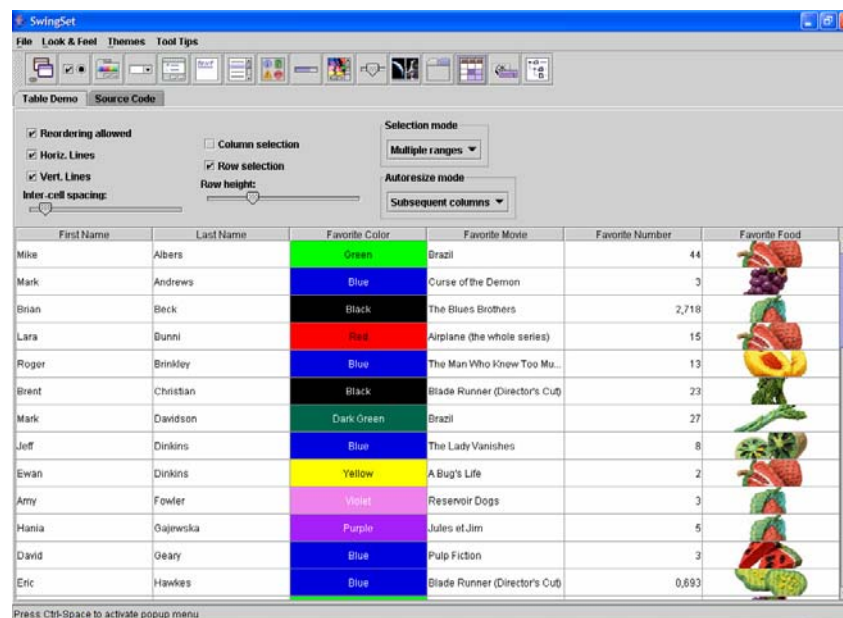


javax.swing.filechooser: suporte ao componente JFileChooser (ferramenta para a escolha de arquivos);

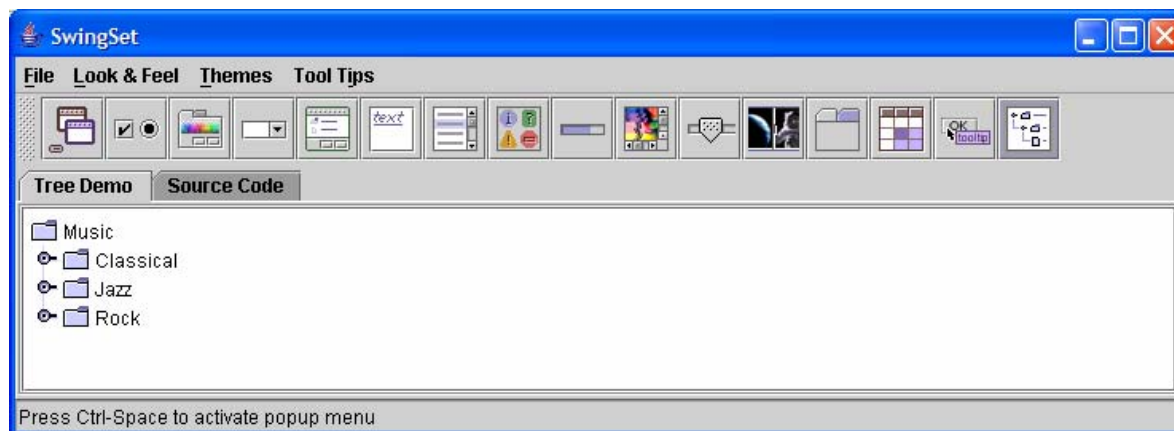


javax.swing.plaf: interfaces que fornecem o suporte a implementação de uso de um look and feel (cores, fontes, temas, etc) nativo ou customizado;

javax.swing.table: prove modelo de dados e visualização para a construção de simples tabelas até grids complexos.



javax.swing.tree: prove o modelo de dados e visualização para a construção de elementos em árvores simples.



javax.swing.undo: contém as classes necessárias para a implementação de “desfazer” dentro do sistema gráfico.

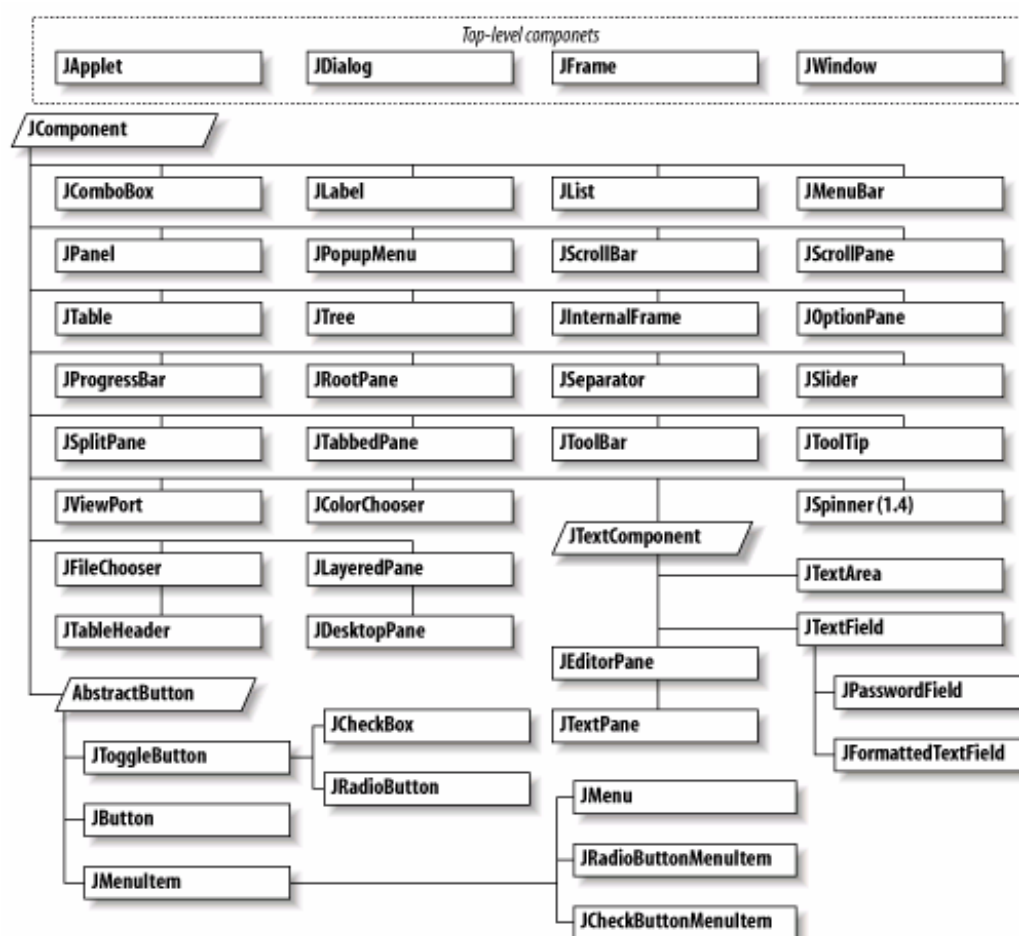
Neste capítulo veremos a construção de interfaces gráficas para aplicações em janelas usando os principais componentes do SWING, tratamento de eventos, aspectos visuais e utilitários.

Entendendo a hierarquia de classes:

O pacote do SWING possui uma superclasse que é responsável por descrever o comportamento de qualquer componente visual. Essa superclasse é `javax.swing.JComponent`.

A partir dela, existem alguns tipos de componentes visuais que foram muito bem abstraídos contruídos, permitindo um grande reaproveitamento dos componentes de interface.

Podemos construir nossas janelas, caixas de mensagens e diálogo a partir dos componentes de alto-nível (top-level components). Um **Top-Level Component** pode conter qualquer outro tipo de componente em seu interior usando o conceito de **Container**. Um **Container** pode conter um conjunto de **Component** que será exibido na tela.



Para se exibir um conjunto de elementos na tela, devemos escolher um Container, que terá dentro dele uma coleção de **JComponent**, organizados por um **LayoutManager** (gerenciador de layout) que dará a geometria da tela em qualquer dimensão.

Quanto mais rica for a interface, mais complexo fica sua construção e desenho, por isso a concepção de comportamento de layout e usabilidade da interface é um passo importante antes de começar a construção da mesma.

Modelo de desenvolvimento de interfaces gráficas

Como a criação de interfaces gráficas está diretamente ligada às necessidades e experiências dos usuários que vão utilizá-las, devemos antes de iniciar a construção de qualquer interface gráfica conhecer suas capacidades.

Usando componentes de GUI (Graphical User Interface) baseados nos componentes do SWING, podemos construir qualquer interface gráfica.

Entretanto, devemos lembrar que quanto maior for o nível iteratividade do usuário, mais tempo levará para se construir a interface e implementar as ações de usabilidade que o usuário deseja.

O foco deste capítulo é mostrar as principais capacidades dos componentes do SWING para que possamos construir interfaces simples e médias. Como a API do SWING é muito extensa, alguns componentes não serão abordados diretamente, entretanto o modelo de uso desses componentes será apresentado permitindo ao leitor entender e usar qualquer componente da SWING API.

Modelo de desenvolvimento do SWING para GUIs

Os componentes “**top-level**” mostrados na figura da hierarquia de classes são categorizados como “**container**”. Um container, pode conter e controlar a exibição de qualquer “**component**”.

Se quisermos construir um interface que tenha uma janela, uma barra de menu e uma barra de status, devemos escolher um “**container**” que receberá esses componentes. Nesse caso usaremos como “**container**” a classe `javax.swing.JFrame` por ser “**top-level**” e por que desejamos que nossa janela tenha todos os controle de controle de exibição.

A forma mais simples de se construir componentes gráficos é especializá-los, ou seja, criamos uma classe que estende o comportamento de outra, dessa forma herdamos todos os métodos da superclasse facilitando a construção de código.

No nosso caso, para o “**top-level**” criaremos uma classe que estende a `javax.swing.JFrame` que chamará `MyFrame` para vermos como construir uma classe para exibição em janela.

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

public class MyFrame extends JFrame {

    public MyFrame() {
        super("TextEditor v1.0");
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        setVisible( true );
    }

    public static void main(String args[]) {
        // cria um objeto de janela para ser iniciado e exibido
        new MyFrame().init();
    }
}
```

Compilando e executando esse código, veremos uma janela sobre a janela da console do terminal.

```
# java MyFrame
```

Como ainda não vimos o tratamento de eventos dos componentes gráficos, para encerrar este aplicativo basta teclar CTRL+C no console do terminal.

Diferentemente das outras tecnologias que usam formulários pré-compilados, a criação de interfaces gráficas em java é 100% escrita em código fonte, permitindo que nossas classes de GUIs sejam abertas e manipuladas em qualquer ferramenta IDE como Sun Java Studio, Eclipse, Oracle JDeveloper, Borland JBuilder, etc.

Gerenciadores de layout e posicionamento.

Vimos até agora os conceitos básicos dos componentes do SWING, basicamente como escrevê-los e exibi-los, entretanto, temos de aprender como posicionar e dimensionar os componentes na área da interface.

Como o SWING é 100% escrito em Java, o posicionamento e dimensionamento dos componentes de interface também deve ser portátil e com garantia de geometria se trocarmos a plataforma de execução, por exemplo de Windows para Linux ou MacOS.

A forma encontrada pelos engenheiros da JavaSoft e da SWING Connection, empresa que desenvolveu o SWING, quando idealizaram as APIs para garantir o posicionamento e geometria dos componentes foi criando um conjunto de gerenciadores de layout que fazem todo trabalho de cálculo para alinhamento, posicionamento e dimensionamento dos componentes independente da plataforma de execução.

Os gerenciadores de layout principais e que resolverão grande parte dos desenhos de tela são:

- `java.awt.FlowLayout`
- `java.awt.GridLayout`
- `java.awt.BorderLayout`
- `java.awt.GridBagLayout`
- Layouts compostos

Quando queremos usar um desses layouts, escolhemos um **“container”** ou um **“top-level”** e usamos o método `setLayout(LayoutManager)`, a partir daí adicionamos os outros componentes que farão parte do desenho da interface, e quando for chamado o método `setVisible(boolean)` do **“container”**, o layout manager escolhido fará os cálculos necessários de dimensionamento e posicionamento para os componentes presentes na interface.

Para os componentes **“top-level”** `javax.swing.JPanel` e `javax.swing.JApplet` o layout manager padrão é o `java.awt.FlowLayout`.

Para os componentes **“top-level”** `javax.swing.JDialog` e `javax.swing.JWindow` e `javax.swing.JFrame` o layout manager padrão é o `java.awt.BorderLayout`.

Entendendo o `java.awt.FlowLayout`

Este gerenciador de layout é o mais simples de todos. E as seguintes características:

- alinhamento padrão ao centro e acima, com deslocamento em linhas (flow = fluxo);
- capacidade de manter o tamanho original dos componentes durante o redimensionamento (mantém o ***preferred size*** dos componentes);
- o container pode receber mais de um componente por área;

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

public class FlowExample extends JFrame {

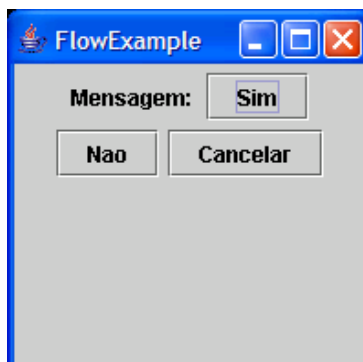
    private JButton sim, nao, cancelar;
    private JLabel mensagem;

    public FlowExample() {
        super("FlowExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        cancelar = new JButton("Cancelar");
        mensagem = new JLabel("Mensagem: ");
        // seta layout para FlowLayout deste top-level
        getContentPane().setLayout( new FlowLayout() );
    }

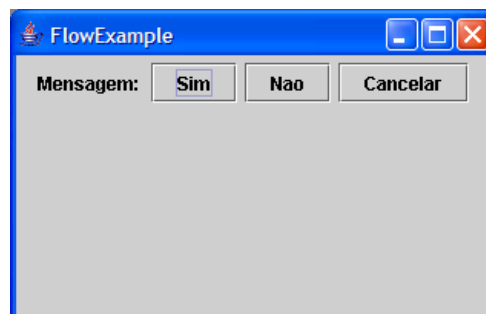
    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        // adiciona componentes
        getContentPane().add( mensagem );
        getContentPane().add( sim );
        getContentPane().add( nao );
        getContentPane().add( cancelar );
        setVisible( true );
    }

    public static void main(String args[]) {
        new FlowExample().init();
    }
}
```

