|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | [Classes](http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/classe.html) |   **Vetores e Matrizes**  **(Mande sua opinião e sugestões sobre este conteúdo)**  [info@mmadeira.org](mailto:madeira@unisul.br) |
| **Vetores**  Um vetor é um conjunto de dados do mesmo tipo armazenados em seqüência na memória do computador.  Podem ser classificados em dois tipos:  a) [aqueles que guardam tipos básicos](http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetores.htm#básicos) - **int**, **float**, **double**, **char** e **boolean**.  b) [aqueles que guardam referências para objetos](http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetores.htm#objetos) (handles).  **Vetores de tipos** **básicos**  Para declarar um vetor do tipo básico, temos as seguintes formas:  a) O vetor é criado com os espaços necessários e inicializados em zero:  int vet [ ] = new int[5];http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetor0.gif  No caso acima poderíamos usar a seguinte alternativa:  int [ ] vet = new int[5];  Esta notação é útil quando queremos declarar mais de um vetor, pois não necessitamos ficar repetindo os colchetes:  char [ ] vet1, vet2, vet3;  Neste caso de **vet** acima, o vetor pode contér valores do tipo **int**. O que criamos foi a reserva de espaço na memória com a capacidade de armazenar 5 valores do tipo **int**.  Observe que o vetor declarado com 5 posições é endereçado começando-se em zero: **vet[0]** até **vet[4]**.  Os tipos **int**, **float**, **double** produzem vetores inicializados com zero. O tipo **char** produz um vetor inicializado com **null** e o tipo **boolean** com seus valores em **false**.  b) O vetor é criado em função de valores iniciais fornecidos. Neste caso o espaço se ajusta automaticamente.  int vet [ ] = { 1,3,5,7,9 }; http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetor1.gif  Todos os vetores possuem um campo - **length** - de apenas leitura que contém seu tamanho.  Para os exemplos acima o contéudo de **length** é 5:  int x;  x = vet.length; // retorna o tamanho do vetor (número de células)  x agora contém 5.  Imagine cada espaço endereçado como uma variável. Ou seja, **vet[0]** pode ser usada como uma variável tem qualquer operação. Por exemplo  vet[1] = 10;  faz com que a posição 1 mude seu conteúdo para 10.http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetor2.gif  Pra cada célula, podemos incrementar, decrementar, realizar qualquer operação como se fosse uma simples variável:  vet[1]++;  vet[1]--;  vet[1] = vet[1] + 3; etc.  Uma função útil de **length** é servir de limite para um **for**:  for(int i = 0; i < vet.length; i++){  vet[i] = i\*2;  }  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetor3.gif  **O vetor de tipos básicos como argumento de um método**  Quando passado como argumento, o vetor é passado por **referência**, ou seja, alterando-se um elemento do vetor no interior do método, altera-se o original:  int vet [ ] = { 1,3,5,7,9 };  func(vet);  public void func( int v[ ] ) {  for(int i = 0; i < v.length. i++)  v[i] = 0;  v[2] = 10;  }  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetor4.gif  No exemplo acima, o valores de **vet** mudam após a execução do método **func**. Isto significa que passamos o **endereço do vetor**, e não uma cópia de seus valores. Em inglês este endereço tem o nome de **handle**. Observe que o nome do vetor dentro do método pode ser igual ou diferente do vetor original (lembre-se de que o método pode ter sido feito por outro programador, e a única coisa que ele tem de se preocupar é como o método é chamado e o que retorna).  A passagem de um elemento de um vetor para dentro de um método respeita o mesmo mecanismo da passagem de qualquer tipo básico, já que para todos os efeitos um elemento se comporta como uma variável comum:  int vet [ ] = { 1,3,5,7,9 };  func2(vet[ 0 ]);  public void func2( int x ) {  x = 3;  }  No exemplo acima, nada acontece com o vetor **vet** após a chamada de **func2**.  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetor5.gif  **Vetores de referências para** **objetos (handles)**  (Obs.: a classe utilizada abaixo é a **StringBuffer**, que é uma seqüência de caracteres, assim como **String**. Não foi usada a classe **String** pois ela tem uma proteção que faz com que seja passada por cópia para dentro de métodos. Este assunto é visto em estudos mais avançados da linguagem).  