Enyo José Tavares Gonçalves
Domingos Sávio Silva Carneiro

Linguagem de Programação II

1ª Edição
Fortaleza - 2010

1

UNIDADE

FUNDAMENTOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS E JAVA

Nesta unidade são introduzidos alguns conceitos de Programação Orientada a Objetos e da Linguagem de programação Java. Estes conceitos são importantes no contexto atual do desenvolvimento de sistemas, uma vez que boa parte das linguagens de programação atuais são orientadas a objeto, fornecendo os conceitos necessários para desenvolvimento de sistemas através de uma das linguagens de programação mais utilizadas no momento: Java.

CAPÍTULO 1

INTRODUÇÃO

Com o desenvolvimento da Ciência da Computação, várias Linguagens de Programação vem sendo criadas, podemos citar: C#, C, C++, Java, COBOL, Lisp, Prolog, Pascal, Smalltalk, Fortran, PHP... Diante deste fato, é natural que nos perguntemos por que tantas linguagens de programação vêm sendo propostas. Vários fatores podem ser citados que justificam a existência de tantas linguagens de programação, dentre eles, podemos citar:

- Propósitos diferentes – Uma linguagem de programação criada para um determinado propósito, pode não ser adequada à outro propósito.
  Ex: A linguagem Pascal foi criada com o intuito de ser utilizada como recurso didático em disciplinas de programação. Portanto, da maneira como foi proposta, não é ideal para o desenvolvimento de um sistema WEB, por exemplo.
- Avanços tecnológicos – Com o passar do tempo, novos conceitos e técnicas de programação são propostos. Portanto, novas linguagens são propostas com base nestes conceitos e técnicas e algumas linguagens existentes evoluem para adequar-se aos benefícios do progresso tecnológico.
  Ex: Podemos citar Smaltalk como exemplo de uma linguagem baseada em novos conceitos propostos. Além disto, C++ é uma evolução de uma linguagem já existente (A linguagem C).
- Interesses comerciais – criação de uma linguagem que fornece uma produtividade maior que as linguagens existentes. Ou necessidade relacionada a um domínio específico (Como a necessidade, que existiu nos anos 90, de uma linguagem para desenvolver sistemas para micro dispositivos).

É desejável que qualquer linguagem de programação proposta possua as seguintes **características**:

- Legibilidade – Diz respeito à facilidade para ler e entender um programa na linguagem.
  Ex: Suponha que em uma linguagem não exista um tipo de dado booleano e um tipo numérico seja usado para representá-lo. Deste modo as operações que envolvem variáveis booleanas são menos legíveis nesta linguagem do que em linguagens que utilizam os valores verdadeiro e falso como valores booleanos. A seguir exemplos da atribuição de valores booleanos nas duas situações:

    - timeOut = 1 (significado não muito claro)
    - timeOut=true (significado claro)
- Redigibilidade – Relacionada a facilidade de escrita de um programa.
- Confiabilidade – Mecanismos oferecidos pela linguagem para incentivar a construção de programas confiáveis.
  Ex: Detecção de erros seja pelo compilador ou através de mecanismos durante a execução.
- Reusabilidade – Possibilidade de reutilizar o mesmo código para diversas aplicações . Ex: Funções pré-definidas e Classes.
- Modularidade – É a propriedade que um sistema tem de ser decomposto em um conjunto de módulos coesos.
- Manutembilidade – Facilidade de realização de manutenções preventivas ou corretivas no programa criado a partir da linguagem.
- Portabilidade – Propriedade que permite o sistema ser executado em vários ambientes operacionais.
- Eficiência – Pode ser eficiência de processamento, ou seja, a resposta para uma operação ser dada no menor tempo possível. Ou eficiência relacionada ao tempo de desenvolvimento.
- Facilidade de aprendizado.

Estas características, são propriedades desejáveis que toda linguagem de programação proposta **deve** oferecer. No entanto, quando as linguagens são comparadas em relação a estas propriedades desejáveis, podemos observar que algumas destacam-se em relação a determinadas propriedades, enquanto outras destacam-se em relação a outras propriedades.
Para exemplificar tomemos as linguagens Java e C. Se as compararmos em relação a reusabilidade, Java oferece mais mecanismos relacionados a esta propriedade do que C. No entanto se as compararmos em relação ao tempo de resposta, possivelmente C terá um resultado melhor que Java.
Além das propriedades desejáveis, cada linguagem de programação é baseada em um **paradigma**. Um paradigma de programação é um modelo, padrão ou estilo de programação suportado por linguagens que agrupam certas características comuns. Os paradigmas mais populares são o paradigma estruturado e o paradigma orientado à objetos.
No paradigma estruturado, o programa é dividido em um conjunto de funções (ou procedimentos) e os dados necessários para resolver um problema. Como exemplo de linguagens que seguem este paradigma podemos citar C e Pascal.
Foi criado com o intuito de aumentar as propriedades desejáveis em relação a programação estruturada como modularização, reutilização, manutembilidade. Introduz um conjunto de

conceitos novos: Herança, polimorfismo, encapsulamento. **Java** e C++ são exemplos de linguagens Orientadas a objeto.

Java é uma linguagem criada por James Gosling em1995 com o intuíto de desenvolver sistemas para eletrodomésticos. Ela possui uma sintaxe semelhante a linguagem C, sendo mais simples e robusta (menos bugs!).

Outras características de Java podem ser citadas:

- Java é sensitive case – Diferencia maiúsculas de minúsculas.
- Independente de plataforma – Programas Java podem ser executados em diferentes sistemas operacionais;
- Sintaxe fortemente tipada – Cada atributo possui um tipo de dados fixo e pode armazenar valores do tipo que foi definido;
- Orientada à objetos – Disponibiliza mecanismos como herança, polimorfismo, encapsulamento;
- Coleta de lixo – Em Java não é necessário desalocar memória não utilizada, pois existe um mecanismo que checa a memória automaticamente liberando as porções que não serão mais utilizadas;
- API Java – Conjunto de classes utilitárias que disponibiliza funcionalidades como interface gráfica, mecanismos de acesso a banco de dados e a arquivos... Bem como a documentação descrevendo-as;
- JVM (Java Virtual Machine) – Para executar um programa Java precisamos de uma maquina virtual. Esta é dependente de plataforma (Sistema operacional);

O funcionamento de um programa java é bastante simples: Inicialmente o código é criado em um arquivo com a extensão .java. Em seguida este arquivo passa pelo processo de **compilação** gerando um arquivo .class conhecido como Bytecode, este arquivo pode ser disponibilizado em qualquer sistema operacional que possua uma JVM instalada.

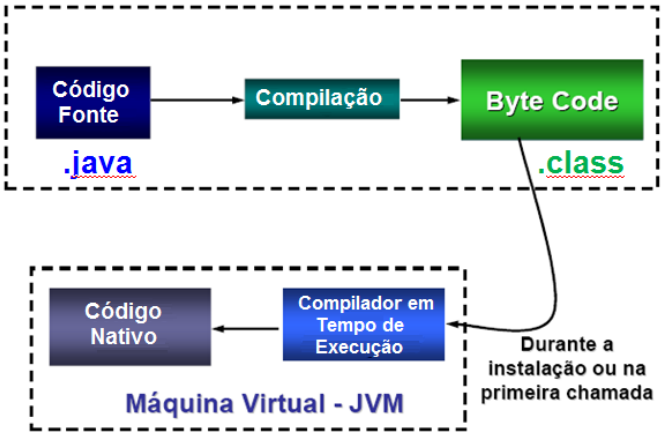
Finalmente, podemos **executar** a aplicação Java. Na primeira execução, a JVM irá converter o arquivo .class em um arquivo de código nativo. Abaixo, uma ilustração das etapas de um programa em Java.

Atualmente Java é uma linguagem que pode ser utilizada para desenvolvimento para desktop, web e desenvolvimento para micro dispositivos. Para tanto, existem três plataformas distintas:

- J2SE (Standard Edition) – Indicada para desenvolvimento Desktop;
- J2EE (Enterprise Edition) – Indicada para desenvolvimento Web;
- J2ME (Micro Edition) – Indicada para desenvolvimento para micro dispositivos.

Para compilar os arquivos .java, basta abrir o prompt de comando, ir até o diretório que se encontra o arquivo .java e digitar:
javac <nome_do_arquivo>.java

Para executar os arquivos .class, basta abrir o prompt de comando, ir até o diretório que se encontra o arquivo .class e digitar:
java <nome_do_arquivo>



Para desenvolver aplicações Java, é necessário o kit do desenvolvedor java, ou SDK (Sun development kit), este disponibiliza JVM, compilador, depurador de código, etc.
Em máquinas que irão apenas executar o código, uma alternativa é o ambiente de execução JRE (Java Runtime Environment).

Questões de Avaliação

1. **Compare as Linguagens C e Java em relação à:**
   a. **Propriedades Desejáveis;**
   b. **Paradigmas;**
2. **Descreva com suas palavras o procedimento para desenvolver um programa Java.**

CAPÍTULO 2

OBJETOS E CLASSES

CLASSES

Programas de computador são maneiras de representar o mundo real, ou seja, o desenvolvimento de sistemas é uma tarefa na qual um problema do mundo real (espaço do problema) é resolvido em um computador (espaço da solução). Linguagens de programação orientadas a objeto representam de maneira bastante intuitiva os elementos do espaço do problema. Vejamos a situação a seguir:

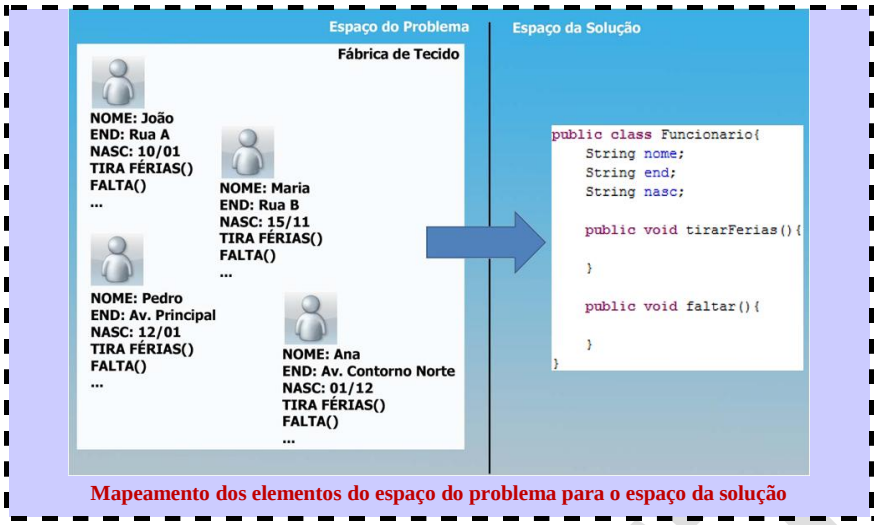
Uma fábrica de tecidos possui os seguintes dados de seus Funcionários:

- João, mora na Rua A, nasceu 10/01. João pode tirar férias e avisar quando for faltar;
- Maria, mora na Rua B, nasceu 15/11. Maria pode tirar férias e avisar quando for faltar;
- Pedro, mora na Avenida Principal, nasceu 12/01. Pedro pode tirar férias e avisar quando for faltar;
- Ana, mora na Rua Av. Contorno Norte, nasceu 01/12. Ana pode tirar férias e avisar quando for faltar;

Caso o dono da fábrica de tecidos solicite a criação de um sistema de cadastro de Funcionários, o primeiro passo é analisar os elementos do espaço do problema (neste caso Funcionários). Esta análise consiste em identificar os dados e comportamentos que cada Funcionário pode realizar no contexto da fábrica.

Deste modo, podemos identificar como dados dos funcionários: nome, endereço e data de nascimento; e como comportamentos, podemos identificar tirar férias e avisar falta.

Estes elementos, após identificados, devem ser codificados no espaço da solução (sistema computacional). Em Java, os elementos do espaço do problema são representados através de *classes*.



Mapeamento dos elementos do espaço do problema para o espaço da solução

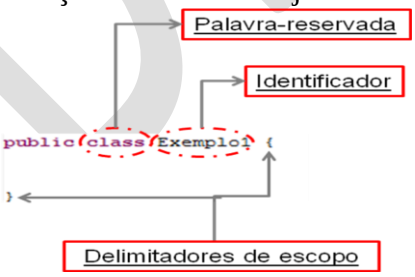
Classe é um dos conceitos mais utilizados em Java, ela é a base para criação de **objetos**. Fazendo uma analogia, podemos compará-la a uma fôrma:

- A partir de uma fôrma podemos criar tantos bolos quanto necessário. Da mesma forma podemos criar vários **objetos** a partir de uma única classe;
- Para criarmos o bolo é preciso termos a fôrma. Da mesma forma, para criar um **objeto** precisamos ter a **classe** antes;

Deste modo quando pensamos em criar programas em Java, inevitavelmente pensamos em criar classes e objetos são criados a partir de classes existentes.

Criamos classes Java em arquivos e estes arquivos devem possuir a extensão .java, sendo que inicialmente colocaremos cada classe em um arquivo diferente. Além disto, o nome do arquivo .java deve ser exatamente o mesmo nome da classe definida no arquivo.

A sintaxe para definição de classes em java é mostrada a seguir:



Por enquanto definiremos todas as nossas classes como public.

As palavras reservadas **public class** são utilizadas, em seguida o identificador (nome) da classe e, por ultimo, as chaves de abertura e fechamento delimitam o escopo (onde a classe começa e onde a classe termina).

Devemos identificar nossas classes da maneira mais intuitiva possível. Nossas classes devem ter nomes como Casa, Funcionario, Execucao, Questao1... Para evitarmos erros na escolha do indentificador de classes, seguem algumas regras:

- Não podemos identificar uma Classe com uma palavra

reservada de Java;

Palavras reservadas da Linguagem Java[1]

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

[1] const e goto são palavras reservadas, mas não são utilizadas pela linguagem.

- O identificador não pode conter espaços;
- Deve ser iniciado com uma letra, _ (símbolo sublinhado) ou $;
- Pode conter números a partir da 2ª posição;
- Não podemos utilizar caracteres especiais[1] (*, -, (,), %, #, @, !, ¨, ´, “, , , ., +. =, <,>, :, ;, ?, |, \, /,) no identificador da classe;
- A primeira letra de cada palavra do identificador da classe deve ser maiúscula. Se não seguirmos esta observação não ocorrerá erro, mas é uma convenção importante;
- Exemplos inválidos de identificadores de classes:

[1] Caracteres como ç e letras acentuadas são aceitas, mas devem ser evitadas.

```
public class Classe Inicial{
}
```
Definição inválida, o identificador não pode conter espaço.

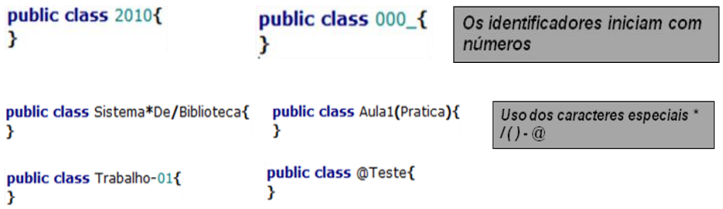
```
public class class{
}
```
Definição inválida, class é uma palavra reservada e não pode ser utilizada como identificador de classe.

```
public class 1Classe{
}
```
Definição inválida, o identificador não pode começar com números

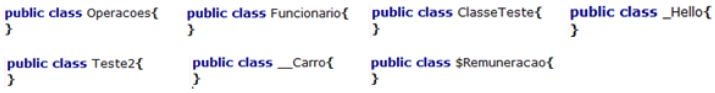
```
public class Classe*{
}
```
*Definição inválida, o identificador não pode ter o caractere **

```
public class default{
}
public class continue{
}
public class super{
}
```
default, continue e super são palavras reservadas

```
public class Classe Principal{
}
public class Conta Corrente{
}
```
Há espaço no identificador das classes



- Exemplos válidos de identificadores de classes:



O que uma classe pode ter?

Uma classe pode conter **atributos** e **métodos**. **Atributos** são entidades que podem guardar valores de um tipo de dados, é como são chamadas as variáveis e constantes em orientação a objetos. **Métodos** definem o comportamento (operações possíveis), é como são chamadas as funções e procedimentos em orientação a objetos.

Atributos

Por enquanto vamos considerar que atributos possuam somente um tipo e um identificador. Identificador é o nome do atributo e o tipo define o tipo de dado que o atributo pode receber. Os pricipais tipos primitivos em java são

Principais tipos	Valores possíveis
boolean	true; false
char	Um caractere: ‘a’ ... ‘z’; ‘A’... ‘Z’ ; ‘1’...’9’...
byte	Inteiros
short	Inteiros
int	Inteiros
long	Inteiros
float	Ponto Flutuante
double	Ponto Flutuante
void	-

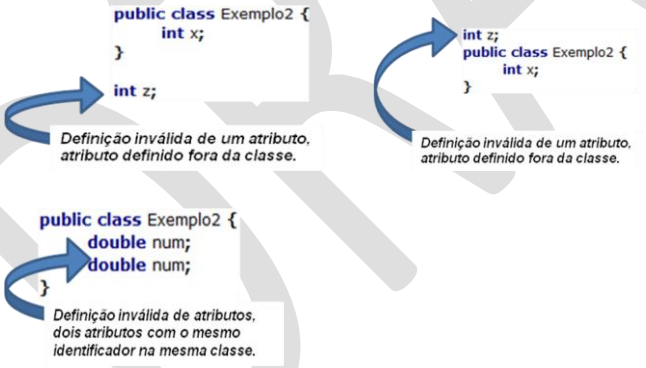
Não podemos definir atributos com o tipo void.
Para valores de ponto flutuante, ponto é usado para separar a parte inteira da parte decimal.

Para representarmos texto como palavras e frases, utilizamos um tipo chamado String. Ex: “Linguagem de Programação II”.
Devemos identificar nossos atributos da maneira mais intuitiva possível. Nossos atributos devem ter nomes como endereco, fone, num1... Para evitarmos erros na escolha do indentificador de atributos, seguem algumas regras:

- A primeira palavra do identificador deve começar com letra minúscula e as demais palavras devem iniciar com

maiúsculas. Se não seguirmos esta observação não ocorrerá erro, mas é uma convenção importante;

- Os atributos são definidos dentro da classe;
- Não podemos definir dois ou mais atributos na mesma classe **com o mesmo identificador**;
- Não podemos identificar um atributo com uma palavra reservada; (idem classes);
- O identificador não pode conter espaços; (idem classes);
- Deve ser iniciado com uma letra, _ (símbolo sublinhado) ou $;
- Pode conter números a partir da 2ª posição; (idem classes);
- Não podemos utilizar caracteres especiais (*, -, (,), %, #, @, !, ¨, ´, “, , , ., +. =, <,>, :, ;, ?, |, \, /,) no identificador do atributo. (idem classes);
- Exemplos inválidos de identificadores de atributos:



- Exemplos válidos de identificadores de atributos:

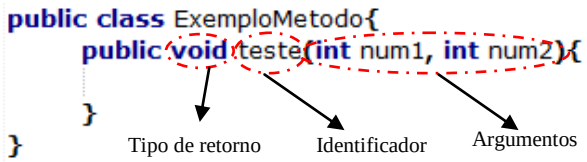
Adicionalmente, podemos inicializar o atributo no momento da definição. A seguir uma ilustração da definição de um atributo juntamente com sua inicialização.

```
public class Uab{
      int codigoCurso = 1;
}
```

Métodos

Um método é criado quando queremos implementar algum comportamento. Ex: Tirar férias, calcular o índice de massa corporal, fazer uma soma, ler dados do teclado, salvar no banco

de dados.
Cada método possui um tipo de retorno, um identificador, um conjunto (zero ou mais) de argumentos (valores que podem ser recebidos) e uma implementação. A implementação do método é delimitada por chaves.



