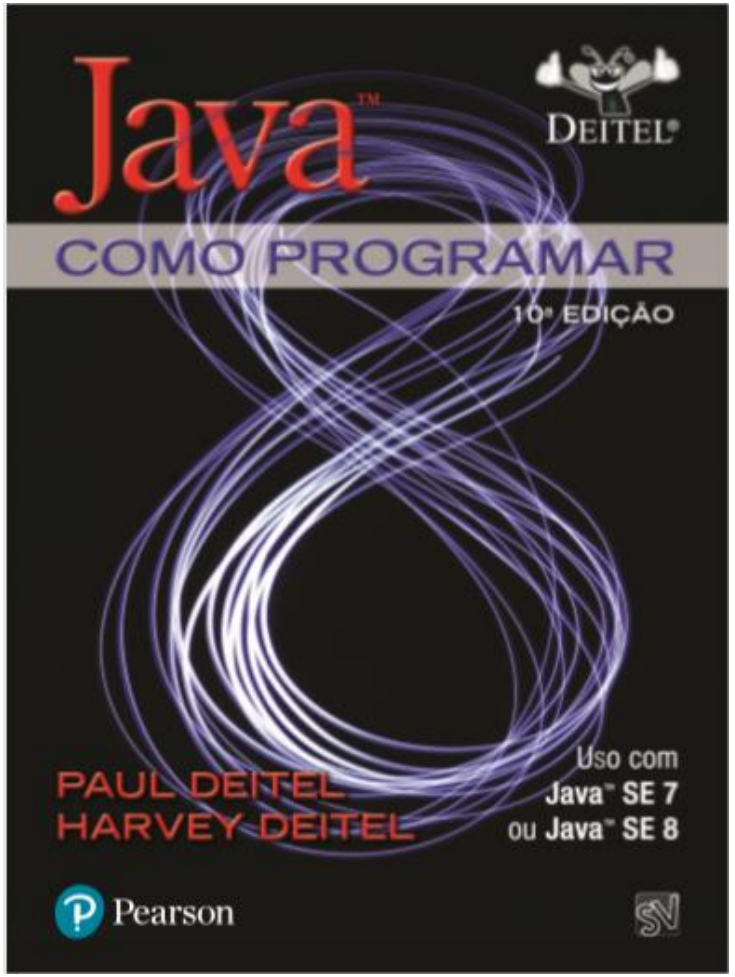
CAP - 2022

JAVA – CAP 2022

Conteúdo do Edital: CAP

- Linguagem de programação JAVA;
- Bibliotecas de classe do Java;
- Classes e Objetos;
- Instruções de controle;
- Módulos de programa em Java;
- Arrays e Arraylists;
- Programação orientada a objetos;
- Tratamento de exceções;
- Componentes GUI;
- Strings, caracteres e expressões regulares;
- Recursão;
- Applets e Java Web Start;
- Multithreading; e
- Serviços Web.



DEITEL, Paul; DEITEL, Harvey. JAVA como Programar. 10.ed. [S.l.]: Pearson Prentice Hall

JAVA – CAP 2022

Classes dos pacotes Java

Classes e Pacotes Java		
java.lang	java.io	java.util
Object	InputStream	Scanner
System	InputStreamReade	Array
String	BufferedReader	ArrayList
Math	File	ArrayDeque
Throwable	Writer	Calendar
Exception		Collections
Runnable		Formatter
Thread		
Wrapper Classes		
Integer		
Byte		
Boolean		
Character		
Float		
Double		

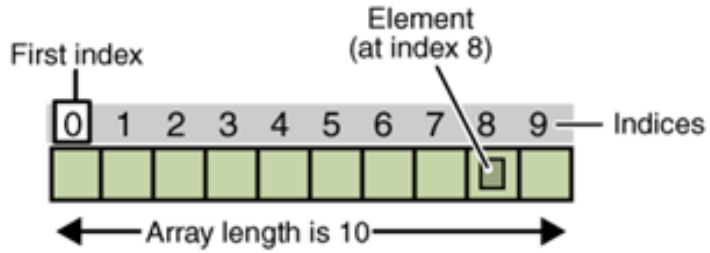
Pacote	Descrição
java.lang	Pacote usual da linguagem
java.util	Utilitários da linguagem
java.io	Pacote de entrada e saída
java.awt	Pacote de interface gráfica
java.net	Pacote de rede
java.sql	Pacote de banco de dados

Pacote	Funcionalidade
java.awt	AWT, ou Abstract Window Toolkit. Contém todas as classes para a criação de aplicações com interfaces gráficas com o usuário (ou GUI, Graphical User Interface).
java.io	Fornece as classes para realizar operações de entrada (input) e saída (output) através do sistema de fluxos de dados (streams) e serialização de objetos.
java.lang	Fornece classes que são fundamentais para a concepção da linguagem de programação Java.
java.net	Fornece as classes para implementar aplicações de rede.
java.util	Para trabalhar com coleções, entrada de dados (Scanner) e componentes de data e hora.
javax.swing	O pacote Swing é um pacote de extensão que complementa o pacote AWT.

Arrays e ArrayList

ARRAYS

Um array é um objeto contêiner que contém um **número fixo de valores de um tipo único**. O comprimento de um array é estabelecido quando o array é criado. Após a criação, seu **comprimento é fixo**.



Cada item em um array é chamado elemento e cada elemento é acessado por seu índice numérico.

Definindo arrays

```
int i[ ] = { 1, 2, 3 };
int i[ ] = new int[3];
String[ ] str = {“um”, “dois”, “três”}
```

ARRAYS

- Objetos array são estruturas de dados consistindo em itens de dados do mesmo tipo relacionados. Arrays tornam conveniente processar grupos relacionados de valores. O comprimento de arrays permanece o mesmo depois que são criados.

- Um array é um grupo de variáveis (chamados elementos ou componentes) que contém valores todos do mesmo tipo. Os arrays são objetos; portanto, são considerados tipos por referência.

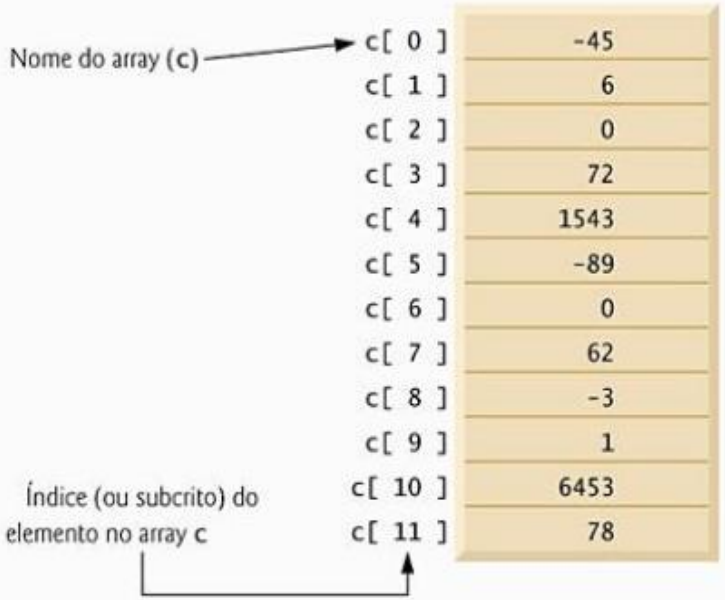
- Em geral consideramos um array na verdade, uma referência a um objeto array na memória. Os elementos de um array podem ser tipos primitivos ou tipos por referência.