Para objetos, que são "variáveis" instanciados de classes, o vetor é inicializado com endereços nulos. Quando um vetor deste tipo é declarado, queremos dizer que estamos reservando um número de células capazes de apontar para objetos do tipo tal.  Exemplo com a classe **StringBuffer**:  StringBuffer vet[ ] = new StringBuffer[3];  Como só estamos reservando o espaço, ainda não temos as seqüências de caracteres realmente. O que a declaração faz é inicializar as células com valores **null**.  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetorObj1.gif  Podemos então guardar dentro destas células endereços o endereço de seqüências de caracteres:  vet[0] = new StringBuffer("Alô");  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetorObj2.gif  Portanto, **vet[0]** é, para todos os efeitos de programação, um objeto do tipo **StringBuffer**.  **Passado como parâmetro**  Na passagem como parâmetro, **vet** é passado por **referência**. Observe este exemplo:  StringBuffer vet[ ] = new StringBuffer[3];  vet[0] = new StringBuffer("Alo1") ;  vet[1] = new StringBuffer("Alo2");  vet[2] = new StringBuffer("Alo3");  met(vet); // **chamando o método com o endereço de vet**  public void met(StringBuffer [ ] v){  v[0].reverse( ) = ; // Mudando o conteúdo de v[0] que equivale a vet[0].  // O método **reverse( )** inverte a seqüência de caracteres.  }  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetorObj3.gif  Veja que o endereço do vetor **vet** foi passado para dentro do método, e a mudança do conteúdo da posição 1 do vetor **v** causou a mudança da posição 1 do vetor orignal **vet**.  **Passando apenas um elemento do vetor**  A passagem de apenas de um elemento de um vetor de objetos para dentro de um método, equivale a passar a referência do objeto lá armazenado. Exemplo:  StringBuffer vet[ ] = new StringBuffer[3];  vet[0] = new StringBuffer("Alo1") ;  vet[1] = new StringBuffer("Alo2");  vet[2] = new StringBuffer("Alo3");  met(vet[0]); // chamando o método com o objeto contido em **vet[0]**  public void met(StringBuffer s){  s.append(" Modificado" ) = ; // Mudando o conteúdo de **s** que aponta para o objeto  // contido em vet[1]. O método **append( )** acrescenta a seqüência "Modificado".  }  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/vetorObj4.gif  Observa-se que **s** aponta para a seqüência armazenada em **vet[0]**. O endereço desta seqüência é passada para o método, ou seja, tando **vet[0]** quanto **s** apontam para o mesmo objeto na memória.  Em resumo, quando o vetor contém objetos, toda passagem é feita por referência.  **Matrizes**  As matrizes em Java equivalem a vetores de vetores, ou seja, em cada célula de um vetor principal é armazenado o endereço de outro vetor.  Exemplo de uma declaração:  int mat[ ][ ] = { { 1,2,3} , {4,5,6} }; http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/mat1.gif  A forma de endereçamenteo segue o esquema :  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/mat2.gif  Para mudarmos o elemento [0][2], usamos esta sintaxe:  mat[0][2] = 0; http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/mat3.gif  Para recuperar o elemento [1][1], usamos:  int v = mat[1][1]; // retornando 5 para **v**.  Podemos declarar matrizes com números de colunas iguais para cada linha de forma dinâmica:  int mat[ ][ ] = new int[3][3]; http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/mat4.gif  Como em Java, matrizes são na verdade vetores de vetores, podemos ter números diferentes de colunas para cada linha.  int mat [ ][ ];  mat = new int[3][]; // Alocamos as linhas;  mat[0] = new int[3]; // alocamos as colunas da linha 0  mat[1] = new int[2]; // alocamos as colunas da linha 1  mat[2] = new int[5]; // alocamos as colunas da linha 2  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/mat5.gif  O que acontece acima, é que temos um vetor base que aponta para outros vetores, mas com a conveniência de usarmos a notação normal de acesso a uma matriz.  O campo **length** está disponível para descobrirmos o tamanho dos vetores quando necessário.  Para **mat** acima, temos:  mat.length retorna 3  mat[0].length retorna também 3  mat[1].length retorna 2  mat[2].length retorna 5  Um **for** que inicializa esta matriz, pode ser escrito da seguinte forma.  for(int lin = 0; lin < mat.length; lin++)  for(int col = 0; col < mat[lin].length; col++)  mat[lin][col] = 1;  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/mat6.gif  **Matrizes como argumentos de métodos**  Neste caso passamos o endereço de toda a matriz através de seu nome. Como nos vetores, a passagem é feita por **referência**.  