Inicial



Redimensionado



Entendendo o `java.awt.GridLayout`

Este layout manager tem como características:

- divide o container em linhas e colunas como uma tabela com células (grid);
- não mantém o tamanho original dos componentes durante o redimensionamento (não mantém o ***preferred size*** dos componentes);
- o container pode receber somente um componente por área;

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

public class GridExample extends JFrame {

    private JButton sim, nao, cancelar;
    private JLabel mensagem;

    public GridExample() {
        super("GridExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        cancelar = new JButton("Cancelar");
        mensagem = new JLabel("Mensagem: ");
        getContentPane().setLayout( new GridLayout(2,2) );
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        getContentPane().add( mensagem );
        getContentPane().add( sim );
        getContentPane().add( nao );
        getContentPane().add( cancelar );
        setVisible( true );
    }

    public static void main(String args[]) {
        new GridExample().init();
    }
}
```

Inicial



Redimensionado



Entendendo o `java.awt.BorderLayout`

Este layout manager tem como características:

- divide o container em cinco áreas, norte, sul, leste, oeste e centro;
- não mantém o tamanho original dos componentes durante o redimensionamento (não mantém o ***preferred size*** dos componentes);
- o container pode receber somente um componente por área;

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

public class BorderExample extends JFrame {

    private JButton sim, nao, cancelar;
    private JLabel mensagem;

    public BorderExample() {
        super("BorderExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        cancelar = new JButton("Cancelar");
        mensagem = new JLabel("Mensagem: ");
        getContentPane().setLayout( new BorderLayout() );
    }

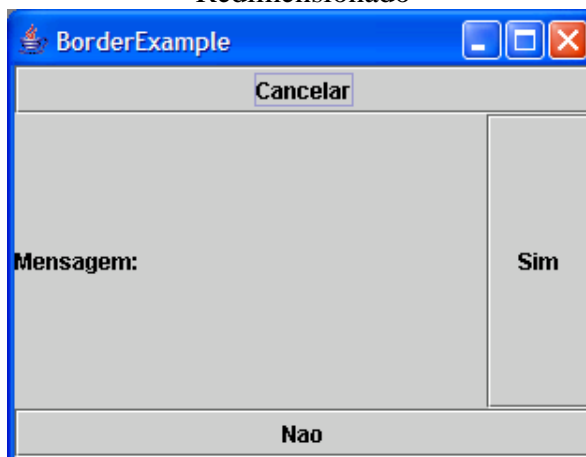
    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        getContentPane().add( mensagem, BorderLayout.CENTER );
        getContentPane().add( sim, BorderLayout.EAST );
        getContentPane().add( nao, BorderLayout.SOUTH );
        getContentPane().add( cancelar, BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String args[]) {
        new BorderExample().init();
    }
}
```

Inicial



Redimensionado



Entendendo o `java.awt.GridBagLayout`

Este gerenciador de layout é o mais flexível de todos, entretanto é o mais complexo e exigirá um grande exercício de definição da interface, pois cada elemento deve conter uma posição inicial, uma posição final, um tamanho, uma escala, um alinhamento e um preenchimento.

Este layout manager tem como características:

- divide o container em linhas e colunas como uma tabela com células (grid);
- dependendo do alinhamento dentro de uma célula, pode ou não manter o **preferred size** dos componentes;
- o container pode receber somente um componente por área;
- cada célula pode ser expandida para ocupar o espaço de uma ou mais células adjacentes usando regras específicas de alinhamento e posicionamento;

```
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Container;
import java.awt.Color;
import javax.swing.*;

public class GridBagExample extends JFrame {

    public GridBagExample() {
        super("GridBagExample");
        getContentPane().setLayout( new GridBagLayout() );
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        gridAdd( getContentPane(), new JButton("1"), 0, 0, 5, 1, 0, 12,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER );
        gridAdd( getContentPane(), new JButton("2"), 0, 1, 1, 1, 0, 0,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("3"), 1, 1, 1, 1, 1, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("4"), 2, 1, 1, 1, 0, 0,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("5"), 3, 1, 2, 1, 0, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("6"), 0, 2, 1, 4, 0, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("7"), 1, 2, 3, 4, 0, 0,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("8"), 4, 2, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("9"), 4, 3, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("10"), 4, 4, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("11"), 4, 5, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("12"), 0, 6, 5, 1, 0, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        setVisible( true );
    }

    ...
}
```

```
private void gridAdd(Container cont, JComponent comp, int x, int y,
    int width, int height, int weightx, int weighty,
    int fill, int anchor) {

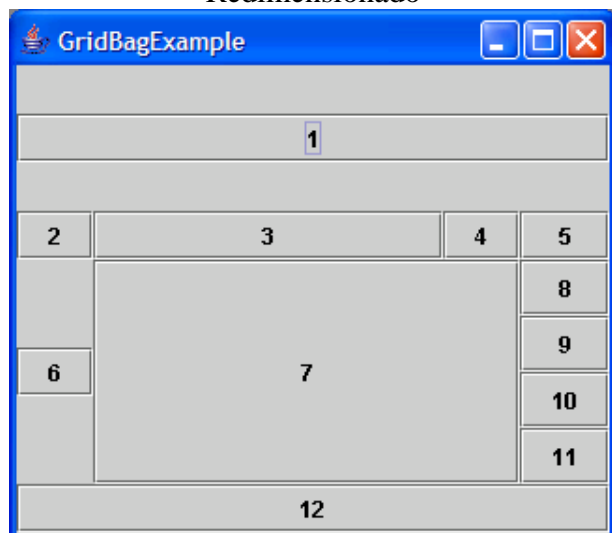
    GridBagConstraints cons = new GridBagConstraints();
    cons.gridx = x;
    cons.gridy = y;
    cons.gridwidth = width;
    cons.gridheight = height;
    cons.weightx = weightx;
    cons.weighty = weighty;
    cons.fill = fill;
    cons.anchor = anchor;
    cont.add(comp, cons);
}

public static void main(String args[]) {
    new GridBagExample().init();
}
}
```

Inicial



Redimensionado



Para um melhor entendimento do GridBagLayout sugiro um estudo mais completo da classe `java.awt.GridBagConstraints` dentro da documentação da Java API, pois ela é a responsável pelo posicionamento, dimensionamento, alinhamento e preenchimento dos componentes dentro de um container que esteja usando o GridBagLayout como gerenciador de layout.

Entendendo os Layout Compostos.

Usando o princípio do mosaíco, e evitando usar o GridBagLayout por ser muito complexo, dividimos uma interface em partes e aplicamos para cada uma delas um **container** com um **layout manager**, e assim colocamos os componentes em vários containers até obter o efeito desejado.

Já sabemos que para colocar mais de um componente dentro de uma área de um container devemos usar o gerenciador de layout FlowLayout, se quisermos dividir uma área em células, usaremos o GridLayout, e para dividir um container em Norte, Sul, Leste, Oeste e Centro, usaremos o BorderLayout.

Sendo assim, a partir da interface abaixo, veremos como ela pode ser dividida para obtermos o efeito desejado.



Código fonte:

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

public class MosaicExample extends JFrame {

    private JPanel botoes;
    private JPanel barraStatus;
    private JScrollPane painelTexto;
    private JButton novo, limpar, salvar, sair;
    private JLabel mensagem, relógio;
    private JTextArea areaTexto;

    public MosaicExample() {
        super("Mosaic Example");
        novo = new JButton("Novo");
        limpar = new JButton("Limpar");
        salvar = new JButton("Salvar");
        sair = new JButton("Sair");
        botoes = new JPanel( new FlowLayout() );
        mensagem = new JLabel("Mensagem: ");
        relógio = new JLabel("Data/Hora: " + new java.util.Date().toString() );
        barraStatus = new JPanel( new FlowLayout() );
        areaTexto = new JTextArea("Digite seu texto aqui: ", 20, 40);
        // no swing JTextArea deve ficar dentro de JScrollPane
        painelTexto = new JScrollPane (areaTexto , JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setLocation( 200, 130 );
        setSize( 640, 480);
        getContentPane().setLayout( new BorderLayout() );
        // definicao da area de botoes
        botoes.add( novo );
        botoes.add( limpar );
        botoes.add( salvar );
        botoes.add( sair );
        // definicao area de mensagens
        barraStatus.add ( mensagem );
        barraStatus.add ( relógio );
        // parte superior da interface contera os botoes
        getContentPane().add( botoes, BorderLayout.NORTH );
        // parte central da interface contera a area de texto
        getContentPane().add( painelTexto, BorderLayout.CENTER);
        // parte inferior da interface contera a area de mensagens
        getContentPane().add( barraStatus, BorderLayout.SOUTH );
        setVisible(true);
    }

    public static void main(String args[]) {
        new MosaicExample().init();
    }
}
```

Construir interfaces gráficas usando vários **containers** compostos de múltiplos gerenciadores de layout é mais simples, entretanto mais trabalhoso pois devemos trabalhar com uma abstração maior das áreas da interface. Devemos desenhar primeiro, imaginando quais containers e tipos de gerenciadores de layout antes de construir o código da interface.

Manipulando aspectos visuais

Todos os componentes do SWING que são originados da superclasse `javax.swing.JComponent`, possuem um conjunto de métodos que nos permite controlar aspectos visuais como fonte, cursor, borda, cor de fonte e cor de fundo.

`public void setCursor(java.awt.Cursor cursor)` - permite trocar o cursor quando este componente estiver em foco.

`public void setBackground(java.awt.Color color)` - permite trocar a cor de fundo do componente.

`public void setForeground(java.awt.Color color)` - permite trocar a cor de frente do componente.

`public void setEnabled(boolean enabled)` - permite habilitar ou desabilitar este componente.

`public void setFont(java.awt.Font font)` - permite escolher uma fonte para este componente.

`public void setBorder(javax.swing.border.Border border)` - permite definir uma borda para este componente.

`public void setToolTipText(String text)` – permite colocar um texto de dica para o componente.

Podemos adicionar elementos visuais em qualquer aplicação SWING ou AWT, veja o método `init()` abaixo com alterações nos aspectos visuais de cores, bordas, fontes e dicas:

```
public void init() {
    // cor de frente
    setForeground( Color.black );
    // cor de fundo
    setBackground( new Color(192,192,192) );
    setLocation( 200, 130 );
    setSize( 640, 480);
    getContentPane().setLayout( new BorderLayout() );

    Border borda = new LineBorder(Color.RED, 2);
    // colocando borda
    barraStatus.setBorder( borda );

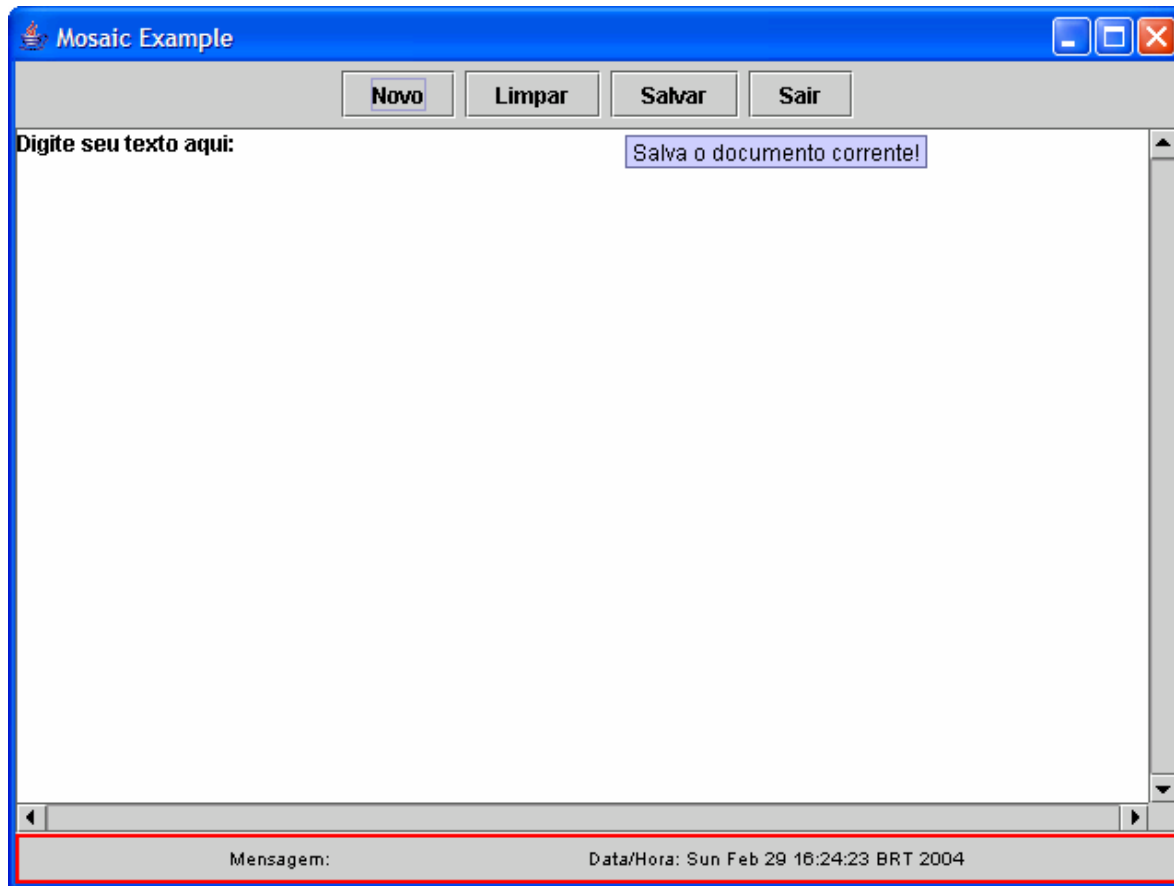
    // alterando cursores
    Cursor hand = new Cursor( Cursor.HAND_CURSOR );
    botoes.setCursor( hand );
    Cursor text = new Cursor( Cursor.TEXT_CURSOR );
    areaTexto.setCursor( text );
    Cursor cross = new Cursor( Cursor.CROSSHAIR_CURSOR );
    barraStatus.setCursor( cross );

    // alterando Fonte
    Font arialBold = new Font("Arial", Font.BOLD, 12 );
    areaTexto.setFont( arialBold );
    Font arial = new Font("Arial", Font.PLAIN, 10 );
    mensagem.setFont( arial );
    relógio.setFont( arial );

    // colocando dicas
    novo.setToolTipText("Cria um novo documento!");
    limpar.setToolTipText("Limpa o documento corrente!");
    salvar.setToolTipText("Salva o documento corrente!");
    sair.setToolTipText("Encerra a aplicação!");

