O estado das variáveis: tipos, acessibilidade e passagens de parâmetro.

Por D´Artagnan Ramos Dias Neto(A.K.A. **Ironlynx**)

1-Introdução:

Nesse artigo, o leitor terá uma breve noção de como se comportam as variáveis no Java, quais são os seus tipos associados, como são acessadas, armazenas em memória e como são passadas para métodos.Esse tutorial está longe de esgotar esses assuntos, mas será uma boa visão aos iniciantes na linguagem Java.

A compreensão de como uma passagem de parâmetro é efetuada, ou como uma variável é acessada e modificada, é fundamental ao aprendizado de qualquer linguagem de programação. Como nunca custa nada lembrar, aí vão alguns conceitos básicos envolvidos que o programador tem que saber_:

-**Classe**: É o modelo ou a forma a partir da qual um objeto é criado.Nela existe a definição das variáveis e seus atributos e como as coisas irão funcionar no seu escopo (dentro dessa classe).Toma-se como padrão a primeira letra do nome da classe em maiúscula (como String, Math, System, entre outras).
-**Objeto**: É uma instância da classe, componente na memória com seus atributos e dados próprios, que utilizam seus métodos para manipular os seus dados internos e que se comunicam entre si através de mensagens (chamadas) a esses objetos.
-**Instância**: é a ocorrência em memória (**heap**) de uma determinada classe, cujos dados são definidos pela classe a qual essa instância faz parte. Só existe após o objeto ser criado.
Percebeu a diferença entre objeto e instância?Exemplo prático (repare os comentários):

```
public class Classe{
        private static String variavel_de_classe ="Sou local a Classe";
        private String i = "Sou uma variável de instância";
        public static void main(String []guj){
                Classe c; /*declara um objeto(referência) c do tipo Classe, na sua
stack(pilha).*/

                c=new Classe(); /*instância,é a ocorrência de c na heap, criada a partir do
new(). Agora , a variável c é visível em memória----representa a classe Classe */

                System.out.println(variavel_de_classe);
                                } //fim do main
 } //fim da classe Classe
```

Por hora, tudo que precisamos saber é que a **stack** armazena variáveis com escopo de método (que serão executadas dentro de métodos) e que a **heap** armazena nossos objetos. Na verdade, fazem muito mais do que isso. Não se preocupe, pois exploraremos bem o assunto mais adiante.

:::**Variáveis e acessibilidade**:::

1.1-As variáveis no Java.

Antes de seguir em frente, devemos levar em conta que a linguagem Java é fortemente tipada, isto é, toda e qualquer variável ou expressão tem um tipo associado conhecido em tempo de compilação. O nome de uma variável deve começar com uma letra e pode possuir os seguintes caracteres:'A'-'Z','a'-'z'.'0'-'9','_' ou qualquer caracter **unicode** que represente uma letra, lembrando que todo nome de variável é sensível a alteração de caixa(case sensitive).
Os tipos em Java são divididos em 2 categorias :

1ª) Em tipos primitivos (**short**, **int**, **boolean**, **char**, **byte**, **long**, **float** e **double**).
Esses tipos primitivos podem ser booleanos (**boolean**) ou numéricos (os demais).
Os numéricos por sua vez podem ser *Integrais*(**byte**, **short**, **int**, **long** e **char**) ou *Pontos Flutuantes*(**float** e **double**).
-limites e valores que cada tipo armazena:
para **byte**, de -128 até 127, (inclusive)-armazena 8bits.
para **short**, de -32768 até 32767, (inclusive)-armazena 16bits.
para **int**, de -2147483648 até 2147483647, (inclusive)-armazena 32bits.
para **long**, de -9223372036854775808 até 9223372036854775807,(inclusive)-armazena 64bits.
para **char**, de '\u0000' até '\uffff'(inclusive), que é, de 0 até 65535 do **Unicode**.
-tipos **booleanos** podem assumir apenas um dos 2 valores: true ou false. Armazenam 8 bits.
para **float**, de -3.4E38 a 3.4E38-32 bits.
para **double**, de -1.7E308 a 1.7E308-64 bits.

2ª)Em tipos referência (que são Classes, Objetos e Interfaces). Veremos mais detalhes sobre esse tipo no próximo tópico.

Obs: Há um terceiro tipo(**Null Type**) que não iremos nos ocupar pois ele não pode ser declarado, por não possuir um nome; nem permitir coerção(conversão de tipo-casting) para esse tipo. Uma referência nula é apenas um possível valor para uma expressão de tipo null.Essa referência pode sempre ser convertida para qualquer tipo referência.Na prática, a única coisa que o programador precisa saber é que null é um literal que pode ser de qualquer tipo referência.

->**Unicode** é um padrão (regulamentado pela norma ISO-10646) que permite definir caracteres cuja representação interna no computador utiliza mais de um byte (ou octeto na nomenclatura ISO) e permite representá-los de forma única não importando o idioma, programa ou plataforma.

1.1.1-Quanto á atribuição de valores:

Devemos levar em conta que uma variável é um nome associado a uma posição (endereço hexadecimal) de memória.Na sua declaração, o compilador "arranja" (aloca) um espaço para ela. A variável tem um tipo associado, por vezes conhecido como "tipo em tempo de compilação", que possui um dos tipos anteriormente citados.Uma variável de um tipo primitivo sempre armazena um valor de seu tipo exato.Em uma variável de tipo referência, deve-se ter uma atenção especial:

-Se ela é uma variável de referência do tipo Classe:
Considerando uma classe Guj, ela pode armazenar uma referência a null ou uma referência a uma instância de Guj, ou qualquer subclasse de Guj.

-Se ela é uma variável de referência do tipo Interface:
Ela pode armazenar uma referência a null ou uma referência a qualquer instância de qualquer classe que implementa essa interface.

-Se ela é referência do tipo Objeto:

a) Se ela é pertencente a um Array:
Se **x** é de um tipo primitivo então uma variável do tipo Array de **x**, pode armazenar uma referência a null ou uma referência a qualquer Array do tipo "Array de **x**"; se **x** é do tipo referência, então a variável do tipo "Array de **x**" pode armazenar uma referência a null, ou uma referência a qualquer Array de um tipo **y**, considerando que **y** tem tipo atribuível à **x**.

b) Se ela é um Objeto:
Uma variável do tipo Object pode armazenar uma referência a null ou uma referência a qualquer Objeto; seja instância de uma Classe, Interface ou Array. Nunca é demais lembrar que Object é a superclasse de todas as outras classes.Portanto, qualquer classe que o leitor crie, já é herdeira (é uma subclasse) de Object e seus respectivos 11 métodos.

1.1.2-tipos identificáveis em uma classe:

Existem 7 tipos de variáveis identificáveis em uma classe Java.São eles:

1-Variável de Classe:
É um campo declarado com a palavra-chave static dentro de uma declaração de classe, ou com (ou sem) static dentro de uma declaração de interface.