- Para referenciar um elemento particular em um array, especificamos o nome da referência para o array e o número de posição do elemento no array. O número de posição do elemento é chamado de índice ou subscrito.

ARRAYS

```
c[ a + b ] += 2;
```

adiciona 2 ao elemento do array `c[11]`. Observe que um nome de array indexado é uma expressão de acesso ao array. Essas expressões podem ser utilizadas no lado esquerdo de uma atribuição para colocar um novo valor em um elemento de array.



ARRAYS

Declarando e criando arrays

Os objetos array ocupam espaço na memória. Como outros objetos, arrays são criados com a palavra-chave new. Para um objeto array, especifique o tipo dos elementos do array e o número de elementos como parte de uma expressão de criação de array que utiliza a palavra-chave new. Tal expressão retorna uma referência que pode ser armazenada em uma variável de array. A declaração e a expressão de criação de arrays a seguir criam um objeto array que contém 12 elementos int e armazenam a referência do array na variável c do array:

```
int[] c = new int[ 12 ];
```

Essa expressão pode ser utilizada para criar o array. Quando um array é criado, cada um de seus elementos recebe um valor padrão — zero para os elementos de tipo primitivo numéricos, false para elementos boolean e null para referências.

```
int[] c; // declara a variável de array
c = new int[12]; // cria o array; atribui à variável de array
```

ARRAYS

Erro comum de programação 7.2

Em uma declaração de array, especificar o número de elementos entre os colchetes da declaração (por exemplo, `int[12] c;`) é um erro de sintaxe.

Um programa pode criar vários arrays em uma única declaração. A declaração seguinte reserva 100 elementos para b e 27 elementos para x:

```
String[] b = new String[ 100 ], x = new String[ 27 ];
```

Quando o tipo do array e colchetes são combinados no início da declaração, todos os identificadores na declaração são variáveis de array. Nesse caso, as variáveis b e x referem-se a arrays `String`. Para maior legibilidade, preferimos declarar apenas uma variável por declaração. A declaração anterior é equivalente a:

```
String[] b = new String[ 100 ]; // cria array b

String[] x = new String[ 27 ]; // cria array x
```

ARRAYS

Quando somente uma variável é declarada em cada declaração, os colchetes podem ser colocados depois do tipo ou depois do nome da variável de array, como em:

```
String b[] = new String[ 100 ]; // cria array b

String x[] = new String[ 27 ]; // cria array x
```

Um programa pode declarar arrays de qualquer tipo. Cada elemento de um array do tipo primitivo contém um valor do tipo de elemento declarado do array. De maneira semelhante, em um array de um tipo por referência, todo elemento é uma referência a um objeto do tipo de elemento declarado do array. Por exemplo, todo elemento de um array `int` é um valor `int` e todo elemento de um array `String` é uma referência a um objeto `String`.

ARRAYS

Criando e Inicializando um Array

```
1  // Figura 7.2: InitArray.java
2  // Inicializando os elementos de um array como valores padrão de zero.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array; // declara o array identificado
9
10       array = new int[ 10 ]; // cria o objeto do array
11
12       System.out.printf( "%s%8s\n", "Index", "Value" ); // títulos de coluna
13
14       // gera saída do valor de cada elemento do array
15       for ( int counter = 0; counter < array.length; counter++ )
16          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17    } // fim de main
18 } // fim da classe InitArray
```

ARRAYS

Criando e Inicializando um Array

```
1  // Figura 7.2: InitArray.java
2  // Inicializando os elementos de um array como valores padrão de zero.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        int[] array; // declara o array identificado
9
10       array = new int[ 10 ]; // cria o objeto do array
11
12       System.out.printf( "%s%8s\n", "Index", "Value" ); // titulos de coluna
13
14       // gera saida do valor de cada elemento do array
15       for ( int counter = 0; counter < array.length; counter++ )
16          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
17    } // fim de main
18 } // fim da classe InitArray
```

Index	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

ARRAYS

Criando e Inicializando um Array

```
1  // Figura 7.3: InitArray.java
2  // Inicializando os elementos de um array com um inicializador de array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // lista de inicializador especifica o valor de cada elemento
9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Index", "Value" ); // titulos de coluna
12
13       // gera saida do valor de cada elemento do array
14       for ( int counter = 0; counter < array.length; counter++ )
15          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16    } // fim de main
17 } // fim da classe InitArray
```

ARRAYS

Criando e Inicializando um Array

```
1  // Figura 7.3: InitArray.java
2  // Inicializando os elementos de um array com um inicializador de a
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        // lista de inicializador especifica o valor de cada elemento
9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf( "%s%8s\n", "Index", "Value" ); // titulos de coluna
12
13       // gera saida do valor de cada elemento do array
14       for ( int counter = 0; counter < array.length; counter++ )
15          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
16    } // fim de main
17 } // fim da classe InitArray
```

Index	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

ARRAYS

Calculando os valores a armazenar em um Array

O aplicativo na Figura 7.4 cria um array de 10 elementos e atribui a cada elemento um dos inteiros pares de 2 a 20 (2, 4, 6, ..., 20). Em seguida, o aplicativo exibe o array em formato tabular. A instrução `for` nas linhas 12–13 calcula o valor de um elemento do array multiplicando o valor atual da variável de controle `counter` por 2 e adicionando 2.

```
1  // Figura 7.4: InitArray.java
2  // Calculando valores a serem colocados em elementos de um array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        final int ARRAY_LENGTH = 10; // declara constante
9        int[] array = new int[ ARRAY_LENGTH ]; // cria o array
10
11       // calcula o valor de cada elemento do array
12       for ( int counter = 0; counter < array.length; counter++ )
13          array[ counter ] = 2 + 2 * counter;
14
15       System.out.printf( "%s%8s\n", "Index", "Value" ); // títulos de coluna
16
17       // gera a saída do valor de cada elemento do array
18       for ( int counter = 0; counter < array.length; counter++ )
19          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20    } // fim de main
21 } // fim da classe InitArray
```

ARRAYS

Calculando os valores a armazenar em um Array

O aplicativo na Figura 7.4 cria um array de 10 elementos e atribui a cada elemento um dos inteiros pares de 2 a 20 (2, 4, 6, ..., 20). Em seguida, o aplicativo exibe o array em formato tabular. A instrução `for` nas linhas 12–13 calcula o valor de um elemento do array multiplicando o valor atual da variável de controle `counter` por 2 e adicionando 2.

```
1  // Figura 7.4: InitArray.java
2  // Calculando valores a serem colocados em elementos de um array.
3
4  public class InitArray
5  {
6     public static void main( String[] args )
7     {
8        final int ARRAY_LENGTH = 10; // declara constante
9        int[] array = new int[ ARRAY_LENGTH ]; // cria o array
10
11       // calcula o valor de cada elemento do array
12       for ( int counter = 0; counter < array.length; counter++ )
13          array[ counter ] = 2 + 2 * counter;
14
15       System.out.printf( "%s%8s\n", "Index", "Value" ); // títulos de coluna
16
17       // gera a saída do valor de cada elemento do array
18       for ( int counter = 0; counter < array.length; counter++ )
19          System.out.printf( "%5d%8d\n", counter, array[ counter ] );
20    } // fim de main
21 } // fim da classe InitArray
```