Por enquanto colocaremos a palavra reservada public antes de cada definição de método.

Devemos identificar nossos métodos da maneira mais intuitiva possível. Nossos métodos devem ter nomes como calcularMedia, imprimeNome, soma... Para evitarmos erros na escolha do indentificador de métodos, seguem algumas regras:

- A primeira palavra do identificador deve começar com letra minúscula e as demais palavras devem iniciar com maiúsculas. Se não seguirmos esta observação não ocorrerá erro, mas é uma convenção importante;
- As mesmas regras de identificadores de classes;

Métodos podem devolver valores. Quando o tipo de retorno é diferente de void, dentro do método devemos retornar um valor adequado ao tipo a ser retornado. Neste caso podemos retornar o valor literal (1 para inteiros, 'a' para char...), retornar o conteúdo de um atributo ou resultado de uma operação.
A palavra reservada return é utilizada para esta finalidade.

```
public class Operacao{
    public int soma(int num1, int num2){
        return num1+num2;
    }
}
```

Neste exemplo temos um método soma que retorna um inteiro resultante da soma de dois inteiros (num1 e num2 são atributos inteiros);

```
public class Operacao{
    public int soma(){
        return 3+2;
    }
}
```

Neste exemplo temos um método soma que retorna um inteiro resultante da soma de dois inteiros (3 e 2);

Qualquer método possui dois momentos: o momento da definição e o momento da chamada.

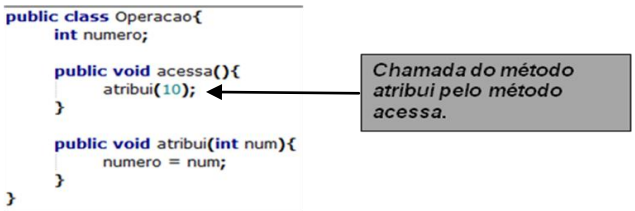
- No 1º Momento (Momento da definição) – O comportamento que o método deve ter é definido;

```
public class Operacao{
    int numero;
    public void atribui(int num){
        numero = num;
    }
}
```

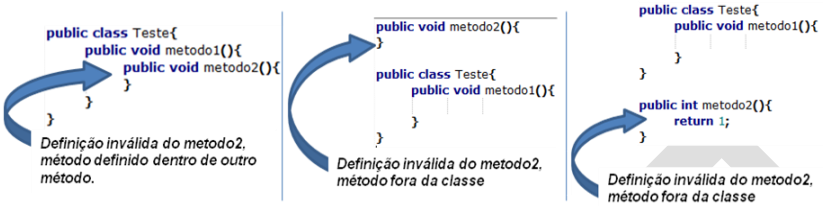
Definição do método atribui.

O comportamento do método atribui é inicializar o valor do atributo numero com o valor do parâmetro num.

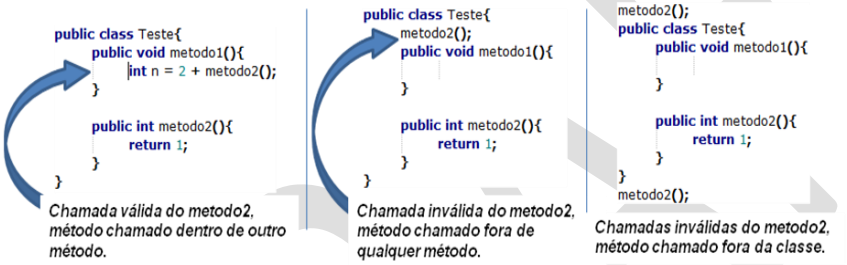
- No 2º Momento (Momento da chamada) – Quando um método que **já foi definido** é acessado para processar as operações.



Nunca esqueça que métodos não podem ser definidos (**1º momento**) dentro de outros métodos. Além disto, não podemos definir métodos fora da classe.



Outra observação importante é que um método pode ser chamado (**2º momento**) somente dentro de outro método. Ele não pode ser chamado solto na classe, nem fora da classe.



Qual a finalidade de criar uma classe?

Uma classe pode ser utilizada como i) base para criação de objetos e/ou ii) ser utilizada como **classe de execução**, ou seja, onde a execução irá iniciar.

A presença do método ***public static void main(String args[])*** caracteriza que a classe é uma classe de execução.

Exemplo de uma classe base simples

```
public class Aluno {
    String nome;
    int idade;

    public void atualizaIdade(){
    }
}
```

Exemplo de uma classe de execução:

```
public class HelloWorld{
    public static void main(String args[]){
        System.out.println("Hello World!");
    }
}
```

O comando System.out.println faz com que o conteúdo colocado entre parentesis seja impresso na tela.

Como inserir comentários na classe?

Existem três tipos possíveis de comentário: O comentário de uma linha, o comentário de várias linhas e o comentário de documentação.

Para criar um comentário de uma linha, basta adicionar // (duas barras) antes do texto do comentário. Iniciamos o comentário de duas linhas com /* (barra seguido de asterisco) e o finalizamos com */ (asterisco seguido de barra). O comentário de documentação é iniciado com /** e encerrado com */, ele pode

ter várias linhas e a partir dele uma documentação da classe pode ser gerada. A seguir exemplos de comentários:

```
// Este comentário é comentário de uma linha.
public class Teste{
        /*
                Comentário de várias linhas
        */
        int x;
        /**
                Comentário de documentação
        */
        public static void main(String args[]){
        }

}
```

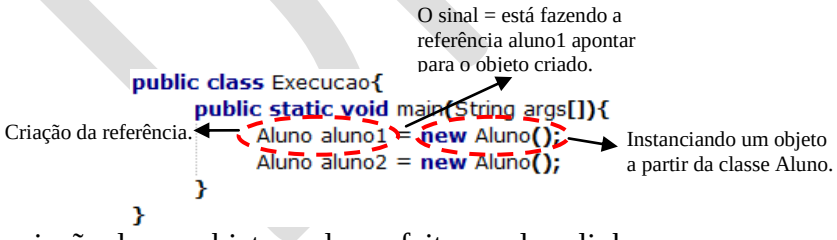
Observe que os comentários podem ser inseridos em qualquer local de nosso arquivo .java.

OBJETOS

Objetos são criados a partir de classes. Portanto, para instanciarmos um objeto, precisamos que a classe tenha sido definida. Lembrando a analogia da fôrma, você tem que ter a fôrma para criar o bolo. Além disto, podemos criar tantos objetos quanto necessário (a partir da fôrma pode criar quantos bolos quisermos).

Em Java os objetos são manipulados a partir de **referências** e são instanciados dentro de uma classe (ou dentro dos métodos da classe).

A palavra-reservada **new** é utilizada para instanciar objetos, seguida do tipo de classe utilizado para criação do objeto e de parêntesis. A seguir um exemplo da criação de uma instância (objeto) a partir da classe Aluno, exibida logo acima.



Uma referência possui um tipo e um identificador: Para o tipo é utilizada qualquer classe já definida e para identificador deve-se seguir as regras de identificador de atributos.

A criação de um objeto pode ser feita em duas linhas.

```
1  public class Execucao{
2          public static void main(String args[]){
3                  Aluno aluno1;
4                  aluno1 = new Aluno();
5          }
6  }
```

Observe que na linha 3 uma referência chamada aluno1 é criada e esta referência poderá apontar para uma instância da classe Aluno. Na linha 4 uma instância de aluno é criada e a referência aluno1 passa a referenciar o objeto.

Outros exemplos de criação de objetos são dados a seguir:

Exemplo 1: Criação de uma classe para representar uma casa e a criação de objetos a partir de casa.

```java
public class Casa {
	int qtdQuartos;
	String corCasa;
	double area;
	int qtdPortas;
	String cidade;


	public void imprimeValores(){
		System.out.print(qtdQuartos);
		System.out.print(qtdPortas);
		System.out.print(area);
		System.out.print(corCasa);
		System.out.print(cidade);
	}
}
```

```java
public class Cidade {

	public static void main(String[] args) {
		Casa casa1 = new Casa();
		Casa casa2 = new Casa();
		Casa casa3 = new Casa();
	}
}
```

Exemplo2: Criação de uma classe de Teste que guarda 3 valores inteiros e uma classe de Execução que instância a classe Teste.

```java
public class Teste{
	int x, y, z;
}
```

```java
public class Execucao{
	Teste t = new Teste();
}
```

Observe que a criação da instância ocorre fora de qualquer método.

Qualquer objeto possuirá as mesmas características da classe utilizada para instanciá-lo. Deste modo, dentro de métodos podemos acessar os membros (atributos e métodos) que o objeto possui através de sua referência seguida do operador ponto (Mais operadores serão vistos no capítulo 3).
Exemplo de acesso aos membros da classe Casa do Exemplo 1 logo acima.

```java
public class Cidade {

	public static void main(String[] args) {
		Casa casa1 = new Casa();

		casa1.qtdQuartos = 3;
		casa1.qtdPortas = 10;
		casa1.area = 100.7;
		casa1.corCasa = "Branca";
		casa1.cidade = "Fortaleza";

		casa1.imprimeValores();
	}
}
```

Os atributos receberam valores nas linhas de 6 a 10 e o método *imprimeValores* foi chamado através da referência *casa1*.

Questões de Avaliação

1. Quais dos identificadores de atributos abaixo não são válidos? Explique o motivo.

a. nome_do_pai	**b. nome**
c. n2	**d. endereço(1)**
e. titulação	**f. salário$1**
g. 13º_salario	**h. 1contador**
i. velocidade inicial	**j. dia&noite**
k. atributo teste	**l. int**
m. dia_e_noite	**n) diaEnoite**

2. **Qual o tipo de dados mais indicado para expressar:**

   a. **O resultado do arremesso de uma moeda;**
   b. **O resultado de um dado;**
   c. **Número sorteado de um bingo;**
   d. **Altura de uma pessoa em metros;**
   e. **Peso em Kg de um navio carregado de ferro;**
   f. **Temperatura do ser humano em ºC;**
   g. **O endereço em um cadastro;**
   h. **O Sexo do Aluno;**

3. **Declare quatro atributos em uma classe, uma de cada tipo primitivo de dados. Na declaração, inicialize-os.**
4. **Crie uma classe base para cadastrar Funcionários. Um funcionário deve possuir nome, endereço, telefone, rg, cpf, data de admissão. Além disto, um funcionário pode tirar férias, faltar e solicitar saída do emprego. Em seguida, crie uma classe de execução chamada Fabrica, nesta classe instancie três Funcionários e inicialize os valores de cada Funcionário criado.**
5. **Encontre os erros nos seguintes trechos de código:**

a.
```
public cass Teste{
        int x;
        double z.
```

b.
```
public class Teste2{
      int num1, num2, n1+n2;
      soma(){
         num1 = 2;
         num2 = 10;
         n1+n2 = num1 + num2;
      }
}
```

c.
```
inicializaValor();
public class Teste3{
      int x;
      inicializaValor();
      public void inicializaValor(){
            x =10;
      }
      public void metodoAuxiliar(){
            inicializa();
      }
}
inicializaValor();
```

d.
```
public class Teste3{
      int x;

      public int inicializaValor(){
            x = 10;
      }
}
```

e.

```
public class Teste3{
	int x;

	public int inicializaValor(){

		public void teste(){
		}

		return 10;
	}
}
```

f.

```
int x;
public class Teste4{
	int x;
	double x;
	String x;
	boolean verdade?;

	public int _soma_1(int x){
		return (x+1);
	}
}
```

g.

```
public class boolean{
	int x = 10;
}
```

h.

```
public class Teste11{

	public void metodo_teste(){
		soma();
	}
}
```

CAPÍTULO 3

ESCOPO E OPERADORES

ESCOPO

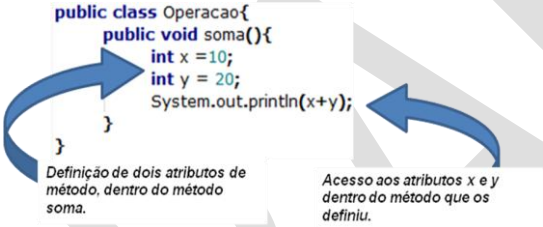
O escopo diz respeito ao local onde um membro (atributo ou método) pode ser acessado. Atributos podem ser classificados em três categorias em relação ao escopo: Instância, Método e Classe. Já os métodos podem ser de classe ou de instância.

Atributos

Atributos de Método

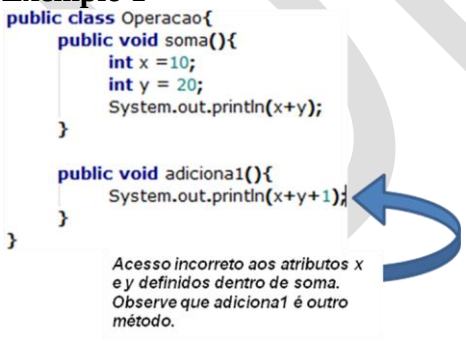
Atributos definidos **dentro** de métodos, são chamados **atributos de método**. Estes podem ser acessados apenas localmente no método que os definiu.

Exemplo

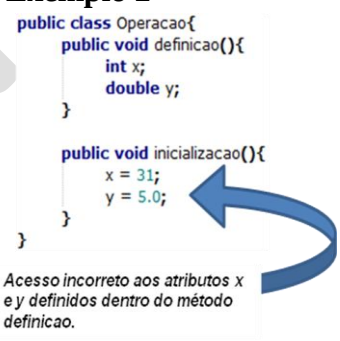


Portanto, se tentarmos acessar um atributo definido dentro de um método dentro de outro método ocorrerá erro.

Exemplo 1



Exemplo 2



Atributos de Instância

Atributos definidos **fora** de métodos são chamados de **atributos de instância**.

Exemplo

```
public class Funcionario{
        String nome;
        int idade;
        double peso;
        float altura;
}
```

Definição de quatro atributos de instância da classe Funcionario

Estes podem ser acessados i) Dentro dos métodos da classe que foi definido normalmente e ii) Através de objetos (instâncias) da classe que foi definido usando o operador ponto.

Exemplo de acesso do atributo de instância dentro de um método da mesma classe.

```
public class Funcionario{
        String nome;
        int idade;
        double peso;
        float altura;

        public void alteraIdade(){
                idade = 25;
        }
}
```

Acesso correto do atributo idade dentro do método alteraIdade.

Exemplo de acesso dos atributos de instância através de uma instância da classe Funcionário.

```
public class Execucao{
        public static void main(String args[]){
                Funcionario f1 = new Funcionario();

                f1.nome = "João";
                f1.idade = 44;
                f1.peso = 65.8;
                f1.altura = 1.75f;
        }
}
```

Acesso correto dos atributos de Funcionario através de uma instância.

Observação: Os atributos de instância devem ser definidos dentro da classe.
Exemplo

```
String nome;
public class Funcionario{
        int idade;
        double peso;
        float altura;
}
```

Definição incorreta do atributo nome. O atributo deve ser definido dentro da classe ou de um método.

Cada instância criada possui valores independentes e individuais em relação aos valores de outras instâncias. Portanto, se alterarmos o valor dos atributos de instância de um objeto, esta alteração não alterará os outros objetos.
Exemplo:

```
public class Execucao{
        public static void main(String args[]){
                Funcionario f1 = new Funcionario();
                Funcionario f2 = new Funcionario();

                f1.nome = "João";
                f1.idade = 44;
                f1.peso = 65.8;
                f1.altura = 1.75f;

                f2.nome = "Maria";
                f2.idade = 29;
                f2.peso = 55.7;
                f2.altura = 1.70f;

                System.out.println(f1.nome);
        }
}
```

Será exibido na tela o valor João.

Atributos de Classe
Os atributos de classe são definidos fora de métodos de maneira

similar aos atributos de instância. A diferença é o uso da palavra reservada **static** antes do tipo de dados do atributo.
Exemplo

```
public class Funcionario{
        static boolean plantaoColetivo = false;
}
```

Atributos de classe são **compartilhados** por todas as instâncias. Ou seja, alteração no valor do atributo através de uma das instâncias reflete nas outras instâncias.
Exemplo:

```
public class Funcionario{
    static boolean plantaoColetivo;
}
```

```
public class Execucao{
    public static void main(String args[]){
        Funcionario f1 = new Funcionario();
        Funcionario f2 = new Funcionario();

        f1.plantaoColetivo = true;
        f2.plantaoColetivo = false;

        System.out.println(f1.plantaoColetivo);
    }
}
```

A saída da tela exibirá false, mesmo tendo atribuído true utilizando f1. Lembre-se que o atributo de classe plantaoColetivo é compartilhado por todas as instâncias.

Outra maneira de acessar um atributo de classe é acessá-lo através do nome da classe ao invés de acessá-lo através de uma referência[1].
Exemplo

```
public class Funcionario{
    static boolean plantaoColetivo;
}
```

```
public class Execucao{
    public static void main(String args[]){
        Funcionario.plantaoColetivo = true;

        System.out.println(Funcionario.plantaoColetivo);
    }
}
```

Será impresso true. Observe que não temos uma instância de Funcionário. Acessamos o atributo de classe através do identificador da própria classe.

[1] Lembre-se que o atributo de classe, como o próprio nome já indica, não necessita de uma instância (objeto). Este tipo de atributo está associado à Classe a qual pertence.

Métodos

Método de Instância

É semelhante ao atributo de instância, ou seja, cada objeto da classe tem o seu individualmente;
Exemplo

```
public class MetodoInstancia{
    public int soma(int x, int y){
        return (x+y);
    }
}
```

O acesso (chamada) do método de instância pode ser realizado dentro da própria classe ou através de uma instância da classe que o definiu.

Exemplo de acesso a um método de instância dentro de um método da mesma classe.

```
public class MetodoInstancia{
    public int soma(int x, int y){
        return (x+y);
    }

    public void chamador(){
        int result = soma(1,10);
    }
}
```

Exemplo de acesso a um método de instância dentro de um método de outra classe.

```
public class Exec{
    public static void main(String args[]){
        MetodoInstancia mi = new MetodoInstancia();
        int result = mi.soma(2,8);
    }
}
```

Método de Classe

Os métodos de classe são definidos de maneira similar aos métodos de instância, porém a palavra reservada static é adicionada antes do tipo de retorno[2].

[2] O método main é um método de classe bastante utilizado. Observe que ao definirmos utilizamos o static.

Exemplo:

```
public class Operacao{
        public static void soma(int x, int y){
                return x+y;
        }
        public static void main(String args[]){
                int result = soma(2+3);
        }
}
```

Um método de classe pode acessar atributos de classe ou atributos definidos dentro dele mesmo.