    // definicao da area de botoes
    botoes.add( novo );
    botoes.add( limpar );
    botoes.add( salvar );
    botoes.add( sair );
    // definicao area de mensagens
    barraStatus.add ( mensagem );
    barraStatus.add ( relógio );
    // parte superior da interface contera os botoes
    getContentPane().add( botoes, BorderLayout.NORTH );
    // parte central da interface contera a area de texto
    getContentPane().add( painelTexto, BorderLayout.CENTER);
    // parte inferior da interface contera a area de mensagens
    getContentPane().add( barraStatus, BorderLayout.SOUTH );
    setVisible(true);
}
```

Vendo a execução do exemplo com o novo método `init()` com vários efeitos visuais novos:



Construir interfaces gráficas usando a API do SWING é uma tarefa trabalhosa, por isso uma definição detalhada dos comportamentos de interface, exibição, fontes, cursores, e dimensões durante a fase de concepção da aplicação é muito importante.

Tipos de componentes visuais

Até agora vimos os componentes visuais mais básicos da API do SWING, como janela (JFrame), botões (JButton), mensagem (JLabel), área de texto (JTextArea), painel de rolagem (JScrollPane) e painel (JPanel).

A partir de agora veremos outros componentes que nos permitirá criar interfaces mais ricas e com uma experiência para usuário mais elaborada, com menus (JMenuBar), manipulando caixa de informação (JOptionPane), campo de texto (JTextField), combos (JComboBox), listas (JList) e tabelas (JTable).

Criando menus para o usuário.

Para se criar uma barra de menus, usamos três classes JMenuBar, JMenu e JMenuItem.

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

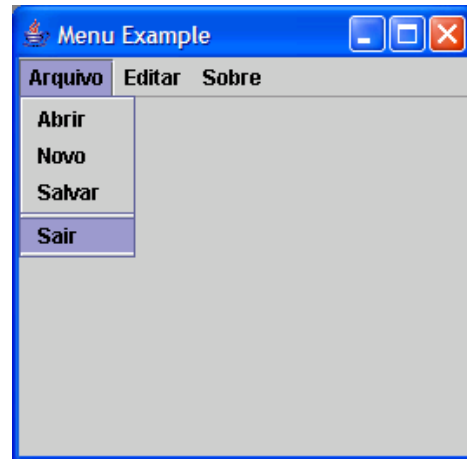
public class MenuExample extends JFrame {

    private JMenuItem abrir, novo, salvar, sair;
    private JMenuItem copiar, colar, recortar;
    private JMenuItem ajuda, info;
    private JMenu arquivo, editar, sobre;
    private JMenuBar menuBar;

    public MenuExample() {
        super("Menu Example");
    }

    public void init() {
        setSize( 400, 400 );
        setLocation( 300, 200 );
        // construindo objetos
        abrir = new JMenuItem("Abrir");
        novo = new JMenuItem("Novo");
        salvar = new JMenuItem("Salvar");
        sair = new JMenuItem("Sair");
        copiar = new JMenuItem("Copiar");
        colar = new JMenuItem("Colar");
        recortar = new JMenuItem("Recortar");
        ajuda = new JMenuItem("Ajuda");
        info = new JMenuItem("Info");
        arquivo = new JMenu("Arquivo");
        editar = new JMenu("Editar");
        sobre = new JMenu("Sobre");
        // construindo menu arquivo
        arquivo.add ( abrir );
        arquivo.add ( novo );
        arquivo.add ( salvar );
        arquivo.addSeparator();
        arquivo.add ( sair );
        // construindo menu editar
        editar.add ( copiar );
        editar.add ( colar );
        editar.add ( recortar );
        // construindo menu sobre
        sobre.add ( ajuda );
        sobre.add ( info );
        // construindo menu
        menuBar = new JMenuBar();
        menuBar.add( arquivo );
        menuBar.add( editar );
        menuBar.add( sobre );
        setJMenuBar( menuBar );
        setVisible(true);
    }

    public static void main(String args[]) {
        new MenuExample().init();
    }
}
```



Trabalhando com caixa de informação e diálogo

Para trabalhar com caixas de informação e diálogos já pronta, usamos a classe `JOptionPane` que nos fornece uma grande variedade de métodos e retorno e opções.

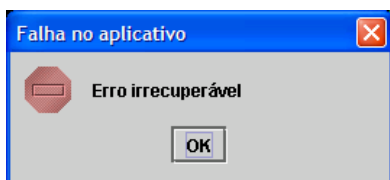
Podemos usar o `JOptionPane` de quatro formas:

- `showConfirmDialog` – confirmação;
- `showInputDialog` – entrada de dado;
- `showMessageDialog` – exibição de mensagem;
- `showOptionDialog` – caixa completa com todas as opções acima.

Exemplos de uso:

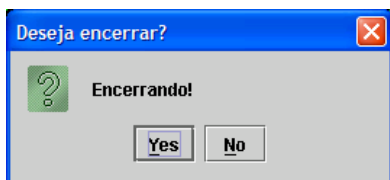
Informando um erro:

```
JOptionPane.showMessageDialog( null, "Erro irre recuperável", "Falha no aplicativo",  
JOptionPane.ERROR_MESSAGE );
```



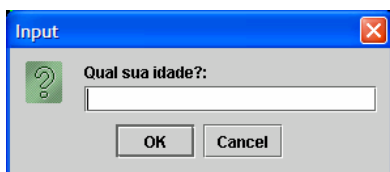
Exibindo uma caixa de opções yes/no:

```
JOptionPane.showConfirmDialog( null, "Encerrando!" , "Deseja encerrar?" ,  
JOptionPane.YES_NO_OPTION );
```



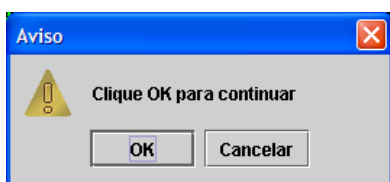
Exibindo uma caixa de entrada de valor:

```
String inputValue = JOptionPane.showInputDialog("Qual sua idade?: ");
```



Exibindo uma caixa de opções customizadas:

```
Object[] options = { "OK", "Cancelar" };  
JOptionPane.showOptionDialog( null, "Clique OK para continuar", "Aviso",  
JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[0] );
```



Trabalhando com botões.

JButton - já vimos em outros exemplos como construir esse tipo de componente.

JRadioButton e ButtonGroup – usamos quando temos opções únicas e exclusivas, tais como sexo, estado civil, etc.

JCheckBox – usamos quando temos opções múltiplas e não exclusivas, tais como bens: casa, carro, moto, etc.

JToggleButton – usamos quando temos para uma determinada opção, ativo ou desativo.

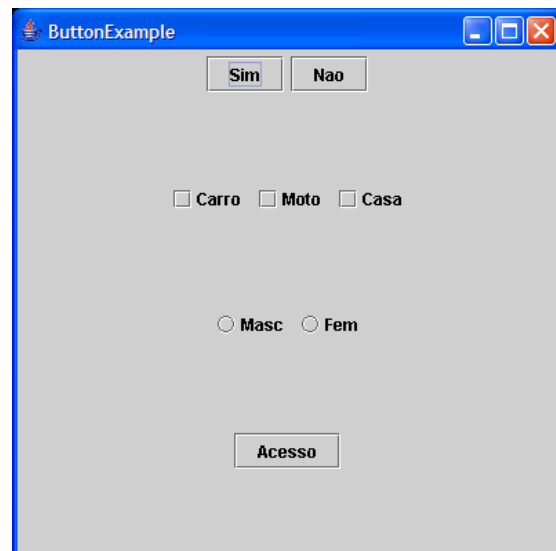
```
import javax.swing.*;
import java.awt.*;

public class ButtonExample extends JFrame {
    private JButton sim, nao;
    private JCheckBox carro, moto, casa;
    private JRadioButton masculino, feminino;
    private JToggleButton luz;
    private JPanel buttons1, buttons2, buttons3, buttons4;

    public ButtonExample() {
        super("ButtonExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        carro = new JCheckBox("Carro");
        moto = new JCheckBox("Moto");
        casa = new JCheckBox("Casa");
        masculino = new JRadioButton("Masc");
        feminino = new JRadioButton("Fem");
        luz = new JToggleButton("Acesso");
    }

    public void init() {
        getContentPane().setLayout( new GridLayout(4,1) );
        setSize(400,400);
        setLocation(300,200);
        buttons1 = new JPanel();
        buttons1.add( sim );
        buttons1.add( nao );
        buttons2 = new JPanel();
        buttons2.add( carro );
        buttons2.add( moto );
        buttons2.add( casa );
        buttons3 = new JPanel();
        ButtonGroup sexo = new ButtonGroup();
        sexo.add( masculino );
        sexo.add( feminino );
        buttons3.add( masculino );
        buttons3.add( feminino );
        buttons4 = new JPanel();
        buttons4.add( luz );
        getContentPane().add( buttons1 );
        getContentPane().add( buttons2 );
        getContentPane().add( buttons3 );
        getContentPane().add( buttons4 );
        setVisible(true);
    }

    public static void main(String arg[]) {
        new ButtonExample().init();
    }
}
```



Trabalhando com campos

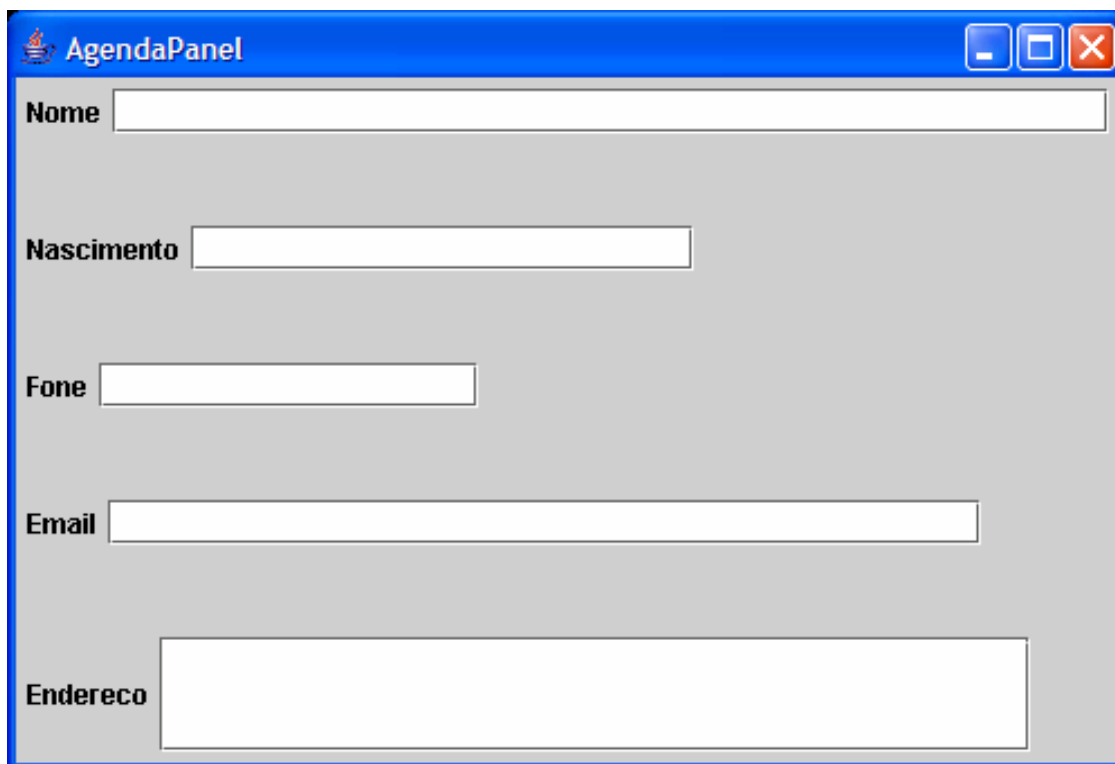
Dentro dos componentes do SWING a entrada de dados pode ser feita, dependendo do caso, usando-se um dos componentes abaixo:

`javax.swing.JTextField` – campo de texto livre, com uma linha, sem formatação;
`javax.swing.JTextArea` – campo de texto livre, com múltiplas linhas, sem formatação;

`javax.swing.JPasswordField` – campo de texto livre, com uma linha, sem formatação, específico para entrada de senhas;

`javax.swing.JFormattedTextField` – campo de texto livre, com uma linha, com formatação;

Exemplo de um formulário:



The image shows a Java Swing window titled "AgendaPanel". The window has a blue title bar with standard Windows-style window controls (minimize, maximize, close). The main content area is light gray and contains five text input fields arranged vertically. Each field is preceded by a label: "Nome", "Nascimento", "Fone", "Email", and "Endereco". The "Nome" field is the longest, followed by "Email". The "Endereco" field is the widest and appears to be a multi-line text area. The "Fone" field is the shortest.