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

ARRAYS

Somando os valores de um Array

```
1  // Figura 7.5: SumArray.java
2  // Calculando a soma dos elementos de um array.
3
4  public class SumArray
5  {
6     public static void main(String[] args)
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // adiciona o valor de cada elemento ao total
12       for (int counter = 0; counter < array.length; counter++)
13          total += array[counter];
14
15       System.out.printf("Total of array elements: %d%n", total);
16    }
17 } // fim da classe SumArray
```

```
Total of array elements: 849
```

ARRAYS

Somando os valores de um Array

```
1  // Figura 7.5: SumArray.java
2  // Calculando a soma dos elementos de um array.
3
4  public class SumArray
5  {
6     public static void main(String[] args)
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // adiciona o valor de cada elemento ao total
12       for (int counter = 0; counter < array.length; counter++)
13          total += array[counter];
14
15       System.out.printf("Total of array elements: %d%n", total);
16    }
17 } // fim da classe SumArray
```

```
Total of array elements: 849
```

ARRAYS

Tratamento de exceções: processando a resposta incorreta

Uma exceção indica um problema que ocorre quando um programa é executado. O nome “exceção” sugere que o problema ocorre com pouca frequência — se a “regra” é que uma instrução normalmente executa corretamente, então o problema representa a “exceção à regra”. O tratamento de exceção ajuda a criar programas tolerantes a falhas que podem resolver (ou tratar) exceções. Em muitos casos, isso permite que um programa continue a executar como se nenhum problema fosse encontrado.

Problemas mais graves podem evitar que um programa continue executando normalmente, exigindo que ele notifique o usuário sobre o problema e, então, termine. Quando a Java Virtual Machine ou um método detecta um problema, como um índice de array inválido ou um argumento de método inválido, ele lança uma exceção, isto é, ocorre uma exceção. Os métodos nas suas classes também podem lançar exceções

ARRAYS

Tratamento de exceções: processando a resposta incorreta

A instrução try

Para lidar com uma exceção, coloque qualquer código que possa lançar uma exceção em uma instrução try.

O bloco try contém o código que pode lançar uma exceção, e o bloco catch contém o código que trata a exceção se uma ocorrer. Podem haver muitos blocos catch para tratar com diferentes tipos de exceções que podem ser lançadas no bloco try correspondente. As chaves que delimitam o corpo dos blocos try e catch são obrigatórias.

ARRAYS

Tratamento de exceções: processando a resposta incorreta

Executando o bloco catch

Quando o programa encontra o valor inválido 14 no array responses, ele tenta adicionar 1 a frequency[14], que está fora dos limites do array — o array frequency tem apenas seis elementos (com índices de 0 a 5). Como a verificação dos limites de array é executada em tempo de execução, a JVM gera uma exceção — especificamente a linha 19 lança ArrayIndexOutOfBoundsException para notificar o programa sobre esse problema. Nesse ponto, o bloco try termina e o bloco catch começa a executar — se você declarou quaisquer variáveis locais no bloco try, agora elas estarão fora do escopo (e não mais existirão), assim elas não estarão acessíveis no bloco catch. O bloco catch declara um parâmetro de exceção (e) do tipo IndexOutOfRangeException. O bloco catch pode lidar com exceções do tipo especificado. Dentro do bloco catch, você pode usar o identificador do parâmetro para interagir com um objeto que capturou a exceção.

ARRAYS

A instrução for aprimorada

A instrução for aprimorada itera pelos elementos de um array sem usar um contador, evitando assim a possibilidade de "pisar fora" do array.

```
for (parâmetro : nomeDoArray)
   instrução
```

```
for (int counter = 0; counter < array.length; counter++)
   total += array[counter];
```

ARRAYS

A instrução for aprimorada

```
1  // Figura 7.12: EnhancedForTest.java
2  // Utilizando a instrução for aprimorada para somar inteiros em um array.
3
4  public class EnhancedForTest
5  {
6     public static void main(String[] args)
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // adiciona o valor de cada elemento ao total
12       for (int number : array)
13          total += number;
14
15       System.out.printf("Total of array elements: %d%n", total);
16    }
17 } // fim da classe EnhancedForTest
```

```
Total of array elements: 849
```

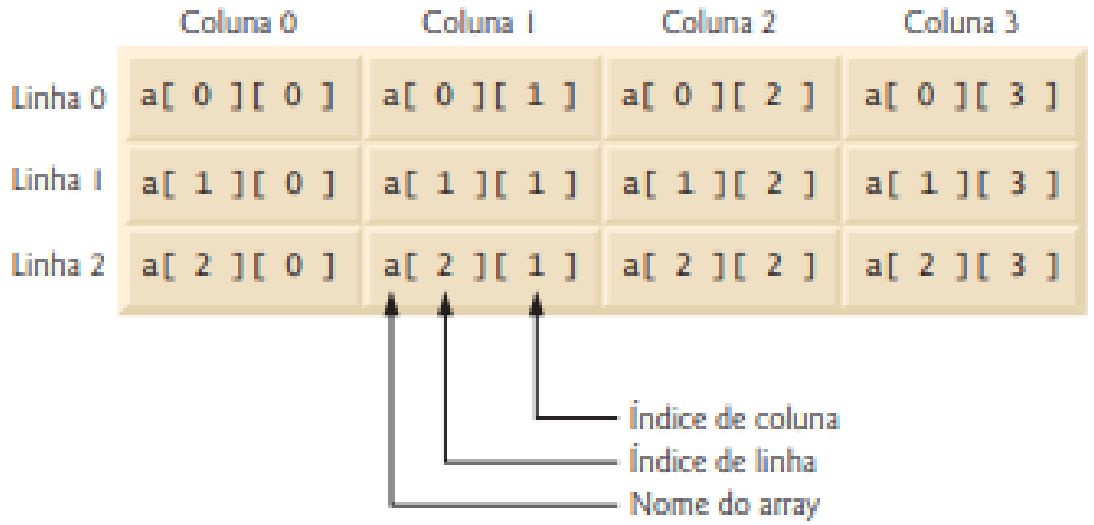
ARRAYS

ARRAYS MULTIDIMENSIONAIS

Arrays multidimensionais com duas dimensões muitas vezes são utilizados para representar tabelas de valores com os dados organizados em linhas e colunas. Para identificar um determinado elemento de tabela, você especifica dois índices. Por convenção, o primeiro identifica a linha do elemento e o segundo, sua coluna. Os arrays que requerem dois índices para identificar cada elemento particular são chamados arrays bidimensionais. (Os arrays multidimensionais podem ter mais de duas dimensões.) O Java não suporta arrays multidimensionais diretamente, mas permite especificar arrays unidimensionais cujos elementos também são arrays unidimensionais, alcançando assim o mesmo efeito. Em geral, um array com m linhas e n colunas é chamado de array **m por n.**

ARRAYS

ARRAYS MULTIDIMENSIONAIS



ARRAYS

ARRAYS MULTIDIMENSIONAIS

Arrays de arrays unidimensionais

Como os arrays unidimensionais, os arrays multidimensionais podem ser inicializados com inicializadores de array em declarações. Um array bidimensional `b` com duas linhas e duas colunas poderia ser declarado e inicializado com **inicializadores de array aninhados** como a seguir:

```
int[][] b = {{1, 2}, {3, 4}};
```

Os valores iniciais são *agrupados por linha* entre chaves. Então, `1` e `2` inicializam `b[0][0]` e `b[0][1]`, respectivamente, e `3` e `4` inicializam `b[1][0]` e `b[1][1]`, respectivamente. O compilador conta o número de inicializadores de array aninhados (representado por conjuntos de chaves dentro das chaves externas) para determinar o número de *linhas* no array `b`. O compilador conta os valores inicializadores no inicializador aninhado de array para uma linha determinar o número de *colunas* nessa linha. Como veremos em breve, isso significa que as *linhas podem ter diferentes comprimentos*.