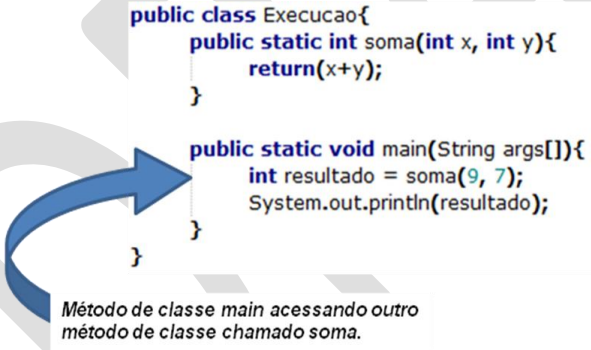
Exemplo:

```
public class Execucao{
        static int idade;  // Atributo de classe

        public static void main(String args[]){
                char sexo;

                idade = 76;  // Acessando o atributo de classe
                sexo = 'M';   // Acessando o atributo definido dentro do próprio método
        }
}
```

Um método de classe pode acessar apenas outros métodos de classe.



Método de classe main acessando outro método de classe chamado soma.

Um método de classe não pode acessar atributos ou métodos que não forem de classe.

Exemplo:

```
public class Execucao{

        int x; // Atributo de instância.

        // Método de instância
        public void acessoIncorreto(){
                System.out.println("Teste");
        }

        // Método de classe
        public static void main(String args[]){
                x = 10;                 // Acesso incorreto.
                acessoIncorreto();  // Acesso incorreto.
        }
}
```

OPERADORES

Na tabela a seguir serão listados e detalhados os principais operadores em Java.

Operador	Finalidade	Tipo de dados que trabalha	Valor retornado da operação
|	ou	Booleano	Booleano
&	e	Booleano	Booleano
!	not	Booleano	Booleano
=	Atribuição	Todos	Depende dos operandos
-, *, /	Operadores matemáticos	Numéricos	Depende dos operandos
+	Soma números e concatena Strings	Numéricos ou String	Depende dos operandos
%	Resto da divisão	Numérico	Numérico
==	Igualdade	Todos	Booleano
!=	Diferença	Todos	Booleano
>, <, >=, <=	Maior, Menor, Maior ou Igual, Menor ou Igual	Numéricos e Caractere	Booleano

Além destes operadores, temos o operador de auto-incremento (++) que é utilizado para adicionar o atributo em uma unidade e o operador de auto-decremento (--) que diminui o valor do atributo em uma unidade. Outro operador bastante comum é são parêntesis, que são utilizados para dar precedência maior a uma dada operação.

A precedência dos operadores é avaliada da esquerda para a direita, sendo que os operadores apresentados possuem a seguinte precedência:

Maior precedência

()
++, --, !
*, /, %
+,-
>,<,>=,<=
==, !=
&,|
?:
=

Menor precedência

Os operadores e, ou e not possuem tabelas lógicas que são utilizadas para calcular seu resultado. A seguir, estão as tabelas lógicas dos três operadores.

Tabela lógica do e (&)		
Valor 1	Valor 2	Resultado
V	V	V
V	F	F
F	V	F
F	F	F

Tabela lógica do ou (|)		
Valor 1	Valor 2	Resultado
V	V	V
V	F	V
F	V	V
F	F	F

Tabela lógica do não (!)	
Valor	Resultado
V	F
F	V

Os operadores podem ser utilizados com valores ou com atributos do tipo que eles trabalham. Exemplos:

```
public class Teste{
        int soma;
        public void soma(){
                soma = 10 +20;
        }
}
```

```
public class Teste{
        int soma, num1, num2;
        public void soma(){
                num1 = 10;
                num2 = 20;
                soma = num1 + num2;
        }
}
```

```
public class Teste{
        boolean result;
        public void operadorE(){
                result = true & true;
        }
}
```

```
public class Teste{
        boolean b1,b2,result;
        public void operadorE(){
                result = b1 & b2;
        }
}
```

O operador + pode ser utilizado em duas situações: i) quando queremos somar valores numéricos e ii) quando queremos concatenar (juntar Strings). Exemplo de concatenação de Strings:

```
public class Teste{
        public static void main(String[] args) {
                String parte1 = "Programação";
                String parte2 = " em Java";
                System.out.println(parte1 + parte2);
        }
}
```

Conversões (Cast)

Valores de tipos numéricos nativos podem ser transformados em outros tipos numéricos também cast nativos através de uma operação conhecida como cast ou conversão explícita. Para efetuar a conversão explícita, basta colocar o tipo de dado para o qual se deseja converter entre parênteses antes da expressão a ser convertida (ex.: Apesar de 100 ser um valor inteiro, o valor (byte)100 será considerado do tipo byte e o valor (double)100; do tipo double).

Se existe perda de precisão em uma atribuição é necessário fazer uma conversão explicitamente. Exemplo:

```
public class Teste{
        public void testa(){
                long longValue = 99;
                int intValue2 = (int)longValue; // Ok!
                int intValue1 = longValue; // ERRO!
        }
}
```

Neste exemplo, a atribuição realizada na linha 5 ocasionará um erro, pois um long não cabe em um int. Já a atribuição da linha 4 não ocasionará erro, pois a conversão (cast foi realizado).

Além disto, observe que algumas conversões poderão ocasionar erros. Por exemplo, se tentarmos converter uma String para

inteiro, ocorrerá erro. Exemplo:

```
public class Teste{
    public void testa(){
        String x = "12";
        int y = (int)x;
    }
}
```

Questões de Avaliação

1. **Crie uma classe para representar uma casa e crie instâncias desta classe em uma classe de execução chamada cidade. OBS.: A classe casa deve possuir atributos nos três escopos possíveis e métodos de classe e de instância.**

2. **Sabendo que x é um atributo int com valor 31 e y é um atributo float com valor 5, calcule as seguintes expressões.**

   a. **x / 5 * y**
   b. **x >= y * 5**
   c. **(y % 2) == 1 | (x * 6) < 50**
   d. **(x / 8 > 5 & !(x / 5 > 10))**
   e. **x+x-!y**
   f. **true & (x*4 > 100)**

3. **Verifique quais das situações são válidas e as que não são válidas, justifique.**

   a. **String valor1 = "Teste";**
      int valor2 = (int)valor1;

   b. **double valor1 = 10;**
      int valor2 = (int)valor1;

   c. **int x = 99;**
      double y = x;

Síntese da Unidade

Nesta unidade foi apresentado um estudo introdutório sobre os fundamentos de Java e Programação Orientada a Objetos. Esta teoria é de fundamental importância pois será utilizada como base para capítulos posteriores.

Bibliografia

ECKEL, B. **Pensando em Java,** 3ª ed, Editora MindView;

SANTOS, R. **Introdução à Programação Orientada a Objetos Usando Java**, Editora Campus;

DEITEL, H. M. e DEITEL, P.J. **Java: Como programar**, 6ª ed, Editora Pearson;

HORSTMANN, C. e CORNEL, G. **Core Java** – Volume I. Fundamentos. Alta Books. 7ª. 2005;

2

UNIDADE

LEITURA DO TECLADO, ESTRUTURAS DE DECISÃO E REPETIÇÃO, VISIBILIDADE E ENCAPSULAMENTO

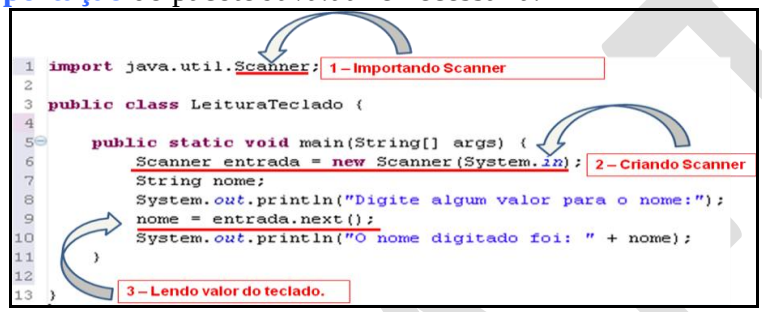
Nesta unidade são introduzidos os conceitos de leitura de dados do teclado via prompt de comando. Estruturas de decisão e de repetição serão demonstradas, juntamente com aplicações. Finalmente, os conceitos de visibilidade e encapsulamento são abordados.

CAPÍTULO 1

LEITURA DE DADOS DO TECLADO

O comando de entrada é utilizado para receber dados digitados pelo usuário. Os dados recebidos devem ser armazenados em variáveis. Uma das formas de leitura de dados que Java disponibiliza é por meio da classe Scanner. Para tanto a **importação** do pacote Java.util é necessária.

Para fazer uso de classes definidas em outros pacotes temos que utilizar o comando de importação. Este comando é semelhante ao include da linguagem c.



É importante ressaltar que dependendo do tipo de dados a ser lido, existe um método diferente na classe Scanner. A seguir, estes métodos e sua respectiva funcionalidade.

Função	Funcionalidade
next()	Aguarda uma entrada em formato String com uma única palavra;
nextLine()	Aguarda uma entrada em formato String com uma ou várias palavras;
nextInt()	Aguarda uma entrada de formato inteiro;
nextDouble	Aguarda uma entrada em formato fracionário;

Observe que não existe o nextChar. Se quisermos ler um caractere, devemos utilizar o método next (o qual retorna uma String lida) juntamente o método charAt (o qual retorna a posição do caractere a ser devolvido).

```
import java.util.Scanner;
public class Leitura{
	public static void main(String args[]){
		Scanner entrada = new Scanner(System.in);
		System.out.println ("Digite um caractere");
		char c = entrada.next().charAt(0);
		System.out.println(c);
	}
}
```

Neste caso, uma String é lida através de entrada.next() e em seguida o primeiro caractere é extraído através de charAt(0).
Uma Observação importante a ser feita é que quando lemos

valores de tipos diferentes vamos necessitar de diferentes Scanners. Para simplificar, vamos criar um Scanner para cada tipo de dados. Portanto se formos ler valores do tipo int, String e double criaremos três Scanners.

```
import java.util.Scanner;
public class Leitura{
	public static void main(String args[]){
		Scanner entrada = new Scanner(System.in);
			Scanner entradaInt = new Scanner(System.in);
			Scanner entradaDouble = new Scanner(System.in);

			String nome;
			String endereco;
			int idade;
			double salario;

			System.out.println ("Digite o nome");
			nome = entrada.nextLine();

			System.out.println ("Digite o endereço");
			endereco = entrada.nextLine();

			System.out.println ("Digite a idade");
			idade = entradaInt.nextInt();

			System.out.println ("Digite o salário");
			salario = entradaDouble.nextDouble();
	}
}
```

Questões de Avaliação

1. **Crie uma classe Funcionario, com os atributos nome, endereco, idade, sexo e o método tirar férias (Este método deverá apenas imprimir na tela que o funcionário tirou férias). Em seguida crie uma classe Fábrica para instanciar Funcionários e atribuir valores para as instâncias de Funcionário através de entrada de dados pelo teclado.**
2. **Faça uma classe que receba dois valores numéricos pelo teclado e imprima o resultado das operações de soma, subtração, multiplicação e divisão entre os dois números.**

CAPÍTULO 2

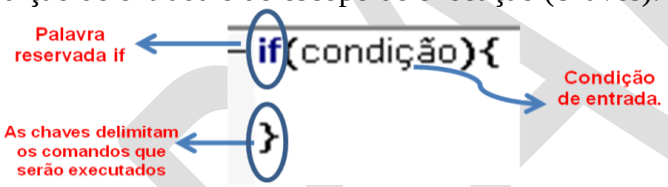
ESTRUTURAS DE DECISÃO E REPETIÇÃO

ESTRUTURAS DE DECISÃO

As estruturas de decisão são utilizadas quando alguma operação (Ex: atribuição, escrever na tela...) está relacionada (depende) da satisfação de uma condição. Ex: Imprimir o nome do funcionário na tela somente se seu salário for igual a 4000. Como recursos de decisão em Java temos a estrutura if, operador ternário e a estrutura switch.

if

O comando if pode ser utilizado para testes de faixa de valor (como x>10) ou testes com valores pontuais (como x==10). A sintaxe do comando if consiste da palavra reservada if, seguida da condição de entrada e do escopo de execução (chaves).



A seguir um exemplo do uso do teste condicional if:

```
public class ExemploTesteCond {
    public static void main(String[] args) {
        int x = 20;
        if(x>10){
            System.out.println("O número é maior que 10");
        }
    }
}
```

O comando de impressão na tela será executado somente se o valor do atributo x for maior que 10. No exemplo a condição é satisfeita, pois o valor de x é 20, e a frase será impressa na tela.

Quando a entrada no if está condicionada a satisfação mais de duas ou mais condições, utilizamos os operadores lógicos. Os principais operadores lógicos em java são:

- &, && - Operadores relacionados ao **e**;
- | , || - operadores relacionados ao **ou**;
- ! – operador relacionado ao **não**.

Diferença entre o operador & e o operador &&

Operador & sempre testa as duas condições. Já o operador &&, se a primeira condição for falsa, ele não testa a segunda e não

executa os comandos do if.
Observe no próximo exemplo que se o valor de x for 2 (por exemplo) o teste condicional irá fazer o teste x>10 e o teste y < 10 não será executado.

```
1  package exemplos;
2
3  public class ExemploTesteCond {
4      public static void main(String[] args) {
5          int x = 20;
6          int y = 5;
7          if((x>10) && (y < 10)){
8              System.out.println("O número é maior que 10");
9          }
10     }
11 }
```

Diferença entre o operador | e o operador ||
O Operador | sempre testa as duas condições. Já o operador ||, se a primeira condição for verdadeira, ele não testa a segunda e executa os comandos do if.

```
import java.util.Scanner;

public class Teste{

        public static void main(String args[]){
            int x = 20;
            int y = 5;
            if(x > 10 || y < 10){
                System.out.println("X é maior que 10");
            }
        }
}
```

Neste caso o teste da segunda condição não é realizado, pois a primeira condição foi satisfeita e o comando de impressão na tela é executado.
O operador not (*!*) também pode ser utilizado no teste condicional.

```
1  package exemplos;
2
3  public class ExemploTesteCond {
4      public static void main(String[] args) {
5          int x = 20;
6          int y = 5;
7          if(!((x>10) && (y < 10))){
8              System.out.println("O número é maior que 10");
9          }
10     }
11 }
```

Através do uso do operador ! no exemplo, o if será executado se uma das condições (ou ambas) não for satisfeita, pois este operador inverte o valor.

else
Às vezes necessitamos que comandos sejam executados somente quando a condição do if não seja aceita. Neste caso necessitamos de um comando else (*se-não*).
O else **sempre** deve ser usado junto com um if e será executado quando a condição do if não é satisfeita.

```java
package exemplos;

public class ExemploSeNao {

    public static void main(String[] args) {
        int idade =30;
        if(idade <= 40){
            System.out.println("Jovem");
        }else{
            if(idade <= 60){
                System.out.println("Meia idade");
            }else{
                System.out.println("Idoso");
            }
        }
    }
}
```

Observe que o else não pode ser colocado separadamente, ele deve vir imediatamente após a chave que fecha o if que ele está aninhado.

Operador Ternário

O operador ternário, como o próprio nome indica, possui três setores que devem especificar o teste a ser realizado, o retorno caso a condição seja satisfeita e o retorno caso a condição não seja satisfeita, respectivamente. O operador ***?*** e ***:*** é utilizado para separar os setores.

Estrutura: *teste?retornoVerdadeiro:retornoFalso;* A seguir um exemplo prático do uso do operador ternário:

```java
import java.util.Scanner;

public class OpTernario {

    public static void main(String[] args) {
        int x;
        Scanner entrada = new Scanner(System.in);

        System.out.println("Digite um valor para x: ");
        x = entrada.nextInt();

        System.out.println(x > 10 ? "Maior que 10" : "Menor que 10" );
    }
}
```

Neste caso se o usuário digitar um valor maior que 10 o texto Maior que 10 será retornado pelo operador, caso contrário o valor Menor que 10 será retornado. Este valor retornado será impresso na tela.

Switch

Quando alguma operação (Ex: atribuição, escrever na tela...) está relacionada a satisfação de uma condição e são possíveis várias condições (casos).

Ex: Imprimir o salário do funcionário na tela somente se seu salário for igual 4000; imprimir sua idade na tela somente se ela for igual a 21;...

O switch pode ser utilizado somente para testes com valores pontuais (como o valor de x igual 10, igual a 11, igual a 15...). ***O switch não pode ser utilizado para testes de faixa de valor (como >10) e pode ser utilizado somente com valores dos tipos int, byte, short ou char.***

A sintaxe do switch é:

```
switch(variavel){
        case valor1:
        case valor2:
}
```

- Inicialmente a palavra switch é utilizada;
- Em seguida temos os parêntesis que com a variável de controle do switch;
- As chaves delimitam os casos e a execução;
- Um switch deve ter vários case que testam um determinado valor;
- Os case possuem dois pontos após o valor;

Exemplo

```
1  package testeSwitch;
2
3  public class TesteSwitch {
4
5      public static void main(String[] args) {
6          int x = 1;
7          switch(x){
8          case 1:
9              System.out.println("número pequeno");
10         case 10:
11             System.out.println("número Grande");
12
13         }
14     }
15 }
```

O resultado da execução deste programa é:

```
número pequeno
número Grande
```

Isto ocorre, pois não utilizamos o comando break. O comando break faz a execução do switch ser interrompida. Caso não o utilizemos para cada case, após a satisfação da condição o switch entrará em todas as opções seguintes. Corrigindo o exemplo anterior temos:

```
1  package testeSwitch;
2
3  public class TesteSwitch {
4
5      public static void main(String[] args) {
6          int x = 1;
7          switch(x){
8          case 1:
9              System.out.println("número pequeno");
10             break;
11         case 10:
12             System.out.println("número Grande");
13             break;
14         }
15     }
16 }
```

Que exibirá como resultado: número pequeno.

Quando quisermos que comandos sejam executados caso nenhuma caso seja satisfeito, utilizamos o caso padrão. Devemos ter as instruções case e por ultimo, ao invés de case valor: devemos utilizar *default:*

```java
package testeSwitch;

public class TesteSwitch {

    public static void main(String[] args) {
        int x = 7;
        switch(x){
        case 1:
            System.out.println("número pequeno");
            break;
        case 10:
            System.out.println("número Grande");
            break;
        default:
            System.out.println("valor não esperado");
            break;
        }
    }
}
```

Neste caso será exibido Valor não esperado.