Para manipularmos os campos formatados, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

Veja o código fonte deste formulário abaixo:

```
// AgendaPanel.java
import javax.swing.*;
import java.awt.*;
import java.text.*;

public class AgendaPanel extends JPanel {

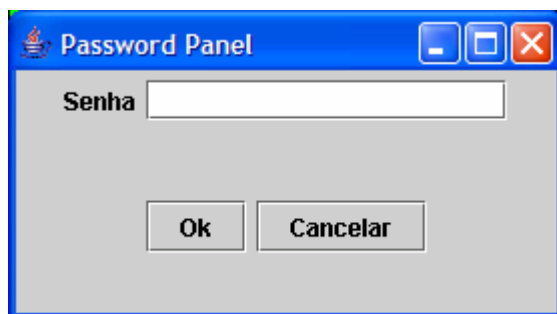
    private JLabel lblNome, lblEmail, lblNascimento, lblFone, lblEndereco;
    private JTextField txtNome, txtEmail, txtFone;
    private JFormattedTextField txtNascimento;
    private JTextArea txtEndereco;
    private Format dateFormatter;

    public AgendaPanel() {
        lblNome = new JLabel("Nome");
        lblEmail = new JLabel("Email");
        lblNascimento = new JLabel("Nascimento");
        lblFone = new JLabel("Fone");
        lblEndereco = new JLabel("Endereco");
        txtNome = new JTextField(40);
        txtEmail = new JTextField(35);
        txtFone = new JTextField(15);
        txtEndereco = new JTextArea("", 3, 35);
        // cria-se o elemento de entrada formatada usando um formato pre-definido
        // atraves da java.text.SimpleDateFormat
        dateFormatter = new SimpleDateFormat("dd/MM/yyyy");
        txtNascimento = new JFormattedTextField( dateFormatter );
        txtNascimento.setColumns( 20 );
    }

    public void init() {
        setLayout( new GridLayout(5,1) );
        FlowLayout esquerda = new FlowLayout( FlowLayout.LEFT );
        // usando Paineis auxiliares do tipo FlowLayout para alinha a esquerda
        // e poder inserir mais um componente por linha do grid
        JPanel auxNome = new JPanel( esquerda );
        auxNome.add( lblNome );
        auxNome.add( txtNome );
        JPanel auxNascimento = new JPanel( esquerda );
        auxNascimento.add( lblNascimento );
        auxNascimento.add( txtNascimento );
        JPanel auxEmail = new JPanel( esquerda );
        auxEmail.add( lblEmail );
        auxEmail.add( txtEmail );
        JPanel auxFone = new JPanel( esquerda );
        auxFone.add( lblFone );
        auxFone.add( txtFone );
        // o JTextArea deve estar sempre de um JScrollPane
        JScrollPane scrollEndereco = new JScrollPane ( txtEndereco,
        JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED, JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED );
        JPanel auxEndereco = new JPanel( esquerda );
        auxEndereco.add( lblEndereco );
        auxEndereco.add( scrollEndereco );
        // adiciona as linhas ao grid
        add( auxNome );
        add( auxNascimento );
        add( auxFone );
        add( auxEmail );
        add( auxEndereco );
    }

    public static void main(String arg[]) {
        AgendaPanel agendaPanel = new AgendaPanel();
        agendaPanel.init();
        JFrame frame = new JFrame("AgendaPanel");
        frame.getContentPane().add( agendaPanel );
        frame.pack();
        frame.setVisible( true );
    }
}
```

Exemplo de leitura de senha:



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

public class PasswordPanel extends JPanel {

    private JLabel lblSenha;
    private JPasswordField txtSenha;
    private JButton ok, cancel;

    public PasswordPanel() {
        lblSenha = new JLabel("Senha");
        txtSenha = new JPasswordField ("",16);
        ok = new JButton("Ok");
        cancel = new JButton("Cancelar");
    }

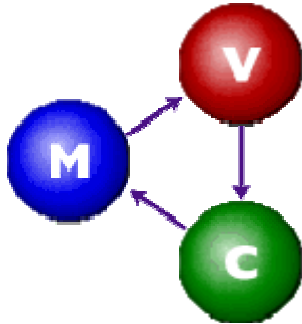
    public void init() {
        setLayout( new GridLayout(2,1) );
        FlowLayout centro = new FlowLayout( FlowLayout.CENTER );
        // usando Paineis auxiliares do tipo FlowLayout para alinha a esquerda
        // e poder inserir mais um componente por linha do grid
        JPanel auxSenha = new JPanel( centro );
        auxSenha.add( lblSenha );
        auxSenha.add( txtSenha );
        JPanel auxBotoes = new JPanel( centro );
        auxBotoes.add( ok );
        auxBotoes.add( cancel );
        add( auxSenha );
        add( auxBotoes );
    }

    public static void main(String arg[]) {
        PasswordPanel passwordPanel = new PasswordPanel();
        passwordPanel.init();
        JFrame frame = new JFrame("Password Panel");
        frame.getContentPane().add( passwordPanel );
        frame.pack();
        frame.setVisible( true );
    }
}
```

Criando classes a partir da superclasse JPanel, como vimos nos últimos dois exemplos, nos permitirá reaproveitar esses componentes em qualquer outro **“top-level container”**, permitindo criar elementos mais modulares.

Exibindo listas, combos e tabelas.

Cada um destes componentes dentro do SWING são divididos em três classes, um “Controller”, um “Model”, e uma “View”. Cada um deles tem uma responsabilidade distinta e baseada no modelo MVC – Model – View – Controller, amplamente utilizando como padrão de construção de componentes visuais.



A responsabilidade das classes que se comportam como **Model**, é representar um modelo de dados e garantir o estado desse modelo.

A responsabilidade das classes que se comportam como **View** é prover uma forma de visualização para aquele modelo naquele estado.

E o **Controller** atua como controlador destas escolhas e visualizações.

Dentro da API do SWING, listas, combos e tabelas são baseadas no modelo MVC, por isso:

- quando formos manipular objetos dentro de uma **lista** na tela usaremos:
 - `javax.swing.JList` – controlador;
 - `javax.swing.ListModel` – modelo;
 - `javax.swing.ListCellRenderer` – visualização;
- quando formos manipular objetos dentro de uma **tabela** na tela usaremos:
 - `javax.swing.JTable` – controlador;
 - `javax.swing.table.TableModel` – modelo;
 - `javax.swing.table.TableCellRenderer` – visualização;
- quando formos manipular objetos dentro de um **combo** na tela usaremos:
 - `javax.swing.JComboBox` – controlador;
 - `javax.swing.ComboBoxModel` – modelo;
 - `javax.swing.ComboBoxEditor` – editor;

Para facilitar o aprendizado faremos um exemplo de cada um desses tipos de componentes.

Trabalhando com JList, ListModel e ListCellRenderer.

A classe `javax.swing.JList` já existe dentro dos componentes do SWING, e para criarmos uma lista com um modelo já existente nos é fornecido a classe `javax.swing.DefaultListModel` que implementa a interface `javax.swing.ListModel`.

Entretanto se quisermos criar nosso próprio modelo de lista podemos criar uma classe que implemente a interface `javax.swing.ListModel` (não indicado, por ser muito complexo) ou que estenda a `javax.swing.DefaultListModel` (indicado, por ser mais simples) onde reescrevemos os métodos que nos forem convenientes. O uso direto da classe `javax.swing.DefaultListModel` como **Model** resolverá grande dos casos de implementação de listas usando `JList` dentro dos componentes do SWING.

Da mesma forma como existe um Model default para as listas, existe uma **View** default também, a `javax.swing.DefaultListCellRenderer`, e se quisermos criar uma visualização especial para uma lista, é preferível criar uma classe que estenda o comportamento da `javax.swing.DefaultListCellRenderer` onde poderemos reescrever somente os métodos que forem necessários.

Exemplo de uso da JList:

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

public class JListExample extends JFrame {

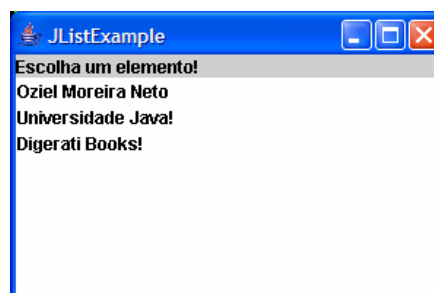
    private JList nomes;
    private DefaultListModel nomesModel;
    private DefaultListCellRenderer nomesRenderer;

    public JListExample() {
        super("JListExample");
        nomesRenderer = new DefaultListCellRenderer();
        nomesModel = new DefaultListModel();
        nomes = new JList( nomesModel );
        nomes.setCellRenderer( nomesRenderer );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        nomesModel.addElement( new String("Oziel Moreira Neto") );
        nomesModel.addElement( new String("Universidade Java!") );
        nomesModel.addElement( new String("Digerati Books!") );

        getContentPane().add ( nomes, BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Escolha um elemento!"), BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String arg[]) {
        new JListExample().init();
    }
}
```



Para manipularmos elementos na lista, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

Trabalhando com JComboBox, ComboBoxModel e ComboBoxEditor.

A classe `javax.swing.JComboBox` já existe dentro dos componentes do SWING, e para criarmos uma caixa de opções com um modelo já existente nos é fornecido a classe `javax.swing.DefaultComboBoxModel` que implementa a interface `javax.swing.ComboBoxModel`.

Entretanto se quisermos criar nosso próprio modelo de caixa de opções podemos criar uma classe que implemente a interface `javax.swing.ComboBoxModel` (não indicado, por ser muito complexo) ou que estenda a `javax.swing.DefaultComboBoxModel` (indicado, por ser mais simples) onde reescrevemos os métodos que nos forem convenientes. O uso direto da classe `javax.swing.DefaultComboBoxModel` como **Model** resolverá grande dos casos de implementação de listas usando JComboBox dentro dos componentes do SWING.

Da mesma forma como existe um **Model** default para as caixas de opções, existe uma **View** default também, a `javax.swing.plaf.basic.BasicComboBoxRenderer`, e se quisermos criar uma visualização especial para uma lista, é preferível criar uma classe que estenda o comportamento da classe `javax.swing.plaf.basic.BasicComboBoxRenderer` onde poderemos reescrever somente os métodos que forem necessários.

O JComboBox possui um elemento diferente das listas, é o **Editor** usado na linha editável que este componente possui internamente. Se desejarmos podemos criar nosso próprio ComboBoxEditor criando uma classe que implemente a `javax.swing.ComboBoxEditor`, ou usando diretamente uma classe de um editor padrão, a `javax.swing.plaf.basic.BasicComboBoxEditor`.

Para tornar um JComboBox editável, devemos usar o método `JComboBox.setEditable (true)`, dessa forma o componente permitirá a entrada de dados via a linha editável.

Exemplo de uso do JComboBox:

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

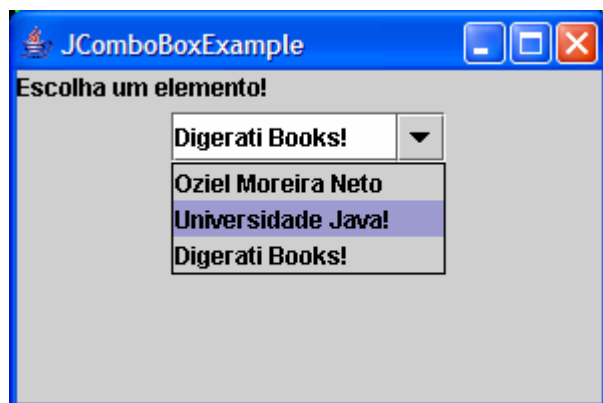
public class JComboBoxExample extends JFrame {

    private JComboBox nomes;
    private DefaultComboBoxModel nomesModel;

    public JComboBoxExample () {
        super("JListExample");
        nomesModel = new DefaultComboBoxModel ();
        nomes = new JComboBox( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        nomesModel.addElement( new String("Oziel Moreira Neto") );
        nomesModel.addElement( new String("Universidade Java!") );
        nomesModel.addElement( new String("Digerati Books!") );
        // tornando o combobox editavel.
        nomes.setEditable( true );
        JPanel auxNomes = new JPanel();
        auxNomes.add( nomes );
        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Escolha um elemento!"), BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String arg[]) {
        new JComboBoxExample().init();
    }
}
```



Para manipularmos elementos na caixa opções e inserir elementos na lista através da linha editável, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

Trabalhando com JTable, TableModel e TableCellRenderer.

A classe `javax.swing.JTable` já existe dentro dos componentes do SWING, e para criarmos uma tabela com um modelo já existente nos é fornecido a classe `javax.swing.table.DefaultTableModel` que implementa a interface `javax.swing.table.TableModel`.

Entretanto se quisermos criar nosso próprio modelo de tabela podemos criar uma classe que implemente a interface `javax.swing.table.TableModel` (não indicado, por ser muito complexo) ou que estenda a `javax.swing.table.DefaultTableModel` (indicado, por ser mais simples) onde reescrevemos os métodos que nos forem convenientes. O uso direto da classe `javax.swing.table.DefaultTableModel` como **Model** resolverá grande dos casos de implementação de listas usando JTable dentro dos componentes do SWING.

Da mesma forma como existe um **Model** default para as caixas de opções, existe uma **View** default também, a `javax.swing.table.DefaultTableCellRenderer`, e se quisermos criar uma visualização especial para uma lista, é preferível criar uma classe que estenda o comportamento da `javax.swing.table.DefaultTableCellRenderer` onde poderemos reescrever somente os métodos que forem necessários.

O JTable, assim como o JComboBox possui um elemento diferente das listas, é o **Editor** usado nas células editáveis que este componente pode fornecer. Se desejarmos podemos criar nosso próprio TableCellEditor devemos criar uma classe que implemente a interface `javax.swing.table.TableCellEditor`, ou usando diretamente uma classe de um editor padrão, a `javax.swing.DefaultCellEditor`.

Exemplo de uso do JTable:

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

public class JTableExample extends JFrame {

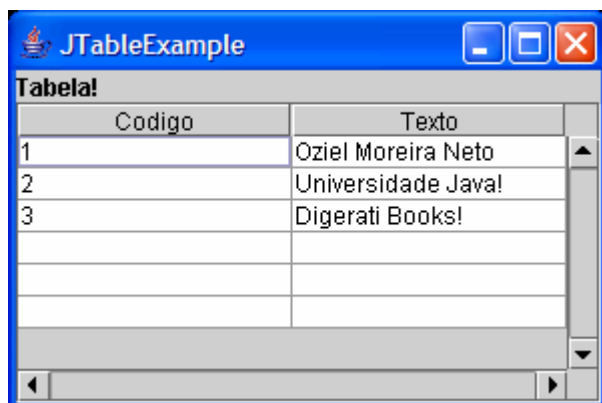
    private JTable nomes;
    private DefaultTableModel nomesModel;

    public JTableExample () {
        super("JTableExample");
        String[] cols = {"Codigo","Texto"};
        nomesModel = new DefaultTableModel ( cols , 3 );
        nomes = new JTable ( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        String[] row1 = {"1","Oziel Moreira Neto"};
        String[] row2 = {"2","Universidade Java!"};
        String[] row3 = {"3","Digerati Books!"};
        nomesModel.insertRow(0, row1 );
        nomesModel.insertRow(1,row2 );
        nomesModel.insertRow(2, row3 );
        JScrollPane auxNomes = new JScrollPane( nomes, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );

        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Tabela!"), BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String arg[]) {
        new JTableExample ().init();
    }
}
```



Para manipularmos elementos na tabela, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

Discussão

Qual o melhor processo para se desenhar uma GUI?

No capítulo atual, as GUIs estão restritas á aplicações stand-alone, como fica uma GUI na WEB?

Quando estamos desenhando GUIs não seria melhor pensar em componentes generéricos?

Exercícios

9. Tratamento de Eventos para GUIs

Todas as ações que o usuário de uma interface gráfica deseja executar como usabilidade de interface deve ser mapeada em eventos que os componentes gráficos suportem.

Cada tipo de componente possui um conjunto de eventos que ele suporta, e os eventos em java também são objetos e proveniente de classes. Então possuem métodos e atributos.

Os eventos são categorizados por recurso (teclado e mouse) e por componente (janela, lista, combo, campo de texto, etc.).

O modelo de tratamento de eventos presente dentro da J2SE 1.4.2 é o mesmo desde a JDK 1.1, ou seja, temos uma grande compatibilidade com versões. Este modelo chama-se modelo de delegação.

Modelo de delegação para tratamento de eventos.

