Unidade 1: Linguagem Java Para Programadores C

Variáveis, Operadores, Estruturas

Prof. Daniel Caetano

Objetivo: Revisar os conceitos da linguagem Java.

Bibliografia: DEITEL, 2005; CAELUM, 2010; HOFF, 1996.

INTRODUÇÃO

Todo curso baseado na linguagem Java exige conhecimentos mínimos iniciais do aluno; a linguagem Java, entretanto, é suficientemente parecida com a linguagem C para que este aprendizado mínimo seja obtido rapidamente. O objetivo desta aula é proporcionar uma uniformização do conhecimento básico, para apoio de todas as aulas subsequentes. Esta aula está dividida nas seguintes seções:

1) Um programa mínimo em Java
2) Variáveis e seus tipos
3) Operadores básicos
4) Controle de Fluxo
5) Tipos Não Nativos
6) Regras de Nomenclatura
7) Classes e Objetos
8) Tratamento de Erros

1. PROGRAMANDO EM JAVA

Uma vez que um programa em Java é muito parecido com um programa em C, será apresentado inicialmente um programa mínimo em Java, de maneira a explicar algumas das principais diferenças de estruturação.

Em Java, o programa (ou **projeto**) mínimo é composto de um único pacote, com uma única classe com um único método **main**. Criando-se o programa no NetBeans, com o nome de projeto de **MeuPrimeiroPrograma**, o código será similar ao que segue:

MeuPrimeiroPrograma.java

```
// Programa mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class MeuPrimeiroPrograma {

   // Método Principal
   public static void main( String args[] ) {
      System.out.println( "Bem vindo ao Java!" );
   }

}
```

Este texto, gravado no arquivo *MeuPrimeiroPrograma.java*, pode ser compilado e executado clicando no botão que tem um triângulo verde no NetBeans. Por ser um programa mínimo, este programa mostra o que minimamente é necessário à construção de um projeto Java organizado:

1) Um pacote
2) Uma classe
3) Um método

Sempre que criamos um projeto no NetBeans, ele cria esses três elementos. Em termos de **organização**, eles funcionam da seguinte forma: Um projeto pode ter vários pacotes. Cada pacote, por sua vez, pode conter várias classes. Finalmente, cada classe pode conter varios métodos.

Uma analogia útil é comparar esta hierarquia com aquela do Microsoft Office.

Projeto:	Microsoft Office
Pacote:	Microsoft Word
Classe:	Documento de Texto
Método:	Corrigir Ortografia

Uma vez que um projeto pode ter um grande número de pacotes, classes e métodos, o NetBeans precisa saber onde é que o programa começa, isto é, **qual código** ele deve executar.

A linguagem Java estabelece que, por padrão, uma classe é sempre executada pelo seu método principal, chamado obrigatoriamente de **main**. Entretanto, isso só resolve o nosso problema se o programa tiver apenas uma classe... mas isso não é o que normalmente ocorre!

Para que resolver esta dificuldade, foi estabelecido um padrão: todo programa Java do NetBeans tem sempre um pacote com o mesmo nome do projeto e, dentro dele, uma classe **com o mesmo nome do projeto**. Quando mandarmos o NetBeans executar um programa, ele sempre começará a execução a partir do método **main** desta classe específica.

1.1. Impressão de Informações

Enquanto no C/C++ usam-se as instruções **cout** e **printf**, em Java pode-se usar:

No Lugar de:	**Use:**
cout << "Texto";	System.out.print("Texto");
cout << var;	System.out.print(var);
cout << "Texto" << endl;	System.out.println("Texto");
cout << "Texto" << var;	System.out.print("Texto" + var);
printf ("Texto %d\n",var);	System.out. printf ("Texto %d\n",var);

1.2. Comentários de Código

Os comentários são fundamentais nos códigos Java, em especial para quem ainda está aprenendo a programar. A linguagem Java permite comentários de 3 formas básicas:

// , /* */ e /** */

1) Comentários de linha (//): o compilador ignorará tudo que estetiver na mesma linha a partir da indicação.