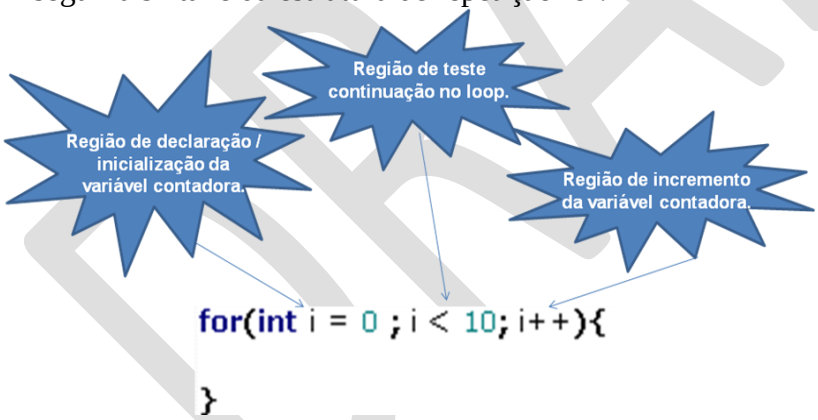
ESTRUTURAS DE REPETIÇÃO

Java disponibiliza três tipos de laço de repetição: for, while e do...while. Eles devem ser utilizados quando necessitamos executar os mesmos comandos uma determinada quantidade de vezes.

NOTA: Faça seus loops de modo que a execução do loop se encerre em determinado momento. Cuidado com loops infinitos.

for

A seguir a sintaxe da estrutura de repetição for:



NOTA: Se seu atributo contador foi definido na estrutura for, ele não poderá ser utilizado após sua execução.

Exemplo do uso do for:

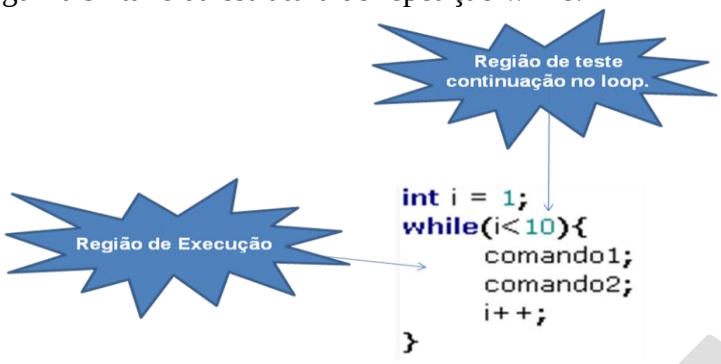
```java
package exemplos;

public class ExemploFor {

    public static void main(String[] args) {
        for(int i = 0; i < 10; i++){
            System.out.println("Iteração "+ i);
        }
    }
}
```

while

A seguir a sintaxe da estrutura de repetição while:



NOTA: Observe que a variável contadora deve ser criada antes do uso da estrutura while e atualizada dentro dela;
Exemplo de uso da estrutura while:

```
1  package exemplos;
2
3  public class ExemploWhile {
4
5      public static void main(String[] args) {
6          int i = 1;
7          while(i <= 10){
8              System.out.println("Iteração "+ i);
9              i++;
10         }
11     }
12 }
```

NOTA: Se apagarmos a linha 9 do programa acontecerá um loop infinito, pois o programa não atualizará o valor de i;

do... while
A seguir a sintaxe da estrutura de repetição while:

```
int i = 1;
do{
    comando1;
    comando2;
    i++;
}while(i <= 10);
```

NOTA: O do...while sempre executará pelo menos uma vez, pois seu teste ocorre no final.
Exemplo do do...while

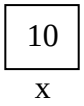
```
1  package exemplos;
2
3  public class ExemploDoWhile {
4
5      public static void main(String[] args) {
6          int i = 1;
7          do{
8              System.out.println("Iteração "+ i);
9              i++;
10         }while(i <= 10);
11     }
12 }
```

NOTA: Se apagarmos a linha 9 do programa java, acontecerá um loop infinito, pois o programa não atualizará o valor de i.

VETORES

Um atributo pode ser visto como uma caixinha que possui um

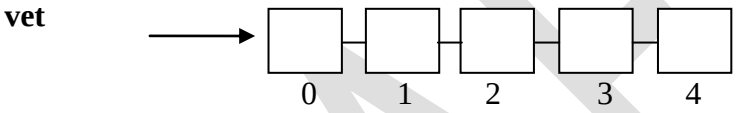
identificador e pode guardar um valor de um determinado tipo. Por tanto se declararmos um atributo int x; na verdade estamos criando um caixinha chamada x que pode guardar números inteiros (o valor 10 por exemplo).



Porém em algumas situações, necessitamos agrupar diversos elementos de **mesmo tipo** em uma única estrutura. Neste caso não podemos utilizar atributos, pois em uma caixinha cabe somente um valor. Necessitamos então de um conjunto de caixinhas de **mesmo tipo**.

A estrutura em Java que cria um conjunto de caixinhas é o vetor. Um vetor é um conjunto de elementos do mesmo tipo, onde cada elemento ocupa uma posição na vetor.

Definimos um vetor em Java da seguinte forma: *int[ ] vet = new int[5];*



Junto ao tipo adicionamos colchetes para explicitar que estamos definindo um vetor. vet é o identificador do vetor. Em seguida criamos um novo vetor através de new, colchetes e a quantidade de posições. Observe que somente na criação (new) definimos a quantidade de posições do vetor.

Existe a possibilidade de criar o vetor e associar valor a todos os seus elementos automaticamente. Da seguinte maneira: int[ ] vet2 = {10, 50, 99); Deste modo, definimos que vet2 é uma referência para um vetor de inteiros, um novo vetor de inteiros de tamanho três é criado e atribuído à vet2. Neste vetor, o elemento da posição 0 tem o valor 10, o elemento da posição 1 tem o valor 50 e o elemento da posição 2 tem o valor 99.

Observe que a primeira posição de qualquer vetor é a posição 0 e que o ultimo elemento do vetor é a posição n-1, onde n é o tamanho do vetor. Portanto, a última posição de um vetor de tamanho 5 é a posição 4.

Para acessarmos os elementos de um vetor é bastante simples, basta utilizar o identificador do vetor seguido de colchetes contendo a posição do elemento a ser acessado. No exemplo anterior, para acessarmos o elemento da posição 1 (segunda posição) do vetor vet2 e imprimi-lo na tela, utilizamos o seguinte comando:

System.out.println(vet2[1]);

E para alterarmos o conteúdo do elemento da posição 2 (terceira posição) do vetor vet, utilizamos o seguinte comando:

vet[2] = 300;

Exemplos:

1) Crie uma aplicação que declare e inicialize um vetor de 5 inteiros lendo seus valores do teclado. Em seguida imprima na

tela quantos elementos iguais a 10 foram lidos.

```
1  package aula_22_09_2010;
2  import java.util.Scanner;
3
4  public class Questao4 {
5      public static void main(String[] args) {
6          Scanner entrada = new Scanner(System.in);
7          int[] v = new int[5];
8          int i = 0;
9          int conta10 = 0;
10
11         while(i < 5){
12             System.out.println("Digite o valor para a posição " + i);
13             v[i] = entrada.nextInt();
14
15             if(v[i] == 10){
16                 conta10++;
17             }
18
19             i++;
20         }
21
22         System.out.println(conta10);
23     }
24
25 }
```

Neste caso, o vetor de tamanho cinco foi criado na linha 7. Na estrutura de repetição que inicia na linha 11 o vetor é populado através da leitura de dados do teclado (linhas 12 e 13) e o controle de quantos valores iguais 10 são lidos é feito pela estrutura condicional da linha 15 através de um atributo chamado conta10 (definido na linha 9). Ao final, a quantidade de valores lidos iguais a 10 é impresso na linha 22.

2) Crie um método que receba um vetor de inteiros e seu tamanho e retorne o valor da soma de seus elementos.

```
1  public class Vetor {
2      public static int somaElementos(int[] v, int n){
3          int result = 0;
4          for(int i = 0; i < n; i++){
5              result = result + v[i];
6          }
7          return result;
8      }
9
10     public static void main(String[] args) {
11         int[] vet = {10, 2, 4, 7};
12         System.out.println(somaElementos(vet, 4));
13     }
14 }
```

O método chama-se somaElementos e em sua execução ele percorre o vetor da primeira posição (posição 0) até a ultima posição (posição n-1) realizando a soma de cada elemento em um atributo chamado result (definido na linha 3).

Questões de Avaliação

1. **Faça um programa que receba um número inteiro e verifique se é par ou ímpar.**
2. **Escreva um método que receba um número e retorne seu fatorial.**
3. **Faça um programa que monte os quinze primeiros termos da sequência de fibonacci. Utilize uma estrutura de repetição. A seguir a sequência: 0 – 1- 1- 2- 3-5- 8...**
4. **Crie um método que receba um vetor de inteiros e seu tamanho e retorne o valor da média aritmética de seus elementos.**

CAPÍTULO 3

VISIBILIDADE E ENCAPSULAMENTO

VISIBILIDADE

Visibilidade diz respeito ao local onde um determinado atributo/método ou classe pode ser acessado. Como um atributo, método e classe podem ser acessados?

Há basicamente três maneiras de acessar (utilizar) um atributo ou método:

- Dentro da própria classe que o está definindo;
- Através da instância (objeto) da classe que o definiu;
- Através de um mecanismo chamado de herança (será abordado na próxima unidade).

Há duas maneiras de utilizar uma classe:

- Através da instância da classe;
- Através de um mecanismo chamado de herança (será abordado na próxima unidade).

Exemplo do acesso de atributos e métodos dentro de um método da própria classe:

```
3  public class TesteAcessoNaClasse {
4      int x;
5      int y;
6      int result;
7
8      public void soma(){
9          result = x + y;
10         /* Acesso dos atributos de
11          * instância dentro da própria classe.*/
12     }
13
14     public void somaMaisUm(){
15         soma(); //Acesso de um método dentro da própria classe.
16         result++;
17     }
18 }
```

Exemplo do acesso de atributos e métodos através da instância da classe:

```
1  package testes;
2
3  public class Base {
4      int x;
5      public void imprime(){
6          System.out.println("OK");
7      }
8  }
```

```java
package testes;

public class Execucao {

    public static void main(String[] args) {
        Base b = new Base();
        b.x = 10;
        // Acesso a atributo através da instância da classe.

        b.imprime();
        // Acesso a método através da instância da classe.
    }
}
```

Java disponibiliza quatro níveis de visibilidade: pública, privada, padrão e protegida. A seguir os níveis de visibilidade serão descritos:

Visibilidade	Aplicado à	Permite Acesso	Palavra Reservada
Pública	Classes, Métodos e Atributos	A partir de qualquer classe	public
Privada	Métodos e Atributos	Apenas dentro da classe que os define	Private
Padrão	Classes, Métodos e Atributos	Apenas a classes do mesmo **pacote**	-
Protegida	Métodos e Atributos	Por classes do mesmo pacote através de herança ou de instanciação; e Por classes de outros pacotes apenas através de herança;	protected

Pacote é um diretório que agrupa as classes no sistema operacional. No código java, informamos que uma classe pertence a um pacote através da palavra reservada package (Veja exemplo anterior).

A seguir, um exemplo relacionado à visibilidade:

```java
package aula9_1;

public class Base {
    /* A seguir, quatro atributos são definidos
     * com as quatro visibilidades possíveis*/
    public int publico;
    protected int protegido;
    String padrao;
    private int privado;
    /* A seguir, quatro métodos são definidos
     * com as quatro visibilidades possíveis*/
    public void imprimePublico(){
        System.out.println("Publico");
    }
    void imprimePadrao(){
        System.out.println("Padrao");
    }
    protected void imprimeProtected(){
        System.out.println("Protected");
    }
    private void imprimePrivado(){
        System.out.println("Privado");
    }
}
```

```java
package aula9_1;

public class InstanciaPacote {

    public static void main(String[] args) {
        Base t = new Base();
        t.publico = 9;
        t.protegido = 5;
        t.padrao = "ANA";
        t.privado = 2;
        /* Não é possível acessar
         * um atributo privado em outra classe*/
        t.imprimePublico();
        t.imprimePadrao();
        t.imprimeProtected();
        t.imprimePrivado();
        /* Não é possível acessar
         * um método privado em outra classe*/
    }
}
```

```java
package aula9_2;
import aula9_1.Base;

public class Instancia {

    public static void main(String[] args) {
        Base t = new Base();
        t.publico = 2;
        /* Os atributos com visibilidade protegida, privada e padrão
         * não podem ser acessados em classes de outro pacote através
         * da instância de Base*/
        t.protegido = 5;
        t.padrao = 2;
        t.privado = "Ana";
    }
}
```

Algumas observações:

- No exemplo, as classes Base e InstanciaPacote estão no mesmo pacote (aula9_1), já a classe Instancia está em outro pacote (aula9_2);
- Dentro dos métodos da classe Base, seus atributos de instância podem ser acessados independente da visibilidade que possuem.
- As linhas 12, 13 e 14 da classe Instancia apresentam erro, pois a visibilidade dos atributos não permite o acesso em classes de outro pacote através do objeto.

Lembre-se que visibilidade é diferente de escopo! Visibilidade envolve palavras reservadas que restringem o acesso.
Além disto, os modificadores de visibilidade (public, protected e private) pode ser aplicados somente à membros (Atributos e métodos) de instância ou de classe, portanto atributos de método **não** podem ter os modificadores public, protected e private.

Erro!
Não podemos associar modificadores de acesso aos atributos de método!

```java
public class Operacao {
    public int x;
    private int y;
    protected int z;
    int w;
    public static int h;
    private static int i;
    protected static int j;
    static int k;

    public int soma(){
        public int a;
        private int b;
        protected int c;
        int k = 5;
        return(x+y+z+w);
    }

    private int subtracao(){
        return(x-y-z-w);
    }

    protected int multiplicacao(){
        return(x*y*z*w);
    }

    int divisao(){
        return(x/y/z/w);
    }
}
```

ENCAPSULAMENTO

Uma classe pode ter atributos (dados) e métodos (operações). Em muitos casos será desejável que os dados não possam ser acessados ou usados diretamente, mas somente através das operações cuja especialidade será a manipulação destes dados
Como analogia, vamos considerar uma câmera fotográfica automática. Quando um usuário da câmera clica o botão para tirar uma foto, diversos mecanismos entram em ação que fazem

com que a velocidade e abertura apropriada do obturador sejam selecionadas, levando em conta o tipo do filme e as condições de iluminação. Para o usuário, os detalhes de como os dados como velocidade, abertura, tipo de filme e iluminação são processados são irrelevantes, o que interessa a ele é que a foto seja tirada.
O mecanismo de seleção da câmera oculta os dados (iluminação, tipo de filme, abertura, velocidade, etc.) e a maneira com que estes são processados, não deixando que o usuário modifique-os dados à vontade. A câmera deixa para o usuário somente uma maneira simplificada de efetuar os cálculos (mais exatamente, pedir à câmera que calcule os dados) e tirar a foto de acordo com os resultados obtidos: o botão que aciona o obturador.

A capacidade de ocultar dados dentro de classes, permitindo que somente métodos especializados (ou dedicados) manipulem estes dados ocultos chama-se **encapsulamento**. Classes que encapsulam os dados possibilitam a criação de programas com menos erros e mais clareza.
Os métodos que manipulam os dados em classes encapsuladas recebem o nome de get e set. Os métodos get retornam o valor do atributo e os métodos set alteram o valor do atributo.
Supondo que o atributo se chame nome, seus métodos de acesso serão setNome e getNome. No caso de um atributo chamado endereco, seus métodos de acesso serão setEndereco e getEndereco.
Exemplo de uma classe encapsulada e do acesso aos atributos da classe via gets e sets:

```
1  package encapsulamento;
2
3  public class ClasseEncapsulada {
4      // atributos privados
5      private int x;
6      private int y;
7
8      // métodos de acesso
9      public int getX(){
10         return(x);
11     }
12     public void setX(int valorX){
13         x = valorX;
14     }
15
16     public int getY(){
17         return(y);
18     }
19     public void setY(int valorY){
20         y = valorY;
21     }
22 }
```

```java
package encapsulamento;

public class AcessaEncapsulada {

    public static void main(String[] args) {
        ClasseEncapsulada ce = new ClasseEncapsulada();

        // Atribuindo o valor 12 para o atributo x através de setX
        ce.setX(12);

        // Atribuindo o valor 100 para o atributo y através de setY
        ce.setY(100);

        // Acessando os valores de x e y através dos métodos get
        System.out.println(ce.getX());
        System.out.println(ce.getY());
    }

}
```

- Observe que os métodos set irão alterar o valor, então tem que receber um argumento do mesmo tipo do atributo para modificá-lo.

- Observe que os métodos get irão devolver o valor, então tem que retornar um valor do mesmo tipo do atributo e retorná-lo.

Questões de Avaliação

1. **Crie uma classe para representar uma casa e crie instâncias desta classe em uma classe de execução chamada cidade. OBS.: A classe casa deve possuir atributos de acordo com os cômodos que possui e cada atributo deverá possuir visibilidades diferentes.**
2. **Encapsule a classe Funcionário do exercício 4 do capítulo 2 da unidade 1.**
3. **Crie uma classe chamada ContaCorrente que possuirá número, agência e saldo. Esta classe deverá ser encapsulada.**

Síntese da Unidade

Nesta unidade foi dada continuidade ao estudo da linguagem de programação Java através dos conceitos de leitura do teclado via prompt de comando, este conceito é importante para tornar nossos programas capazes de interagir com o usuário através do recebimento de valores. As estruturas de decisão if, if...else e switch e as estruturas de repetição for, while e do...while foram demonstradas através da sintaxe e de exemplos práticos. Finalmente, visibilidade e encapsulamento conceituados e exemplificados.

Bibliografia

ECKEL, B. **Pensando em Java,** 3ª ed, Editora MindView;

SANTOS, R. **Introdução à Programação Orientada a Objetos Usando Java**, Editora Campus;

DEITEL, H. M. e DEITEL, P.J. **Java: Como programar**, 6ª ed, Editora Pearson;

HORSTMANN, C. e CORNEL, G. **Core Java** – Volume I. Fundamentos. Alta Books. 7ª. 2005;

3

UNIDADE

HERANÇA, POLIMORFISMO E TRATAMENTO DE EXCEÇÕES

Nesta unidade continuamos estudando os conceitos da Programação Orientada a Objetos aplicados à Linguagem de programação Java. Conheceremos as principais técnicas OO para reutilização de código através da herança de classes, criação de código polimórfico e como tornar nossos programas mais robustos com os mecanismos de tratamento de exceções.