2-Variável de Instância:
É um campo declarado dentro de uma declaração de classe sem a palavra-chave static.

3-componentes de um Vetor(Array):
São variáveis sem nome que são criadas e inicializadas com valores padrão sempre que um novo objeto que é um array é criado.

4-parâmetros para método:
São nomes de argumentos cujos valores são passados para um método.Para cada parâmetro declarado em uma declaração de método, uma nova variável de parâmetro é criada cada vez que um método é invocado.
5-parâmetros para Construtor:
São nomes de argumentos em que seus valores são passados para um construtor. Para cada parâmetro declarado em uma declaração de um construtor, uma nova variável de parâmetro é criada cada vez que uma instância é criada ou uma chamada explícita ao construtor é executada.
6-parâmetro para tratar uma exceção:
É criado cada vez que uma exceção é capturada por uma cláusula catch em um bloco **try**-**catch**. A nova variável é inicializada com o Objeto associado com a exceção.
7-variáveis locais:
São variáveis locais aquelas limitadas ao escopo (alcance) de uma instrução ou método. Quando um fluxo de controle entra em um bloco ou uma instrução for, uma nova variável é criada para cada variável local declarada nessas instruções ou campos. Uma instrução de declaração de uma variável local pode conter uma expressão que inicializa a variável. Uma variável local com uma expressão de inicialização não é inicializada, entretanto, até a sua instrução de declaração que a declara ser executada. Se não fosse uma excepcional situação, poderíamos dizer que uma variável local é sempre criada quando uma instrução ou bloco a qual foi declarada é executada. Uma excepcional condição ocorre quando em uma instrução switch, aonde é possível o controle de entrada no bloco pular a execução de uma instrução que declara uma variável local.Seu ciclo de vida é no máximo o método a qual está inserida. Vejamos um simples código para reforçar o que foi escrito:

```
import java.io.*;

 public class Variaveis{
        private float sal; //1
        private static String nome; //2
        private final static  int ID=10; //3

     Variaveis(){ //4
                }
     Variaveis(float x,String y){ //5
      this.sal=x;
      this.nome=y;              }

   public static void main(String []args){
        Variaveis v = new Variaveis(); //6 v é local a main
         v.sal=5000;
         nome="Paulo";

        Object o[] = new Object[]{"J","U","G"}; //7

      exibir(o);//8
```

```
        System.out.println(+v.sal); //9
        System.out.println(nome);  //10
        System.out.println(id);  //11


     try{ //12
          PrintWriter grava = new PrintWriter(new
FileWriter("c:\\entrada.txt"));
          grava.println("Registro :"+id);
          grava.println("Nome :"+nome);
          grava.println("Salario :"+v.sal);
          grava.close();
        }
     catch(Exception erro){ //13
        erro.printStackTrace();
        System.err.println("Erro de Arquivo!");
        }
                                          }

   protected static void exibir(Object []object){ //14
         for(int i=2;i>=0;i--){
           System.out.print(object[i]);        }
           System.out.println();
                                                 }

              } //fim da classe Variáveis
```

Nessa classe temos a presença de vários tipos de variáveis.Vamos analisar cada trecho (siga os comentários numerados):
Em **(1),** temos uma variável de instância sal do tipo float;
Em **(2),** temos uma variável de classe (repare o static) chamada nome do tipo String;
Temos em **(3)** uma outra variável de classe (ID) com a palavra-chave **final**.Isso indica que a ela só pode ser atribuído um valor uma única vez.E uma vez atribuído, ela terá sempre esse mesmo valor (é uma *constante*).
Temos em **(4)** e **(5)** um construtor vazio (sem parâmetros) e um recebendo os parâmetros de construtores **x** e **y** respectivamente.Esse trecho de código referentes a construtores é totalmente desnecessário (é apenas para ilustrar o tipo de variável), pois nenhuma variável é passada na criação da instância de classe em **(6).** Poderia deixar sem construtor nenhum, pois java proveria um vazio para classe. Mas jamais poderia deixar somente o construtor com parâmetros -em **(5)-,** pois daria um erro de compilação, uma vez que em **(6):**

```
Variaveis v=new Variaveis();
```

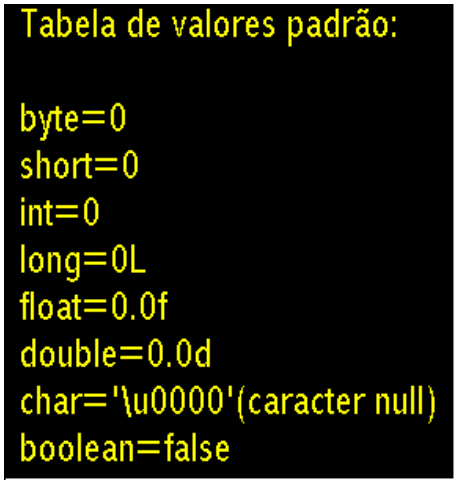
Nenhum valor é passado a instância da classe criada no método main!
*Nota:v é uma instância de classe, mas é local a **main**, ela pode acessar variáveis de classe e de instância, mas não pode ser acessada de lugar algum senão o próprio método em que é criada.*
Continuando:
Em **v.sal=5000;** atribuímos a variável de instância sal o valor 5000
repare que é necessário utilizar uma instância da classe (v) para acessar sal.Isso ocorre porque ela não é estática (static).Em **nome= "Paulo";**

podemos atribuir diretamente porque ela é de classe.Esse modificador(static) garante á variável que só haverá uma instância dela em memória.
Em **(7),** criamos um array do tipo Object cujos índices receberam J, U e G como parâmetros.
Em **(8),** fazemos uma chamada ao método **exibir**() passando a variável de referência o como parâmetro.
Em **(9), (10)** e **(11)** passamos parâmetros ao método **println**() para mostrá-los na tela.Repare que as duas variáveis de classe (com o static) são passadas diretamente (sem a variável de referência v), sal, que é de instância, precisa ser acessada da forma ***objeto.variável;***
Em **(12),** temos o início de um bloco **try**-**catch**, **try** tenta executar o código para gravar um arquivo de texto em disco. Para isso usamos um reescritor de caracteres da classe java.io.FileWriter (por isso é necessário dar um import java.io.*; e uma instancia chamada **grava** de java.io.PrintWriter que através das chamadas ao método println(parametro) nos permite gravar em disco os dados.Logo depois fechamos o recurso utilizado através de **grava**.**close**();
Colocando o programa para executar(***javac Variaveis.java***) Será visto na tela:
GUJ
5000.0
Paulo
10

E, no seu drive **C**: será criado o arquivo **entrada**.**txt** com os dados:
Registro :10
Nome: Paulo
Salario :5000.0