2) Comentários multi-linhas (/* */): o compilador ignorará tudo que estiver entre o marcador /* e o marcador */.

3) Comentários tipo "JavaDoc" (/** */): o compilador ignorará tudo que estiver entre o marcador /** e o marcador */, de forma similar aos comentários multilinhas. Entretanto, é possível colocar algumas "instruções" dentro destes comentários (todas iniciadas com "@"), como @author, que são processadas por um programa especial chamado "JavaDoc", para gerar documentação do seu programa automaticamente.

2. VARIÁVEIS NATIVAS

As variáveis do Java funcionam, basicamente, da mesma maneira que as da linguagem C... ou seja, é preciso declará-las com um tipo. A forma de declarar uma variável é (dados entre colchetes [] são opcionais; dados entre menor/maior <> são obrigatórios):

<tipo da variável> <nome da variável> [= valor] ;

Exemplo:

```
int var;
int var = 5;
```

Os tipos básicos existentes são:

boolean	true/false
byte	número de 8 bits, com sinal (-128 a 127)
char	número de 16 bits, sem sinal (0 a 65535)
int	número de 32 bits, c/ sinal (-2.147484E+09 a +2.147484E+09)
long	número de 64 bits, c/ sinal (-9.223372E+18 a +9.223372E+18)
float	número de 32 bits, ponto flutuante
double	número de 64 bits, ponto flutuante

Note que números inteiros longos devem ser sufixados com a letra **L** e números de ponto flutuante float devem ser sufixados com a letra **F**. Uma construção como a feita abaixo vai gerar um erro de compilação:

long meuNumero = 10000000000;

A forma correta de especificar é:

long meuNumero = 10000000000L;

O mesmo valento para um número como:

float meuFloat = 3.4F

Neste caso, esquecer o **F** não vai causar erro na compilação, mas pode causar erros de arredondamento. Aaixo, o código adaptado para apresentar alguns dos tipos de valores:

MeuPrimeiroPrograma.java

```
// Programa não tão mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class MeuPrimeiroPrograma {

   // Método Principal
   public static void main( String args[] ) {
      byte umByte = 120;
      int umInteiro = 10000;
      long umLongo = 10000000000L;
      float umFlutuante = 10.37F;
      double umDuplo = 10.37;

      System.out.println( "Bem vindo ao Java!" );
      System.out.println( "Byte: " + umByte);
      System.out.println( "Inteiro: " + umInteiro );
      System.out.println( "Longo: " + umLongo );
      System.out.println( "Flutuante: " + umFlutuante );
      System.out.println( "Duplo: " + umDuplo );
   }
}
```

3. OPERAÇÕES EM JAVA

As operações básicas em Java são as mesmas do C:

+	soma	==	Comparação de igualdade
-	subtração	!=	Comparação de diferença
*	multiplicação	<=	Comparação de menor ou igual
/	divisão	>=	Comparação de maior ou igual
%	resto de divisão	<	Comparação menor que
		>	Comparação maior que

Exemplo de uso:

MeuPrimeiroPrograma.java

```
// Programa não tão mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class MeuPrimeiroPrograma {

// Método Principal
public static void main( String args[] ) {
   Integer idade;
   Integer idadeMaisUm;

   idade = 18;
   idadeMaisUm = idade + 1;

   System.out.println(idade);
   System.out.println(idadeMaisUm);
   }
}
```

4. CONTROLE DE FLUXO

Bem, até agora foi falado muito sobre as estruturas do Java, mas muito pouco sobre como construir a lógica dos programas, que são as partes que realmente executam trabalho. Como em qualquer outra linguagem clássica, segue-se uma lógica estruturada seqüencial, que pode ser construída basicamente por estruturas de sequência, seleção e repetição. Em especial, estas estruturas são idênticas àquelas encontradas na linguagem C/C++.