Os arrays multidimensionais são mantidos como *arrays de arrays unidimensionais*. Portanto, o array `b` na declaração anterior é na realidade composto de dois arrays unidimensionais separados — um que contém o valor na primeira lista de inicializadores aninhados `{1, 2}` e outro que contém o valor na segunda lista de inicializadores aninhados `{3, 4}`. Portanto, o próprio array `b` é um array de dois elementos, cada um desses elementos é um array unidimensional de valores `int`.

ARRAYS

ARRAYS MULTIDIMENSIONAIS

Arrays bidimensionais com linhas de diferentes comprimentos

A maneira como os arrays multidimensionais são representados os torna bem flexíveis. De fato, os comprimentos das linhas no array `b` *não* precisam ser os mesmos. Por exemplo,

```
int[][] b = {{1, 2}, {3, 4, 5}};
```

cria array de inteiros `b` com dois elementos (determinados pelo número de inicializadores de array aninhados) que representam as linhas do array bidimensional. Cada elemento de `b` é uma *referência* a um array unidimensional de variáveis `int`. O array `int` da linha `0` é um array unidimensional com *dois* elementos (`1` e `2`), e o array `int` da linha `1` é um array unidimensional com *três* elementos (`3`, `4` e `5`).

ARRAYS

ARRAYS MULTIDIMENSIONAIS

Criando arrays bidimensionais com expressões de criação de arrays

Um array multidimensional com o *mesmo* número de colunas em cada linha pode ser criado com uma expressão de criação de array. Por exemplo, as linhas a seguir declaram o array `b` e atribuem a ele uma referência para um array três por quatro:

```
int[][] b = new int[3][4];
```

Nesse caso, utilizamos os valores literais `3` e `4` para especificar o número de linhas e o número de colunas, respectivamente, mas isso *não* é necessário. Programas também podem utilizar variáveis para especificar as dimensões do array, porque *new cria arrays em tempo de execução, não em tempo de compilação.* Os elementos de um array multidimensional são inicializados quando o objeto array é criado.

Pode-se criar um array multidimensional em que cada linha tem um número *diferente* de colunas, como mostrado a seguir:

```
int[][] b = new int[2][];   // cria 2 linhas
b[0] = new int[5]; // cria 5 colunas para a linha 0
b[1] = new int[3]; // cria 3 colunas para a linha 1
```

As instruções anteriores criam um array bidimensional com duas linhas. A linha `0` tem *cinco* colunas, e a linha `1`, *três* colunas.

ARRAYS

ARRAYS MULTIDIMENSIONAIS

```
1  // Figura 7.17: InitArray.java
2  // Inicializando arrays bidimensionais.
3
4  public class InitArray
5  {
6     // cria e gera saída de arrays bidimensionais
7     public static void main( String[] args )
8     {
9        int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Values in array1 by row are" );
13       outputArray( array1 ); // exibe array1 por linha
14
15       System.out.println( "\nValues in array2 by row are" );
16       outputArray( array2 ); // exibe array2 por linha
17    } // fim de main
18
19    // gera saída de linhas e colunas de um array bidimensional
20    public static void outputArray(int[][] array )
21    {
22       // faz um loop pelas linhas do array
23       for ( int row = 0; row < array.length; row++ )
24       {
25          // faz um loop pelas colunas da linha atual
26          for ( int column = 0; column < array[ row ].length; column++ )
27             System.out.printf( "%d  ", array[ row ][ column ] );
28
29          System.out.println(); // inicia nova linha de saída
30       } // fim do for externo
31    } // fim do método outputArray
32 } // fim da classe InitArray
```

JAVA – CAP 2022

ARRAYS

ARRAYS MULTIDIMENSIONAIS

```
1  // Figura 7.17: InitArray.java
2  // Inicializando arrays bidimensionais.
3
4  public class InitArray
5  {
6     // cria e gera saída de arrays bidimensionais
7     public static void main( String[] args )
8     {
9        int[][] array1 = { { 1, 2, 3 }, { 4, 5, 6 } };
10       int[][] array2 = { { 1, 2 }, { 3 }, { 4, 5, 6 } };
11
12       System.out.println( "Values in array1 by row are" );
13       outputArray( array1 ); // exibe array1 por linha
14
15       System.out.println( "\nValues in array2 by row are" );
16       outputArray( array2 ); // exibe array2 por linha
17    } // fim de main
18
19    // gera saída de linhas e colunas de um array bidimensional
20    public static void outputArray(int[][] array )
21    {
22       // faz um loop pelas linhas do array
23       for ( int row = 0; row < array.length; row++ )
24       {
25          // faz um loop pelas colunas da linha atual
26          for ( int column = 0; column < array[ row ].length; column++ )
27             System.out.printf( "%d  ", array[ row ][ column ] );
28
29          System.out.println(); // inicia nova linha de saída
30       } // fim do for externo
31    } // fim do método outputArray
32 } // fim da classe InitArray
```

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

ARRAYS

ARRAYS MULTIDIMENSIONAIS

A seguinte instrução `for` aninhada soma os valores de todos os elementos no array a:

```
int total = 0;

for (int row = 0; row < a.length; row++)
{
   for (int column = 0; column < a[row].length; column++)
      total += a[row][column];
}
```

ARRAYS

A classe Arrays

A classe Arrays ajuda a evitar reinventar a roda fornecendo métodos static para manipulações de array comuns. Esses métodos incluem sort para classificar um array (isto é, organizar os elementos em ordem crescente), binarySearch para procurar um array classificado (isto é, determinar se um array contém um valor específico e, se contiver, onde o valor está localizado), equals para comparar arrays e fill para inserir valores em um array. Esses métodos são sobrecarregados para arrays de tipo primitivo e arrays de objetos.

```
1  // Figura 7.22: ArrayManipulations.java
2  // Métodos da classe Arrays e System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations
6  {
7     public static void main( String[] args )
8     {
9        // classifica doubleArray em ordem crescente
10       double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort( doubleArray );
12       System.out.printf( "\ndoubleArray: " );
13
```

ARRAYS

A classe Arrays

```
14          for ( double value : doubleArray )
15             System.out.printf( "%.1f ", value );
16
17          // preenche o array de 10 elementos com 7s
18          int[] filledIntArray = new int[ 10 ];
19          Arrays.fill( filledIntArray, 7 );
20          displayArray( filledIntArray, "filledIntArray" );
21
22          // copia array intArray em array intArrayCopy
23          int[] intArray = { 1, 2, 3, 4, 5, 6 };
24          int[] intArrayCopy = new int[ intArray.length ];
25          System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length );
26          displayArray( intArray, "intArray" );
27          displayArray( intArrayCopy, "intArrayCopy" );
28
29          // compara a igualdade de intArray e intArrayCopy
30          boolean b = Arrays.equals( intArray, intArrayCopy );
31          System.out.printf( "\n\nintArray %s intArrayCopy\n",
32             ( b ? "==" : "!=" ) );
33
34          // compara a igualdade de intArray e filledIntArray
35          b = Arrays.equals( intArray, filledIntArray );
36          System.out.printf( "intArray %s filledIntArray\n",
37             ( b ? "==" : "!=" ) );
38
39          // pesquisa em intArray o valor 5
40          int location = Arrays.binarySearch( intArray, 5 );
```