Este modelo está composto de três elementos:

1. Criamos uma classe que trata um tipo de evento, esta classe deve implementar um das interfaces filhas de `java.util.EventListener`, dependendo do tipo de evento que deseja tratar, a esta classe chamamos de **listener**,
2. O componente de interface registra um **listener** (tratador de eventos) através do método `addXXXListener(Listener)`, onde XXX é o tipo de evento, criado para tratar este tipo de evento. Ou seja, basta consultar a documentação do componente para sabermos quais tipos de eventos ele é capaz de tratar através dos **listeners** que ele pode registrar.
3. Internamente, quando o usuário executar aquela ação, se houver um **listener** (tratador de eventos) registrado para ela, então a JVM criará o objeto de evento específico e delegará o tratamento para o **listener** registrado.

Dentro da J2SE 1.4.2, as interfaces que definem as classes de eventos que podemos usar e tratar estão agrupadas nos pacotes: `java.awt.event` (Eventos Genéricos) e na `javax.swing.event` (Eventos Específicos de alguns componentes do SWING)

Implementando o tratamento de eventos.

Para se tratar um evento devemos escolher qual sua categoria, por exemplo, dentro de um objeto de desenho Canvas, podemos tratar o movimento do mouse. Uma vez escolhida a categoria, criamos a classe de interface gráfica com uma classe interna para manipular tal categoria de evento facilitando a manipulação dos componentes da classe de interface.

Essa classe interna pode implementar a interface listener do evento escolhido ou estender a classe adaptadora do evento escolhido.

Extendendo a classe de interface gráfica e classe interna que tratará um tipo de evento:

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

public class CanvasPanel extends JFrame {

    private Canvas areaDesenho;

    public CanvasPanel() {
        super("Desenho livre!");
        areaDesenho = new Canvas();
    }

    public void init() {
        areaDesenho.setForeground( Color.BLACK );
        areaDesenho.setBackground( Color.WHITE );
        areaDesenho.setSize( 400, 400 );
        // registra para o canvas, o listener capaz de tratar
        // os eventos de movimento do mouse
        areaDesenho.addMouseListener ( new MouseMotionHandler() );
        getContentPane().add ( areaDesenho );
        setSize( 410, 410 );
        setVisible(true);
    }

    // classe interna para tratar evento do mouse
    class MouseMotionHandler extends MouseMotionAdapter {

        // escolhido tratar o evento de movimento arrastar do mouse
        public void mouseDragged(MouseEvent me) {
            // recupera o objeto Graphics e desenha no Canvas
            // de acordo com a posicao do mouse
            areaDesenho.getGraphics().drawString("**", me.getX(), me.getY() );
        }

    }

    public static void main(String arg[]) {
        new CanvasPanel().init();
    }
}
```

Quando o evento é gerado na interface, ele é delegado para ser tratado dentro do objeto `MouseMotionHandler` através do método chamado de acordo com o evento daquela categoria.

Tratando eventos comuns.

Todos os componentes do SWING possuem um conjunto de eventos que estão disponíveis e podem ser tratados. Esses eventos comuns e mais genéricos podem ser tratados através dos **listeners**:

Evento	Listener	Adaptadora
ComponentEvent	ComponentListener	ComponentAdapter
FocusEvent	FocusListener	FocusAdapter
InputMethodEvent	InputMethodListener	InputMethodAdapter
KeyEvent	KeyListener	KeyAdapter
MouseEvent	MouseListener	MouseAdapter
MouseMotionEvent	MouseMotionListener	MouseMotionAdapter
MouseWheelEvent	MouseWheelListener	MouseWheelAdapter

Usamos no exemplo acima, um listener `MouseMotionHandler`, que foi construído através de uma classe adaptadora `MouseMotionAdapter`. Veja agora o mesmo **listener** no caso de ele implementar a interface `MouseMotionListener`.

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

public class CanvasPanel2 extends JFrame {
    private Canvas areaDesenho;
    public CanvasPanel() {
        super("Desenho livre!");
        areaDesenho = new Canvas();
    }
    public void init() {
        areaDesenho.setForeground( Color.BLACK );
        areaDesenho.setBackground( Color.WHITE );
        areaDesenho.setSize( 400, 400 );
        // registra para o canvas, o listener capaz de tratar
        // os eventos de movimento do mouse
        areaDesenho.addMouseListener ( new MouseMotionHandler2() );
        getContentPane().add ( areaDesenho );
        setSize( 410, 410 );
        setVisible(true);
    }
    // classe interna para tratar evento do mouse
    class MouseMotionHandler2 implements MouseMotionListener {
        public void mouseMoved(MouseEvent me) {
            // sem implementacao
        }

        // escolhido tratar o evento de movimento arrastar do mouse
        public void mouseDragged(MouseEvent me) {
            // recupera o objeto Graphics e desenha no Canvas
            // de acordo com a posicao do mouse
            areaDesenho.getGraphics().drawString("X", me.getX(), me.getY() );
        }
    }
    public static void main(String arg[]) {
        new CanvasPanel2().init();
    }
}
```

Vimos que podemos escrever um **listener** de duas formas, mas a melhor forma ainda é entender as classes **adaptadoras** quando houverem, e implementar a **interface** de um **listener** quando a adaptadora não houver.

Tratando eventos de janelas.

As classes de janelas, JWindow, JDialog e JFrame, possuem alguns tipos de eventos para manipulação de seus estados de aberta, fechada, maximizada, minimizada, etc.

Evento	Listener	Adaptadora
WindowEvent	WindowListener	WindowAdapter
WindowFocusEvent	WindowFocusListener	WindowFocusAdapter
WindowStateEvent	WindowStateListener	WindowStateAdapter

Para fechar uma janela, devemos implementar um tratador de eventos para os eventos da categoria WindowEvent.

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

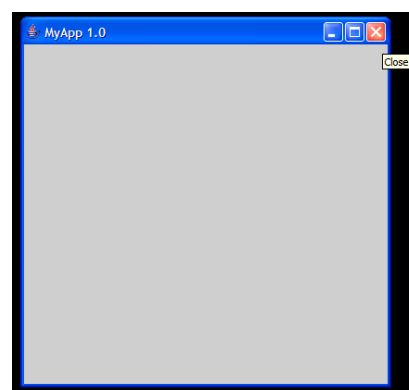
public class MyApp extends JFrame {

    public MyApp () {
        super("MyApp 1.0");
    }

    public void init() {
        setForeground(Color.BLACK);
        setBackground(Color.GRAY);

        // podemos usar uma classe interna sem nome
        // AnonymousInnerClass para tratar eventos,
        // muitas IDEs farao assim ao inves de definir
        // classes completas.
        addWindowListener( new WindowAdapter () {
            // quando clicar no X para fechar, encerrara a aplicacao.
            public void windowClosing(WindowEvent we) {
                System.exit( 0 );
            }
        });
        setSize(400,400);
        setLocation(200,150);
        setVisible(true);
    }

    public static void main(String arg[]) {
        new MyApp().init();
    }
}
```



Vimos que podemos criar classes internas dentro da definição de método, essa prática não é indicada no dia a dia para outras funcionalidades, exceto para o tratamento de eventos simples de interface gráfica, por serem de difícil leitura, entendimento e ir um pouco contra as boas práticas de construção de software.

Até aqui entendemos como os eventos de interface funcionam e a três formas de construir classes que manipulam eventos. A partir de agora vamos ver como manipular eventos dos principais componentes gráficos do SWING.

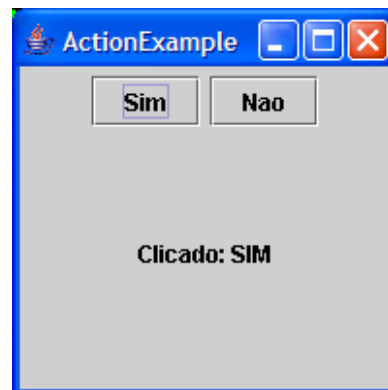
Tratando eventos de botões e menus.

Para tratar os eventos de botões (JButton, JRadioButton, JCheckBox, JToggleButton) ou itens de um menu (JMenuItem), criaremos as classes que manipulam esses eventos a partir da interface `java.awt.event.ActionListener`, esta interface tem somente um método `actionPerformed(ActionEvent e)`, por isso não existe uma classe adaptadora para este tipo de evento.

Evento	Listener	Adaptadora
ActionEvent	ActionListener	Não Disponível

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

public class ActionExample extends JFrame {
    private JButton sim, nao;
    private JLabel mensagem;
    private JPanel buttons1, buttons2;
    public ActionExample() {
        super("ActionExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        mensagem = new JLabel("Clicado: ");
    }
    public void init() {
        // registrando listener para os botoes.
        ButtonHandler buttonHandler = new ButtonHandler();
        sim.addActionListener( buttonHandler );
        nao.addActionListener( buttonHandler );
        // montando tela
        getContentPane().setLayout( new GridLayout(2,1) );
        setSize(200,200);
        setLocation(300,200);
        buttons1 = new JPanel();
        buttons1.add( sim );
        buttons1.add( nao );
        buttons2 = new JPanel();
        buttons2.add( mensagem );
        getContentPane().add( buttons1 );
        getContentPane().add( buttons2 );
        // classe anonima para tratar evento de fechar a tela
        addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        }); // fim classe anonima
        setVisible(true);
    }
    // classe interna para tratar eventos dos botoes
    class ButtonHandler implements ActionListener {
        // quando um dos botoes for clicado gerando o evento de Action
        // podemos tratá-lo e descobrir de qual botão veio.
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == sim ) {
                mensagem.setText("Clicado: SIM");
            } else if ( ae.getSource() == nao ) {
                mensagem.setText("Clicado: NAO");
            }
        }
    }
    public static void main(String arg[]) {
        new ActionExample().init();
    }
}
```



Exemplo de eventos em menus:

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

public class MenuExample extends JFrame {
    private JMenuItem abrir, novo, salvar, sair;
    private JMenu arquivo;
    private JMenuBar menuBar;

    public MenuExample() {
        super("Menu Example");

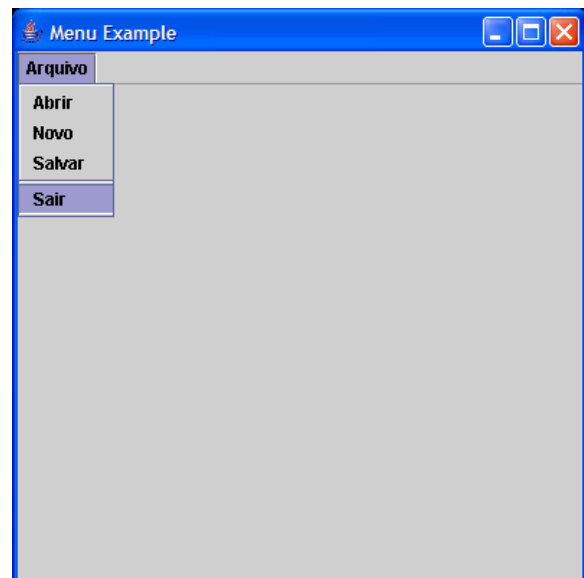
    }

    public void init() {
        setSize( 400, 400 );
        setLocation( 300, 200 );
        // construindo objetos
        abrir = new JMenuItem("Abrir");
        novo = new JMenuItem("Novo");
        salvar = new JMenuItem("Salvar");
        sair = new JMenuItem("Sair");
        arquivo = new JMenu("Arquivo");
        //registrando listener
        MenuHandler mh = new MenuHandler();
        abrir.addActionListener( mh );
        novo.addActionListener( mh );
        salvar.addActionListener( mh );
        sair.addActionListener( mh );
        // construindo menu arquivo
        arquivo.add ( abrir );
        arquivo.add ( novo );
        arquivo.add ( salvar );
        arquivo.addSeparator();
        arquivo.add ( sair );
        // construindo menu
        menuBar = new JMenuBar();
        menuBar.add( arquivo );
        setJMenuBar( menuBar );
        setVisible(true);
        addWindowListener( new WindowHandler() );
    }

    // listener capaz de tratar action event
    class MenuHandler implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == sair ) {
                System.exit(0);
            } else {
                System.out.println( ae );
            }
        }
    }

    // listener capaz de tratar window event
    class WindowHandler extends WindowAdapter {
        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    }

    public static void main(String args[]) {
        new MenuExample().init();
    }
}
```



Inserindo teclas de atalhos.

Podemos tratar eventos de atalhos em alguns componentes, e geralmente são usados para facilitar a usabilidade da interface.

Os objetos de menu que provem da class JMenuItem, possuem um método `setAccelerator(KeyStroke)` que registra uma tecla de atalho o item de menu, os elementos JMenu possuem um método `setMnemonic(char)`, que seta a tecla de atalho para o menu.

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

public class MenuExample2 extends JFrame {

    private JMenuItem abrir, novo, salvar, sair;
    private JMenu arquivo;
    private JMenuBar menuBar;

    public MenuExample2() {
        super("Menu Example");
    }

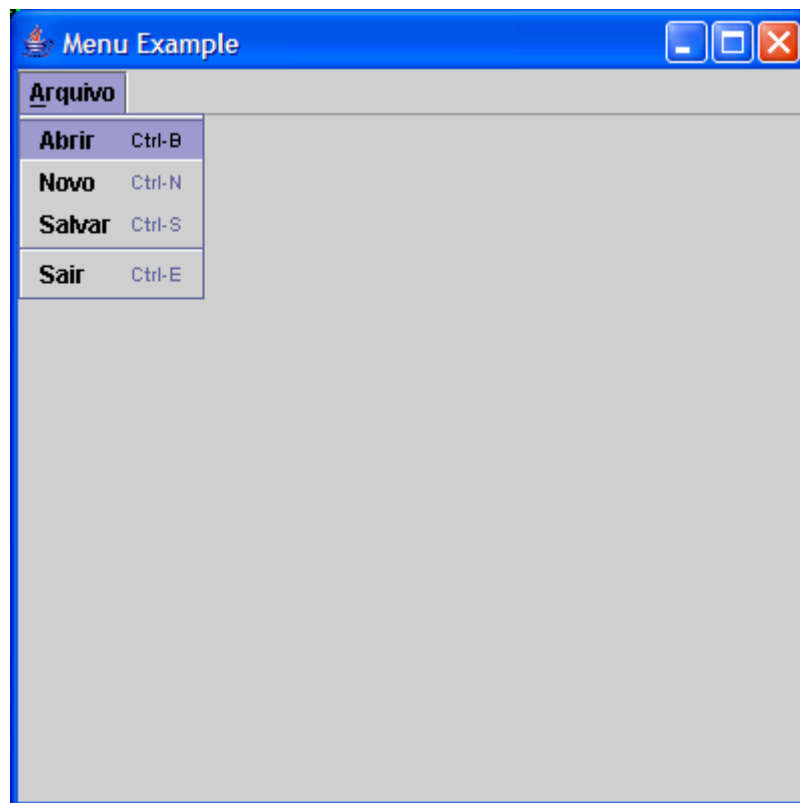
    public void init() {
        setSize( 400, 400 );
        setLocation( 300, 200 );
        // construindo objetos
        abrir = new JMenuItem("Abrir");
        novo = new JMenuItem("Novo");
        salvar = new JMenuItem("Salvar");
        sair = new JMenuItem("Sair");
        arquivo = new JMenu("Arquivo");
        //registrando listener
        MenuHandler2 mh = new MenuHandler2();
        abrir.addActionListener( mh );
        novo.addActionListener( mh );
        salvar.addActionListener( mh );
        sair.addActionListener( mh );
        // construindo menu arquivo
        arquivo.add ( abrir );
        arquivo.add ( novo );
        arquivo.add ( salvar );
        arquivo.addSeparator();
        arquivo.add ( sair );
        // setando teclas de atalho
        arquivo.setMnemonic('A');
        abrir.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_B, InputEvent.CTRL_MASK,
false) );
        novo.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_N, InputEvent.CTRL_MASK,
false) );
        salvar.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_S, InputEvent.CTRL_MASK,
false) );
        sair.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_E, InputEvent.CTRL_MASK,
false) );

        // construindo menu
        menuBar = new JMenuBar();
        menuBar.add( arquivo );
        setJMenuBar( menuBar );
        setVisible(true);
        addWindowListener( new WindowHandler2() );
    }

    // listener capaz de tratar action event
    class MenuHandler2 implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == sair ) {
                System.exit(0);
            } else {
                System.out.println( ae );
            }
        }
    }
}
```

```
    }  
}  
  
// listener capaz de tratar window event  
class WindowHandler2 extends WindowAdapter {  
    public void windowClosing(WindowEvent we) {  
        System.exit(0);  
    }  
}  
  
public static void main(String args[]) {  
    new MenuExample2().init();  
}  
}
```