Nota-se que na tela e no arquivo o numero inteiro 5000 passado a **float** sal foi convertido para 5000.0. Isso ocorre porque a conversão do tipo **float** para **int** é permitida.Veremos uma tabela de conversão de tipos primitivos no final desse tópico.
Em **(13),** temos uma cláusula **catch** com o parâmetro **erro** (do tipo Exception).Se um erro ao tentar gravar esse arquivo **entrada.txt** ocorrer, a execução do programa será desviada para catch,e a chamada a **erro**.**printStackTrace**(); exibirá na tela a pilha de erros a partir do ponto em que ocorreram.A chamada a System.**err**.**println**() é somente para tornar mais explícita (e personalizado)ao programador que um erro ocorreu.
E temos, finalmente em **(14),** o corpo do método **exibir()** que recebe um array de Object como parâmetro, conta seus índices do final para o início, para mostrá-los na tela(transformará J,U,G em **GUJ**).
Nesse exemplo(**Variaveis**.java) vimos vários tipos de variáveis, mas devemos observar que isso é apenas um humilde pedaço do poderoso mundo que é Java.

No que tange a variáveis vale a pena salientar que:
-Quando instanciamos uma variável com um Objeto associado, essa variável não guarda o próprio objeto, e sim um endereço (uma "referência") para esse objeto ser acessado na memória.
-Uma dada variável não pode ser declarada mais de uma única vez, ou ocorreria um erro de compilação. Ela pode ser inicializada uma única vez, e escrita/lida quantas vezes for necessário.
-O custo de declarações locais é **NULO** em termos de instruções utilizadas pela CPU para alocação de espaço. Já o custo de inicializar essas variaveis deve ser considerado.
-variáveis locais também são chamadas de variáveis de pilha, temporárias e ou de método.
-Variáveis de instância também são chamadas de atributos, propriedades ou campos.
-Uma variável de instância e uma variável de método podem possuir o mesmo nome; nesse caso uma referência this deve ser usada dentro do método a onde está localizada a variável local para chamar a variável de instância.(this faz referencia ao objeto "classe" a qual está inserido).
-**Toda** variável local tem que ser inicializada antes de ser usada.
-Variáveis de Instância são inicializadas com valores padrão caso não seja feito explicitamente pelo Programador assim que um objeto da classe a qual está inserida é instanciado.
-Variáveis de um tipo mais restritivo (por exemplo **int**) não podem receber um conteúdo de um tipo mais abrangente(por exemplo double), pois dará erro de compilação!Será necessário uma **coerção** (ou casting) para permitir isso. Ex:
double **d**=(double)**i**; //onde i é um inteiro
Vejamos uma tabela de valores **default** de variáveis não explicitamente inicializadas:



O número de conversões e tipos são tão grandes em Java, que esse assunto merece um tutorial á parte só para isso.Por hora, mostraremos uma tabela com as conversões permitidas entre os tipos primitivos (as que ocorrem com mais frequência) em seus programas Java, só a título de curiosidade:

Conversão de Ampliação do Tipo primitivo(são 19 possíveis):

é permitido de:

byte para short, int, long, float, ou double
short para int, long, float, ou double
char para int, long, float, ou double
int para long, float, ou double
long para float ou double
float para double

---

*Nota: Conversões de Ampliação de um tipo numérico primitivo não causam perda no valor da ordem de grandeza da variável, desde que ela(a classe ou método) esteja com o modificador **strictfp**, que força todos os métodos a aderirem as regras **IEE754**, tornando possível saber o comportamento de números **double** ou **float**, independente da plataforma.Caso não esteja com esse modificador, poderão ocorrer perdas em números **float** e **double**.Em números **int** e **long**, o uso de **strictfp** evita uma perda de valores na sua conversão para **float** ou **double** arredondando para um valor válido mais próximo possível.*

Conversão de Redução do Tipo primitivo(são 23 possíveis):

é permitido de:

byte para char
short para byte ou char
char para byte ou short
int para byte, short, ou char
long para byte, short, char, ou int
float para byte, short, char, int, ou long
double para byte, short, char, int, long, ou float

Nota: Conversões de Redução de um tipo numérico primitivo podem causar perdas na ordem de grandeza da variável e perder a precisão quanto ao seu valor real.

Obs: Variáveis de instância e métodos, também são chamados de membros de classe e podem fazer uso de modificadores de acesso, ao contrário das variáveis locais.Veremos mais sobre o assunto no próximo tópico.

2- Acessibilidade á Membros

Antes de falar de modificadores, devemos lembrar que se uma classe não está acessível para outra, seus membros também não estarão, não importando os modificadores usados para eles.
Vamos aos principais modificadores (não só de variáveis, como de classes e métodos também):

-Modificador de classe, método e atributo **public**:
Classes **public** podem ser instanciadas por qualquer objeto livremente.
Métodos e atributos são acessados sem restrições.

-Modificador de classe e método **abstract**:
Só se aplica a classes, interfaces e métodos.
Classes com o modificador abstract não permitem ser instanciadas por qualquer objeto, sendo indicado para definir superclasses genéricas.
Obs.: Toda interface é implicitamente pública e abstrata;
métodos abstratos não possuem corpo, mas devem ser implementados na classe que implementa a interface.

-Modificador de classe, método e variável **final**:
Uma classe **final,** é uma classe que não permite que seja criada subclasses dela.Ou seja, não permite herança.
Métodos **final** impedem que um método seja subscrito na subclasse que extende a classe a qual este método faz parte.
Variáveis declaradas **final** não podem possuir valor ou referências que sejam alteradas (são uma constante), incluso nisso argumentos passados a métodos.
Obs.: No caso de objetos, a referência para eles que é **final**, não o próprio objeto.

-Modificador de classe **default**:
É quando não é declarado explicitamente nenhum dos outros modificadores, sendo acessível por classes contidas no mesmo pacote.
Obs.: Variáveis com escopo de método(locais) continuam locais a onde foram declaradas, e não podem ser acessadas fora deles.

-Modificador de método e atributo **protected**:
Permite que um método somente possa ser invocado na classe em que é definido e nas suas subclasses. O mesmo vale para o acesso aos atributos.

-Modificador de método e atributo **private**:
O modificador **private** permite a invocação do método somente na classe a qual foi definido.
Um atributo **private** possui somente escopo de classe, não podendo ser acessado diretamente de qualquer classe ou subclasse que implemente a classe a qual está definido.