CAPÍTULO 1

HERANÇA

Neste capítulo conheceremos um dos principais recursos das linguagens orientadas a objetos, herança.
Quando programamos pensando em objetos separamos todos os elementos do espaço do nosso problema em classes, para então criarmos, ou instanciarmos nossos objetos. Mas ao classificarmos nossos objetos podemos também relacioná-los conceitualmente.
Normalmente quando pensamos no conceito de alguma classe podemos generalizá-la ou especializá-la. Por exemplo, a classe dos macacos. Podemos ter uma classe mais genérica, os mamíferos. Ou, tomado o sentindo oposto podemos ter um macaco mais específico, um chimpanzé. O que temos em comum a todas essas classes é a relação de generalização ou de especialização. Um mamífero é uma idéia mais genérica que um macaco que é mais genérico que um chimpanzé, ou o contrário, um chimpanzé, é mais específico que um macaco que é mais específico que um mamífero. Mas o que isso tem a ver com as nossas linguagens de programação? Herança.
Todo chimpanzé é um macaco e todo macaco é um mamífero. Desta forma eles guardam consigo características em comum. Todo mamífero mama, portanto, todo macaco mama e todo chimpanzé também.
Ao criarmos a classe dos macacos como uma especialização dos mamíferos, criamos uma classe a partir de outra, portanto, a nova classe herdará todos os atributos e métodos da classe mais genérica. A isso damos o nome de reutilização de código ou herança.
Com a reutilização de código temos os seguintes principais benefícios:

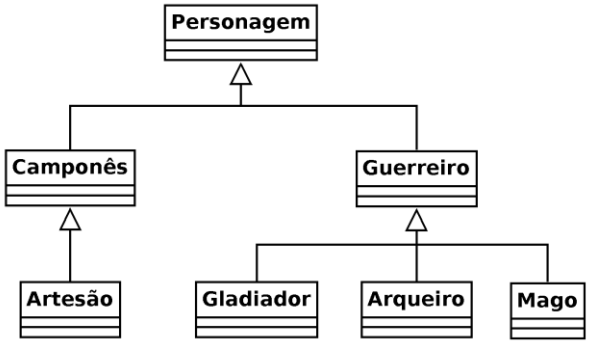
- Economia de tempo de programação e
- Construção de códigos mais confiáveis, pois serão baseados em outros já testados;

Ao criamos uma classe utilizando outra como base precisamos apenas adaptar as diferenças ou adicionar os novos atributos e métodos.

Hierarquia de classes

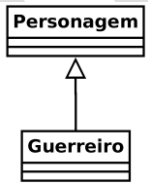
Analisaremos o uso da técnica de herança na elaboração de um pequeno jogo de computador. No qual temos alguns personagens que realizam determinadas tarefas.
Neste jogo os personagens possuem os papéis de um guerreiro, um camponês, um gladiador, um artesão, um mago ou um

arqueiro. Eles pertencem a tribos e estas precisam defender-se de ataques externos ou atacar outras tribos para conseguir mais recursos para sua sobrevivência.
Imediatamente identificamos uma relação entre os diversos papéis dos personagens do jogo e classes em Java. Cada um dos papéis podem ser classes de objetos no jogo. Além disso, identificamos semelhanças e diferenças entre os diversos papéis. Por exemplo, um arqueiro deve possuir mais atributos em comum com um gladiador que com um artesão. Mas mesmo papéis distintos podem ser agrupados todos como sendo personagens. Desta forma podemos estabelecer uma hierarquia entre os diversos tipos do jogo, como na figura abaixo.



As setas apontam para as classes mais genéricas.
A classe Personagem seria a mais genérica de todos. E todos os outros elementos são um tipo de personagem. Os papéis Gladiador, Arqueiro e Mago foram agrupados como sendo Guerreiros. E um artesão como um tipo especial de camponês.
Como a construção de toda essa hierarquia de classe está fora do escopo deste capítulo, analisaremos apenas alguns ramos. Analisaremos primeiramente o seguinte ramo:

O Java não permite herânca múltipla. Uma classe só pode herdar de apenas uma classe. Veremos na próxima unidade como podemos associar a uma classe diversas interfaces, de forma a simular herança múltipla.



Superclasses x subclasses

O código Java para a hierarquia de classes desta figura segue logo abaixo.
Como todas as classes são públicas criamos um arquivo exclusivo para cada uma delas, com os mesmos nomes das classes.

```java
// Figura 3.1.1: Personagem.java
// A classe Personagem

public class Personagem {
    private char   sexo;
    private double energia;
    private String nome;
    private int    x, y;

    // Retorna o sexo do personagem
    public char getSexo()
    {
        return sexo;
    }

    // Retorna a energia do personagem
    public double getEnergia()
    {
        return energia;
    }

    // Configura a energia do personagem
    public void setEnergia(double energia)
    {
        if (energia >= 0 || energia <= 100)
            this.energia = energia;
    }

    // Retorna o nome do personagem
    public String getNome()
    {
        return nome;
    }

    // Move o personagem da sua posição atual para a nova
    // posição informada
    public void mover(int x, int y)
    {
        int stepX = 0, stepY = 0;

        stepX = this.x < x ? 1 : -1;
        stepY = this.y < y ? 1 : -1;

        while (this.x != x)
            this.x += stepX;

        while (this.y != y)
            this.y += stepY;
    }

    // Construtor padrão
    Personagem()
    {
        sexo = 'm';
        energia = 0.0;
        nome = "Person";
        x = y = 0;
    }

    // Construtor com 5 parâmetros
    Personagem(char sexo, double energia, String nome, int x, int y)
    {
        this.sexo = sexo;
        this.energia = energia;
        this.nome = nome;
        this.x = x;
        this.y = y;
    }
}
```

A variável this é uma variável especial, ela é uma referencia a instância do objeto em execução. Como a classe Personagem possue um atributo com o mesmo nome do parâmeto do método. O this esta sendo utilizado para diferenciar o atributo do parâmetro do método.

Estes métodos com o mesmo nome da classe são chamados construtores da classe. Eles são chamados automaticamente pelo compilador ao criarmos um objeto do tipo da classe. Sua função é inicializar os valores dos atributos da classe afim de garantir que os objetos estejam em um estado consistente desde o início do seu uso.

A classe Personagem possui os atributos sexo, energia, nome e as posições x e y. Todos estes atributos foram definidos como private, limitando a visibilidade destas variáveis ao escopo da classe. Para obtermos os valores destes atributos fora da classe foram criados os métodos getSexo(), getEnergia(), getNome(). O nome, a energia, o sexo e a posição do personagem podem ser definidos no momento da criação de algum objeto desta classe, através do construtor com cinco parâmetros, um para cada atributo. Caso tenhamos que alterar o nível de energia de algum personagem devemos chamar o método setEnergia(). Ele garante

que o nível de energia de qualquer personagem sempre estará entre zero e cem. Para modificarmos a posição de algum personagem chamamos o método mover(). Este irá mover gradativamente o personagem para a nova posição.

Sobrecarga de métodos

Um outro recurso utilizado na codificação da classe Personagem foi a sobrecarga de métodos. Observe que nesta classe temos dois métodos com o mesmo nome. Os dois métodos são os construtores da classe. Ambos possuem o mesmo nome da classe. A sobrecarga de métodos permite criarmos vários métodos com o mesmo nome, no entanto, devem possuir assinaturas diferentes. Ou seja, apesar dos métodos terem o mesmo nome eles devem ser diferentes em pelo menos um dos seguintes critérios:

- A quantidade de argumentos;
- o tipo dos argumentos;
- ou o tipo de retorno do método.

Ao chamarmos um método sobrecarregado o compilador Java saberá diferenciar qual método utilizar de acordo com os critérios listados acima.
Normalmente utilizamos métodos com o mesmo nome quando estes realizam as mesmas tarefas ou são muito semelhantes, como o caso dos nossos dois construtores. Ambos são utilizados para inicializar nossos objetos do tipo Personagem, porem podemos fazer isso de duas formas, uma atribuindo valores padrões, ou atribuindo valores específicos que desejarmos.

Estendendo classes

Como a classe Personagem está no nível mais alto da nossa hierarquia, esta não estende nenhuma outra classe. Diferentemente da classe Guerreiro, codificada a seguir.

```
1// Figura 3.1.1: Guerreio.java
2// A classe Guerreiro
3
4public class Guerreiro extends Personagem {
5    private double forca;
6
7    // Construtor padrão
8    Guerreiro()
9    {
10       super('m', 70.0, "Guerreiro", 0, 0);
11       forca = 10.0;
12   }
13
14   // Construtor com 6 parâmetros
15   Guerreiro(char sexo, double energia, String nome, int x, int y,
16           double forca)
17   {
18       super(sexo, energia, nome, x, y);
19       this.forca = forca;
20   }
21
```

Apesar da classe Personagem não estender uma classe ela herda da classe Object. Toda a hierarquia de classes em Java é oriunda desta classe. Portanto toda classe em Java herda direta ou indiretamente da classe Object.

```
22      // Retorna a forca do Guerreiro
23      double getForca()
24      {
25          return forca;
26      }
27
28      // Diminui a energia do adversário de acordo com a força
29      public void bater(Guerreiro g)
30      {
31          g.setEnergia(g.getEnergia() - forca);
32
33          setEnergia(getEnergia()-1);
34      }
35
36      // Aumenta a energia do guerreiro
37      public void alimentar(double energia)
38      {
39          setEnergia(getEnergia() + energia);
40      }
41  }
```

Observe na linha 4 do arquivo da classe Guerreiro a palavra chave **extends**. Através deste comando fazemos com que a classe Guerreiro estenda a classe Personagem. Ou seja, a classe Guerreiro terá todo o código da classe Personagem como parte do seu próprio código além do código incluído na classe propriamente dita. Sendo um guerreiro uma especialização de um personagem, além da força definida na linha 5, um guerreiro possui uma posição (x,y), um nome, um nível de energia e um sexo. Podendo-se ainda movê-lo e modificar o seu nível de energia.

Além dos métodos da classe base, foram acrescentados os métodos bater() e alimentar(). Estes modificam o nível de energia de um oponente ou o do próprio guerreiro. Ambos os métodos fazem chamadas aos métodos setEnergia() e getEnergia() herdados de Personagem.

Utilizando a nomenclatura da linguagem Java, dizemos que a classe Personagem é superclasse da classe Guerreiro. Por isso o uso da palavra chave 'super' nos construtores da classe Guerreiro. Sempre que precisamos utilizar algum método ou atributo da superclasse podemos utilizar a referência 'super' para acessá-los. Além disso, se Personagem é superclasse de Guerreiro então dizemos que Guerreiro é uma subclasse de Personagem.

Apesar da idéia original do jogo um Guerreiro puder lutar com qualquer outro tipo de personagem, inclusive guerreiros, o método bater() espera como parâmetro apenas um outro guerreiro. Veremos no próximo capítulo, com a técnica de polimorfismo, como poderemos modificar este tipo de código para funcionar da forma esperada.

Membros Protected

Vamos incluir um novo personagem ao código do jogo. O novo elemento será o arqueiro. Seu objetivo é lutar ou defender sua tribo. Diferentemente de um guerreiro qualquer, um arqueiro luta utilizando flechas e estas possuem uma quantidade limitada.

Mas antes de analisarmos o código da nova classe Arqueiro, precisaremos modificar a classe Personagem para que permita o acesso aos atributos energia, posições x e y. Isso por que um

arqueiro sendo mais ágil poderá mover-se mais rápido que outro personagem como um camponês ou gladiador. E ao usufruir de porções mágicas, estas lhe garantirão energia além do comum.
O acesso direto a estes atributos está restrito a classe Personagem, pois foram declarados como private. E os métodos públicos de Personagem mover() e setEnergia() não permitem manipular os atributos como desejado. Desta forma, modificaremos o acesso às variáveis energia, x e y da classe como a seguir:

```
protected double energia;
protected int    x, y;
```

A alteração consiste apenas na modificação do especificador de acesso. Antes era private agora **protected**. Este novo especificador é um meio termo entre os especificadores public e private. Atributos protected podem ser acessados de dentro da classe na qual foram declarados bem como a partir dos descendentes desta classe. Qualquer outro ponto do código não possui visibilidade a estes atributos. As variáveis energia, x e y continuam restritas a classe Personagem, mas agora abertas também para todos os descendentes dela.

Sobrescrita de métodos

Agora conheceremos como podemos modificar um método já codificado realizando apenas as adaptações necessárias com a sobrescrita de métodos.
Com a modificação da classe Personagem podemos analisar a classe Arqueiro. Seu código segue abaixo.

```
1 // Figura 3.1.1: Arqueiro.java
2 // A classe Arqueiro
3
4 public class Arqueiro extends Guerreiro {
5     private int flechas;
6
7     // Construtor padrão
8     Arqueiro()
9     {
10        super('m', 70.0, "Arqueiro", 0, 0, 10);
11        flechas = 5;
12    }
13
14    // Retorna a quantidade de flechas disponíveis
15    public int getFlechas()
16    {
17        return flechas;
18    }
19
20    // Acrescenta mais flechas
21    public void incFlechas(int flechas)
22    {
23        this.flechas += flechas;
24    }
25
26    // Aumenta a energia do arqueiro acima do normal
27    public void beberElixir(int quant)
28    {
29        energia += quant;
30    }
31
```

```
32      // Diminui a energia do adversário de acordo com a força
33      @Override // Sobreescreve o método da superclasse
34      public void bater(Guerreiro g)
35      {
36          super.bater(g);
37
38          flechas--;
39      }
40
41      // Move o arqueiro da sua posição atual para a nova
42      // posição informada
43      @Override
44      public void mover(int x, int y)
45      {
46          int stepX = 0, stepY = 0;
47
48          stepX = this.x < x ? 5 : -5;
49          stepY = this.y < y ? 5 : -5;
50
51          while (this.x != x &&
52                  Math.abs(Math.abs(this.x)-Math.abs(x))>=Math.abs(stepX))
53              this.x += stepX;
54
55          while (this.y != y &&
56                  Math.abs(Math.abs(this.y)-Math.abs(y))>=Math.abs(stepY))
57              this.y += stepY;
58      }
59  }
```

A palavra chave @Override ajuda ao compilador identificar o métodos a serem sobrescritos. Ela está disponível apartir da versão 5 do Java.

A classe Arqueiro foi definida como subclasse de Guerreiro. Suas flechas estão na variável privada flecha com os métodos incFlecha(). Este deverá ser chamado caso um guerreiro encontre novas flechas para incrementa à sua sacola de flechas. O método getFlecha() informa ao jogador a quantidade de flechas ainda disponíveis. Por padrão todo arqueiro começa com cinco flechas, como especificado no construtor padrão da classe. Observe que os métodos beberElexir() e mover() alteram diretamente as variáveis energia, x e y herdadas de Personagem. Sendo as variáveis x e y incrementadas cinco posições por vez, isso torna o Arqueiro mais veloz que os outros personagens.

Observe também que os métodos herdados bater() e mover() estão sendo redefinidos na classe Arqueiro. Esta reescrita sobrescreve os métodos herdados. Ou seja, agora ao chamarmos os métodos bater() ou mover() para um objeto Arqueiro não iremos mais utilizar os métodos das superclasses, mas o sobrescritos na classe Arqueiro.

O método mover() substitui completamente o método sobrescrito. Já o método bater() realiza uma pequena alteração. Ele diminui as flechas disponíveis reaproveitando o código herdado ao chamá-lo na linha 35.

CAPÍTULO 2

POLIMORFISMO

Neste capítulo conheceremos outro conceito-chave da programação orientada a objetos, o polimorfismo. No capítulo anterior aprendemos a criar uma hierarquia de classes para os elementos dos nossos programas. Até então nosso objetivo era reutilizar código de classes mais genéricas para construirmos classes mais específicas. Agora seguiremos um caminho inverso na forma de utilizar nossos objetos. Em vez de programarmos no específico programaremos no geral.

Considere por exemplo a classe dos veículos. A partir dessa classe podemos criar diversas outras mais específicas, como navios, carros, aviões, etc. Uma capacidade, portanto um método, que todo veículo possui é locomover-se. Desta forma, criar classes mais específicas a partir de veículo, necessariamente, deve-se sobrescrever este método, pois um barco flutua e desloca-se sobre a água, um carro roda sobre uma via e um avião voa.

Em um sistema com diversos objetos do tipo veículo, referenciados por um vetor deste mesmo tipo, podemos locomover todos eles uniformemente. Observe que quando pensamos em locomover os veículos do nosso vetor estamos abstraindo qualquer veículo. Não importa se cada um dos objetos do vetor sejam um barco, um carro ou um avião. Ao chamarmos o método locomover de cada um deles, eles saberão comporta-se adequadamente de acordo com o seu subtipo. Desta forma podemos constatar que a mesma mensagem enviada a diversos objetos de um mesmo tipo, veículo, tem muitas formas de resultado, isso é polimorfismo.

Exemplo de polimorfismo

Demonstraremos o comportamento polimórfico em Java adicionando alguma ação ao jogo de computador iniciado no capítulo anterior. Mas antes vamos incluir mais dois personagens ao jogo: mago e camponês. A classe do personagem mago segue abaixo.

```java
// Figura 3.1.1: Mago.java
// A classe Mago

public class Mago extends Guerreiro {

    // Construtor padrão
    Mago()
    {
        super('m', 90.0, "Mago", 0, 0, 15);
    }

    // Construtor com 6 parâmetros
    Mago(char sexo, double energia, String nome, int x, int y,
            double forca)
    {
        super(sexo, energia, nome, x, y, forca);
    }

    // Diminui a energia do adversário de acordo com a força
    @Override // Sobreescreve o método da superclasse
    public void bater(Guerreiro g)
    {
        super.bater(g);
        super.bater(g);

        energia -= (getForca() - 2);
    }

    // Move o mago da sua posição atual para a nova
    // posição informada
    @Override
    public void mover(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void decansar()
    {
        energia+=3;
    }
}
```

A classe Mago da mesma forma que a classe Arqueiro também estende a classe Guerreiro. Nela, temos dois construtores, um padrão e um com a possibilidade de configurarmos todos os atributos da classe. O padrão é criarmos Magos com um pouco mais de energia que os demais Guerreiros. Isso por que um mago utiliza parte da sua energia para combater os seus inimigos. Este fato está codificado no método bater(). Um mago é capaz de ferir duas vezes mais que um arqueiro, no entanto consome da sua energia o equivalente a uma força sua. Desta forma, precisa de um período de descanso para recuperar suas energias. Para tanto, temos o método descansar(). Este deve ser chamado ocasionalmente para recompor as energias do Mago.
Outra diferença entre um Mago e um Arqueiro é o fato de poder se tele-transportar. O método mover altera imediatamente a posição do mago, ao contrário dos outros Personagens onde se movimentavam gradativamente.
A seguir temos o código da classe camponês.

```java
// Figura 3.1.1: Campones.java
// A classe Campones

public class Campones extends Personagem {

    // Construtor padrão
    Campones()
    {
        super('m', 50.0, "Campones", 0, 0);
    }

    // Construtor com 5 parâmetros
    Campones(char sexo, double energia, String nome, int x, int y)
    {
        super(sexo, energia, nome, x, y);
    }
}
```

Esta classe é bem simples. Reaproveita todos os métodos e atributos de Personagem. Isso por que um camponês é o mais elementar dos personagens. Ele não possui nenhuma habilidade ou característica especial.
Vamos agora, criar uma classe Arena para dar alguma atividade ao jogo. Para o nosso estudo sobre polimorfismo iremos acionar os mecanismos de mobilidade dos personagens.
O código da classe Arena segue abaixo:

```java
// Figura 3.1.1: Guerreio.java
// A classe Jogo

import java.util.Scanner;
import java.util.Random;

public class Jogo {
    Personagem[] bots;
    Personagem jogador;

    /* Construtor padrão */
    Jogo()
    {
        bots = new Personagem[5];
    }

   /* Configurações do Jogo */
   void config()
   {
       /* Escolhendo personagem do jogador */
       Scanner entrada = new Scanner(System.in);
       System.out.print("Escolha um personagem:\n");
       System.out.print(" 0 => Campones\n 1 => Arqueiro\n 2 => " +
               "Mago\nOpcao[Default:Campones]:");

       int op = entrada.nextInt();
       switch (op)
       {
           case 2:
               jogador = new Mago();
               break;
           case 1:
               jogador = new Arqueiro();
               break;
```

```
35              case 0:
36              default:
37                  jogador = new Campones();
38          }
39
40          /* Escolhendo aleatoriamente os adversários */
41          Random randNum = new Random();
42          for (int i=0; i<5; i++)
43          {
44              op = randNum.nextInt(3);
45              switch (op)
46              {
47                  case 2:
48                      bots[i] = new Mago();
49                      break;
50                  case 1:
51                      bots[i] = new Arqueiro();
52                      break;
53                  case 0:
54                  default:
55                      bots[i] = new Campones();
56              }
57          }
58      }
59
60      /* Ação do Jogo */
61      void run()
62      {
63          int x, y;
64
65          /* Mover personagem do jogador */
66          Scanner entrada = new Scanner(System.in);
67          System.out.print("Digite a nova posicao:\n");
68          System.out.print("x => ");
69
70          x = entrada.nextInt();
71          System.out.print("y => ");
72
73          y = entrada.nextInt();
74
75          System.out.println("Movendo o seu personagem:" +
76                  jogador.getNome());
77          jogador.mover(x, y);
78
79          /* Mover outros personagens */
80          Random randNum = new Random();
81
82          for (int i=0; i<5; i++)
83          {
84              x = randNum.nextInt(80);
85              y = randNum.nextInt(25);
86
87              System.out.println("Movendo o personagem:" +
88                      bots[i].getNome() + Integer.toString(i));
89              bots[i].mover(x, y);
90          }
91      }
92  }
```

O método .nextInt(int) de Random retorno um número aleatório entre 0 e o primeiro inteiro anterior ao valor do parâmetro informado.