Usamos ALT+A para abrir o menu arquivo, e se quisermos teclamos CTRL+LETRA para acionar um dos itens do menu.



Tratando eventos de textos.

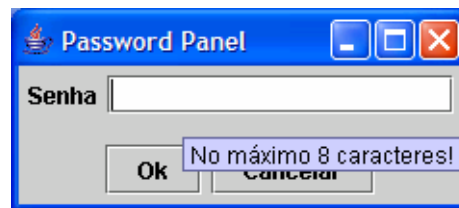
Os eventos de texto geralmente estão associados a validação de caracteres que estão sendo inseridos durante a digitação. O mais comum é evitar que determinados caracteres sejam inseridos ou que o tamanho da cadeia de texto não ultrapasse um valor estipulado. Podemos usar dois tipos de listener para isso:

Evento	Listener	Uso
CaretEvent	CaretListener	Tratar a posição do cursor
ActionEvent	ActionListener	Identificar se o ENTER foi acionado

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

public class PasswordPanel extends JPanel {
    private JLabel lblSenha;
    private JPasswordField txtSenha;
    private JButton ok, cancel;
    public PasswordPanel() {
        lblSenha = new JLabel("Senha");
        txtSenha = new JPasswordField("",16);
        ok = new JButton("Ok");
        cancel = new JButton("Cancelar");
    }
    public void init() {
        setLayout( new GridLayout(2,1) );
        FlowLayout centro = new FlowLayout( FlowLayout.CENTER );
        // usando Painéis auxiliares do tipo FlowLayout para alinha a esquerda
        // e poder inserir mais um componente por linha do grid
        JPanel auxSenha = new JPanel( centro );
        auxSenha.add( lblSenha );
        auxSenha.add( txtSenha );
        JPanel auxBotoes = new JPanel( centro );
        auxBotoes.add( ok );
        auxBotoes.add( cancel );
        add( auxSenha );
        add( auxBotoes );
        // adicionando tratamento de evento
        txtSenha.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                if ( ae.getSource() == txtSenha ) {
                    String texto = txtSenha.getText();
                    if ( texto.length() > 8 ) {
                        texto = texto.substring(0,7);
                        txtSenha.setText(texto);
                    }
                }
            }
        } );
    }

    public static void main(String arg[]) {
        PasswordPanel passwordPanel = new PasswordPanel();
        passwordPanel.init();
        JFrame frame = new JFrame("Password Panel");
        frame.getContentPane().add( passwordPanel );
        frame.pack();
        frame.setVisible( true );
    }
}
```



Tratando eventos de listas.

Os eventos das listas geralmente se referem a identificar qual ou quais elementos de uma lista na interface foram selecionados. Esses eventos são muito utilizados para melhorar a intuitividade da interface gráfica.

Dentro do SWING, usamos as seguinte categorias para implementar a manipulação de tais eventos:

Evento	Listener	Uso
ActionEvent	ActionListener	Inserção de novo elemento
ItemEvent	ItemListener	Seleção de um elemento
ListSelectionEvent	ListSelectionListener	Seleção de um elemento

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

public class ListExample extends JFrame {
    private JList lista1, lista2;
    private JPanel panel1, panel2, panel;
    private DefaultListModel modelLista1, modelLista2;

    public ListExample() {
        super("List Example!");
        modelLista1 = new DefaultListModel();
        modelLista2 = new DefaultListModel();
        lista1 = new JList( modelLista1 );
        lista2 = new JList( modelLista2 );
        panel1 = new JPanel();
        panel2 = new JPanel();
        panel = new JPanel();

    }

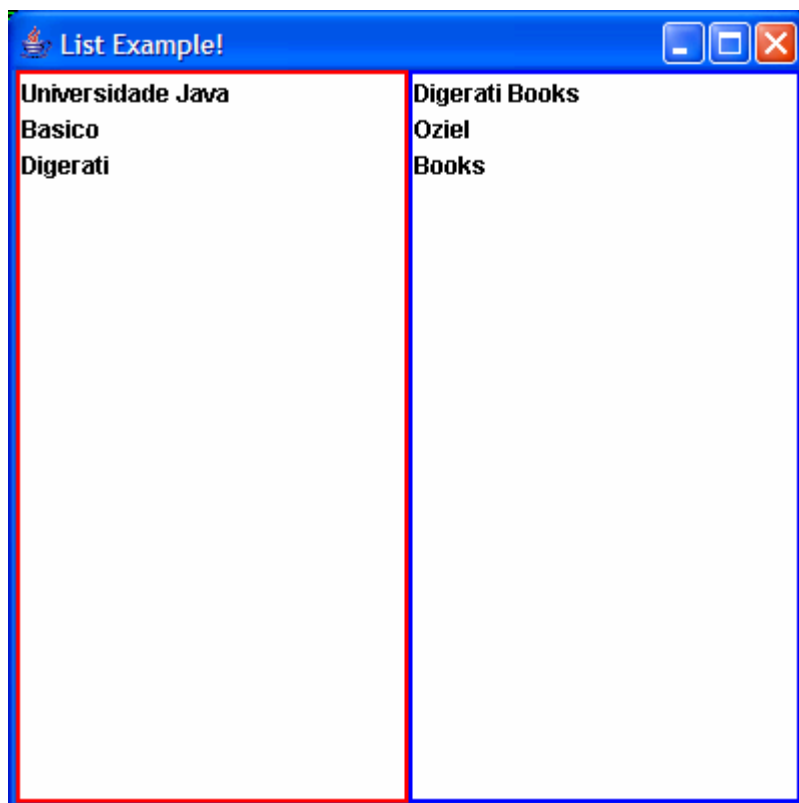
    public void init() {
        setLocation(200,200);
        setSize(400,400);
        modelLista1.addElement( new String("Universidade Java") );
        modelLista1.addElement( new String("Basico") );
        modelLista1.addElement( new String("Digerati") );
        modelLista2.addElement( new String("Digerati Books") );
        modelLista2.addElement( new String("Oziel") );
        modelLista2.addElement( new String("Books") );
        lista1.setSelectionMode( ListSelectionModel.SINGLE_INTERVAL_SELECTION );
        lista2.setSelectionMode( ListSelectionModel.SINGLE_INTERVAL_SELECTION );
        panel1.setLayout( new BorderLayout() );
        panel1.add( lista1, BorderLayout.CENTER );
        panel1.setBorder( new LineBorder( Color.RED, 2 ) );
        panel2.setLayout( new BorderLayout() );
        panel2.add( lista2, BorderLayout.CENTER );
        panel2.setBorder( new LineBorder( Color.BLUE, 2 ) );
        panel.setLayout( new GridLayout(1,2) );
        panel.add( panel1 );
        panel.add( panel2 );
        ListHandler lh = new ListHandler();
        lista1.addMouseListener( lh );
        lista2.addMouseListener( lh );
        ListSelectionHandler lsh = new ListSelectionHandler();
        lista1.addListSelectionListener( lsh );
        lista2.addListSelectionListener( lsh );
        getContentPane().add( panel, BorderLayout.CENTER );
        setVisible(true);
    }
}
```

//... continua

```
class ListHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent me) {
        Object obj = null;
        if ( me.getSource() == lista1 ) {
            obj = lista1.getSelectedValue();
        } else if ( me.getSource() == lista2 ) {
            obj = lista2.getSelectedValue();
        }
        System.out.println(obj);
    }
}

class ListSelectionHandler implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent le) {
        if ( le.getSource() == lista1 ) {
            Object[] selected = lista1.getSelectedValues();
            for (int i=0; i<selected.length; i++) System.out.println( selected[i]
);
        } else if ( le.getSource() == lista2 ) {
            Object[] selected = lista2.getSelectedValues();
            for (int i=0; i<selected.length; i++) System.out.println( selected[i]
);
        }
    }
}

public static void main(String args[]) {
    new ListExample().init();
}
}
```



Experimente selecionar um elemento ou múltiplos elementos usando CTRL, você verá que esses eventos são tratados de formas distintas, e cada um pelo seu listener.

Tratando eventos de combos.

Os eventos das caixas de seleção (ComboBox) geralmente se referem a identificar qual elemento foi selecionado na lista, ou identificar se um novo elemento foi inserido.

Evento	Listener	Uso
ActionEvent	ActionListener	Inserção de novo elemento
ItemEvent	ItemListener	Seleção de um elemento

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

```
public class JComboBoxExample extends JFrame {

    private JComboBox nomes;
    private DefaultComboBoxModel nomesModel;

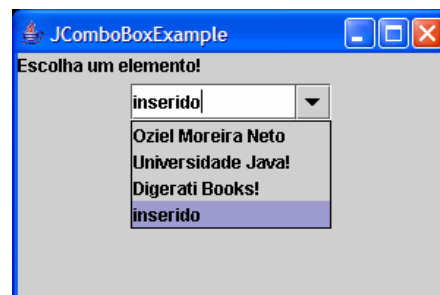
    public JComboBoxExample () {
        super("JComboBoxExample");
        nomesModel = new DefaultComboBoxModel ();
        nomes = new JComboBox( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        nomesModel.addElement( new String("Oziel Moreira Neto") );
        nomesModel.addElement( new String("Universidade Java!") );
        nomesModel.addElement( new String("Digerati Books!") );
        // tornando o combobox editavel.
        nomes.setEditable( true );
        JPanel auxNomes = new JPanel();
        auxNomes.add( nomes );
        // adiciona evento ao editor para nova entrada
        nomes.getEditor().addActionListener( new ActionListener() );
        // adiciona evento para selecao de elemento
        nomes.addItemListener( new ItemHandler() );
        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Escolha um elemento!"), BorderLayout.NORTH );
        setVisible( true );
    }

    class ActionHandler implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            Object obj = nomes.getEditor().getItem();
            nomesModel.addElement ( obj );
            System.out.println( obj );
        }
    }

    class ItemHandler implements ItemListener {
        public void itemStateChanged(ItemEvent ie) {
            System.out.println( nomes.getSelectedIndex() );
        }
    }

    public static void main(String arg[]) {
        new JComboBoxExample().init();
    }
}
```



Experimente selecionar um elemento e inserir um elemento, você verá que esses eventos são tratados de formas distintas, e cada um pelo seu listener.

Tratando eventos de tabelas.

Podemos manipular alguns eventos direto pelo JTable.

Listener	Evento	Uso
MouseListener	MouseEvent	Recuperar qual elemento o mouse clicou
ListSelectionListener	ListSelectionEvent	Seleção de elementos da tabela

O TableModel também possui um listener específico e bem completo para o tratamento de eventos para tabelas.