-Modificador de método, classes aninhadas e variáveis de classe **static**:
O modificador **static** permite que um método possa ser invocado sem a utilização de um objeto.
O uso de **static** permite que uma variável de classe possua uma única referência em memória.
Métodos **static** não conseguem acessar diretamente variáveis ou métodos não **static**, não podem ser sobrescritos, mas podem ser redefinidos.
Variáveis e métodos **static** são membros unicamente da classe a qual estão inseridos, não fazendo parte de nenhuma instância em particular.
Obs.(1): As variáveis estáticas(**static**) não são estáticas no sentido que não podem ser mudadas(isso seria **final**), mas sim por não permitirem serem criadas dinamicamente (leia-se em execução) para uma dada instância de um objeto.
Obs.(2): Uma classe aninhada é uma classe definida dentro de outra classe.
Há muitos outros modificadores como synchronized (para permitir acesso de apenas uma thread por vez a um método ou bloco), transient (permite a JVM excluir uma variável de instância da lista de seriação (muito importante para se poupar memória quando não se deseja persistir esse atributo em disco), entre outros. Mas os mostrados aqui são os principais para quem está começando com o Java.

Depois dessa carga teórica, está na hora de fixarmos os conceitos aqui comentados, pois serão úteis na sua vida de programador.Vamos a prática observando as três classes a seguir:

```
package acessibilidade;

 public class Membros{
        private static String tipoDoUsuario= "";

        protected static String verificarStatus(int id){
                if(id==1)
                   tipoDoUsuario= "Administrador";
                else
                  if(id<=10)
                    tipoDoUsuario= "Moderador";
                  else
```

```
                    tipoDoUsuario= "Usuario";

                return tipoDoUsuario;             }

    }


package acessibilidade;
import java.io.*;

 public class Arquivo extends Membros{

    private  static PrintWriter grava;
    private  static BufferedReader lerArq;
    private  static String arquivo;

    public Arquivo(String arquivo){
       this.arquivo=arquivo;      }

 // método gravarArquivo();
  public static void gravarArquivo(String nome,int id){

    try
         {
              grava =
              new PrintWriter(new BufferedWriter(
               new FileWriter("c:\\"+(arquivo),true),1*1024*1024));
              /* crio um arq no dir C para receber um nome dado pelo
usuário */
              grava.println("Nome:"+nome);
              grava.println("Id:"+id);
              grava.println("Status:"+verificarStatus(id));
            grava.close();
         }
    catch (IOException erro)
         {   System.err.println("Erro ao Gravar Arquivo!!!");
             erro.printStackTrace();
             System.exit(1);
         }
                                    } //fim de grava arquivo


 // método exibeArquivo();
  public static void exibirArquivo(){

     try
          {    lerArq=new BufferedReader(new FileReader(arquivo));
               String linha= null;
                // le linha por linha do arquvo e mostra na tela
               while ((linha = lerArq.readLine()) != null)
                  System.out.println(linha);

               lerArq.close();
           }
     catch (IOException erro) {
                System.err.println("Erro de Leitura!!!: "+ erro);
```

```
                erro.printStackTrace();
            }
    }
} //fim da classe Arquivo
```

```
import acessibilidade.Arquivo;

public class Principal{
        public static void main(String []guj){
                Arquivo a=new Arquivo("guj.txt");
                a.gravarArquivo("Paulo",1);
                a.gravarArquivo("Daniel",10);
                a.gravarArquivo("Ana",100);
                a.exibirArquivo();
        }
} //fim da classe Principal
```

Observando a classe Membros, logo na primeira linha está escrito:
package acessibilidade;
O que é isso?
Um pacote(**package**) serve para agrupar classes relacionadas, ao compilar um dos seus componentes (por exemplo **Membros**.java)será criado um diretório com o nome do pacote criado(no caso acessibilidade), com o .**class** do membro compilado. Para compilar um **package**:
javac pacote/Programa.Java
e para rodar:
java pacote.Programa (no nosso caso será necessário compilar **Principal**, que contém nosso método **main**).

Nota: a instrução import nome.Do.Package.Classe; permite uma classe utilizar uma classe dentro desse **package** (no caso de nome **Classe**).

Depois de compilar os 3 arquivos e rodar Principal.java veremos a saída no prompt de comando(tela):

Nome:Paulo
Id:1
Status:Administrador
Nome:Daniel
Id:10
Status:Moderador
Nome:Ana
Id:100
Status:Usuario
Press any key to continue...

Veremos também, que foi criado no diretório C: um arquivinho de nome **guj**.**txt** com os mesmos itens mostrados no prompt. Analisando a classe Membros, notamos uma variável de classe ***tipoDoUsuario ,*** que é privada, logo nenhuma outra classe ou subclasse poderá acessá-la!

-Qual será a vantagem disso?
É simples: -Além do acesso, quanto mais restritivo for um atributo, maior será o controle de um programador sobre ele, evitando uma futura inconsistência nos dados do seu programa.
Continuando, temos nessa mesma classe, um método protegido chamado verificarStatus() que retornará o tipo de usuário de acordo com o Id(que seria o número de registro) passado para o método. O fato de ser protegido permite ao método ser visível apenas para suas subclasses e classes no mesmo pacote (no caso **Arquivo** que *herda* de **Membros**).
Observando a classe **Arquivo**, temos 3 variáves de classe(grava,lerArq e arquivo) e 2 métodos de classe, *gravarArquivo* e *exibirArquivo* (obs.:métodos de classe possuem o modificador static, métodos de instância, não).Note o construtor da classe recebendo arquivo como parâmetro.Isso deixa nossa variável inicializável quando criarmos uma instância de **Arquivo**(o que será feito na classe **Principal**).Observando o método *gravarArquivo*, vemos que ele recebe 2 parâmetros(um nome e um id) e que dentro de um bloco **try**-**catch** esses dados serão persistidos para o disco.O bloco dentro desse método é quase igual ao dentro da classe **Variaveis** (de tópicos anteriores), exceto pela presença de um **BufferedWriter** e dos parâmetros true e 1*1024*1024 passados a **FileWriter** e **BufferedWriter** respectivamente.
O que é um **buffer**?**Buffer** é uma área reservada da memória que armazena temporariamente os dados que serão persistidos pelo seu arquivo. Ela é muito mais rápida do que gravar/ler diretamente do disco, o que gera uma otimização na sua entrada/saída para o disco. Por isso usamos um **BufferedWriter** e um **BufferedReader**. O valor true em **FileWriter** permite o modo *append*(anexar) no arquivo **guj**.**txt**. Se executarmos esse programa 2 vezes veremos que o arquivo foi duplicado e não sobscrito!O valor 1*1024*1024 especifica o valor de **BufferedWriter**.Quando o **buffer** se enche, os dados são realmente persistidos em disco.É importante notar a chamada de á ***verificarStatus*(id)** dentro de **grava**.***println***(); pois isso fará que seja persistida o retorno do método da classe **Membros** que é o nosso atributo privado *tipoDoUsuario*!Algo que o mecanismo de herança de Java sabe aproveitar muito bem!
Vejamos o método ***exibirArquivo***(): temos um leitor de caracteres(**FileReader**) que lerá nosso arquivo **guj**.**txt** e usamos a variável *lerArq*,que é uma instância de **BufferedReader** que através de seu método ***readLine***() nos permite ler linha a linha **guj**.**txt** e mostrar na tela o seu conteúdo, o que é feito no trecho:

```
while ((linha = lerArq.readLine()) != null)
        System.out.println(linha);
```