Na linguagem Java, a estrutura de sequência é automática: basta colocarmos as instruções que realizam trabalho na ordem correta que o Java automaticamente as executará em sequência. Entretanto, para as estruturas de seleção e repetição existem, de fato, instruções explícitas.

4.1. Instruções de Seleção

O Java fornece uma número suficiente de instruções de seleção, que podem ser **aninhadas**, isto é, uma colocada dentro da outra. Estas instruções são as de construção do tipo **if ~ else** e as construções do tipo **switch ~ case**. Muitas vezes é possível substituir uma delas pela outra, mas há casos em que é mais cômodo usar uma ou outra, em favor da legibilidade.

Estrutura if ~ else

A estrutura if ~ else serve quando temos uma decisão simples a fazer, ou seja: existem duas possibilidades e nunca ambas ocorrem ao mesmo tempo: ou isso, ou aquilo. Por exemplo: um aluno é aprovado se tiver média maior ou igual a 50. Então, a verificação é: "Se nota é maior ou igual a 50, aluno aprovado. Caso contrário, aluno reprovado". Em Java isso poderia ser expresso da seguinte forma:

```
if (notaDeProva >= 50) {   // se nota de prova maior ou igual a 50...
       aprovado = 1;       // aluno aprovado
```

```
        }
else    {                   // caso contrário...
        aprovado = 0;       // aluno não aprovado
        }
```

As estruturas if~else podem ser aninhadas, ou seja, podemos ter ifs dentro de ifs. Por exemplo: se o aluno passasse caso tivesse nota de prova menor que 50 mas tivesse 70 ou mais nos trabalhos, a estrutura ficaria assim:

```
if (notaDeProva >= 50) {          // se nota de prova maior ou igual a 50...
        aprovado = 1;             // aluno aprovado
        }
else    {                         // caso contrário... (nota de prova < 50)
        if (notaDeTrabalho >= 70) {// se nota trabalho maior ou igual a 70...
                aprovado = 1;     // aluno aprovado
                }
        else {                    // caso contrario (prova<50 e trabalho<70)
                aprovado = 0;     // aluno não aprovado
                }
        }
```

Também é possível executar operações lógicas mais complexas dentro da seleção, usando as lógicas E (**&&**), OU (**||**) e NÃO (**!**) dentro do if. Por exemplo, o if anterior pode ser escrito assim:

```
if (notaDeProva >= 50 || notaDeTrabalho >= 70) { // se nota de prova >= a 50
        aprovado = 1;                            // OU a de trabalho >= 70...
        }                                        // aluno aprovado
else    {                                        // caso contrário... (prova<50
        aprovado = 0;                            // E trabalho < 70)
        }                                        // aluno não aprovado
```

Embora um pouco mais complexa à primeira vista, esta notação facilita a leitura, reduz o código e pode tornar o programa mais eficiente. Lembrando a tabela verdade das operações lógicas:

A	Operação	B	Resultado
FALSO	*OU (||)*	FALSO	FALSO
FALSO	*OU (||)*	VERDADEIRO	VERDADEIRO
VERDADEIRO	*OU (||)*	FALSO	VERDADEIRO
VERDADEIRO	*OU (||)*	VERDADEIRO	VERDADEIRO
FALSO	*E (&&)*	FALSO	FALSO
FALSO	*E (&&)*	VERDADEIRO	FALSO
VERDADEIRO	*E (&&)*	FALSO	FALSO
VERDADEIRO	*E (&&)*	VERDADEIRO	VERDADEIRO
-	*NÃO (!)*	FALSO	VERDADEIRO
-	*NÃO (!)*	VERDADEIRO	FALSO