Listener	Evento	Uso
TableModelListener	TableModelEvent	Manipular insercao, selecao e atualização da tabela

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

public class JTableExample extends JFrame {

    private JTable nomes;
    private DefaultTableModel nomesModel;

    public JTableExample () {
        super("JTableExample");
        String[] cols = {"Codigo","Texto"};
        nomesModel = new DefaultTableModel ( cols , 3 );
        nomes = new JTable ( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        String[] row1 = {"1","Oziel Moreira Neto"};
        String[] row2 = {"2","Universidade Java!"};
        String[] row3 = {"3","Digerati Books!"};
        nomesModel.insertRow(0, row1 );
        nomesModel.insertRow(1, row2 );
        nomesModel.insertRow(2, row3 );
        JScrollPane auxNomes = new JScrollPane( nomes, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );
        // adiciona o listener para o model
        nomesModel.addTableModelListener( new TableModelHandler() );
        // adiciona o listener para o jlist
        nomes.addMouseListener( new MouseTableHandler() );

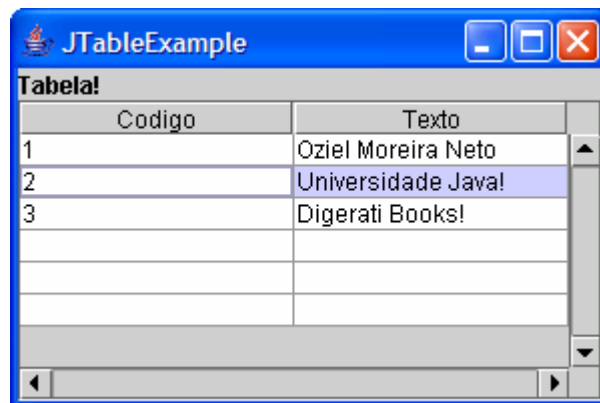
        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Tabela!"), BorderLayout.NORTH );
        setVisible( true );
    }

    //... continua
}
```

```
class MouseTableHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent me) {
        int row = nomes.getSelectedRow();
        int column = nomes.getSelectedColumn();
        System.out.println( nomesModel.getValueAt(row, column) );
    }
}

class TableModelHandler implements TableModelListener {
    public void tableChanged(TableModelEvent te) {
        int row = te.getFirstRow();
        int column = te.getColumn();
        System.out.println( nomesModel.getValueAt(row, column) );
    }
}

public static void main(String arg[]) {
    new JTableExample ().init();
}
}
```



Experimente selecionar um elemento e alterar um elemento, você verá que esses eventos são tratados de formas distintas, e cada um pelo seu listener.

Discussão

Numa aplicação devemos separar claramente as responsabilidades, em classes de dados (representam o modelo de negócio), classes de serviços (aquelas que acessam recursos, como arquivos, bancos, etc) das classes de GUI?

Porque devemos fazer essa separação?

Significa então que é nas classes de Eventos definidas dentro das GUIs que faremos as chamadas as classes de negócio e classes de serviço?

Exercícios

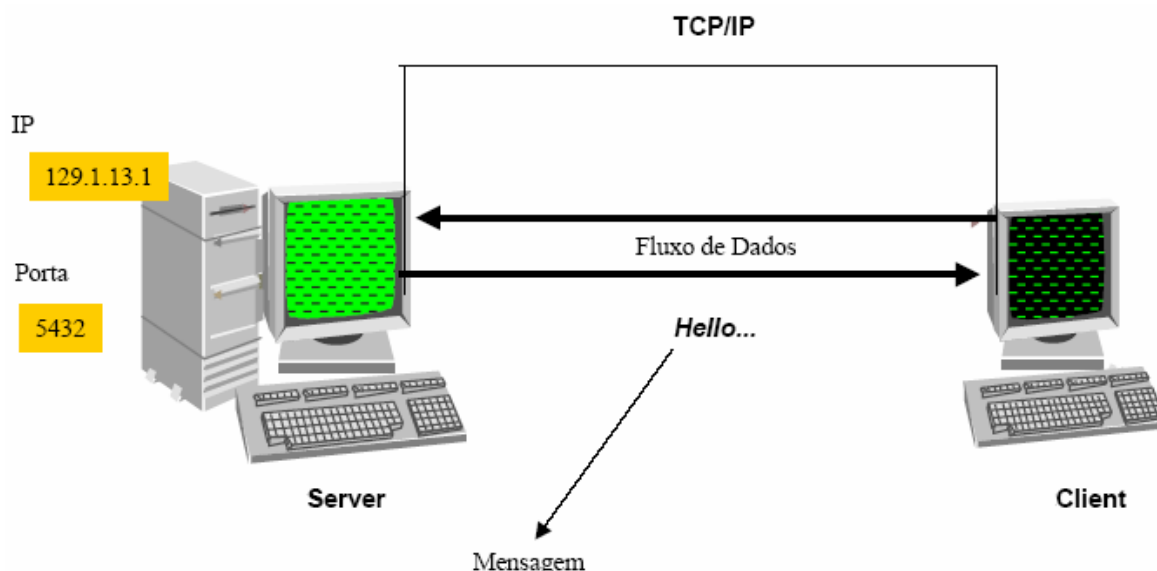
10. Programação para Redes (Network Programming)

A tecnologia Java, usando a API **java.net** possibilita a comunicação distribuída no modelo client/server através do uso de Datagram Sockets e Stream Sockets.

Com Stream Sockets (Soquetes de Fluxo), podemos estabelecer conexão entre um cliente e servidor, e enquanto perdurar a conexão podemos trocar mensagens, este fluxo de mensagem é controlado, caso haja algum problema entre as mensagens enviadas e recebidas, elas podem ser corrigidas. Este serviço é chamado orientado a conexão, o protocolo utilizado é TCP.

Já os Datagram Sockets (Soquetes Datagramas), as mensagens são transmitidas em pacotes, entretanto, o fluxo de troca de mensagens não são controlados, isto quer dizer, que eles podem ser perdidos (nunca chegar ao seu destino, por exemplo) entre outras coisas. Este serviço é chamado serviço sem conexão, o protocolo utilizado é UDP.

Ambos os dois modelos devem ser suportados pelos serviços de rede TCP/IP, onde um servidor deve possuir um endereço de rede IPV4 ou IPV6, onde serão enumeradas portas para trocas de mensagens.



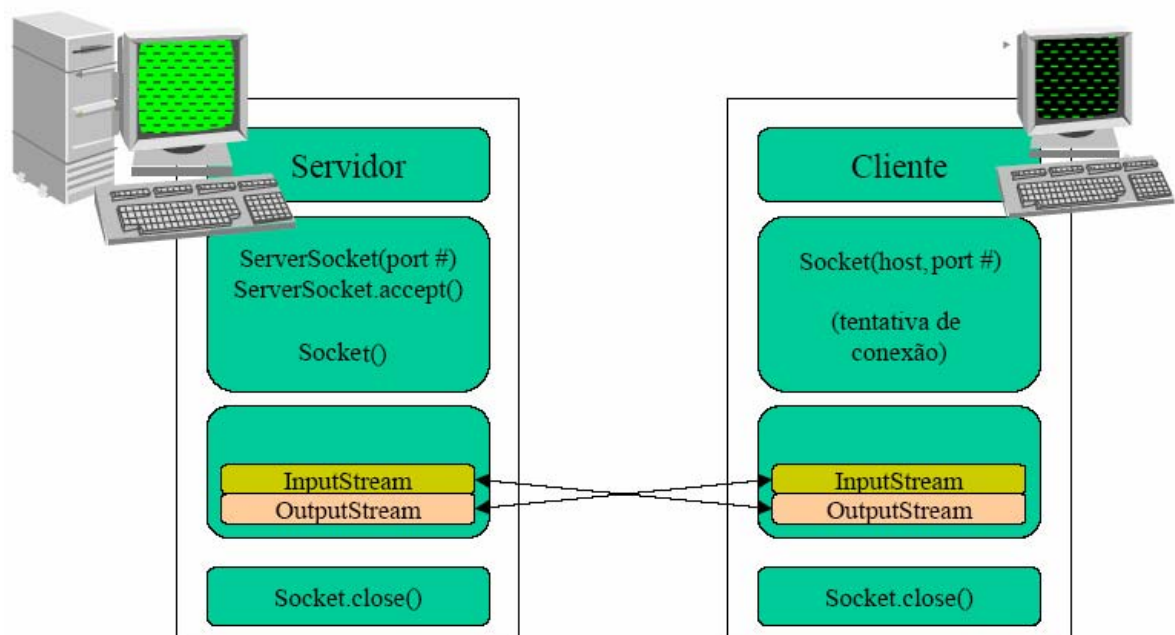
Então um Socket, é determinado por par (IP:PORT). Os endereço IP dependem da implementação. IPV4: 255.255.255.255 e IPV6: AA:FF:AA:FF:AA:FF

As portas variam de 0 a 65535. Sendo que as portas 0 até a 1024 são reservadas para o sistema operacional e serviços de rede.

Trabalhando com Stream Sockets

Numa máquina que suporte os serviços TCP/IP, registramos uma porta de serviço que atuará como Servidor. Neste caso, usamos o objeto `java.net.ServerSocket`.

O objeto `java.net.ServerSocket`, no servidor faz o bind (registro) de uma porta que é capaz de receber conexões dos clientes, e quando a conexão é estabelecida através do método `java.net.ServerSocket.accept()`, é retornado um objeto `java.net.Socket`, onde podemos usar um `java.io.InputStream` ou `java.io.OutputStream` para a troca de dados.



No cliente, criamos um objeto `java.net.Socket`, passando o IP e PORTA de serviço do `java.net.ServerSocket` que está registrado no servidor, se a conexão for estabelecida, ficam disponíveis dois objetos de Streamers, um `java.io.InputStream` e outro `java.io.OutputStream` para troca de dados entre o cliente e o servidor.

Codificando um Servidor e um Cliente simples

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

public class SimpleServer {

    public static void main(String args[]) {

        ServerSocket s= null;
        try {
            s = new ServerSocket(5432);
        } catch (IOException e) {
            e.printStackTrace();
        }
        for (;;) {
            try {
                Socket s1 = s.accept();
                OutputStream slout = s1.getOutputStream();
                DataOutputStream dos = new DataOutputStream(slout);
                // Envia a mensagem
                dos.writeUTF("Hello Net World!");
                dos.close();
                s1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Executando: # java SimpleServer

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

public class SimpleClient {

    public static void main(String args[]) {
        try {
            // Endereço IP é Localhost
            Socket s1 = new Socket("127.0.0.1", 5432);
            InputStream is = s1.getInputStream();
            DataInputStream dis = new DataInputStream(is);
            System.out.println(dis.readUTF());
            dis.close();
            s1.close();
        } catch (ConnectException connExc) {
            System.err.println("Could not connect to the server.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Em outro terminal: Executando: # java SimpleClient

Discussão

Que tipos de sistemas, aplicações e programas podemos desenvolver usando a comunicação em rede?

Que tipos de dados podem ser passados pela rede? Depende do Stream usado?

Em aplicações que rodarão em redes heterogêneas, como fica a comunicação e o dados?

Exercícios

11. Programação multi-tarefa (MultiThreading Programming)

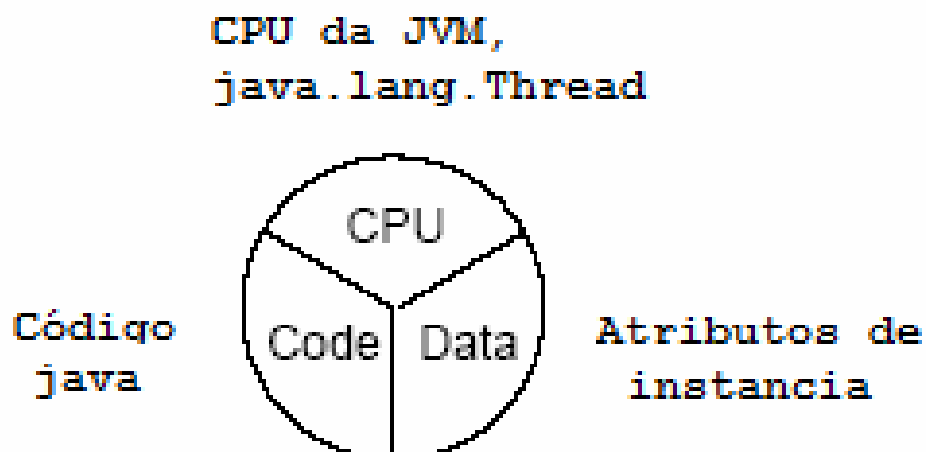
Threads são linhas ou fluxos de execução em paralelo de programa. Também conhecida como programação concorrente ou Multithreading programming.

A plataforma Java permite que seus componentes de runtime executem ao mesmo tempo várias Threads ou Linhas de Execução.

O processo de execução de mais que duas Threads ao mesmo tempo é chamado de Multithreading.

As Threads são enfileiradas e escalonados pelo Scheduler da JVM. Quando o computador tem somente uma CPU física, é executada somente uma Thread por ciclo de CPU, e quando o computador físico tem mais de uma CPU física, são executadas mais de uma Thread por ciclo de CPU.

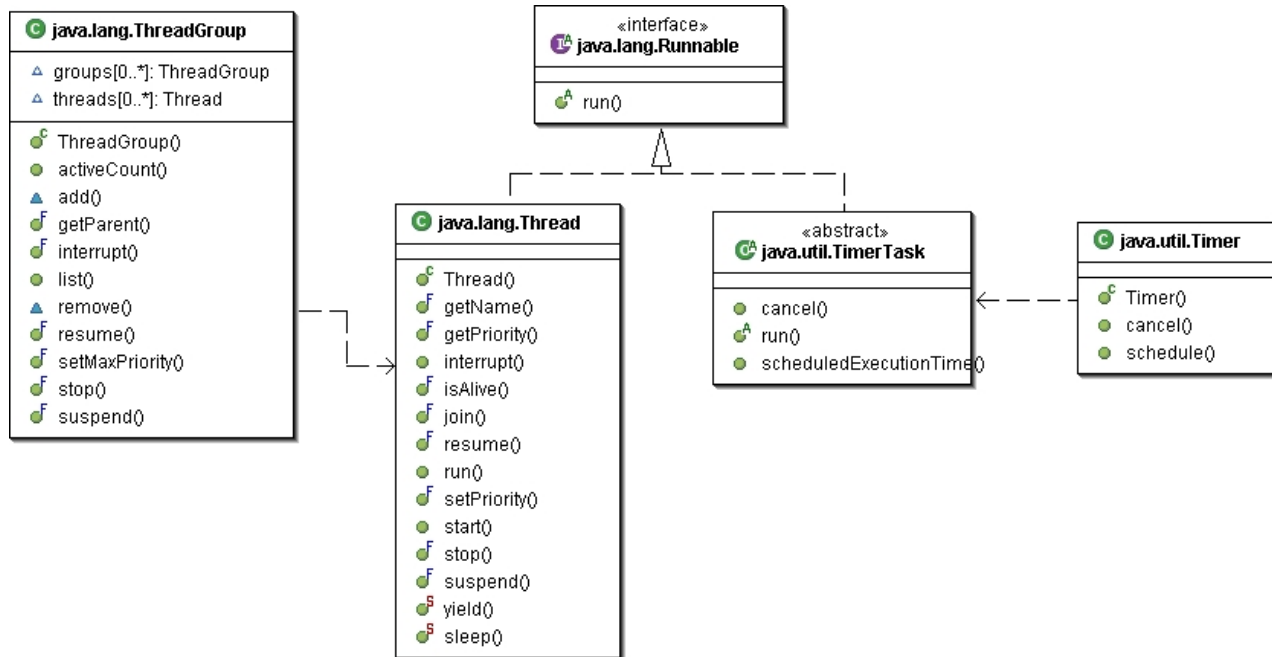
Dentro da computação moderna e orientada a objetos, definimos uma Thread como sendo uma linha de execução que contem um trecho de código, que usa uma área de memória e ocupa uma porção de tempo de uma CPU.



Java Thread API: Classes e interfaces

Interfaces: `java.lang.Runnable`

Classes: `java.lang.Thread`, `java.util.TimerTask`, `java.util.Timer`, `java.lang.ThreadGroup`.



Estas são as classes principais que utilizamos quando estamos desenvolvendo aplicações Multithreading.

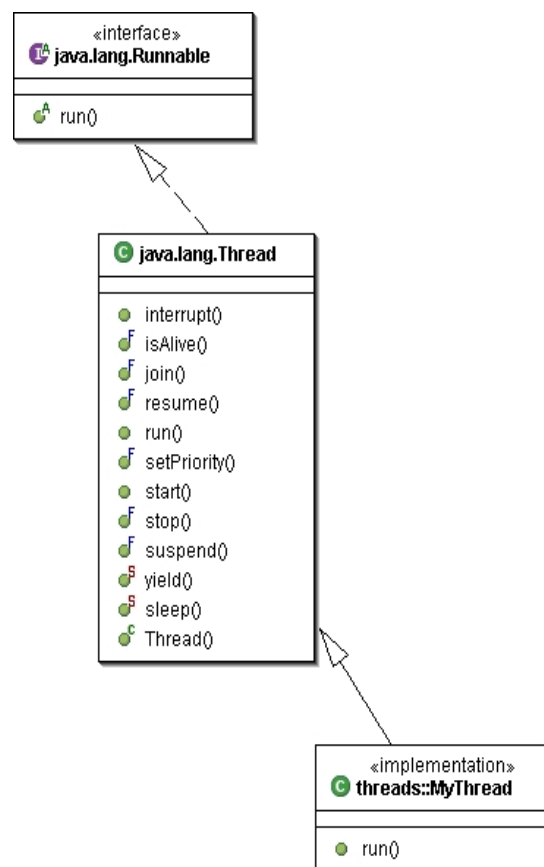
As classes `Thread`, `ThreadGroup` e a interface `Runnable` são usadas quando queremos construir aplicações que manipulem múltiplas execuções em paralelo, e que geralmente tem tempo de execução definido e finito.

As classes `Timer` e `TimerTask` são usadas quando queremos construir aplicações que manipulem processos e execuções que geralmente tem tempo de execução cíclico e finito.

Construindo Threads

Podemos construir e executar uma Thread de duas maneiras distintas, seguindo os diagramas abaixo:

1. Extendendo a Classe java.lang.Thread



Criamos uma classe Java, que estende a classe `java.lang.Thread`.

Nesta classe devemos reescrever o método `public void run()`, e neste método implementamos o comportamento (código) que queremos que seja executado como uma linha de execução independente.