A classe arena possui um vetor com cinco adversários chamado bots, e uma variável jogador para o guerreiro do usuário. No construtor da classe é criado o vetor de personagens. No entanto, os bots serão escolhidos aleatoriamente somente no momento de inicialização do jogo com a chamada ao método config(). Neste mesmo método também é perguntado ao usuário com qual personagem ele deseja jogar. Observe que tanto o vetor bots como a variável jogador são do tipo Personagem.

Vamos agora analisar o método run(). A ação do jogo consiste em movimentar as peças do jogo por uma arena ou tabuleiro imaginário. O código do método run() pode ser dividido em duas seções: movimento do jogador e movimento dos bots. A

primeira parte do método pergunta ao usuário para onde deseja mover o seu personagem. Definida a nova posição é chamado o método mover() do jogador e este altera sua posição para o novo local. A segunda parte do código consiste no mesmo processo, porém para o bots. Como estes são controlados pelo computador, suas posições são definidas aleatoriamente. A chamada ao método mover(), dos bots, atualiza as posições de cada um dos cinco adversários.
Devemos observar que todas as chamadas ao método mover() partiram de variáveis(bots e jogador) do tipo Personagem, porém em nenhum momento criamos qualquer objeto deste tipo. Todos os objetos criados são Magos, Arqueiros ou Camponeses.
Mas esperamos que cada um dos Personagens mova-se de acordo com o tipo deles. Um mago se tele-transporta, um arqueiro corre e um camponês caminha. E isso é o comportamento polimórfico no código run().
Quando estamos programando a chamada ao método mover() dentro de run() não sabemos qual método será efetivamente executado. Não conhecemos até o momento da inicialização do jogo o tipo dos objetos referenciados pelo vetor bots ou pela variável jogador. Estamos pensando globalmente, ou seja, estamos apenas movendo as peças do jogo, estamos movendo personagens, mas agindo localmente, cada uma das peças sabe como mover-se.
Dinamicamente, durante a execução do código run(), é determinado qual método mover() deverá ser chamado de acordo com o subtipo do objeto referenciado pelas variáveis.

Classes e métodos abstratos

No exemplo anterior o método mover() estava implementado em todas as classes do jogo. No entanto, nem sempre é possível conhecermos todo o código de uma classe. Para estudarmos este tipo de situação vamos adicionar mais um método às classes no nosso jogo. Agora criaremos o método exibir() com uma imagem para as classes dos nossos personagens: Mago, Arqueiro e Campones. Da mesma forma que tínhamos o método mover() polimórfico, exibir() também deverá ser. Portanto devemos incluí-lo nas superclasses Personagem e Guerreiro. No entanto elas serão abstratas. Pois não sabemos como exibir um Personagem ou um Guerreiro qualquer. O código para cada uma das classes segue abaixo. Primeiramente vamos alterar a classe Personagem.

```
abstract public class Personagem {
```

No início da declaração da classe Personagem adicionamos a palavra chave abstract, e no final do corpo da classe após o método mover() adicionamos o método exibir(), abaixo:

```
70
71      abstract void exibir(char[] tabuleiro);
72  }
```

A declaração do método exibir() está correta, ele não possui código. Este é um método abstrato, sem código.
Também alteramos a classe Guerreio, adicionando a palavra chave abstract no início da declaração da classe. O restante da classe permanece igual à anterior. Ela também é abstrata porque não possui nenhum código para o método exibir() herdado de Personagem.

```
abstract public class Guerreiro extends Personagem {
```

Acrescentamos os métodos exibir() em cada um dos três tipos de personagens criados. Os métodos imprimem na tela um ícone em ascii-art representando uma imagem para cada um dos personagens.
Para o camponês temos um martelo,

```
public void exibir()
{
    System.out.print("                              ?MMMMMM8 .\n" +
                     "                           8:. .=MMMII=M=    .\n" +
                     "                        M.  MNN~D$,D  .+MM$MIZ\n" +
                     "                    .M  77,$8DI$Z 7MD+I$7 M\n" +
                     "                ..  =7+8DOMMII7.M  .\n" +
                     "              M .=ND8= Z7M$:NMD Z\n" +
                     "           M. N=?OD8Z=7$.OD8NDNM\n" +
                     "            N$D+IDDM=  =M  8=.\n" +
                     "           .~M7,M,M ...  .M      $\n" +
                     "           N .D=MN .M            ..\n" +
                     "            M  N  N         M      .\n" +
                     "              M.MO           .       .\n" +
                     "                             7        .\n" +
                     "                              =       ,.\n" +
                     "                              ~        ~\n" +
                     "                               +        M\n" +
                     "                                .       M\n" +
                     "                               .M        M\n" +
                     "                                .         $\n" +
                     "                                 M          .\n" +
                     "                                  .         :.\n" +
                     "                                  M          M\n" +
                     "                                  ,.          M.\n" +
                     "                                  .M          ?M\n" +
                     "                                   .=      .MODM\n" +
                     "                                    M    +,=M\n" +
                     "                                     N = 8M\n");
}
```

para o arqueiro uma pena

```
public void exibir()
{
    System.out.println("      M7,\n" +
                       "      M M\n" +
                       "      MM Z+\n" +
                       "       M    M\n" +
                       "       M+ D   M\n" +
                       "        M   M   MM\n" +
                       "        MD   M    MM\n" +
                       "         M    M     M\n" +
                       "          M    MD    ZM\n" +
                       "          MM    MM     M\n" +
                       "           MM    +M   + MM\n" +
                       "            ZM     MMM   M\n" +
                       "              M     MM    +M\n" +
                       "                M   ~ DM    MM\n" +
                       "                DM     MM   MD7\n" +
                       "                  MM    MM ,M, 7\n" +
                       "                    ZMMMD~MMM   ,\n" +
                       "                          MMM\n" +
                       "                             MD\n" +
                       "                               MM\n" +
                       "                                 MD\n" +
                       "                                  MM\n" +
                       "                                    MM");
}
```

e para o mago temos um chapéu:

```
public void exibir()
{
    System.out.println("     MMMMMMM$\n" +
                       "       .MM   .MMM\n" +
                       "         MM      MMN\n" +
                       "         MM       M:\n" +
                       "         MM       MM\n" +
                       "         MM       MM\n" +
                       "         MM        MM?\n" +
                       "         MM         :MM$\n" +
                       "          MM           MMM\n" +
                       "           MM            MMM~\n" +
                       "           MM               MM:\n" +
                       "           MM               MM,     8MMMM~M\n" +
                       "           MM             MMM,MMMMMM8   .7MM\n" +
                       "          +M           MMMM   MMN         MMM\n" +
                       "          MM      MMMMN   MMM~          MMM I\n" +
                       "         MM +MMMMM   DMMMM          DMMM\n" +
                       "        7M   M   NMMMMM          7MMMM\n" +
                       "        $MMMMMMM,          MMMMM,\n" +
                       "   ZM   MD           .MMMMMM:\n" +
                       "7MM          $MMMMMMM\n" +
                       "M+OMMMMMMMMM\n");
}
```

Agora vamos entender por que as classes Personagem e Guerreiro são abstratas e seus métodos exibir() não possuem código. A definição das imagens acima está diretamente relacionada ao conceito de todas as classes que elas ilustram. No entanto, não saberíamos determinar uma imagem para as classes Personagem ou Guerreiro. Isso por que não temos clareza sobre quem são. Um personagem, por exemplo, pode ser qualquer coisa. Portanto não conhecemos completamente os seus códigos e devemos programá-las como abstratas.

Uma classe abstrata possui pelo menos um método abstrato, ou seja, sem código. O fato de uma classe ser abstrata implica também que não poderemos instanciar objetos dessa classe. Pois

não seria possível chamar o método abstrato para um objeto dessa classe. Desta forma, devemos necessariamente herdá-la e na nova classe herdeira definirmos um método para todos os métodos abstratos. Como fizemos com as classes Campones, Arqueiro e Mago.

Métodos e classes final

Em Java toda classe pode ser estendida, e todo método pode ser sobrescrito. Com a palavra chave final podemos modificar este comportamento padrão do Java. Quando não queremos que uma classe possa ser estendida devemos declará-la como final.
Por exemplo, tomemos o caso das classes Arqueiro, Mago e Campones. Caso quiséssemos que estas fossem as últimas classes no nível hierárquico deveríamos declará-las como final. Isso evitaria que outro programador criasse uma subclasse a partir delas. Abaixo um exemplo como tornar a classe Arqueiro final. Desta forma não poderemos criar subclasses de Arqueiro mais especializadas.

```
final public class Arqueiro extends Guerreiro {
```

Da mesma forma os métodos, o padrão é ao redeclará-los em uma subclasse sobrescrevê-los. Caso quiséssemos inibir a sobrescrita de um método deveríamos declará-lo como final. Abaixo um exemplo como tornar o método getSexo() de Personagem final. Desta forma não poderemos sobrescrevê-lo nas subclasses de Personagem.

```
final public char getSexo()
```

O principal argumento em favor do uso de classes ou métodos final é o desempenho. Isto por que o compilador pode determinar em tempo de compilação qual método deve chamar, isso é conhecido como vinculação estática. Diferentemente da vinculação dinâmica que somente em tempo de execução o compilador poderá determinar qual o método será executado de acordo com o tipo específico do objeto usado.

CAPÍTULO 3

TRATAMENTO DE EXCEÇÕES

Vamos agora estudar sobre tratamento de exceções. Exceções são problemas que podem ocorrer durante a execução dos nossos programas, neste capítulo vamos conhecer como o Java resolve estas situações.
Quando programamos nossos sistemas desejamos que funcionem como esperado, no entanto, problemas acontecem mas eles são as exceções e não a regra. Portanto tratar estas exceções é fundamental para garantirmos a continuidade da execução dos nossos programas. Desta forma tornamos nossos programas mais robustos e tolerantes a falhas.

Blocos try & catch

Vamos analisar um programa exemplo para entendermos como implementamos o tratamento de exceções em Java. Nosso exemplo consiste em um programa para realizarmos a divisão de dois números informados pelo usuário. Inicialmente fazemos a leitura dos valores fornecidos pelo usuário e em seguida calculamos a divisão e exibimos o valor calculado.
Apesar da simplicidade do cálculo nosso programa pode ter que enfrentar um grande problema. E se o usuário informar o valor zero como o divisor? Na nossa aritmética não temos essa resposta, portanto o nosso programa não poderá calcular a divisão e o programa será finalizado. No entanto seria conveniente informarmos ao usuário o problema ocorrido e possivelmente orientá-lo como saná-lo.
Poderíamos nos proteger desse tipo de situação, onde o nosso programa é terminado de forma inesperada, colocando algumas verificações antes de tentarmos realizar a operação, porém incorremos em outro inconveniente. Misturamos código que realiza a tarefa pretendida com código para tratar possíveis problemas. A seguir o código exemplo.

```java
// Arquivo: DivisaoZero.java
// A classe DivisaoZero

import java.util.Scanner;

public class DivisaoZero {
    public static void main(String[] args)
    {
        Scanner scanner = new Scanner(System.in);
        float result;
        int a, b;

        try {
            a = scanner.nextInt();

            b = scanner.nextInt();

            result = divisao(a, b);

            System.out.print(result);
        }
        catch(ArithmeticException exception)
        {
            System.err.printf("Argumentos inválidos. " +
                    "Divisão por zero.");
        }
    }

    public static float divisao(int a, int b)
    {
        float result;

        result = (float)a / b;

        return result;
    }
}
```

Neste exemplo utilizamos blocos try e catch. Colocamos entre as chaves do bloco try a porção de código que queremos proteger bem como a fonte de uma possível exceção. No bloco catch, incluímos o código que será executado se um problema aritmético, como a divisão por zero, ocorrer.
Dentro do bloco 'try' realizamos a leitura de dois inteiros, a e b, e passamos como parâmetros para o método divisão. Este é o ponto do código onde poderá ocorrer uma falha. Observe que a operação de divisão está dentro da função divisão. A porção de código que estamos protegendo é a exibição do valor da divisão.
No bloco catch, apenas exibimos uma mensagem para o usuário informando o motivo da falha na execução do programa.
Quando executamos este programa e informamos os valores corretos, nenhum problema ocorre e ele pode ser finalizado sem problemas.

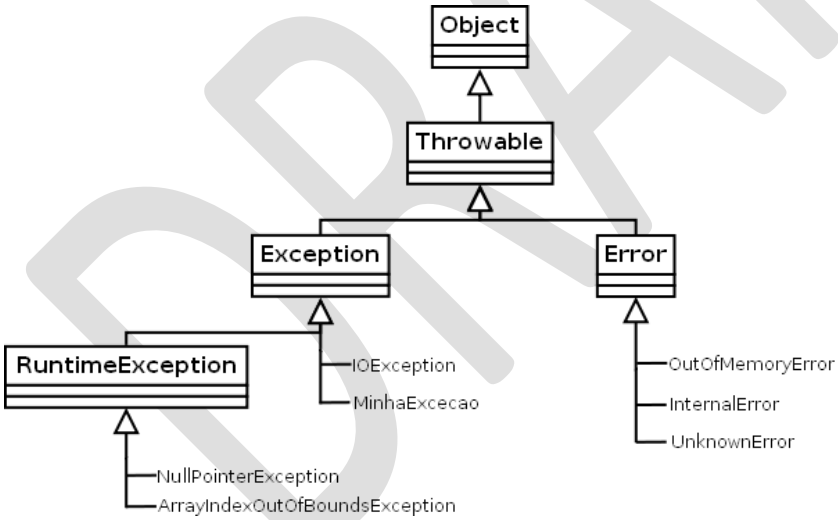
Captura de Exceções

Caso tenhamos informado à variável b o valor zero, a instrução de divisão no método divisao lança uma exceção do tipo ArithmeticException.

Quando uma exceção é lançada o Java busca por um bloco try/catch que possa tratá-la. Caso não o encontre o programa é finalizado imediatamente.
A divisão por zero considerada do nosso exemplo, lançada dentro do método divisão é tratada pelo bloco catch do método main. Ou seja, no momento da tentativa de divisão o fluxo de execução do programa é interrompido para ser tratada a exceção. Como o primeiro bloco try está no método chamador mais externo, main, a exceção é tratada neste ponto.
Após ser capturada e tratada a exceção, não há retorno ao método divisão. O fluxo do programa é alterado para continuar após o bloco catch executado.

Hierarquia de Exceções

Neste exemplo capturamos apenas um tipo de exceção, mas podemos capturar vários tipos de exceções. Para cada uma devemos criar um bloco catch. A ordem em que os diversos blocos catch são escritos é a mesma ordem que são considerados.
O Java oferece uma hierarquia de classes de exceções. A figura abaixo ilustra algumas das principais classes já disponíveis no Java.



Quando criamos um bloco catch para capturar uma exceção todas as suas subclasses desta exceção também são capturadas, de acordo com o processamento polimórfico.
Por exemplo, um bloco catch que capture exceções Throwable captura todos os tipos de exceções.
Além das classes de exceções disponíveis podemos criar as nossas próprias exceções. Estas devem estender alguma das classes da hierarquia de exceções do Java para que possam ser capturadas pelo mecanismo de tratamento de exceções.

Antes de criar uma nova classe de exceção é interessante verificar se Java já não disponibiliza essa classe.

Bloco finally

Além dos blocos try e catch também temos os blocos finally. Estes ficam sempre depois do último bloco catch.
Utilizamos os blocos finally quando precisamos garantir que uma porção de código sempre seja executada caso uma exceção ocorra ou não. Mesmo quando executamos instruções return, break ou continue dentro do bloco try ou catch ou ainda quando o próprio catch lança uma exceção. Por exemplo, algumas situações quando alocamos recursos(arquivos, conexões de rede, etc) e ocorre uma exceção, o fluxo normal do tratamento se exceções seria interromper a execução do programa e tratar a exceção, no entanto, precisamos garantir que os recursos previamente alocados sempre sejam liberados antes do fim de um método ou do programa. A seguir termos a estrutura do uso de blocos finally.

```
try
{
    // Instruções de aquisição de recursos
}
catch(TipoExcecao1 excecao1)
{
    // Tratamento de exceção
}
catch(TipoExcecao2 excecao2)
{
    // Tratamento de exceção
}
finally
{
    // Instruções de liberação de recursos
}
```

Lançando exceções

Agora que aprendemos como tratamos exceções aprenderemos como lançá-las. Para lançarmos exceções utilizamos a palavra chave throw. O exemplo a seguir ilustra o uso deste recurso.

```
try {
    System.out.println("Lançando uma exceção");

    throw new Exception();
}
catch(Exception exception)
{
    System.err.printf("Exceção capturada");
}
```

Temos dentro do bloco try o lançamento de uma exceção do tipo Exception. Observe que a exceção é primeiramente criada com a palavra chave new e depois lançada com o throw.
Logo em seguida temos um bloco catch que captura nossa exceção e exibi uma mensagem informando do tratamento da exceção.
Uma exceção ela pode também ser relançada em um bloco catch, caso não tenhamos como tratar completamente a exceção no dado nível em que foi capturada.

```
try {
    System.out.println("Lançando uma exceção");

    throw new Exception();
}
catch(Exception exception)
{
    System.err.printf("Exceção capturada");

    System.out.println("Relançando a exceção");

    throw exception;
}
```

Síntese da Unidade

Nesta unidade foi dada continuidade ao estudo dos conceitos de Orientação a Objetos aplicados a linguagem de programação Java. Aprendemos o conceito de herança para reutilizarmos códigos já implementados em outras classes para criamos novas, especializando os seus comportamentos. Estudamos a criação de código polimórfico e conhecermos os mecanismos de tratamento de exceções para proteger o código desta situações indesejadas tornando-o portanto mais robusto.