Estrutura switch ~ case

A estrutura do tipo switch~case existe para situações em que temos um número finito de tarefas a executar, dependendo de um único valor. Por exemplo, se o programa imprime o estado atual de um MP3 player e os estados possíveis são: 0- parado, 1- tocando, 2- pausa e 3- erro, isso poderia ser feito com um switch na seguinte forma:

```
switch(estadoDoMP3) {
case 0:
       System.out.println("MP3 parado");
       break;
case 1:
       System.out.println("MP3 tocando");
       break;
case 2:
       System.out.println("MP3 em pausa");
       break;
case 3:
       System.out.println("Erro na leitura do MP3");
       break;
default:
       System.out.println("Erro desconhecido");
       break;
}
```

Note que para cada valor de estado, um determinado "case" será executado. A instrução break faz com que a execução pule para a primeira linha após os delimitadores { } do switch. Caso não se use a instrução break, a execução continua com o caso seguinte, até encontrar um break.

Note também o caso especial "default". Embora nem sempre estritamente necessário, é uma boa prática de programação usá-lo, pois ajuda a diagnosticar problemas de lógica no software. No exemplo acima, qualquer estadoDoMP3 diferente de 0 a 3 causará a execução do caso "default". Como não é possível fazer a menor idéia do que isso seja (já que o valor do estadoDoMP3 só deveria ser de 0 a 3) este caso especial imprime uma mensagem dizendo que um erro desconhecido ocorreu.

4.2. Instruções de Repetição

Como o C e o C++, o Java fornece uma número mais que suficiente de instruções de repetição, que podem ser **aninhadas**, isto é, uma colocada dentro da outra. Estas instruções são as de construção do tipo **for**, **while** e **do ~ while**. Via de regra é possível substituir uma delas pela outra, mas há casos em que é mais cômodo usar uma ou outra, em favor da legibilidade.

Estrutura for

A instrução for é usada quando é necessário realizar uma tarefa por um número determinado de vezes. Ela compreende 4 partes: uma inicialização, um teste de finalização, uma descrição de incremento e, finalmente, o trecho que deve ser repetido.

Por exemplo: se for necessário imprimir um determinado texto 3 vezes, podemos usar uma estrutura for da seguinte forma:

```
for (int i = 0; i < 3; i = i + 1) {
        System.out.printf ("Texto número %d \n", i);
}
```

O Java lê este trecho de código como:

Repita o trecho de código entre {} respeitando as seguintes regras:

1) Faça i (o contador) valer 0
2) Verifique se i é menor que 3. Se sim, execute passo 3. Se não, termine o for.
3) Execute o trecho entre { }
4) Faça i = i + 1 (ou seja, some 1 ao contador)
5) Volta ao passo 2.

O resultado desta estrutura (freqüentemente chamada de *loop* ou *laço*) é:

Texto número 0
Texto número 1
Texto número 2

Estrutura while

A instrução while é usada quando queremos que um dado conjunto de instruções seja executado até que uma dada situação ocorra. A instrução while compreende 2 partes: um teste de finalização e o trecho que deve ser repetido.

Por exemplo: se for necessário imprimir um determinado texto até que o usuário digite 0 como entrada, podemos usar uma estrutura while da seguinte forma:

```
Scanner input = new Scanner(System.in); // Não se importe com a função disso!
int dado = -1;

while (dado != 0) {
System.out.println ("Digite o número zero!");
dado = input.nextInt();    // Lê um dado do teclado
}
```

O Java lê este trecho de código como "Repita o trecho de código entre {} respeitando as seguintes regras:"

1) Verifique se dado é diferente de 0. Se sim, execute passo 2. Se não, fim do while.
2) Execute o trecho entre { }
3) Volta ao passo 1.

Note que agora a inicialização da variável e a atualização da mesma, ao contrário do que normalmente acontece com o for, **não é** responsabilidade da estrutura while, e sim do código que está antes do *loop* e do código do *loop* respectivamente.