```
//MyThread.java

public class MyThread extends Thread {

    // atributos de instancia
    // que será usado nesta Thread

    public void run() {

        // código que será executado
        // em paralelo com outras
        // Threads

    }

}
```

Para executar um objeto da classe `MyThread` como uma Thread, fazemos:

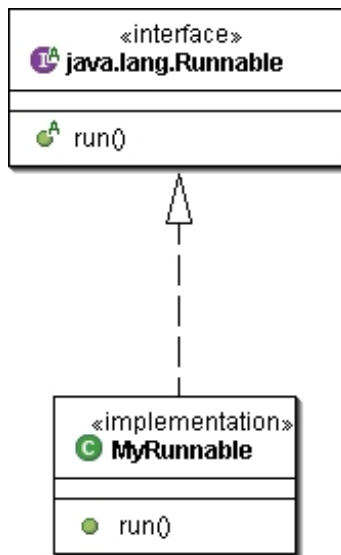
```
public class ThreadTest {

    public static void main(String a[]) {
        MyThread mt = new MyThread();
        mt.start();
    }

}
```

A classe `Thread` representa a CPU virtual da JVM, e quando criamos um objeto que estende esta CPU, estamos especializando uma instância da CPU virtual, que nos permite executar o trecho de código definido no método `run()` em uma nova linha de execução, e no método `start()`, agendamos o início da execução.

2. Implementando a interface java.lang.Runnable



Criamos uma classe Java, que implementa a `java.lang.Runnable`.

Nesta classe devemos reescrever o método `public void run()`, e neste método implementamos o comportamento (código) que queremos que seja executado como uma linha de execução independente.

```
// MyRunnable.java

public class MyRunnable extends Runnable {

    // atributos de instancia
    // que será usado nesta Thread

    public void run() {

        // código que será executado
        // em paralelo com outras
        // Threads

    }

}
```

Para executar um objeto da classe `MyRunnable` como uma `Thread`, fazemos:

```
public class RunnableTest {

    public static void main(String a[]) {
        MyRunnable mr = new MyRunnable ();
        Thread cpu = new Thread( mr );
        cpu.start();
    }

}
```

A interface `java.lang.Runnable` marca este objeto como sendo um `Runnable`, entretanto a CPU virtual que deve ser alocada para esta `Thread` deve ser inicializada, e para isso criamos uma instancia do objeto `Thread`, e no método `start()`, agendamos o início da execução.

Manipulando Threads

Existe um conjunto principal de métodos presentes na classe `java.lang.Thread`, que nos permite manipular o estado das Threads.

Método	Descrição
<code>void start()</code>	Elege uma Thread para execução. O Scheduler vai executá-la quando houver CPU disponível.
<code>void interrupt()</code>	Indica que a Thread em execução deve ser seguramente finalizada.
<code>static void sleep(long)</code>	Suspende a execução de uma Thread pelo tempo especificado, o tempo é um valor em <i>milisegundo</i> . Principal vantagem: Economiza processamento de CPU. Aplicação: Controle Diversos, processo de sincronização, etc.
<code>static void yield()</code>	Suspende a execução da Thread corrente e dá a oportunidade de execução para a thread de maior prioridade da fila de execução.
<code>void stop()</code>	Interrompe abruptamente a execução da thread. Método não seguro.
<code>boolean isAlive()</code>	Verifica se a Thread está ativa (se foi iniciada e ainda seu processamento não terminou).

Podemos ainda manipular as prioridades da Threads, fazendo com que alguns processos ocupem mais tempo de CPU que outros.

Método	Descrição
<code>setPriority()</code>	Ajusta o valor da prioridade da Thread
<code>getPriority()</code>	Recupera o valor prioridade da Thread

Como a JRE foi desenhada para ser portátil, o valor das prioridades são dependentes de plataforma, e não devemos usar número absolutos, devemos usar as constantes de valor da classe Thread

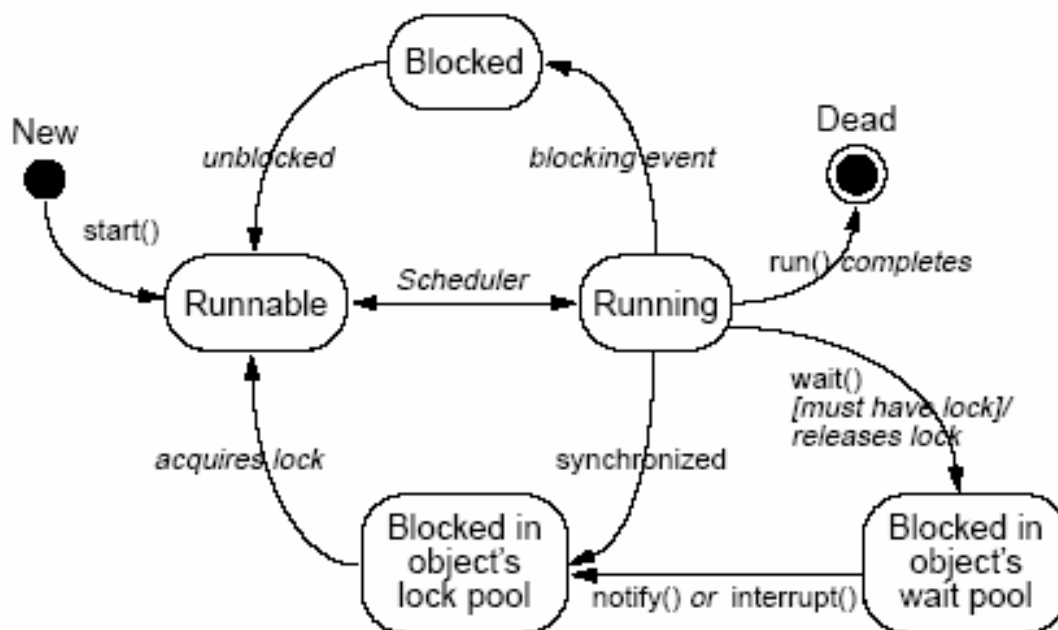
Atributo	Descrição
<code>Thread.MAX_PRIORITY</code>	Representa o valor de prioridade máxima
<code>Thread.NORM_PRIORITY</code>	Representa o valor de prioridade normal
<code>Thread.MIN_PRIORITY</code>	Representa o valor de prioridade mínima

Máquina de estados

Para garantir a escalabilidade das aplicações Multithreading, a estabilidade da JRE e o escalonamento dos processos que nossas aplicações estão executando, devemos conhecer a máquina de estado das Threads.

Toda vez usamos um dos métodos da classe Thread, estamos alterando o estado da Thread, e algumas seqüências de alterações de estados são proibidas ou não indicadas.

Na figura abaixo, vemos a máquina de estados.



O ciclo superior da máquina de estados, é o ciclo padrão de seqüência de execução, onde uma Thread vai ser interrompida por que a CPU da máquina está sendo compartilhada por outros processos.

O ciclo inferior da máquina de estados, é usado pelo scheduler quando sincronizamos o acesso á métodos dos objetos, controlando concorrência.

O “object wait pool” desenhado fora dos ciclos padrão de estados, é uma extensão do mecanismo de sincronismo, e usamo-lo quando, além de sincronizar o acesso á métodos, queremos coordenar a ordem de execução de uma ou mais Threads, usando o “object wait lock”.

A implementação do controle de ordem de execução de Threads não é trivial, e depende da arquitetura envolvida.

Controlando a concorrência.

Quando construímos aplicações multithreading, devemos nos preocupar com a concorrência de acesso á objetos compartilhados entre as threads.

A forma mais simples de controlar a concorrência, é usar o controle de sincronismo e semáforos presentes na linguagem java:

- usando o modificador de método `synchronized`;
- usando um bloco de código `synchronized (obj) { };`

Dessa forma toda e qualquer Thread que desejar executar o método, ou aquele bloco de código, antes ela deverá conseguir adquirir o lock do objeto, e se houver outra Thread em execução neste método ou bloco, o Scheduler da JVM vai enfileirar o acesso ao método ou bloco sincronizado, impedindo que duas threads acessem essas áreas sincronizadas ao mesmo tempo.

```
public class SyncStack {  
    private Stack stack;  
  
    public SyncStack() {  
        stack = new Stack();  
    }  
  
    public synchronized void put(Object obj) {  
        stack.add( obj );  
    }  
  
    public Object get(int index) {  
        Object obj = null;  
        synchronized ( stack ) {  
            obj = stack.get( index );  
        }  
        return obj;  
    }  
  
    public synchronized void clear() {  
        stack.removeAllElements();  
    }  
}
```

Vemos na classe acima:

1. os métodos que foram sincronizados usando o modificador **synchronized**:

```
public synchronized void put(Object obj) {  
    public synchronized void clear() {
```

2. o bloco de código sincronizado:

```
synchronized ( stack ) {  
    obj = stack.get( index );  
}
```

Controlando a execução (estados de espera).

Vários mecanismos existentes podem parar a execução de uma Thread temporariamente.

Seguindo este tipo de suspensão, a execução pode ser retomada a partir do ponto de onde parou, aumentando o tempo de execução somente.

O método ***sleep()*** foi descrito anteriormente e foi usado para parar uma Thread por um período de tempo determinado.

Às vezes é apropriado suspender a execução de uma Thread indefinidamente até que outra Thread libere a execução.

Um par de métodos está disponível na API para isto. Eles são chamados ***suspend ()*** e ***resume ()***.

Uma outra forma de fazer isso, de forma mais segura é usar o “object lock wait”. A classe `java.lang.Object` disponibiliza um conjunto de métodos que nos permite bloquear um objeto e desbloqueá-lo quando desejarmos.

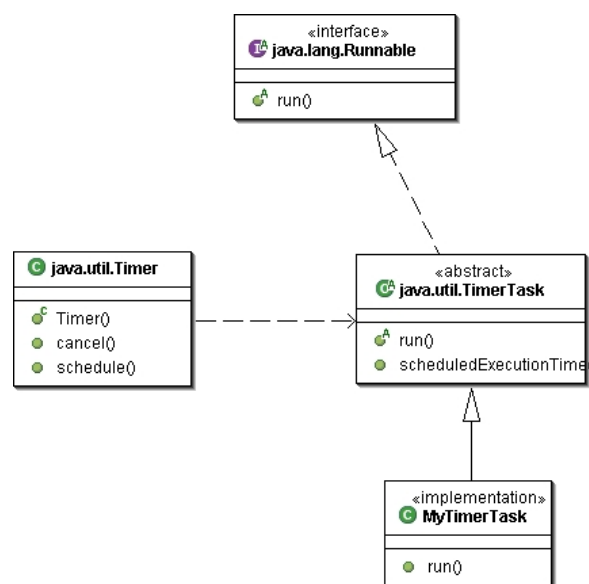
Método	Descrição
<code>wait()</code>	Indica que o objeto atual está bloqueado até acontecer a notificação de desbloqueio.
<code>notify()</code>	Indica que o objeto atual está desbloqueado. Notifica a Thread atual que o objeto foi liberado.
<code>notifyAll()</code>	Indica que o objeto atual está desbloqueado. Notifica todas as Threads que tem referencia deste objeto que ele foi liberado.

Construindo e Executando Timers

O uso de Timers se faz necessários quando queremos criar uma Thread com execução cíclica e recorrente.

A Java2 Standard Edition, traz uma classe que controla definida por `java.util.Timer`, e que tem um método `schedule()`, que recebe uma instância da classe `java.util.TimerTask` (tarefa do timer).

A classe `java.util.TimerTask` é abstrata, tendo o método `run()` abstrato. Assim, quando queremos criar uma tarefa de timer, estendemos o comportamento da classe `java.util.TimerTask`.



```

public class MyTimerTask extends TimerTask {

    public void run() {
        System.out.println ( System.currentTimeMillis() );
    }
}

public class TestMyTimerTask {

    public static void main(String args[]) throws Exception {
        MyTimerTask mtt = new MyTimerTask();
        Timer timer = new Timer();
        timer.schedule( mtt, 10, 1000 );
        Object obj = new Object();
        synchronized( obj ) {
            obj.wait();
        }
    }
}
    
```

Discussão

Qual a melhor maneira de se especificar um Thread? Estender a classe `java.lang.Thread` ou implementar a interface `java.lang.Runnable`?

Nas GUIs usamos as Threads para preencher barras de progresso de acordo com o andamento de um processo, nas aplicações WEB faremos isso?

Quando criamos um servidor que necessita atender mais de um usuário ao mesmo tempo, usaremos os conceitos de Threads?

Usando todos os elementos de Threads (Thread, Monitor, Task, etc), podemos fazer aplicativos complexos e ricos no gerenciamento de processos, que tipos de aplicações são essas?

Exercícios