ARRAYS

A classe Arrays

```
41
42          if ( location >= 0 )
43             System.out.printf(
44                "Found 5 at element %d in intArray\n", location );
45          else
46             System.out.println( "5 not found in intArray" );
47
48          // pesquisa em intArray o valor 8763
49          location = Arrays.binarySearch( intArray, 8763 );
50
51          if ( location >= 0 )
52             System.out.printf(
53                "Found 8763 at element %d in intArray\n", location );
54          else
55             System.out.println( "8763 not found in intArray" );
56       } // fim de main
57
58       // gera saída de valores em cada array
59       public static void displayArray( int[] array, String description )
60       {
61          System.out.printf( "\n%s: ", description );
62
63          for ( int value : array )
64             System.out.printf( "%d ", value );
65       } // fim do método displayArray
66    } // fim da classe ArrayManipulations
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

Fill() - preenche com um intervalo
arrayCopy() - Copia os elementos de um Array
binarySearch() - realiza pesquisa binária
Equals() - faz a comparação dos elementos

ARRAYS

A classe ArrayList

A Java API fornece várias estruturas de dados predefinidas, chamadas coleções, usadas para armazenar grupos de objetos relacionados na memória. Essas classes fornecem métodos eficientes que organizam, armazenam e recuperam seus dados sem a necessidade de conhecer como os dados são armazenados. Isso reduz o tempo de desenvolvimento de aplicativos. Você já usou arrays para armazenar sequências de objetos. Arrays não mudam automaticamente o tamanho em tempo de execução para acomodar elementos adicionais. A classe de coleção ArrayList (pacote java.util) fornece uma solução conveniente para esse problema — ela pode alterar dinamicamente seu tamanho para acomodar mais elementos. O T (por convenção) é um espaço reservado — ao declarar um novo ArrayList, substitua-o pelo tipo dos elementos que você deseja que o ArrayList armazene.

ARRAYS

A classe ArrayList

Por exemplo:

```
ArrayList<String> list;
```

declara list como uma coleção ArrayList que só pode armazenar Strings. Classes com esse tipo de espaço reservado que podem ser usadas com qualquer tipo são chamadas classes genéricas. Somente tipos não primitivos podem ser usados para declarar variáveis e criar objetos das classes genéricas. Mas o Java fornece um mecanismo conhecido como boxing, que permite que valores primitivos sejam empacotados como objetos para uso com classes genéricas

Por exemplo:

```
ArrayList<Integer> integers;
```

declara integers como um ArrayList que só pode armazenar Integers. Ao inserir um valor int em um ArrayList, o valor int é empacotado como um objeto Integer, e quando você obtém um objeto Integer de um ArrayList, e então atribui o objeto a uma variável int, o valor int dentro do objeto é desempacotado.

ARRAYS

A classe ArrayList

Método	Descrição
`add`	Adiciona um elemento ao *final* do `ArrayList`.
`clear`	Remove todos os elementos do `ArrayList`.
`contains`	Retorna `true` se o `ArrayList` contém o elemento especificado; caso contrário, retorna `false`.
`get`	Retorna o elemento no índice especificado.
`indexOf`	Retorna o índice da primeira ocorrência do elemento especificado no `ArrayList`.
`remove`	Sobrecarregado. Remove a primeira ocorrência do valor especificado ou o elemento no índice especificado.
`Size`	Retorna o número de elementos armazenados em `ArrayList`.
`trimToSize`	Corta a capacidade do `ArrayList` para o número atual de elementos.

Figura 7.23 | Alguns métodos e propriedades da classe `ArrayList<T>`.

JAVA – CAP 2022

Estatística de questões de JAVA

CAP		50 QUESTÕES
2011	9	18,00%
2012	13	26,00%
2013	9	18,00%
2014	5	10,00%
2015	7	14,00%
2016	6	12,00%
2017	9	18,00%
2018	7	14,00%
2019	9	18,00%
2020	6	12,00%

2021	5	10,00%

JAVA – CAP 2022

QUESTÕES

1) Em relação ao uso de arrays na linguagem Java, avalie as afirmativas a seguir.

I - Um array é um grupo de variáveis que contém valores todos do mesmo tipo.

II - O primeiro elemento em cada array tem um índice um.

III - Um arraylist é semelhante a um array, mas pode ser dinamicamente redimensionado.

Das afirmativas acima, apenas:

A)I está correta.
B)II está correta.
C)III está correta.
D)I e III estão corretas.
E) Nenhuma alternativa está correta

QUESTÕES

2) Sobre a linguagem Java, marque V para as afirmativas verdadeiras e F para as falsas.

( ) O Java é uma linguagem fortemente tipada, ele requer que todas as variáveis tenham um tipo.
( ) Uma declaração do método final nunca pode mudar; assim, todas as subclasses utilizam a mesma implementação do método.
( ) Um método final em uma superclasse pode ser sobrescrito como uma subclasse, garantindo que a implementação do método final será utilizada por todas as subclasses diretas e indiretas na hierarquia.
( ) Os operadores aritméticos *, /, %, + e - têm, todos, o mesmo nível de precedência.
( ) Métodos que são declarados *private* são implicitamente final, porque não é possível sobrescrevê-los como uma subclasse.

A sequência está correta em
Alternativas
A) F, V, F, V, F.
B) V, F, V, F, V.
C) F, F, V, V, F.
D) V, V, V, V,F.
E) V, V, F, F, V.

JAVA – CAP 2022

QUESTÕES

3) Com relação à linguagem de programação JAVA, coloque F (falso) ou V (verdadeiro) nas afirmativas abaixo e assinale a opção que apresenta a sequência correta.

( ) Um array pode armazenar muitos tipos de valores diferentes.

( ) Um índice de array deve ser normalmente do tipo float.

( ) Argumentos de linha de comando são separados por vírgula

( ) A instrução “for” aprimorada permite aos programadores iterar pelos elementos em um array sem utilizar um contador.

A) (F) (F) (F) (V)

B) (F) (F) (V) (V)

C) (F) (V) (V) (V)

D) (V) (F) (F) (F)

E) (F) (V) (F) (V)

QUESTÕES

Questões EAOP 2022/2023

44) Analise os trechos de código A e B implementados em linguagem Java e, em seguida, marque a opção correta.

A	B

for (int i = 0; i <= 10; i++) {
  if (i % 2 != 0)
  continue;
  System.out.println(i);
}
	int i = 0;
	while (i <= 10) {
	  if (i % 2 != 0)
	  continue;
	  System.out.println(i);
	  i++;
	}

a) Os trechos de código A e B imprimem os mesmos valores na tela sem provocar loop infinito.
b) O trecho de código A provoca um loop infinito.
c) O trecho de código B provoca um loop infinito.
d) Os trechos de código A e B provocam um loop infinito.

JAVA – CAP 2022

QUESTÕES

CAP - 2011

5) Analise o código em JAVA a seguir.

```
package prova;

public class Main {

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

Sabendo-se que o código acima foi escrito e executado utilizando o IDE NetBeans 6.0.1, assinale a opção correta referente ao valor da variável total que será impresso, considerando o array c[] aumentado para 13.