Bibliografia

ECKEL, B. **Pensando em Java,** 3ª ed, Editora MindView;

SANTOS, R. **Introdução à Programação Orientada a Objetos Usando Java**, Editora Campus;

DEITEL, H. M. e DEITEL, P.J. **Java: Como programar**, 6ª ed, Editora Pearson;

HORSTMANN, C. e CORNEL, G. **Core Java** – Volume I. Fundamentos. Alta Books. 7ª. 2005;

4

UNIDADE

INTERFACES, INTERFACE GRÁFICA E JDBC

Nesta unidade estudaremos alguns recursos mais avançados da Linguagem de programação Java. Como o uso de interfaces de classes, uma aplicação gráfica simples utilizando a biblioteca Swing e por fim consultamos a um banco de dados utilizando os mecanismos da JDBC.

CAPÍTULO 1

INTERFACES

Agora vamos conhecer o conceito de interfaces. Não confundir interfaces com interfaces gráficas, assunto do próximo capítulo. As interfaces que estudaremos neste capítulo são uma espécie de contrato que os objetos assumem.
Podemos imaginar interfaces como classes, ou melhor, tipos completamente abstratos, sem implementação alguma. Interfaces definem o que um objeto deve fazer e não como fazer. Quando definimos uma interface declaramos apenas quais métodos um objeto, que venha implementá-la, deve necessariamente possuir.
Ao pensarmos em interfaces estamos mais interessados no que um objeto faz do que como ele faz. Padronizamos como coisas, objetos ou sistemas podem interagir entre si.
Uma das principais aplicações das interfaces é fazer com que objetos de classes completamente distintas possam oferecer um conjunto de métodos em comum. Desta forma, podemos virtualmente enxergá-los como de um mesmo tipo.

Exemplo Balanço

Vamos estudar um exemplo para entendermos o uso de interfaces. Considere um sistema de controle de uma empresa, onde temos diversas classes com as suas respectivas hierarquias de classes. Por exemplo, nesse sistema empresarial teríamos uma classe para representar os clientes da empresa, outra para tratar dos colaboradores e uma terceira com as informações sobre o estoque da empresa. Naturalmente estas classes não devem pertencer a uma mesma hierarquia, ou seja, conceitualmente são desconexas.
Porém, precisamos criar um algoritmo no nosso sistema que faça uma espécie de balanço da empresa, some e subtraia tudo que é despesa e rendimentos na empresa. No caso de um cliente, este gera receitas, um funcionário custos e o estoque possuem um valor.
Se considerássemos apenas essas classes estaríamos restringindo nosso balanço somente para esses elementos, no entanto, sabemos que podemos ter outros participantes como: saldo de contas bancárias, empréstimos, títulos a receber, títulos a pagar, etc. Desta forma precisamos considerar estes diversos elementos como todos sendo de um mesmo tipo, participando do nosso balanço.
Precisamos, portanto, enxergar todos eles a partir de uma mesma ótica. Ou melhor, através de uma interface comum. As

interfaces do Java nos auxiliam exatamente neste tipo de situação. Temos diversas classes completamente distintas, mas que precisam ser consideradas, pelo menos em um aspecto específico, como todas relacionadas. Com uma interface podemos fazer com que todos os objetos exponham um conjunto de métodos comuns, assim podem ser considerados como do mesmo tipo.
Vamos criar uma interface Balanco, o código segue abaixo.

```
1 // Arquivo: Balanco.java
2 // A interface Balanco
3
4 public interface Balanco {
5     float getBalanco();
6 }
```

Enquanto uma classe pode herdar apenas de uma outra classe, As interfaces podem herdar de mais de uma interface.

Nossa interface Balanco é bem simples. Possui apenas um método getBalanco(). Este retorna o valor associado à entidade que estiver sendo considerado no cálculo do balanço da empresa. Por exemplo, para uma classe funcionário retorna o valor do salário do mesmo.
Observe que a definição da interface esta em um arquivo exclusivo para a interface. A declaração da interface inicia com a palavra-chave interface. Todos os métodos são abstratos e públicos, ou seja, não devemos fornecemos implementação para nenhum método. Além disso, caso tenhamos algum atributo associado à interface este será automaticamente considerado público, estático e final.
Vejamos como utilizamos a definição da interface Balanco no sistema de controle da empresa. Vamos considerar a classe Funcionário com a implementação da interface Balanco. Abaixo temos o código desta classe.

Apesar de em Java só podermos herdar de uma única classe, podemos implementar quantas interfaces quisermos.

```
1  // Arquivo: Funcionario.java
2  // A classe Funcionario
3
4  public class Funcionario implements Balanco {
5      String nome;
6      float salario;
7
8      Funcionario(String nome, float salario)
9      {
10         this.nome = nome;
11         this.salario = salario;
12     }
13
14     @Override
15     public float getBalanco() {
16         return -salario;
17     }
18 }
```

Na definição da classe Funcionario informamos que ela implementa a interface Balanco. Desta forma, a classe concreta Funcionario informa ao compilador que possui o compromisso

de fornecer um código para todos os métodos da interface. Pois se assim não o fizer, a classe será considerada abstrata e deve ser declarada como tal.
Como informado no código, a classe Funcionário sobrescreve o método getBalanco() da interface Balanco. O método retorna o valor do salário do funcionário negativo. A declaração do método sobrescrito deve obedecer exatamente assinatura presente na interface sendo implementada.
Vamos criar a classe Cliente também implementando a interface Balanco. O código da classe segue abaixo.

```
1// Arquivo: Cliente.java
2// A classe Cliente
3
4public class Cliente implements Balanco {
5    String nome;
6    double[] compras;
7
8    Cliente(String nome)
9    {
10       this.nome = nome;
11
12       compras = new double[5];
13
14       // Valores fixos por simplicidade
15       compras[0] = 1700.00;
16       compras[1] = 510.00;
17       compras[2] = 312.87;
18       compras[3] = 35.63;
19       compras[4] = 843.33;
20   }
21
22   @Override
23   public double getBalanco() {
24       double result = 0;
25
26       for (int i=0; i<5; i++)
27           result += compras[i];
28
29       return result;
30   }
31}
```

Para simplificar nossa implementação consideramos os valores das compras dos clientes de forma fixa. Observe que o método getBalanco() retorna o somatório das compras do cliente.
Vamos agora visualizar a construção do algoritmo do método para cálculo do balanço da empresa.

```java
// Arquivo: testeInterfaces.java
// A classe testeInterfaces

public class testeInterfaces {

    public static void main(String[] args) {
        Balanco[] elementos = new Balanco[3];
        double result = 0;

        elementos[0] = new Funcionario("Isabela", 300.00);
        elementos[1] = new Funcionario("Eduardo", 500.00);
        elementos[2] = new Cliente("Summer Tech");

        for (int i=0; i<3; i++)
        {
            result += elementos[i].getBalanco();
        }
    }
}
```

Temos um vetor elementos referenciando os diversos elementos participantes do balanço. Observe que criamos instâncias de classes distintas, Funcionario e Cliente, sendo referenciadas pelo mesmo vetor.
O cálculo do balanço consiste apenas em somar todos os valores associados às entidades contábeis da empresa. Polimorficamente, chamamos o método getBalanco() para cada um dos elementos do vetor e somamos os valores retornados.

CAPÍTULO 2

INTERFACE GRÁFICA

Neste capítulo vamos aprender como construir aplicações que possuam interfaces gráficas com os usuários. As GUIs (Graphical User Interfaces – Interfaces gráficas do usuário) são interfaces que permitem a interação do usuário com os nossos programas de forma muito mais amigável que as interfaces de texto utilizadas nos capítulos anteriores.
Podemos interagir com os componentes gráficos das GUIs tais como botões, menus, ícones através do mouse ou teclado. Temos duas opções de componentes gráficos no Java, o Swing e o AWT.

AWT x Swing

O AWT (Abstract Windows Toolkit) foi a primeira biblioteca gráfica do Java. Foi construída utilizando código nativo dos objetos gráficos das plataformas sobre as quais o Java é executado. A principal vantagem do AWT é performance, pois é basicamente um mapeamento dos componentes nativos dos sistemas operacionais, no entanto possui o grande inconveniente de não ter todos os componentes disponíveis em todas as plataformas.
O swing é uma outra API para interfaces gráficas do Java, uma evolução do AWT. Diferentemente do AWT procura gerar os componentes da interface sem a necessidade dos recursos dos sistemas operacionais da plataforma onde estiver sendo executado. Desta forma sua aparência é bem mais uniforme independente do sistema utilizado. No entanto, apresenta menor performance e maior consumo de memória.

Construção da Aplicação

Vamos conhecer alguns dos componentes do Swing com a criação de uma pequena aplicação para o controle do cadastro de clientes de uma loja fictícia. Neste exemplo poderemos conhecer os principais componentes utilizados nas aplicações com interfaces gráficas.

Diversas IDEs oferem ferramentas para o projeto de aplicações GUI, o que facilita muito a criação deste tipo de aplicação. No entanto, o código gerado é diferente de IDE para IDE. Desta forma o código criado neste capítulo será todo escrito a mão.

JFrame

A classe JFrame é utilizada para criarmos nossas janelas. É um componente Swing com os atributos e elementos básicos de uma janela. Como uma barra de títulos e botões para minimizar,

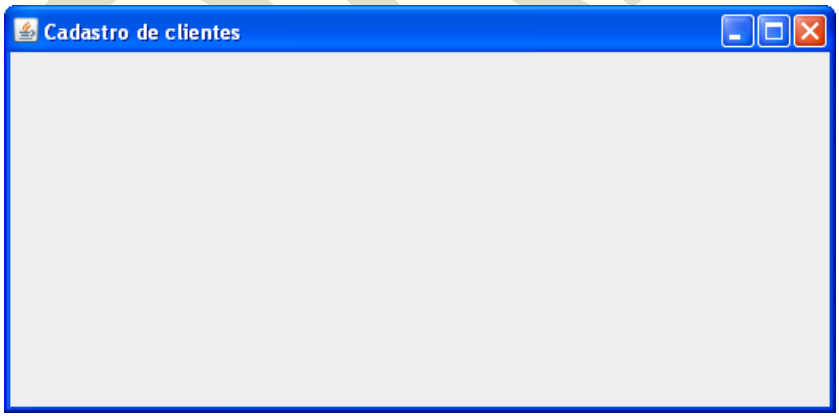
maximizar e fechar a janela. Sempre que formos criar uma janela devemos herdar de JFrame e adicionamos os demais elementos da interface nesta janela, segue o código desta janela.

```
1 import javax.swing.JFrame;
2
3 public class clientFrame extends JFrame {
4
5     public clientFrame()
6     {
7         super("Cadastro de clientes");
8     }
9 }
10
```

Para testarmos nossa janela criamos a classe testeCap1 com o método main a seguir:

```
1 import javax.swing.JFrame;
2
3 public class testeCap1 {
4
5     public static void main(String[] args) {
6         clientFrame fClient = new clientFrame();
7
8         fClient.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9         fClient.setSize(530,260);
10        fClient.setVisible(true);
11    }
12 }
```

Logo após instanciarmos um objeto de clientFrame, configuramos a ação de fechamento da janela da aplicação para ao fechá-la terminar a aplicação, o padrão é simplesmente ocultar a janela. Configuramos o seu tamanho e então tornamos a janela visível. Ao executarmos nossa aplicação obtemos a janela exibida abaixo.



Nessa nossa primeira janela ainda não temos nada útil, apenas a exibimos com o título da janela 'Cadastro de clientes' passado para o construtor de JFrame. Agora passaremos a incrementar nossa primeira aplicação gráfica.

JMenuBar, JMenu e JMenuItem

Vamos incluir primeiramente em nossa janela um menu. Nosso menu será bem simples. Teremos duas opções: Cliente e Ajuda.

O código para criação do menu foi incluído em um método privado createMenu() na classe clientFrame. Este método deve ser chamado pelo construtor da classe.
Porém antes de passarmos ao código para criação do menu devemos importar as classes necessárias para a sua construção. No início do arquivo clientFrame.java devemos incluir o seguintes imports.

```
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
```

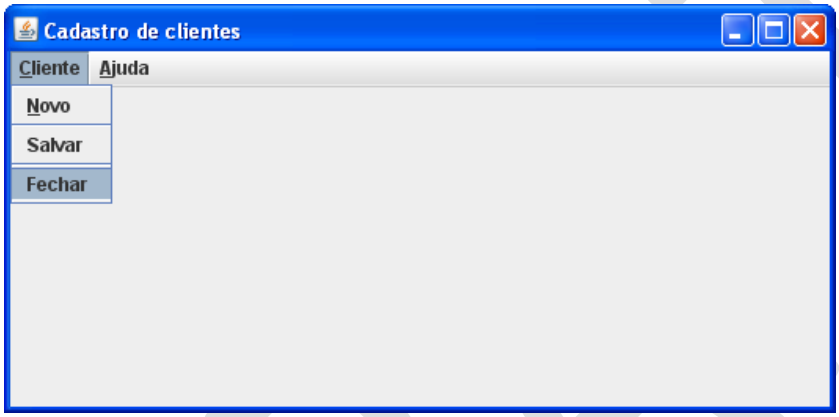
A classe JMenuBar cria a barra de menu no início da janela para acomodar os menus e seus itens. Estes são criados com as classes JMenu e JMenuItem, respectivamente. Analisaremos o código a seguir para aprendermos a criar menus.

```
46    private void createMenu()
47    {
48        // Cria menu cliente
49        JMenu menuCliente = new JMenu("Cliente");
50        menuCliente.setMnemonic('C');
51
52        menuItemCliente = new JMenuItem[3];
53
54        // Cria item novo do menu cliente
55        menuItemCliente[0] = new JMenuItem("Novo");
56        menuItemCliente[0].setMnemonic('N');
57        menuCliente.add(menuItemCliente[0]);
58
59        // Cria separador
60        menuCliente.addSeparator();
61
62        // Cria item salvar do menu cliente
63        menuItemCliente[1] = new JMenuItem("Salvar");
64        menuItemCliente[1].setMnemonic('S');
65        menuCliente.add(menuItemCliente[1]);
66
67        // Cria separador
68        menuCliente.addSeparator();
69
70        // Cria item salvar do menu cliente
71        menuItemCliente[2] = new JMenuItem("Fechar");
72        menuItemCliente[2].setMnemonic('F');
73        menuCliente.add(menuItemCliente[2]);
74
75        // Cria menu ajuda
76        JMenu menuAjuda = new JMenu("Ajuda");
77        menuAjuda.setMnemonic('A');
78
79        menuItemAjuda = new JMenuItem[1];
80
81        // Cria item novo do menu ajuda
82        menuItemAjuda[0] = new JMenuItem("Sobre...");
83        menuItemAjuda[0].setMnemonic('o');
84        menuAjuda.add(menuItemAjuda[0]);
85
86        // Cria a barra de menu
87        JMenuBar barra = new JMenuBar();
88        setJMenuBar(barra);
89        barra.add(menuCliente);
90        barra.add(menuAjuda);
91    }
```

Observe que os vetores menuItemCliente e menuItemAjuda estão declarados como atributos privados da classe clientFrame.

```
private JMenuItem[] menuItemCliente;
private JMenuItem[] menuItemAjuda;
```

Inicialmente criamos o menu cliente. Configuramos, através da chamada ao método setMnemonic(), a letra C para compor com a tecla alt a tecla de atalho deste menu. Desta forma, ao pressionarmos as teclas Alt+c são exibidos os itens do menu Cliente. Estes são criados em seguida.
Criamos os itens Novo, Salvar e Fechar. A cada item configuramos também uma letra para as teclas de atalhos e adicionamos o item criado ao menuCliente. A ordem com que adicionamos é a ordem que aparecerão no menu. Observe ainda que entre a criação dos itens acrescentamos ao menuCliente separadores. Estes separadores são barras horizontais no menu dividindo os itens em grupos. A figura abaixo mostra o menu criado.



Porém antes de testarmos nossa aplicação com seu novo menu, criamos o menu 'Ajuda' com um apenas um item: 'Sobre...'. Os três pontos deste item indicam que ao escolhermos esse item ele irá apresentar uma nova janela.
No final do método createMenu() criamos a barra de menus e incluímos nela os menus Cliente e Ajuda. A chamada ao método setJMenuMar() configura a barra criada como a barra de menu da janela sendo instanciada.

Gerenciadores de Layout

Vamos agora criar um formulário com os campos para as informações de cadastro do cliente. Mas antes de construirmos o formulário propriamente dito, vamos entender como os componentes serão dispostos na janela. Essa organização dos componentes é atribuída aos gerenciadores de layout.
Os componentes em uma janela podem ser posicionados com um valor fixo ou controlados por um gerenciador. A única vantagem em utilizar um posicionamento fixo é o maior controle com relação à posição dos elementos na janela, no entanto é bem trabalhoso configurar cada elemento, informando tamanho, localização e limites. Atribuir esta tarefa a um gerenciador de layout torna a construção de interfaces gráficas bem mais rápida e simples.

Dentre os vários gerenciadores de layout disponíveis em Java, utilizaremos os seguintes:

- FlowLayout: O gerenciador mais simples. Os componentes são dispostos da esquerda para direita não ordem que são adicionados. Quando a janela termina os próximos elementos são posicionados abaixo.
- BorderLayout: A janela é dividida em cinco regiões: NORTH, SOUTH, EAST, WEST e CENTER. Este gerenciador só permite, portanto, cinco componentes na área da janela. Veremos no nosso exemplo que podemos estender esta quantidade de elementos incluindo painéis. Estes criam uma subárea onde podemos incluir mais elementos.

JLabel, JTextField e JPanel

Vamos dividir o conteúdo da janela em três partes. Para cada uma delas iremos ter um painel para abrigar os componentes. A primeira porção da tela, logo abaixo do menu, teremos uma região com o título do formulário, logo abaixo, na porção central um painel com os campos do formulário. A parte inferior botões e uma barra de status.
O código a seguir cria os dois primeiros painéis com o título, e os campos do formulário.