Depois temos nossa classe **Principal**, que contém o método ***main***() aonde criaremos a instância de **Arquivo** a,que recebe **guj**.**txt** como parâmetro, e chamamos 3 vezes ***gravarArquivo***() passando diferentes parâmetros(o que fará ***verificarStatus***() de **Membros** trabalhar!). Depois exibiremos seu conteúdo na tela através de a.***exibirArquivo***(). Veja que na primeira linha importamos *acessibilidade*.**Arquivo**; que nos permite trabalhar com os métodos dessa classe.O leitor pode estar se perguntando:
-Se eu importar *acessibilidade*.**Membros**; e criar uma instância de classe:
Membros m=new Membros();
Chamando:
m.verificarStatus(10);
Posso acessar diretamente o método ***verificarStatus***() certo?
ERRADO!Será recebida uma msg do tipo:

verificarStatus(int) has protected access in acessibilidade.Membros

A instanciação (criação) só funcionaria se ***verificarStatus***() tivesse acesso público (**public**).
Como não custa nada lembrar, só conseguimos acessar membros protegidos no mesmo pacote ou via herança! Jamais por instância!(Isso serve para atributos e métodos!).

.::3-Alocação em Memória e Passagens de Parâmetros::.

3.1-As áreas em memória.

Passagens de parâmetro em Java é um assunto que mesmo programadores experientes na linguagem tropeçam ao tentar explicar como ela ocorre. Antes de irmos direto ao assunto, é necessário termos idéia de como as variáveis são alocadas na memória. Como já foi dito antes, uma variável é um nome associado a uma posição (endereço hexadecimal) de memória.Na sua declaração, o compilador "arranja" (aloca) um espaço para ela. Como isso ocorre?É simples: A VM (máquina virtual em inglês) define várias áreas de execução de dados que são utilizadas durante a execução de um programa (algumas criadas no *start-up* da VM e destruídas quando ela termina seu trabalho, outras existentes durante o ciclo de vida de uma **thread**), mas devemos considerar duas áreas cruciais para a alocação e execução de dados na sua memória principal (RAM) que são utilizadas pela Java Virtual Machine (JVM) durante a execução de um programa: a **stack** e o **heap**. Vamos á elas e as demais áreas de dados em execução:

A **stack** ("pilha") é uma área com suporte direto ao processador, através de seu apontador de pilha(stack pointer-sp).Ele empilha (faz um"push") para criar nova memória, e desempilha("pop") para liberar aquela memória;ou seja, cabe ao sp guardar o endereço do próximo endereço vago na **stack** (o topo da stack). A **stack** é uma forma extremamente rápida para alocar espaço, perdendo apenas para registradores (que são localizados dentro dos processadores), mas o programador não tem qualquer controle ou evidência de que seu programa fará uso deles. O compilador sabe (durante a criação do ***.class***) o tamanho e o tempo de vida de cada dado que será armazenado na **stack**, porque através desse conhecimento é que será gerado o código para movimentar o **sp** (conforme aloca ou libera espaço na memória).O limite da pilha (por default o tamanho da **stack** é 220KB na VM da Sun em máquinas usando Windows) restringe o que pode ser armazenado nela.Em geral, armazena variáveis locais, chamadas a métodos com seus parâmetros, variáveis temporárias para algum cálculo e referências a objetos (não os objetos em si, que ficam na **heap**).

A **heap** ("monte") é um local da memória que armazena todos os objetos que serão utilizados no seu programa (por isso é conhecido também como pool de memória). Quando um objeto é instanciado (geralmente através do operador new), esses objetos criados e seus respectivos parâmetros são automaticamente alocados na **heap** (através de seu construtor). Quando um método que utiliza o objeto é finalizado, uma exceção ocorre, ou o número de referências ao objeto cai a zero, ou **threads** que utilizam o mesmo morrem (são finalizadas); Ele -o objeto- fica passível de ser coletado pelo **Garbage Collector**, apesar de não sabermos quando isso ocorrerá (ficará a cargo da Máquina Virtual).

Existem outras áreas de dados em execução como a **MethodArea** (área de método), compartilhada entre todas as **threads** da JVM, armazenando estruturas como o Pool de Constantes, campos e dados de métodos; o **Pool de Constantes** (pertencente a **MethodArea**), que contém vários tipos de constantes, incluindo de literais numéricas (**int**, **float**...) que são conhecidas em tempo de compilação até métodos e campos de referências que devem ser resolvidas em tempo de execução; e a **pilha de métodos nativos**, que permite usar métodos escritos em outras linguagens, como C, mas sua existência depende da implementação da JVM(algumas permitem,outras não), e o **pc register** . A VM pode suportar diversas **threads** ao mesmo tempo (na verdade quase, elas são **concorrentes**, não **simultâneas**!). Cada **thread** possui seu próprio registrador contador de programas (**pc register**). De qualquer ponto, cada **thread** está executando o código de um único método, o método corrente para aquela **thread**. Se o método não é nativo, o registrador contém o endereço da instrução da VM que está sendo executada, se o método não é nativo, seu valor é indefinido. Esse registrador (pc) é grande o suficiente para armazenar um endereço de retorno, ou um ponteiro nativo específico de uma dada plataforma. O programador não possui qualquer controle direto sobre esse registrador ou qualquer área de dados que alocam/liberam memória (são transparentes ao programador)!

Essas áreas de memória são apenas para dar uma visão maior da plataforma Java em si, mas para nós, programadores, o que nos interessa é a **stack** e a

heap, pois precisamos delas para saber como se processam as passagens de parâmetro em Java. O leitor já se perguntou o que ocorre em memória quando um simples objeto é alocado?Dada uma classe **Guj**, resolvemos criar uma instância **g** em memória de **Guj**. Ao fazermos:

Guj g=new **Guj**();

É utilizada uma área em memória da **stack** (com a variável de referência "g") para armazenar o endereço de memória **heap** aonde foi criada (logicamente) a estrutura do objeto **Guj**! Note que ao ser feita a declaração na **stack (**de **Guj g;)** o objeto ainda não existe em memória realmente!Pois só quando fazemos **new Guj();** alocamos esse objeto na área **heap** (aonde nossos objetos residem!). Quando criamos uma nova instância de **Guj** e atribuímos g a essa instância, passam a existir apenas uma área de memória em **heap**, mas existem duas áreas para as variáveis de referência na nossa pilha (**stack**). Exemplo:

Guj *g1*=new **Guj**();
g1=**g**; //apontam para o mesmo local em heap!