(A) 153
(B) 154
(C) 155
(D) 156
(E) 157

JAVA – CAP 2022

QUESTÕES

CAP - 2012

28) Com relação ao tratamento de variáveis na linguagem JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package p1;
public class P1 { // NetBeans IDE 7.2.1

    public static void main(String[] args) {
        final int ARRAY_LENGTH = 3;
        int array[] = new int[ARRAY_LENGTH];

        int t=0;

        for (int counter = 0; counter < array.length; counter++)
        {
             array[counter] = 2 + 2 * counter;
        }

        System.out.printf("%s%8s\n","Index", "Value");
        for (int counter = 0; counter < array.length; counter++) {
        System.out.printf("%s%8s\n", counter, array[counter]);

        t = t + counter + array[counter];
        System.out.printf("%s%8s\n","Somatório ", t);
    }
  }
}
```

Dentre as opções abaixo, qual apresenta o valor impresso da variável "t" ao final da execução do programa?

(A) 11
(B) 12
(C) 13
(D) 14
(E) 15

JAVA – CAP 2022

QUESTÕES

CAP - 2012

37) Considerando as estruturas de controle na linguagem de programação JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package p7;
public class P7 {
    public static void main(String[] args) {

        int [] string = {8,8,8};
        int total = 0;
        for (int number:string)
            total += number;
            System.out.println(total);
    }
}
```

Dentre as opções abaixo, qual apresenta o valor que será impresso por esse programa ao final de sua execução?

(A) 888
(B) 88
(C) 32
(D) 24
(E) 16

JAVA – CAP 2022

QUESTÕES

CAP - 2012

38) Considerando a sintaxe da linguagem JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package cap9;

import java.lang.*;
public class CAP9 {

public static void main(String[] args) {

        int [] d = {4};
        int a=1;
        int b=2;
        int c=3;
        System.out.print(a+b+c+d[0]);
    }
}
```

Assinale a opção que contém uma instrução que pode ser omitida sem afetar a execução do programa.

(A) int c=3;
(B) int b=2;
(C) int a=1;
(D) int [] d = {4};
(E) import java.lang.*;

JAVA – CAP 2022

QUESTÕES

CAP - 2012

39) Analise o programa em JAVA abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package cap8;
public class CAP8 {

    public static void main(String[] args) {

        int j=1;
        int t=1;

        int [] x = {1,2};
        for (int i=0; i<x.length; i++) { t += x[i] = x[i] * j++; }

        System.out.print(t);
    }
}
```

O que será impresso por esse programa?

(A) 10
(B) 9
(C) 8
(D) 7
(E) 6

JAVA – CAP 2022

QUESTÕES

CAP - 2012

42) Com relação a funções e procedimentos na linguagem JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package p2;
public class P2 {
    public static void main(String[] args) {

        P2 f = new P2();
        f.d(10);
    }
    public static long f( long n) {
        if (n <=1) {
            return 1;
        }
        else {
            return n * f ( n-1);
        }
    }
    public void  d(long n) {
        for (int c=0; c <= n; c++ ) {
            System.out.printf("%d! = %d\n", c, f(c));
        }
}
}
```

Dentre as opções abaixo, qual apresenta o valor da variável "c" quando f(c)=24?

(A) 4
(B) 5
(C) 8
(D) 10
(E) 11

JAVA – CAP 2022

QUESTÕES

CAP - 2012

43) Considerando a construção de algoritmos no que tange às estruturas básicas de controle, analise o programa em JAVA abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package p6;
public class P6 { // NetBeans IDE 7.2.1 || D6 1018
    public static void main(String[] args) {
        int c=20;
        while (c <=20 )
        {
            ++c;
        }
        System.out.println(c);
    }
}
```

Dentre as opções abaixo, qual apresenta o valor que será impresso por esse programa ao final de sua execução?

(A) 22
(B) 21
(C) 20
(D) 19
(E) 10

JAVA – CAP 2022

QUESTÕES

CAP - 2012

44) Considerando a definição e utilização de matrizes em JAVA, analise o programa abaixo, desenvolvido no ambiente NetBeans 7.2.1:

```
package cap4;
public class CAP4 {

    public static void main(String[] args) {
            int [][]matriz = new int[2][2];
            int t = 0;

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

Dentre as opções abaixo, qual apresenta o valor que será impresso por esse programa ao final de sua execução?

(A) 12347
(B) 11248
(C) 11237
(D) 01236
(E) 01124

Métodos

A experiência mostrou que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenos e simples pedaços, ou módulos. Essa técnica é chamada dividir para conquistar. Os métodos ajudam a modularizar programas.

Módulos de programa em Java

Você escreve programas Java combinando novos métodos e classes com aqueles predefinidos disponíveis na Java Application Programming Interface (também chamada Java API ou biblioteca de classes Java) e em várias outras bibliotecas de classes. Classes relacionadas são agrupadas em pacotes de modo que possam ser importadas nos programas e reutilizadas.

Métodos

Dividir para conquistar com classes e métodos Classes e métodos ajudam a modularizar um programa separando suas tarefas em unidades autocontidas. As instruções no corpo dos métodos são escritas apenas uma vez, permanecem ocultas de outros métodos e podem ser reutilizadas a partir de várias localizações em um programa. Uma motivação para modularizar um programa em métodos e classes é a abordagem dividir para conquistar, que torna o desenvolvimento de programas mais gerenciável, construindo programas a partir de peças mais simples e menores. Outra é a capacidade de reutilização de software — o uso de classes e métodos existentes como blocos de construção para criar novos programas.

Métodos

Relação hierárquica entre chamadas de método Como você sabe, um método é invocado por uma chamada de método, e quando o método chamado termina sua tarefa, ele retorna o controle e possivelmente um resultado para o chamador. Uma analogia a essa estrutura de programa é a forma hierárquica de gerenciamento (Figura 6.1). Um chefe (o chamador) solicita que um trabalhador (o método chamado) realize uma tarefa e informe (retorne) os resultados depois de completar a tarefa. O método chefe não tem conhecimento sobre como o método trabalhador realiza suas tarefas designadas. O trabalhador também pode chamar outros métodos trabalhadores, sem que o chefe saiba. Esse “ocultamento” dos detalhes de implementação promove a boa engenharia de software. A Figura 6.1 mostra o método chefe se comunicando com vários métodos trabalhadores de uma maneira hierárquica. O método chefe divide as responsabilidades entre os vários métodos trabalhadores. Aqui, trabalhador1 atua como um "método chefe" para trabalhador4 e trabalhador5.

Figura 6.1 | Relacionamento hierárquico de método trabalhador/método chefe.

Métodos

A classe Math e seus métodos

```
Math.sqrt(900.0)
```

```
System.out.println(Math.sqrt(900.0));
```

Os argumentos de método podem ser constantes, variáveis ou expressões. Se `c = 13.0`, `d = 3.0` e `f = 4.0`, então a instrução