int mat[ ][ ] = new int[2][2]; // declaramos uma matriz dois X dois.  func( mat ); // chamada : passamos para o método apenas o nome.  public void func( int m[ ][ ] ) { // aqui estamos passando o endereço da matriz  for(int lin = 0; lin < m.length; lin++)  for(int col = 0; col < m[lin].length; col++)  m[lin][col] = 1;  }  Neste caso, após a execução de **func**, os valores de **mat** mudaram.  Para passar apenas um velor, este se comporta como uma variável do tipo da matriz e passada por **cópia**.  int mat[ ][ ] = new int[2][2]; // declaramos uma matriz dois X dois.  func( mat[1][0] ); // chamada : passamos apenas uma cópia do elemento [1][0]  public void func( int x ) { // aqui estamos passando o endereço da matriz  x = 3; // **não altera** o valor de mat[1][0].  }  Assim como nos vetores, as matrizes podem guardar tanto tipos básicos como objetos e todas as regras são as mesmas dos vetores anteriormente explicadas.   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Classes   |  | | --- | | Classes(Mande sua opinião e sugestões sobre este conteúdo) [info@mmadeira.org](mailto:madeira@unisul.br) | | Classe Uma classe é uma forma do programador criar novos tipos de dados, de tal forma que possa representar no computador um elemento do problema a ser resolvido. No processo de criar este novo tipo, estaremos criando um **tipo abstrato de dados**.  Para ilustrar o processo, vamos imaginar a necessidade de criar no computador uma forma de representar lâmpadas, que será utilizada em um pequeno simulador gráfico de funcionamento das mesmas. Para isto vamos criar a classe lâmpada.  class Lampada {  ...  }  Uma classe tenta representar o elemento do problema apenas com os detalhes deste que são necessários. Neste processo, donominado **abstração**, não consideramos aquilo que não interessa para resolver nosso problema.  Observando uma lâmpada, podemos dizer que ele possui **atributos e operações** possíveis de ser realizadas com ela. Atributos Entre os atributos que nos interessam, vamos selecionar estes:   * Potência (de 1 a 200 watts); * Voltagem (de 6 a 220 volts); * Estado (ligada ou desligada); * Cor.   Estes atributos são os dados que cada lâmpada guarda.  Em Java, poderemos aumentar nossa definição da classe **Lampada** para conter estes atributos, na forma de **variávies e objetos** (veremos o que são objetos mais a seguir):  class Lampada {  private int potencia; // guarda a potência em watt.  private int voltagem; // guarda a voltagem em volts.  private Color cor; // guarda a cor da lâmpada  private boolean estado; // guarda o estado:  // ligado (true) ou desligado (false)  }  Observamos que, para cada atributo, nós incluímos um especificar de acesso, neste caso a palavra-chave **private**. Isto chama-se esconder a informação, fazendo com que o que o usuário da classe não possa diretamente alterar os valores guardados por estes campos. Este procedimento deriva de uma série de necessidades da Engenharia de Software, onde a representação dos dados não deve ficar exposta, contribuindo para que erros em uma parte do programa não se propaguem para outras. Mas isto é outro assunto. O que nos interessa aqui é que os atributos não podem ser manipulados diretamente.  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/publicPrivate.gif Operações Podemos realizar algumas operações sobre a lâmpada, que denotam o seu comportamento:   * Ajustar e ler sua potência; * Ajustar e ler sua voltagem; * Ajustar e ler sua sua cor; * Ligar e Desligar; * Mandar que ela se desenhe na tela.   Estas operações, direta ou indiretamente, manipulam os atributos privados (**private**).  Vale observar que alguns atributos e operações não são apenas da lâmpada real, mas também do domínio da solução do problema, que em nosso caso é criar uma simulação no computador.  Acrescentamos à classe estas operações, que são implementadas como **métodos**:  class Lampada {  // atributos  private int potencia;  private int voltagem;  private Color cor;  private boolean estado;  // Métodos  public void setPotencia(int p){  if(p > 1 && p < 200) // uma crítica para não existirem lâmpadas fora dos parâmetros.  potencia = p;  else  potencia = 0;  // Esta alternativa é ruim, pois vai permitir criar lâmpadas  // com potência zero.  // Mais adiante será estudado o tratamento de excessões e  // veremos  // uma melhor forma de lidar com este tipo de erro.  }  public int getPotencia( ){  return potencia;  }  public void setVoltagem(int v){  if(v > 6 && v < 220) // Outra crítica  voltagem = v;  else  voltagem = 0;  }  public int getVoltagem( ){  return voltagem;  }  public void setEstado(boolena e){  estado = e; // true para ligado e false para desligado  }  public boolean getEstado( ){  return estado;  }  public void setCor(Color c){  cor = c; // Veja a página 498 para ver como funcioana a classe Color  }  public Color getCor( ){  return cor;  }  public String toString( ){  // aqui nossa lâmpada retorna uma string com seus dados  String est;  if(estado == true)  est = "ligado";  else  est = "desligado";  return "Potencia = " + potemcia + " " +  "Voltagem = " + voltagem + " " +  "Estado = " + est + " " +  "Cor = " + cor;  }  }  Nosso intuito é escrever um programa no modo gráfico para a apresentação das lâmpadas, onde elas serão desenhadas coloridas, de acordo com o valor do atributo **cor**. Nesta implementação inicial, para efeito de testes, ainda não desenharemos a lâmpada no modo gráfico. No entanto, usaremos o método **toString()** para "vermos" os valores dos atributos.   |  | | --- | | **Em resumo: uma classe possui atributos e operações:**  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/classe.gif  **Que são implementadas, ou codificadas, por variáveis, objetos e métodos.** |  Objetos Uma classe é uma espécie de "forma", de onde podemos gerar exemplares que se comportam como a classe descreve. Este processo é denominado **instanciação** e os exemplares **objetos**.  É interessante observar que a classe está para o tipo, assim como o objeto está para a variável. Dizemos que as intâncias dos tipos básicos chamam-se variávies, enquanto para as classe, objetos.   |  | | --- | | tipo -> variável  classe -> objeto |   Para criar um objeto de lâmpada na memória do computador, usamos o operador **new**:  new Lampada( );  Isto reserva uma região de memória para conter os atributos do objeto. Mas também precisamos de um nome que possamos manipular, como fazemos como uma variável:  Lampada lp1 = new Lampada( );  Dizemos que **lp1** mantém uma referência para o objeto na memória.  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/lp1a.jpg  No momento de criação do objeto acima, valores iniciais foram armazenados em cada atributo. Veremos mais para frente como mudar este comportamento.  Se quizermos mudar algum destes conteúdos, podemos usar os métodos **set** que definimos para a classe. O **set** significa ajuste em inglês. Usamos este nome seguido do nome do atributo por ser uma padronização utilizada, indispensável no desenvolvimento de **JavaBeans**.  Exemplo para o atributo **potencia**:  lp1.setPotencia(60);  http://inf.unisul.br/~madeira/disciplinas/teciii/tutorial/lp1b.jpg  Veja um programa **Applet** que utiliza a classe descrita acima:  import java.awt.\*; import javax.swing.\*;  class Lampada {  // atributos private int potencia; private int voltagem; private Color cor; private boolean estado; // Métodos public void setPotencia(int p){  // uma crítica para não existirem lâmpadas fora dos parâmetros. if(p > 1 && p < 200)  potencia = p;  else  potencia = 0;  // Esta alternativa é ruim, pois vai permitir criar lâmpadas com  // potência zero.  // Mais adiante será estudado o tratamento de excessões e veremos  // uma melhor forma de lidar com este tipo de erro.  } public int getPotencia( ){  return potencia;  } public void setVoltagem(int v){  if(v > 6 && v < 220) // Outra crítica  voltagem = v;  else  voltagem = 0;  } public int getVoltagem( ){  return voltagem;  }  public void setCor(Color c){  cor = c;  }  public void setEstado(boolean e){  estado = e; // true para ligado e false para desligado  } public boolean getEstado( ){  return estado;  } public String toString( ){  // aqui nossa lâmpada retorna uma string com seus dados String est; // Auxílio para apresentar na tela if(estado == true)  est = "ligado";  else  est = "desligado";  return "Potencia = " + potencia + " " + "Voltagem = " + voltagem + " " + "Estado = " + est + " " + "Cor = " + cor; }  }  // usando a classe **Lampada** acima public class TestaLampada extends JApplet{  public void init( ) {  JTextArea saida = new JTextArea(); Container c = getContentPane(); c.add(saida);  // cria a lâmpada l1 Lampada l1 = new Lampada();   // apresenta o conteúdo de seus atributos na tela String atributos = l1.toString(); saida.setText(atributos);  // ajustando alguns atributos  l1.setPotencia(60); l1.setVoltagem(110); Color cor = new Color(255,0,0); // veja página 498 do livro de Java  l1.setCor(cor);  l1.setEstado(true);   // verificando como ficou: atributos = l1.toString(); saida.append("\n" + atributos);  }  }  Como exercício, crie mais objetos do tipo **Lampada** e chame seus métodos. | | |