Apesar de ser relativamente grande, o espaço na **stack** e na **heap** são limitados (pelo algoritmo empregado em dada VM e pelo seu espaço físico de memória principal disponível). Um sistema prevalente (que aloja todos os seus objetos em RAM) armazenando milhões de objetos na memória ou um programa que abusasse de recursividade criando inúmeras variáveis locais a cada iteração fatalmente estourariam os limites da **heap** e da **stack** . Alguns comandos que devemos saber para otimizar nossos programas:
*__***Atenção***__!Não são aplicáveis a todas as VMs (mas principalmente a HotspotVM)!

-Alterar o tamanho da sua **stack** padrão:

java -**Xss**:**tamanho *SeuPrograma***

-Maior tamanho da **stack** possível:

java –**Xoss**

-Maior tamanho possível da **heap**:

java -**XX**:+**AggressiveHeap**

Isso foi só uma curiosidade, mas tome cuidado ao passar esses parâmetros, pois gerar um caos no seu programa!Dê uma passada em http://java.sun.com/docs/hotspot/VMOptions.html para saber mais!

3.2-Passagens de Parâmetro em Java:

Existe um mito em Java que precisa ser esclarecido:
-Objetos são passados por referência, tipos primitivos são passados por

valor.Certo?**ERRADO**!**Todos os parâmetros em Java são passados por valor**, a diferença é que, no caso de objetos, são passados referências a esses objetos, nunca os próprios objetos! Deve ser observado que os valores das variáveis são sempre primitivos ou referências a objetos, jamais os objetos propriamente ditos!Vamos a exemplos práticos :

```
public class Parametros{


  public static void main(String []args){
         int y=0;
         Object x=null;
            tipoPrimitivo(y);
            tipoReferencia(x);
         System.out.println(y);
         System.out.println(x);
                                              }
  public static int tipoPrimitivo(int parametro){
            parametro=30*10;
     return parametro;
                                            }
  public static Object tipoReferencia(Object o){
        String a= "GUJ";
        o=a; /*NOTA:atribuição permitida pois
               String também é um Object(todas
               as classes extendem java.lang.Object).
               Poderia ser executada a atribuição o="GUJ"
               diretamente sem problemas. */
     return o;
                                                    }
 } //fim da classe Parametros
```

Compile e rode.(**javac Parametros.java** e **java Parametros**)
Reparou na tela?Veja:

0
null

E não:

300
GUJ

Porque? Porque na chamada a ***tipoPrimitivo*(y);** uma cópia do valor inicial da variável (**0**) é passado como parâmetro ao método, não a variável em si, quando esse método termina, a variável permanece inalterada, o mesmo vale para a chamada a ***tipoReferencia*(x)** , pois uma cópia da variável é passada, não a própria! É o mesmo tipo de cópia que acontece quando fazemos uma instrução de atribuição a uma variável!Jamais será passado a própria variável

ao método, pois se fosse possível fazê-lo em Java, seu conteúdo seria modificado e isso caracterizaria uma passagem por referência! Na próxima seção veremos mais sobre esse assunto e alguns erros que devemos evitar programando!

3.3-Alguns erros a evitar.

Veja a seguinte classe:

```
public class Confusao{ //classe com muitos erros propositais!!! ;)

     private static String s;

     public static void mudaValor(int a[]){
         for(int i=0;i<a.length;i++)
           a[i]+=17;
       }

     public static int mudaValor(int a){
        a*=10;
       return a;
                                        }
     public static String mudaValor(String s){
        s=s.concat(" O Maior Forum De Usuarios Java Do Brasil");

          return s;                              }
     public static void mudaValor(int[] ref,int x,int y){ //troca os
//elementos em um array
        int temp;
        temp=ref[x];
        ref[x]=ref[y];
        ref[y]=temp;
                                                          }
     public static void bolha(int []a){
          for(int i=0;i<a.length;i++) //controla o número de passagens
           for(int j=0;j<a.length-1;j++) //controla as comparações e
//trocas
              if(a[j]>a[j+1]) //compara o elemento com o seguinte,se for
//maior
                mudaValor(a,j,j+1); //haverá a troca.
                                       }
     public static void main(String []GUJ){
        int a[]={123,6111,52,13,2};

        s=new String("GUJ");
        //super exemplo de Overloading-sobrecarga de métodos
        mudaValor(a); //muda o valor do array
        System.out.println(a); //array eh um objeto sera exibido seu
//endereço Heap

        mudaValor(a[2]); //muda o valor da variavel
        System.out.println(a[2]);
```

```
        mudaValor(s); //tenta mudar o valor da String
        System.out.println(s);

        bolha(a);
        System.out.println(a);

        }
}
```

Compile (***javac Confusao.java***)e rode (***java Confusao***).Uma classe perfeita não?Ela é perfeita para se jogar no lixo!Os piores erros são o que passam na compilação e na execução, pois estes são difíceis de serem depurados!Nesse exemplo será fácil de ver os erros, mas nem sempre será assim!

Temos 5 métodos de classe além do ***main***(), sendo que 4 deles com mesmo nome(**mudaValor**), mas assinaturas distintas(a assinatura de um método é composta de nome+lista de parâmetros). A essa possibilidade de métodos possuírem mesmo nome, mas diferentes listas de argumentos é chamada de ***overloading*** (sobrecarga), cabendo ao *Interpretador* selecionar qual método deve ser executado combinando a lista de argumentos com o parâmetro passado a chamada do método. A essa associação de atributos feita em tempo de compilação, é chamada de ***early binding*** (ligação prematura) ou ***static binding*** (ligação estática); que é executada pelo Carregador de Classe (**ClassLoader**), procurando os membros estáticos na sua classe. Se existirem, eles serão carregados!

NOTA:Quando executamos métodos não estáticos, ou algum método sofre ***overriding*** (é subscrito) numa subclasse, o compilador não sabe qual método será chamado e isso só é resolvido em tempo de execução, processo chamado de **late binding** (ligação tardia), **dinamic binding**(ligação dinâmica) ou a melhor definição, **virtual method invocation** (invocação de método virtual) pois os métodos "existem" no momento da execução(nenhum código extra é gerado).