Note também que, se a variável **dado** for inicializada com o valor zero (ao invés de -1, como foi no exemplo), o trecho de código entre { } no while não será executado nenhuma vez, pois o teste é feito antes de qualquer coisa ser executado.

Estrutura do~while

A instrução do~while é usada quando é desejado que um dado conjunto de instruções seja executado até que uma dada situação ocorra, mas é necessário garantir que o conjunto de instruções seja executado **pelo menos uma vez**. A instrução do~while compreende 2 partes: um trecho que deve ser repetido e um teste de finalização. Usando o mesmo exemplo da instrução while, se for necessário imprimir um determinado texto até que o usuário digite 0 como entrada, podemos usar uma estrutura do~while da seguinte forma:

```
Scanner input = new Scanner(System.in); // Não se importe com a função disso!

do {
System.out.println ("Digite o número zero!");
dado = input.nextInt(); // Lê um dado do teclado
} while (dado != 0);
```

O Java lê este trecho de código como: "Repita o trecho de código entre {} respeitando as seguintes regras:"

1) Execute o trecho entre { }
2) Verifique se dado é diferente de 0. Se sim, volte ao passo 1. Se não, termine o do-while.

Note que neste caso não foi necessário inicializar a variável, já que ela é lida dentro do *loop* antes de ser testada. De qualquer forma, assim como no caso do while, no do~while a inicialização da variável e a atualização da mesma, **não é** responsabilidade da estrutura do~while, e sim do código que está antes do *loop* e do código do *loop* respectivamente.

Note também que, independente do valor inicial da variável **dado**, o trecho de código entre { } no do~while será executado **pelo menos uma vez** já que o teste só é feito depois que o conteúdo do *loop* tiver sido executado uma vez.

5. TIPOS NÃO NATIVOS (Classes Wrappers)

Tipos de dados não nativos são tipos de dados que são criados por meio de classes. A utilidade disso é criar tipos de dados que além de armazenar informação, também sejam capazes de realizar tarefas específicas com essa informação.

Vejamos o caso dos textos. Em C é bastante enfadonho trabalhar com textos. Devemos declarar um vetor de caracteres:

```
char texto[30] = "Meu texto";
```

E se quisermos saber quantas letras possui esse texto, precisamos usar uma função do C:

```
int tamanho = strlen(texto);
```

Em Java esse processo é mais simples. Vejamos como declarar uma variável texto:

```
String texto = "Meu texto";
```

Note que não é preciso se preocupar com o tamanho máximo do texto. Se eu quiser saber quantas letras tem esse texto, basta chamar a função **length** do próprio texto:

```
int tamanho = texto.length();
```

A classe String "encapsula" todas as complexidades envolvidas ao lidar com textos e nos fornece várias facilidades. Por não fazer parte da **linguagem Java**, mas sim da biblioteca Java, chamamos estes tipos de dados de "tipos não nativos".

Alguns outros tipos de dados não nativos foram criados, para representar os tipos nativos:

Boolean	true/false
Byte	número de 8 bits, com sinal (-128 a 127)
Char	número de 16 bits, sem sinal (0 a 65535)
Integer	número de 32 bits, com sinal (-2.147484E+09 a +2.147484E+09)
Long	número de 64 bits, com sinal (-9.223372E+18 a +9.223372E+18)
Float	número de 32 bits, ponto flutuante
Double	número de 64 bits, ponto flutuante

Observe que eles se inicial **todos** com letra maiúscula: todos os nomes de classe em Java devem ser grafados com a primeira letra maiúscula.

O uso é idêntico ao visto anteriormente:

```
Byte umByte = 100;
```

ou

```
Integer i = 5;
```

A vantagem com relação aos tipos nativos, é que estes tipos não nativos são capazes de, por exemplo, converter uma String para um número.