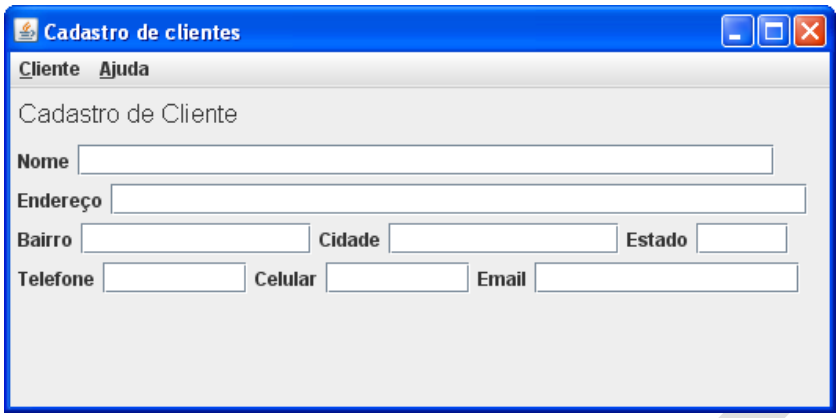
```
79      private void createForm()
80      {
81          // Cria layout
82          setLayout(new BorderLayout());
83
84          // Cria panel título
85          panelTitulo = new JPanel();
86          panelTitulo.setLayout(new FlowLayout(FlowLayout.LEFT));
87
88          titulo = new JLabel("Cadastro de Cliente");
89          titulo.setFont(new Font("Dialog", Font.PLAIN, 16));
90          panelTitulo.add(titulo);
91          add(panelTitulo, BorderLayout.NORTH);
92
93          // Cria panel cadastro
94          panelCadastro = new JPanel();
95          panelCadastro.setLayout(new FlowLayout(FlowLayout.LEFT));
96
```

```
97          // Labels
98          labels = new JLabel[8];
99          labels[0] = new JLabel("Nome");
100         labels[1] = new JLabel("Endereço");
101         labels[2] = new JLabel("Bairro");
102         labels[3] = new JLabel("Cidade");
103         labels[4] = new JLabel("Estado");
104         labels[5] = new JLabel("Telefone");
105         labels[6] = new JLabel("Celular");
106         labels[7] = new JLabel("Email");
107
108         // Fields
109         fields = new JTextField[8];
110         fields[0] = new JTextField(40);
111         fields[1] = new JTextField(40);
112         fields[2] = new JTextField(13);
113         fields[3] = new JTextField(13);
114         fields[4] = new JTextField(5);
115         fields[5] = new JTextField(8);
116         fields[6] = new JTextField(8);
117         fields[7] = new JTextField(15);
118
119         for (int i=0; i<8; i++)
120         {
121             panelCadastro.add(labels[i]);
122             panelCadastro.add(fields[i]);
123         }
124         add(panelCadastro, BorderLayout.CENTER);
125     }
126
```

Da mesma forma que construímos o menu, criamos um método separado para criarmos o formulário. O método privado createForm() também deve ser chamado pelo construtor da janela.
Neste formulário iremos utilizar novos componentes: o JLabel cria rótulos e o JTextField cria campos de texto. Rótulos são os textos fixos, normalmente utilizados para explicar o significado de um campo de texto. Já os campos de textos são utilizados para apresentar algum valor.
No início do método createForm() configuramos o gerenciador de layout da janela. Escolhemos o BorderLayout. Desta forma dividimos a janela em regiões. Utilizaremos três regiões: Norte, central e sul.
Na porção norte temos o painel panelTitulo com o título do formulário. A este é adicionado apenas um rótulo, titulo, na fonte Dialog com o tamanho 16.
Na parte central, o painel panelCadastro recebe oito rótulos e oito campos de texto. Para estes foram criados um vetor de JLabel e um outro de JTextField. Estas variáveis são privadas a classe da janela. O código destas variáveis segue abaixo.

```
private JLabel[] labels;
private JTextField[] fields;
private JLabel titulo;
private JPanel panelTitulo, panelCadastro;
```

Na figura a seguir temos a janela criada com a inclusão dos novos componentes.



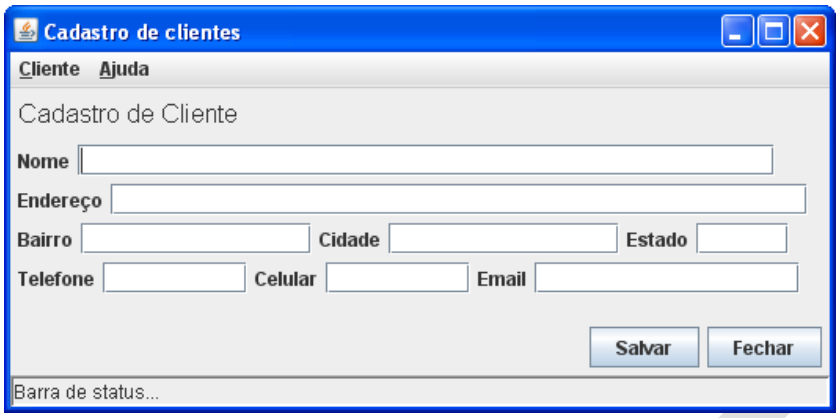
JButton

Por fim construímos a porção final da janela. Para esta ultima parte da janela também criamos um método separado que deverá ser chamado pelo construtor da janela. O código segue logo abaixo.
As variáveis panelBase, PanelBotoes, botoes e status são privadas a classe clientFrame.

```
private JButton[] botoes;
private JPanel panelBotoes, panelBase;
private JLabel status;
```

```
127     private void createBase()
128     {
129         // Cria panel base
130         panelBase = new JPanel();
131         panelBase.setLayout(new BorderLayout());
132 
133         // Cria panel botões
134         panelBotoes = new JPanel();
135         panelBotoes.setLayout(new FlowLayout(FlowLayout.RIGHT));
136 
137         botoes = new JButton[2];
138         botoes[0] = new JButton("Salvar");
139         botoes[1] = new JButton("Fechar");
140 
141         panelBotoes.add(botoes[0]);
142         panelBotoes.add(botoes[1]);
143         panelBase.add(panelBotoes, BorderLayout.EAST);
144 
145         // Criar barra de status
146         status = new JLabel("Barra de status... ");
147         status.setFont(new Font("Dialog", Font.PLAIN, 12));
148         status.setBorder(new BevelBorder(BevelBorder.LOWERED));
149         panelBase.add(status, BorderLayout.SOUTH);
150 
151         add(panelBase, BorderLayout.SOUTH);
152     }
```

No método createBase() criamos o painel panelBase que irá receber um outro painel(panelBotoes) com dois botões e um rótulo(status) que serve como barra de status.
Os botões são criados instanciando os objetos do vetor do tipo JButton. Estes são adicionados ao painel panelBotoes.
A barra de status é criada utilizando o rótulo status com a sua borda modificada para um efeito de baixo relevo.
Ao final do método o painel panelBase é adcionado a porção sul da janela. Podemos observar a janela criada na sua forma final na figura abaixo.



Tratamento de eventos

Apesar da interface esta criada ela não realiza nenhum ação. Vamos adicionar alguns tratadores de eventos nos botões e nos itens do menu. Ao selecionarmos com o mouse ou o teclado então alguma ação será executada.
Para isso vamos criar classes privadas dentro da classe clientFrame para os eventos que desejamos tratar. Ou seja, os três itens do menu cliente e o item do menu ajuda. Os botões Salvar e Fechar possuem as mesmas funções que estes itens, portanto utilizarão os mesmo tratadores de evento.

```
185     private class NovoHandler implements ActionListener {
186         public void actionPerformed(ActionEvent event)
187         {
188             for (int i=0; i<8; i++)
189             {
190                 fields[i].setText("");
191             }
192         }
193     }
194     private class SalvaHandler implements ActionListener {
195         public void actionPerformed(ActionEvent event)
196         {
197             status.setText("Cliente salvo");
198         }
199     }
200     private class FechaHandler implements ActionListener {
201         public void actionPerformed(ActionEvent event)
202         {
203             System.exit(0);
204         }
205     }
206     private class SobreHandler implements ActionListener {
207         public void actionPerformed(ActionEvent event)
208         {
209             JOptionPane.showMessageDialog(null, "Construção GUI", "Sobre",
210                     JOptionPane.PLAIN_MESSAGE);
211         }
212     }
213 }
```

Todas as classes criadas implementam a interface ActionListener e sobrescrevem o método actionPerformed(). Agora utilizaremos essas classes classe para configurarmos os devidos componentes aos seus respectivos tratadores de evento.
Utilizaremos novamente um método privado na classe clientFrame chamado pelo seu construtor.

```
168     private void configAction()
169     {
170         SalvaHandler salvaHandler = new SalvaHandler();
171         FechaHandler fechaHandler = new FechaHandler();
172         NovoHandler novoHandler = new NovoHandler();
173         SobreHandler sobreHandler = new SobreHandler();
174 
175         botoes[0].addActionListener(salvaHandler);
176         botoes[1].addActionListener(fechaHandler);
177 
178         menuItemCliente[0].addActionListener(novoHandler);
179         menuItemCliente[1].addActionListener(salvaHandler);
180         menuItemCliente[2].addActionListener(fechaHandler);
181 
182         menuItemAjuda[0].addActionListener(sobreHandler);
183     }
```

Criamos um objeto de cada tipo dos tratadores de evento e chamamos o método addActionListener() com o respectivo objeto para cada elemento da interface que deverá executar alguma ação caso seja selecionado.
Pronto agora podemos testar nossa aplicação e ela realizará alguma atividade.

CAPÍTULO 3

JDBC

Neste capítulo vamos conhecer a JDBC API (Java Database Connectivity API). A JDBC é um conjunto de classes e interfaces para permitir a comunicação dos programas Java com bancos de dados relacionais.
Com o JDBC podemos desenvolver aplicações para qualquer banco de dados, de tal forma que caso for preciso trocar de banco não será necessário modificar o código Java. Isso por que para cada tipo de banco de dados o JDBC possui um driver que compatibiliza as especificidades dos diversos bancos com a forma de trabalhar do JDBC.

Banco de Dados Relacional

Um banco de dados relacional é uma forma de armazenarmos dados em tabelas podendo-se estabelecer relações entre os dados destas tabelas.
Utilizamos a linguagem SQL para enviar comandos aos bancos de dados. Temos basicamente quatro importantes instruções para manipularmos e consultarmos os dados nos bancos de dados: SELECT, INSERT, DELETE e UPDATE
Imaginando um banco com uma tabela de clientes com os campos nome, endereço e fone. Poderíamos realizar as seguintes operações:

- Consultar dados

```
SELECT * FROM clientes;
```

- Inserir dados

```
INSERT INTO clientes
VALUES ('Pettersen', 'Storgt 20', '3456-1548');
```

- Remover dados

```
DELETE FROM clientes
WHERE nome='Pettersen';
```

- Atualizar dados

```
UPDATE clientes
SET endereco='Rua Java, 46'
WHERE nome='Pettersen';
```

Naturalmente a linguagem SQL possui outras instruções, até mesmo as apresentadas aqui permitem outras formas de uso. Porém não é o objetivo deste livro abordar em profundidade o tópico banco de dados, mas conhecermos como podemos utilizá-los em nossas aplicações Java.

Neste livro utilizaremos o MySQL para gerenciar o banco de dados que iremos criar para acessar via JDBC.

Instalação do MySQL

Além do MySQL existem diversos outros sistemas gerenciadores de banco de dados, tais como: Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL e outros.

O MySQL é um sistema gerenciador de banco de dados Open-Source. Informações completas sobre o MySQL podem ser obtidas no endereço www.mysql.com. A seguir iremos instalar e configurar um usuário para acessarmos o banco.

Instalação:

1. Visite dev.mysql.com/downloads/mysql/ e baixe o instalador adequado para a sua plataforma (Linux, Windows, Macintosh, etc). Para o propósito deste livro o pacote Windows Essential no Microsoft Windows ou o pacote Standard é suficiente.

2. Execute o instalador mysql-essential-5.1.52-win32.msi para iniciar a instalação no Microsoft Windows. Clique em Next>

3. Escolha Typical para o Setup Type e clique em Next>. Então clique em install

Quando a instalação estiver completa, clique em Next> duas vezes e, depois em Finish para iniciar a configuração do servidor.

Para configurar o servidor siga os seguintes passos:

1. Clique em Next>, então selecione Standard Configuration e clique em Next> novamente.

2. Você tem a opção de instalar MySQL como um serviço Windows, o que habilita o servidor MySQL ser executado automaticamente sempre que o Windows for inicializado. Para nossos exemplos, isso é desnecessário, portanto, desmarque Install as a Windows Service e marque Include Bin Directory in Windows PATH. Isso permitirá utilizar os comandos MySQL no prompt de comando do Windows.

3. Clique em Next> e, então, clique em Execute para realizar a configuração do servidor.

4. Clique em Finish para fechar o assistente.

Instalação do MySQL Connector/J

Para usar o MySQL com o JDBC, é necessário instalar o MySQL Connector/J. Esse software é um driver que permite que programas usem a JDBC para interagir com o MySQL

1. Visite dev.mysql.com/downloads/connector/j/ e baixe o Conector/J. O arquivo para download é mysql-connector-java-5.1.13.tar.gz.

2. Abra o arquivo mysql-connector-java-5.1.13.tar.gz com um extrator de arquivos e extraia o seu conteúdo para a unidade C:\. Isso irá criar um diretório mysql-connector-java-5.1.13.

Configuração de uma conta de usuário MySQL

Vamos agora configurar um login e senha para conectarmos com o MySQL.

1. Abra um prompt de comando e inicie o servidor do banco de dados.

mysqld.exe

Ele simplesmente inicia o servidor sem exibir nada.

2. Abra outro prompt de comando e inicie o monitor MySQL

Mysql –h localhost –u root

3. No prompt mysql>, digite

USE mysql;

4. Em seguida criamos e configuramos a conta developer. Digite os seguintes comandos

```
create user 'developer'@'localhost' identified by 'developer';
grant select, insert, update, delete, create, drop, references,
    execute on *.* to 'developer'@'localhost';
```

5. Digite o comando

exit;

para encerrar o Monitor MySQL.

Criação do Banco de Dados no MySQL

Para acessarmos o banco de dados primeiramente precisamos criá-lo. Digite o script abaixo em um arquivo com o nome bancoloja.sql e execute no MySQL.

```
create database if not exists loja_bd;

use loja_bd;

create table clientes (
    id int(10) unsigned not null auto_increment,
    nome varchar(50),
    endereco varchar(255),
    fone varchar(15),
    primary key (id));

insert into clientes(nome, endereco, fone)
    values ('Francisco', 'Rua Pedro I, 316', '4281-9738');
insert into clientes(nome, endereco, fone)
    values ('Marcius', 'Av. Pontes Vieira, 753', '3298-3282');
insert into clientes(nome, endereco, fone)
    values ('Fabiula', 'Av. Dedé Brasil, 1700', '9587-6582');
```

Antes de executarmos o script para criação do banco de dados de exemplo, devemos inicializar o serviço do banco de dados MySQL. Abra um terminal de linha de comando e digite o comando:

mysqld.exe

Para executar o script abra outro terminal de linha de comando e conecte-se ao MySQL:

mysql –h localhost –u developer –p

Em seguida execute o script com o seguinte comando

source bancoloja.sql

Neste momento o banco de dados é criado e adicionado a sua única tabela clientes alguns dados de exemplo. Agora podemos acessar e manipular os dados presentes no banco.

Consultando o Banco de Dados

O código a seguir é uma pequena aplicação para conectar-se ao banco de dados criado e realizar uma consulta.

```java
// Arquivo testeJDBC.java
// A classe testeJDBC

import java.sql.Connection;
import java.sql.ResultSetMetaData;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.ResultSetMetaData;

public class testeJDBC {
    static final String database_url = "jdbc:mysql://localhost/loja_bd";
    public static void main( String args[])
    {
        Connection connec = null;
        Statement state = null;
        ResultSet result = null;
        ResultSetMetaData meta = null;

        try {
            connec = DriverManager.getConnection(database_url,
                    "developer", "java");

            state = connec.createStatement();

            result = state.executeQuery("SELECT * FROM clientes");

            meta = result.getMetaData();


           System.out.print(meta.getColumnName(1)+ " ");
           System.out.print(meta.getColumnName(2) + " ");
           System.out.println(meta.getColumnName(3) + " ");
           System.out.println(meta.getColumnName(4) + " ");

           while (result.next())
           {
               System.out.print(result.getInt(1)+ " ");
               System.out.print(result.getString(2)+ " ");
               System.out.print(result.getString(3)+ " ");
               System.out.println(result.getString(4));
           }
       }
       catch (SQLException sqlExcept)
       {
           sqlExcept.printStackTrace();
       }
       finally
       {
           try
           {
               result.close();
               state.close();
               connec.close();
           }
           catch( Exception except)
           {
               except.printStackTrace();
           }
       }
   }
}
```

Para conectarmos ao banco utilizamos uma chamada ao método getConnection() da classe DriverManager. Este método recebe como parâmetros a url com o nome do banco de dados, o nome do usuário e a senha a ser utilizada para conectar-se ao banco.
O resultado deste método é um objeto Connection. Através deste objeto iremos enviar comandos e receber dados do banco.
Os comandos são enviados através de objetos Statement criados com a chamado ao método createStatement() da classe

Connection. Com um objeto Statement passamos os diversos comandos SQL através dos seguintes métodos:

- executeQuery: Executa um comando SQL e obtém um resultado
- excuteUpdate: Executa um comando SQL que não retorna resultado (update, delete, insert)

No caso desse exemplo realizamos uma consulta, portanto chamamos o método executeQuery(). Este método retorna um objeto do tipo ResultSet. O ResultSet possui o conjunto de informações geradas pela execução da consulta. Neste caso, o conteúdo da tabela clientes.
No momento que o ResultSet é criado este aponta para uma posição antes da primeira linha da tabela. A chamada ao método next(), o cursor do ResultSet é alterado para a primeira posição com dados. Nas chamadas sucessivas são alterados para as próximas linhas da tabela. Enquanto encontra linhas ele retorna True, após a última linha é retornado False.
Os dados são obtidos com a chamada aos métodos getInt() e getString(). Eles recebem como parâmetro o índice da coluna da qual devem retornar o valor. Observe que para coluna de uma tabela temos um tipo associado, ou seja, além da getInt() e da getString() temos:

- getLong(): inteiros longos
- getFloat(): números de ponto flutuante.
- getDouble(): números de ponto flutuante com maiores precições
- getBignum(): decimal
- getDate(): data
- getTime(): hora
- getBoolean(): bit

Ao final do método, no bloco finally, todos os objetos são fechados e a aplicação finalizada.
Também é utilizado um objeto do tipo ResultSetMetaData para obter os nomes dos campos da consulta.
Para executarmos o nosso exemplo devemos executar a partir do prompt de comando o seguinte comando:

Java –classpath .;c:\mysql-connector-java-5.1.13\mysql-connector-java-5.1.13-bin.jar testeJDBC

Síntese da Unidade

Nesta unidade conhecemos alguns recursos avançados da linguagem Java. Com interfaces aprendemos a criar objetos que podem comportar-se como mais de um tipo, criamos uma

aplicação utilizando os recursos da toolkit Swing para interargimos com usuário através do ambiente gráfico e por fim aprendemos a conectar e consultar tabelas em banco de dados utilizando a JDBC API.

Bibliografia

ECKEL, B. **Pensando em Java,** 3ª ed, Editora MindView;
SANTOS, R. **Introdução à Programação Orientada a Objetos Usando Java**, Editora Campus;
DEITEL, H. M. e DEITEL, P.J. **Java: Como programar**, 6ª ed, Editora Pearson;
HORSTMANN, C. e CORNEL, G. **Core Java** – Volume I. Fundamentos. Alta Books. 7ª. 2005;