Em teoria essa classe deveria adicionar 17 á todos os membros do array a, através da chamada a primeira função ***mudaValor***() e mostrá-lo na tela. Deveria através da chamada a ***mudaValor*(a[2]);** mudar o valor da variável (o que não ocorre). A terceira chamada era para retornar a String modificada "GUJ O Maior Forum De Usuarios Java Do Brasil", mas só GUJ é retornado. E a chamada ao método bolha, era para permitir exibir no método println() o array a ordenado do menor para o maior. Nota-se que é exibido uns caracteres estranhos (do tipo **[I@10b62c9**) assim como na primeira chamada a ***println***(). Porque isso ocorre?Porque é passado uma referência a um array em ambos os casos! Todos os objetos tem associados a eles uma referência a própria classe, quando uma referência a um array é passada ao método ***println***() há uma chamada ao método ***ToString*** que retornará o nome da classe a qual esse objeto é uma instância(no caso [I-de Inteiro), um separador "@", e uma representação hexadecimal representando o código *hash* desse objeto.Veja o formato:

getClass().getName() + '@' + Integer.toHexString(hashCode())

Nota-se que nas duas chamadas a ***println***(a) serão retornadas a mesma String!Isso se deve porque o objeto está num único local na tabela hash (esse endereço de memória de mentirinha que é @numero_hexadecimal).Não

devemos passar um objeto a ***println***() a menos que se queira mostrá-lo dessa forma, senão poderá ser necessário subscrever o método ***println***()(para retornar um objeto). Prosseguindo com a análise do código, a chamada ao primeiro ***mudaValor*(a)** e ao método ***bolha*(a)** tem êxito, o problema foi passar uma referência ao método ***println***() para exibir na tela! Deve-se usar um laço for para poder exibir os valores corretos na tela:

```
for(int i=0;i<a.length;i++)
   System.out.println(a[i]);
```

Dessa forma, todos os índices serão percorridos, pois ocorre uma passagem por valor (índice após índice) a cada interação do laço.(Nota: **a.length** retorna o tamanho total do vetor).
Prosseguindo, temos a segunda chamada a ***mudaValor*()** na qual é passado **a[2].**Exibirá 69 na tela, logo a operação teve êxito, certo?ERRADO!Deveria exibir 690, correspondente ao **mudaValor()** que recebe um tipo inteiro e multiplica por 10, mas o valor da posição 3 do array (assim como o array todo) foi modificado pela chamada ao primeiro **mudaValor()**na qual foi passado o array **a**.Ele não pode ser exibido(o array **a** modificado com os novos valores) porque foi passado erroneamente a **println()** também! Além do mais, quando se passa um índice (um elemento) individual de um array de tipo de dados primitivos essa passagem é feita por valor como uma simples variável! Elementos individuais de um array de tipo referência(Object, String...) são passados como uma referência a esse objeto!O valor passado (69) não foi modificado, pois a cópia de **a[2]** ficou limitada a execução no escopo do método!
A terceira chamada a **mudaValor()** é tudo o que você não pode fazer em Java! Não há como mudar o valor de uma determinada String numa passagem para um método porque **Strings** são **imutáveis**!E nesse caso, o gasto de memória foi um desperdício! (Em relação ao que pode ser feito!). Jamais tente modificar **Strings** via método. Quando precisar de **Strings** modificáveis, use java.lang.**StringBuffer**(com sincronização) ou **StringBuilder** (sem sincronização, portanto mais rápida!).
A chamada a ***bolha*(a);** faz executar um algoritmo de ordenação(do menor para o maior) conhecido como **BubbleSort** que ordenará o array **a**. Esse método é ótimo para demonstração, mas péssimo em eficiência (Quando precisar de eficiência, procure usar um algoritmo como **MergeSort**, por exemplo). Repare que esse método faz várias passagens pelo array comparando pares sucessivos em cada passagem. Se um par estiver em uma ordem crescente, ou os valores forem iguais, nada será mudado. Se um par estiver na ordem decrescente, seus valores são trocados no array. No trecho: ***mudaValor*(a,j,j+1);** haverá uma chamada para trocar a posição de dois elementos para ficarem em ordem crescente. Deve ser notado que uma referência **ref** é passada junto a dois números inteiros que representam os índices do array, e **temp** será a variável que armazenará o conteúdo temporariamente de um dos valores para evitar que qualquer conteúdo seja perdido durante as atribuições. É necessário reparar bem nessa última assinatura do método ***mudaValor(int[] ref,int x,int y).*** É a popular operação

de **swap** (troca) de variáveis que veremos mais a fundo no próximo tópico. Uma provável solução para a classe Confusao pode ser :

```
public class Solucao{ //classe com as devidas correcoes


     public static void mudaArray(int a[]){
         for(int i=0;i<a.length;i++)
           a[i]+=17;
        }

     public static int mudaElemento(int a){
            a*=10;
          return a;
                                        }
     public static void troca(int[] ref,int x,int y){
        int temp;
        temp=ref[x];
        ref[x]=ref[y];
        ref[y]=temp;
                                                      }
     public static void bolha(int []a){
          for(int i=0;i<a.length;i++)
           for(int j=0;j<a.length-1;j++)
              if(a[j]>a[j+1])
                troca(a,j,j+1);
                                       }
     public static void main(String []GUJ){
        int a[]={123,6111,52,13,2};
        String s="GUJ";

          mudaArray(a);
        for(int i=0;i<a.length;i++) //para exibir os elementos modificados
         System.out.println(a[i]+"\n");

        a[2]=mudaElemento(a[2]); //passagem por valor
        System.out.println("Valor do elemento na posicao 3 do
array:"+a[2]+"\n");

        s=s.concat(" O Maior Forum De Usuarios Java Do Brasil");
        System.out.println(s+"\n");

        bolha(a); //ordena os elementos
        for(int i=0;i<a.length;i++) //para exibir os elementos ordenados
              System.out.println(a[i]);

        }

 }
```

Acabamos com esse **overloading** desnecessário, o programador deve escrever os métodos mais intuitivos (de fácil compreensão) possíveis, logo foi eliminada aquela população de ***mudaValor*** em série por ***mudaArray***, ***mudaIndice*** e troca(lembrando que o que manipulava String foi eliminado!). Para

mudaArray e ***bolha***(que já funcionavam no exemplo anterior) apenas incluímos o laço for para correta exibição dos elementos na tela. Em :
a[2]=mudaElemento(a[2]);
Fez-se necessária essa atribuição porque é o único modo de se receber o resultado da cópia passada como parâmetro para ***mudaElemento***(), devido a passagem ser por valor.
Já para a String eliminamos o gasto de ter aquele método inútil na memória e fizemos a concatenação (via ***concat***()) que adiciona uma String ao final da String **s** (de valor "GUJ"). É importante perceber que o objeto String é imutável mais sua variável de referência na **stack** não!
Nesse exemplo, existem 3 objetos String em heap ("GUJ"," O Maior Forum De Usuarios Java Do Brasil" e "GUJ O Maior Forum De Usuarios Java Do Brasil"), mas apenas um está sendo referenciado("GUJ O Maior Forum De Usuarios Java Do Brasil"). Os demais são considerados perdidos! Já pensou num programa ter zilhões de concatenações e Strings novas sendo criadas á todo momento? Um **OutOfMemoryError** na sua aplicação poderia ser catastrófico! Pense bem antes de precisar concatenar muitas Strings (A classe StringBuffer e StringBuilder são para isso!).E a **heap** pode ser relativamente vasta. Mas não é infinita.
Essa classe de "Solução" é perfeita?Não. Longe disso. Ela apenas serve para ilustrar que é melhor que a anterior, mas várias otimizações podem (e devem!) ser feitas. O uso do modificador static em excesso é uma forma pobre de programação (e **proceduralização** do Java!) e o único intuito no exemplo em uso é evitar a criação de instâncias da classe para chamar os métodos. E o método ***mudaElemento*** poderia ser eliminado apenas colocando um **if** em ***mudaArray*** verificando se é dado elemento do array e efetuar a operação desejada quando este for encontrado. Na vida real de programador é necessário ser mais atencioso (e menos preguiçoso!) para observar as melhorias que sempre podem ser feitas em um programa.