6. REGRAS DE NOMENCLATURA

Você deve ter reparado que ao longo das explicações são usadas diferentes composições de letras para nomes das coisas em Java. Por exemplo: os nomes das classes começam sempre com letras maiúsculas e os nomes dos métodos sempre com letras minúsculas. Essa é uma convenção adotada por todos os profissionais Java por facilitar muito a leitura e a compreensão do código; adicionalmente, esta convenção é obrigatória nos exames de certificação da Sun/Oracle.

Assim, segue abaixo um breve resumo da convenção:

1) Cada arquivo *.java* pode conter **apenas** uma classe pública, ou seja, uma única classe que pode ser usada por um programa maior. Se, por algum motivo, um arquivo *.java* contiver mais que uma classe, todas as classes excedentes devem ser declaradas como *private*, ficando indisponíveis para classes em outros arquivos.

2) Um arquivo .java deve ter **um nome exatamente igual** ao de sua classe pública. Assim, se a classe pública chama-se **BemVindo**, o nome do arquivo será **BemVindo.java**.

3) Convenção: nomes de pacotes/pastas, arquivos, classes, métodos e campos só podem conter caracteres alfanuméricos (A-Z, 0-9) e underline/underscore (_) e **nunca** devem começar com números. **Nunca** use espaços nem caracteres acentuados e/ou especiais.

4) Convenção: nomes de classes devem sempre começar com letra maiúscula. Nomes de métodos e atributos/variáveis devem sempre começar com letras minúsculas. Se o nome da classe, método ou atributo/variável tiver mais de uma palavra, como "meu campo" ou "minha classe", os espaços devem ser eliminados e a primeira letra de cada palavra deve ser feita maiúscula, como em **meuCampo** e **MinhaClasse**.

5) Convenção: use sempre nomes de pastas, arquivos, classes, métodos e campos que descrevam bem seu significado, mas sempre tão curtos quanto possível.

7. ORIENTAÇÃO A OBJETOS

Orientação a objetos é um conceito de representação da realidade em que os componentes são objetos e não funções ou estruturas de dados. Em outras palavras, se nos modelos estruturados os componentes eram definidos de acordo com características intrínsecas à implementação, nos modelos orientados a objetos estes componentes se baseiam objetos (entidades) do mundo real.

- O mundo real é composto de objetos que interagem entre si.
- Um modelo orientado a objetos é composto de objetos que interagem entre si.

Da teoria de sistemas, temos que um sistema é um conjunto de entidades que interagem entre si a fim de produzir um resultado comum. Assim, é natural o uso de "objetos programa" a fim de compor um sistema computacional.

7.1. Como São os Objetos?

No mundo real, objetos podem ser animados ou inanimados, mas qualquer um deles possui carcterísticas que podem ser classificadas como atributos ou comportamentos.

Exemplos de objetos: átomos, veículos, vias, pessoas...

Isso faz com que exista uma diferença semântica muito pequena entre modelo e a realidade que ele representa, proporcionando maior clareza.

Vantagens principais:

- **Concepção do sistema mais simples**: a transição da *realidade* para o *modelo* é facilitada.

- **Compreensão do modelo é simples**: como o *modelo* é mais próximo da *realidade*, a compreensão do modelo por quem compreende o problema real é quase automática.
- **Gerenciamento do sistema mais simples**: assim como na realidade, os objetos são estáveis na solução de um problema, ou seja, os objetos mudam muito pouco; quando é necessário resolver problemas ligeiramente diferentes, modificamos a forma com que os objetos interagem e não os objetos em si.

Mas afinal, o que são objetos em programação?

Em programação (e, de certa forma também na vida real), um objeto é um ente caracterizado por um conjunto de operações e um estado, caracterizados por *métodos* e *campos*, podendo ainda ser compostos por outros objetos.

Note que um objeto é uma estrutura similar à uma "estrutura de dados"; porém, além de "dados", um objeto pode armazenar também "funções". Em um objeto os dados são chamados de *atributos* e as funções são chamadas de *métodos*.