Nota: O tamanho da **heap** é calculado baseado na memória física (**RAM**) da máquina e é feito pelo algoritmo que faz alocação de espaço para tenta usar o máximo possível. Na verdade, ele tenta usar metade da memória disponível, mas se a memória for inferior a 160MegaBytes, ele (o algoritmo alocador de espaço) tentará usar o máximo possível.
Nota2: Deve-se salientar que em Java o programador não é responsável pela liberação de memória, não possuindo qualquer capacidade de manipulação direta dos objetos residentes em memória **heap**. Em C, por exemplo, o programador é responsável pela liberação da memória em uso.Isso é perigoso!Em Java, o coletor de lixo(**Garbage Collector**) se encarrega de limpá-la pelo programador, evitando erros como acesso a dados que foram desalocados e estouro pela não liberação de objetos.O programador não tem o menor controle sobre o coletor de lixo, que é uma **thread** de baixa prioridade da Máquina Virtual, cuida de todo o processo.O máximo que é possível fazer é chamar System.***gc***(); para solicitar a execução do coletor, mas não é garantida a sua execução.Cuide para que o objeto que tenha que ser coletado não possua nenhuma referência a ele (ou uma atribuição **null** depois de seu uso), pois só assim o coletor poderá entrar em ação.

3.4-O que é passagem por referência?

A passagem por referência permite passar uma variável para uma função em C++(o que em Java seria um método) e retornar essas mudanças ao seu chamador. O conteúdo da variável muda de verdade!E um posterior uso e/ou chamada da variável mostraria essa mudança.
Na Linguagem C, pode se obter o mesmo efeito de uma passagem por referência utilizando ponteiros, mas não é a mesma coisa que uma passagem feita em C++, essa sim, passagem de referência!Será visto o porque.
Antes de tudo, o que são ponteiros?(Ok, programador Java não tem obrigação de saber disso, mas é sempre bom ter uma idéia!).
Ponteiros são variáveis (ou valores) que contém um endereço(de uma posição na memória).
Em C, quando uma função precisa alterar o valor de um parâmetro, o programa deve passar para a função um ponteiro para o endereço de memória da variável.Em C++, isso é simplificado com o uso de uma referência, que é um apelido (um segundo nome) para uma variável que faz uma ligação direta para uma posição de memória. O uso da referência elimina a necessidade do ponteiro e, quaisquer operações (como incremento ou decremento) afetarão o valor da variável referenciada. Ao passar um parâmetro por ponteiro (em C), seu valor é copiado na **stack** e passado ao método.Se esse ponteiro for desreferenciado (**Obs**.:desreferenciar um ponteiro é o processo de acessar o valor em uma posição específica na memória), se chegará na mesma área de memória apontada pelo ponteiro inicial, o que dá a impressão de uma passagem por referência,o que não ocorre uma vez que é uma cópia do ponteiro que aponta para o mesmo lugar do original, mas o ponteiro original passado não pode ser alterado!Em Java, uma cópia da variável de referência a objeto quando é passada, executa a grosso modo, um processo similar ao feito em C, só que sem ponteiros explícitos, pois a referência ao objeto "aponta" a um local em memória!Exemplo (só para visualização) de uma função/método de troca(**swap**) de variáveis em 3 linguagens distintas :

->Método em Java (passagem por referência a objeto):

```
void troca(int[] ref,int x,int y){
      int temp;
      temp=ref[x];
      ref[x]=ref[y];
      ref[y]=temp;
} //troca feita em Java
```

-Chamada ao método:

troca(ref, x, x+1);

->Função em C(passagem por referência por ponteiros):

```
void troca(int *x, int *y){
      int temp = *x;
      *x = *y;
      *y = temp;
                          } //troca feita em C
```

-Chamada á função:

troca(&x,&y);

->Função em C++(passagem por referência):

```
void troca(int& x, int& y){
  int temp = x;
  x = y;
  y = temp;
                       } //troca feita em C++
```

-Chamada á função:

troca(x,y);

Apesar de não ser escopo desse tutorial fica aqui só para constar:
int *x e **int *y** são os dois ponteiros que a função recebe como parâmetros.
O operador **&(uppersign)** é chamado de *operador de endereços* em C(que são passados aos ponteiros pela chamada da função).
Em C++, o operador **&** é o *operador de referência* (ele fica geralmente associado ao tipo da variável para se distinguir do operador de endereços de C) para as que forem passadas a função ***troca***.
Se em Java fosse possível trocar dois objetos diretamente, teríamos a passagem de referência configurada na linguagem!

3.4.1-Porque devemos evitar a passagem por referência?

A passagem por referência mistura a entrada e a saída (dos dados) no código. Em Java, sabemos que uma variável passada em uma chamada de método não será alterada por operações inerentes a esse método. Em linguagens que utilizem esse tipo de passagem, não podemos afirmar isso. A passagem por referência torna confusa e pouco intuitiva a interface de um método tornando difícil sua compreensão.
Existem 2 motivos para querermos utilizar a passagem por referência: Reduzir o custo da chamada a um método, evitando a cópia de grandes quantidades de dados em memória, o que é facilmente solucionável em Java, pois os parâmetros passados em Java são meras cópias de referências e um objeto, por maior que ele seja, jamais será passado a um método; e, o principal motivo, que é o fato de poder realmente mudar o conteúdo de uma variável e essa mudança pode ser vista no código cliente. A melhor solução é refatorar seu código o quanto possível usando valores de retorno nos seus métodos para esse propósito. Deve-se atentar para a possibilidade de ser necessário utilizar múltiplos valores de retorno (o que em Java não é permitido) eliminando pedaços desnecessários do seu código, tratando corretamente possíveis exceções e/ou fragmentando o método em vários métodos menores para em cada um retornar uma parte da possível resposta. Assim ficará mais intuitivo do que criar um método gigantesco que tudo faz.

Até a próxima!

Notas Bibliográficas e Referências:

JLS(Java Language Specification):
http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html
Excelente Artigo sobre passagens de parâmetros em Java:
http://www.yoda.arachsys.com/java/passing.html (por Jon Skeet, Dale King e Chris Smith)