Exemplos de objetos do mundo real:

TV	- Liga	- Canal
	- Desliga	- Volume
	- Muda canal	- Estado(ligada/desligada)
Carro	- Liga	- Cor
	- Desliga	- Velocidade
	- Acelera	- Quilometragem (odômetro)
	- Breca	- Portas

7.2. Classes x Objetos

É importante, neste momento, diferenciar uma classe de um objeto. Uma classe é um conceito genérico: pessoa, carro, cliente. Um objeto, por outro lado, é um conceito específico: Alberto, Mustang Azul, O Comprador do Mustang Azul do Alberto. Uma classe pode ser considerada como um "molde" de um objeto.

Exemplo:
Classe: Carro
Objetos: Carro vermelho, Carro azul, Ferrari etc.

Quando programamos, nós programamos **classes**. Para realizar uma tarefa, podemos precisar de objetos específicos e, então, solicitamos ao Java que construa um objeto com base em uma classe específica. Um objeto é criado da seguinte forma:

```
ClasseDoObjeto nomeDoObjeto = new ClasseDoObjeto(parâmetrosDeCriação);
```

Por exemplo, para criar a Pessoa chamada Alberto, uma possibilidade seria essa:

```
Pessoa alberto = new Pessoa("Alberto");
```

Observe que o nome do objeto não precisa ter relação nenhuma com os atributos do objeto. O código a seguir, por exemplo, realiza a mesma tarefa que o código anterior:

```
Pessoa umaPessoa = new Pessoa("Alberto");
```

7.3. Chamando Métodos

Uma vez que um objeto tem atributos e métodos, pode ser que desejemos solicitar a algum objeto que ele nos realize uma tarefa específica. Por exemplo, podemos querer solicitar que um personagem de um jogo se desenhe na tela. Isso poderia ser feito conforme indicado abaixo:

```
personagem.desenhar(tela);
```

Observe que primeiramente foi indicado o nome do objeto (personagem), seguido do método (desenhar) e, como parâmetro, foi passado o nome de outro objeto (tela).

De maneira geral, uma chamada a um método segue o seguinte formato:

```
[dono_do_metodo.]<nome_do_metodo>([parametros_do_metodo]);
```

7.4. Principais Propriedades da Orientação a Objetos

As principais caracterísitas das classes de objetos constituem também as fundações do modelo orientado a objetos. Estas características são: encapsulamento, polimorfismo e a herança.

ENCAPSULAMENTO

É a propriedade que permite que um objeto seja tratado como uma "caixa preta". O interior do objeto, ou seja, "como" ele realiza as tarefas é invisível para os clientes daquele objeto. Os clientes só podem se comunicar com um objeto através da *interface* deste objeto, sendo que a interface de um objeto nada mais é do que a definição de quais mensagens ele "sabe" responder.

Exemplo: o carro é um objeto que pode ser tratado como uma caixa preta; uma pessoa pode dirigir sem saber como funciona o motor do carro.

Note que essa propriedade permite que pensemos em termos de "classe de análise" antes de pensarmos em "classe de projeto". Nas "classes de análise" são definidas, basicamente, as interfaces dos objetos. Posteriormente, nas "classes de projeto", é que existirá a preocupação em como fazer tais objetos funcionarem a partir da interface estabelecida.

POLIMORFISMO

Polimorfismo é uma propriedade que permite que um objeto que conheça uma determinada *interface*, pode se comunicar, isto é, trocar mensagens com qualquer outro objeto que respeite aquela *interface,* independentemente de qual seja o tipo do objeto com quem está se comunicado. Em outras palavras, dois objetos que conheçam uma mesma *interface* podem se comunicar, independentemente de quais sejam suas classes.

Exemplo: Se Carro Azul e Caminhonete Vermelha possuem a mesma interface, que é conhecida por João, então:

Objeto	Ação	Objeto		Objeto	Ação	Objeto
João	Dirige	Carro azul	=>	João	Dirige	Caminhonete Vermelha

Trocando em miúdos, se carro e caminhonete possuem a mesma interface de operação que é conhecida por João, então João saberá operar tanto o carro quanto a caminhonete, mesmo que o objeto caminhonete tenha sido inventado muito tempo depois da criação do objeto João.

HERANÇA

Herança é a propriedade que nos permite criar uma nova classe especificando que ela "é uma" outra classe também. Por exemplo, se temos a classe "Pessoa", podemos criar a classe "Trabalhador" dizendo que *Trabalhador é uma Pessoa.* Assim, um objeto da classe Trabalhador vai também possuir todos os atributos e métodos de um objeto da classe Pessoa (como *nome*, por exemplo).

De forma mais rigorosa, podemos dizer que herança é a propriedade que permite que, ao especializar uma classe, os objetos da nova classe preservem todos os comportamentos e atributos dos objetos da classe original, ou seja, os comportamentos e atributos são *herdados*. Em outras palavras, a nova classe (mais especializada) continua a respeitar a *interface* estabelecida pela classe original.

Exemplo:

classe: Pessoa	ação: dirige	**classe: Veículo**
objeto: joao		***subclasse: Carro*** (é um Veículo)
objeto: carla		*objeto: carroAzul*
		subclasse: Caminhonete (é um Veículo)
		objeto: caminhoneteVermelha

Se objetos da classe Pessoa conhecem a interface da classe Veículo, conhecem também a interface das classes Carro e Caminhonete, que são classes **especializadas** da classe Carro original. Assim, se *joao* e *carla* são objetos da classe Pessoa e *carroAzul* é um objeto da classe Carro e *caminhoneteVermelha* é um objeto da classe Caminhonete, então tanto objeto *joao* quanto o objeto *carla* podem interagir com carroAzul e caminhoneteVermelha.

Muitas linguagens, incluindo C++ e Java, utilizam a propriedade da Herança para implementar o Polimorfismo. O Java inclui também o tipo "interface" para esta finalidade.

8. TRATAMENTO DE ERROS

Em muitas situações da programação, uma sequência de tarefas pode ser interrompida pela ocorrência de algum tipo de erro. Por exemplo: se solicitarmos ao usuário que digite um número, para que possamos fazer uma conta, e o usuário digitar um texto, haverá um problema para realizar a conta. Para este tipo de situação, existe uma estrutura de tratamento de erros denominada *try ~ catch ~ finally*.

A idéia é colocar dentro de um bloco "try" toda a sequência de instruções propensas a erro e, para cada tipo de erro, acrescentar um bloco "catch". O bloco finally existe caso desejemos que algumas operações sejam executadas sempre, ocorra um erro ou não; caso típico é desfazer a conexão com o banco de dados.

O código abaixo mostra um exemplo de uso de try-catch-finally:

Main.java

```
// Programa não tão mínimo em Java
package meuprimeiroprograma;

// Classe Principal
public class Main {

// Método Principal
public static void main( String args[] ) {

   try {
      String numeroDigitado = "5";
      double numero = Double.parseDouble(numeroDigitado);
      double resultado = numero / 2.0;
      System.out.println(resultado);
      }
   catch (NumberFormatException ex) {
      System.out.println("Número inválido!");
      }
   finally {
      System.out.println("Programa encerrado!");
      }
   }

}
```

Execute o programa e veja o resultado. Depois disso, modifique o valor inicial de "numeroDigitado" para "5A", ao invés de "5". Execute novamente o programa e veja o que ocorre.

9. BIBLIOGRAFIA

CAELUM, FJ-11: Java e Orientação a Objetos. Acessado em: 10/01/2010. Disponível em: < http://www.caelum.com.br/curso/fj-11-java-orientacao-objetos/ >

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - 6a ed. São Paulo: Pearson, 2005.

HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.