```
System.out.println(Math.sqrt(c + d * f));
```

calcula e imprime a raiz quadrada de `13.0 + 3.0 * 4.0 = 25.0` — a saber `5.0`. A Figura 6.2 resume vários métodos da classe `Math`. Na figura, *x* e *y* são do tipo `double`.

Métodos

A classe Math e seus métodos

Método	Descrição	Exemplo

`abs(x)`	valor absoluto de *x*	`abs(23.7)` é `23.7`
		`abs(0.0)` é `0.0`
		`abs(-23.7)` é `23.7`

`ceil(x)`	arredonda *x* para o menor inteiro não menor que *x*	`ceil(9.2)` é `10.0` `ceil(-9.8)` é `-9.0`
`cos(x)`	cosseno trigonométrico de *x* (*x* em radianos)	`cos(0.0)` é `1.0`
`exp(x)`	método exponencial e^x	`exp(1.0)` é `2.71828` `exp(2.0)` é `7.38906`
`floor(x)`	arredonda *x* para o maior inteiro não maior que *x*	`floor(9.2)` é `9.0` `floor(-9.8)` é `-10.0`
`log(x)`	logaritmo natural de *x* (base *e*)	`log(Math.E)` é `1.0` `log(Math.E* Math.E)` é `2.0`
`max(x,y)`	maior valor de *x* e *y*	`max(2.3, 12.7)` é `12.7` `max(-2.3, -12.7)` é `-2.3`

Método	Descrição	Exemplo
`min(x,y)`	menor valor de *x* e *y*	`min(2.3, 12.7)` é `2.3` `min(-2.3, -12.7)` é `-12.7`
`pow(x,y)`	*x* elevado à potência de *y* (isto é, x^y)	`pow(2.0, 7.0)` é `128.0` `pow(9.0, 0.5)` é `3.0`
`sin(x)`	seno trigonométrico de *x* (*x* em radianos)	`sin(0.0)` é `0.0`
`sqrt(x)`	raiz quadrada de *x*	`sqrt(900.0)` é `30.0`
`tan(x)`	tangente trigonométrica de *x* (*x* em radianos)	`tan(0.0)` é `0.0`

Métodos

continuação

```
5  public class MaximumFinder
6  {
7     // obtém três valores de ponto flutuante e localiza o valor máximo
8     public static void main(String[] args)
9     {
10       // cria Scanner para entrada a partir da janela de comando
11       Scanner input = new Scanner(System.in);
12
13       // solicita e insere três valores de ponto flutuante
14       System.out.print(
15          "Enter three floating-point values separated by spaces: ");
16       double number1 = input.nextDouble(); // lê o primeiro double
17       double number2 = input.nextDouble(); // lê o segundo double
18       double number3 = input.nextDouble(); // lê o terceiro double
19
20       // determina o valor máximo
21       double result = maximum(number1, number2, number3);
22
23       // exibe o valor máximo
24       System.out.println("Maximum is: " + result);
25    }
26
27    // retorna o máximo dos seus três parâmetros de double
28    public static double maximum(double x, double y, double z)
29    {
30       double maximumValue = x; // supõe que x é o maior valor inicial
31
32       // determina se y é maior que maximumValue
33       if (y > maximumValue)
34          maximumValue = y;
35
36       // determina se z é maior que maximumValue
37       if (z > maximumValue)
38          maximumValue = z;
39
40       return maximumValue;
41    }
42 } // fim da classe MaximumFinder
```

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Figura 6.3 | O método declarado pelo programador `maximum` com três parâmetros `double`.

JAVA – CAP 2022

API JAVA

Pacote	Descrição
java.awt.event	O **Java Abstract Window Toolkit Event Package** contém classes e interfaces que permitem tratamento de evento para componentes GUI em nos pacotes `java.awt` e `javax.swing` (Veja o Capítulo 12, "Componentes GUI: parte 1", e o Capítulo 22, "Componentes GUI: parte 2".)
java.awt.geom	O **Java 2D Shapes Package** contém classes e interfaces para trabalhar com as avançadas capacidades gráficas bidimensionais do Java. (Veja o Capítulo 13, "Imagens gráficas e Java 2D".)
java.io	O **Java Input/Output Package** contém classes e interfaces que permitem aos programas gerar entrada e saída de dados. (Veja o Capítulo 15, "Arquivos, fluxos e serialização de objetos".)
java.lang	O **Java Language Package** contém classes e interfaces (discutidas por todo este texto) que são exigidas por muitos programas Java. Esse pacote é importado pelo compilador em todos os programas.

Pacote	Descrição
java.net	O **Java Networking Package** contém classes e interfaces que permitem aos programas comunicar-se via redes de computadores, como a internet. (Veja o Capítulo 28, em inglês, na Sala Virtual do livro.)
java.security	O **Java Security Package** contém classes e interfaces para melhorar a segurança do aplicativo.
java.sql	O **JDBC Package** contém classes e interfaces para trabalhar com bancos de dados. (Veja o Capítulo 24, "Acesso a bancos de dados com JDBC".)
java.util	O **Java Utilities Package** contém classes e interfaces utilitárias que permitem armazenar e processar grandes quantidades de dados. Muitas dessas classes e interfaces foram atualizadas para suportar novos recursos lambda do Java SE 8. (Veja o Capítulo 16, "Coleções genéricas".)
java.util.concurrent	O **Java Concurrency Package** contém classes utilitárias e interfaces para implementar programas que podem realizar múltiplas tarefas paralelamente. (Veja o Capítulo 23, "Concorrência".)
javax.swing	O **Java Swing GUI Components Package** contém classes e interfaces para componentes GUI Swing do Java que fornecem suporte para GUIs portáveis. Esse pacote ainda usa alguns elementos do pacote `java.awt` mais antigo. (Veja o Capítulo 12, "Componentes GUI: parte 1" e o Capítulo 22, "Componentes GUI: parte 2".)
javax.swing.event	O **Java Swing Event Package** contém classes e interfaces que permitem o tratamento de eventos (por exemplo, responder a cliques de botão) para componentes GUI do pacote `javax.swing`. (Veja o Capítulo 12, "Componentes GUI: parte 1" e o Capítulo 22, "Componentes GUI: parte 2".)
javax.xml.ws	O **JAX-WS Package** contém classes e interfaces para trabalhar com serviços da web no Java. (Consulte o Capítulo 32, em inglês, na Sala Virtual.)
pacotes `javafx`	JavaFX é a tecnologia da GUI preferida para o futuro. Discutiremos esses pacotes no Capítulo 25, "GUI do JavaFX: parte 1" e nos capítulos "JavaFX GUI" e "Multimídia", em inglês, na Sala Virtual.
Alguns pacotes Java SE 8 usados neste livro	
java.time	O novo **pacote** Java SE 8 **Date/Time API** contém classes e interfaces para trabalhar com datas e horas. Esses recursos são projetados para substituir as capacidades de data e hora mais antigas do pacote `java.util`. (Veja o Capítulo 23, "Concorrência".)
java.util.function e java.util.stream	Esses pacotes contêm classes e interfaces para trabalhar com as capacidades de programação funcional do Java SE 8. (Veja o Capítulo 17, "Lambdas e fluxos Java SE 8".)

Figura 6.5 | Pacotes da Java API (um subconjunto).

Strings, Caracteres especiais e Expressões Regulares

Strings, Caracteres especiais e Expressões Regulares

Strings (Classe - java.lang.String)

Uma String é uma **seqüência de caracteres**. Na linguagem de programação Java, strings são objetos.

A plataforma Java fornece a classe String para criar e manipular strings.

Criando Strings

A maneira mais comum de criar uma String é como no exemplo a seguir:

```
String saudacao = "Olá mundo!";
```

Neste caso, "Olá mundo!" é uma string literal, uma série de caracteres envolvida por aspas. Sempre que encontrar uma string literal em seu código, o compilador cria um objeto String com o seu valor, neste caso, “Olá Mundo!”.

Como acontece com qualquer outro objeto, você pode criar objetos String usando a palavra-chave new e um construtor. A classe String tem treze construtores que permitem a você fornecer o valor inicial da seqüência utilizando diferentes fontes, como uma matriz de caracteres:

```
char [] aloArray = {'A', 'l', 'o', ' ', 'M', 'u', 'n', 'd', 'o','.'};

String alo = new String (aloArray);
System.out.println (alo);
```

A última linha do trecho de código irá exibir “Alo Mundo.”

Strings, Caracteres especiais e Expressões Regulares

Nota: A classe String é **imutável**, de modo que, uma vez que é criado um objeto String não pode ser alterado. A classe String tem um número de métodos, alguns dos quais serão discutidos a seguir, que aparecem para modificar strings. Como as strings são imutáveis, o que esses métodos realmente podem fazer é criar e retornar uma nova seqüência de caracteres que contém o resultado da operação.

String Length

Os métodos usados para obter informações sobre um objeto são conhecidos como *métodos de acesso.* Um método de acesso que você pode usar com strings é o **length()** que retorna o número de caracteres contidos no objeto string. Após as duas seguintes linhas de código serem executadas, len será igual a 12:

```
String exemplo = "CURSO ADONAI";
int len = exemplo.length ();
```

Strings, Caracteres especiais e Expressões Regulares

Concatenando Strings

A classe String inclui um método para concatenar duas strings:
string1.concat(string2);

Isso retorna uma nova string que é string1 com string2 adicionado a ela no final.
Você também pode usar o **concat()** método com strings literais, como no exemplo abaixo:
"Meu nome é ".concat("Alexandre Eugênio");
Strings são mais comumente concatenadas com o operador + , desta forma:
"Olá," + "mundo" + "!"

o que resulta em

"Olá, mundo!"

O operador + é amplamente utilizado em declarações print:

String string1 = "ADONAI";
System.out.println("CURSO " + string1);

Isto irá imprimir

CURSO ADONAI

Nota: A linguagem de programação Java não permite que strings literais possuam espaços entre às linhas em arquivos fonte, assim você deve usar o operador de concatenação + no final de cada linha em uma string com multiplas linhas.

```
String quote = "Agora é a hora boa para todos" +
      "homens de vir em auxílio do seu país."
```

Essa concatenação pode ser combinada com outros tipos. Para cada tipo que não é uma String o método **toString()** deste será chamado para representar sua saída na forma de uma string.

Strings, Caracteres especiais e Expressões Regulares

Criando Strings formatadas

A classe java.io.PrintStream possui os métodos printf() e format() utilizados para gerar uma saída formatada.

A classe String tem um método equivalente format() que retorna uma String objeto em vez de um objeto PrintStream
O método estático format() da classe String permite que você crie uma string formatada que você pode reutilizar, ao invés de utilizar apenas a impressão.

```
System.out.printf("O valor da variável é float %f, enquanto o valor da" +
                  "variável inteiro é %d e a string é%s", floatvar, intVar, stringVar);

String fs = String.format("O valor da variável é float %f, enquanto o valor da variável inteiro é %d e a string é %s", floatvar, intVar, stringVar);
```

Manipulação de caracteres

A classe String possui vários métodos para examinar o conteúdo de seqüências de caracteres, caracteres ou substrings achados dentro de uma seqüência, caso de mudança, e outras tarefas.

Strings, Caracteres especiais e Expressões Regulares

Retornando caracteres e Substrings pelo índice

Você pode obter o caractere em um índice específico dentro de uma seqüência, invocando o método charAt(). O índice do primeiro caractere é 0, enquanto o índice do último caractere é length()-1

Por exemplo, o código a seguir obtém o caractere no índice 9, em uma seqüência:

String frase = "CURSO ADONAI";

System.out.print(frase.charAt(9)); // N

O método substring possui duas versões, como mostrado na tabela a seguir:

Os Métodos substring da classe String

Método	Descrição
String substring(int beginIndex, int endIndex)	Retorna uma nova string que é uma subseqüência desta cadeia. O primeiro argumento inteiro especifica o índice do primeiro caractere. O segundo argumento inteiro é o índice do último caractere + 1.
String substring(int beginIndex)	Retorna uma nova string que é uma subseqüência desta cadeia. O argumento inteiro especifica o índice do primeiro caractere. Aqui, o substring retornada se estende até o final da string original.

Exemplo: String str = “CURSO ADONAI”.substring(4, 8); // O AD

Strings, Caracteres especiais e Expressões Regulares

Outros métodos manipulação de Strings

Método	Descrição
String[] split(String regex)	Procura por uma parte, conforme especificado pelo argumento string (que contém uma expressão regular) e divide essa string em um array de strings.
String trim()	Retorna uma cópia dessa seqüência com os espaços em branco laterais removidos.
String toLowerCase() e String toUpperCase()	Retornam uma cópia da string convertida em minúsculas ou maiúsculas respectivamente.

Strings, Caracteres especiais e Expressões Regulares

Procurando por caracteres e substrings em uma String

Métodos de pesquisa na classe String

Método	Descrição
int indexOf(int ch)	Retorna o índice da primeira ocorrência do caractere especificado. **Caso não encontre retorna -1**
int lastIndexOf(int ch)	Retorna o índice da última do caractere especificado. **Caso não encontre retorna -1**
boolean contains(CharSequence s)	Retorna true se a string contém a seqüência de caracteres especificado.

Nota: CharSequence é uma interface que é implementada pela classe String.
Portanto, você pode usar uma seqüência de caracteres como um argumento para o método contains().

Strings, Caracteres especiais e Expressões Regulares

Substituição de caracteres e substrings em uma String

Método	Descrição
String replace(char oldChar, char newChar)	Retorna uma nova seqüência de caracteres resultante da substituição de todas as ocorrências de oldChar nessa seqüência de caracteres por newChar.
String replace(CharSequence target, CharSequence replacement)	Substitui cada subseqüência de uma cadeia de caracteres que corresponde a seqüência alvo pela a seqüência de substituição especificada.

Exemplo replace(CharSequence target, CharSequence replacement)

```
String str = “aaa”;
System.out.println(str.replace("aa", "b")); // ba
```

Mais Métodos

length, charAt, equals, pool, equalsIgnoreCase, compareTo, regionsMatches, startsWith, indexOf, substring, concat, replace, toUpperCase, trim, toCharArray, valueOf

Strings, Caracteres especiais e Expressões Regulares

StringBuilder (Classe - java.lang.StringBuilder)

Mutáveis

Capacidade inicial 16 caracteres

Métodos - capacity, ensureCapacity, setLength, setCharAt, getChars, append, insert, reverse
Tokenização de Strings com split

Strings, Caracteres especiais e Expressões Regulares

Caracteres (Classe – java.lang.Character)

Métodos

isDefined, isDigit, isJavaIdentifierStart, isJavaIdentifierPart, isLetter, isLetterOrDigit, isLowerCase, isUpperCase, digit, forDigit, charValue, toString

Classes do pacote java.util.regex

Classe – java.util.regex.Pattern
Métodos - compile, matcher, matches, pattern, split

Classe - java.util.regex.Matcher
Métodos- find, group, lookinAt, matches, pattern, replaceFirst, replaceAll

Strings, Caracteres especiais e Expressões Regulares

Expressões regulares

Classes de caracteres predefinidas

Caractere	Correspondências
\d	Qualquer dígito
\w	Qualquer caractere de palavra
\s	Qualquer caractere de espaço em branco
\D	Qualquer não dígito
\W	Qualquer caractere não palavra
\S	Qualquer caractere não espaço em branco

Quantificadores [ganancioso, ? (relutante ou preguiçoso)]

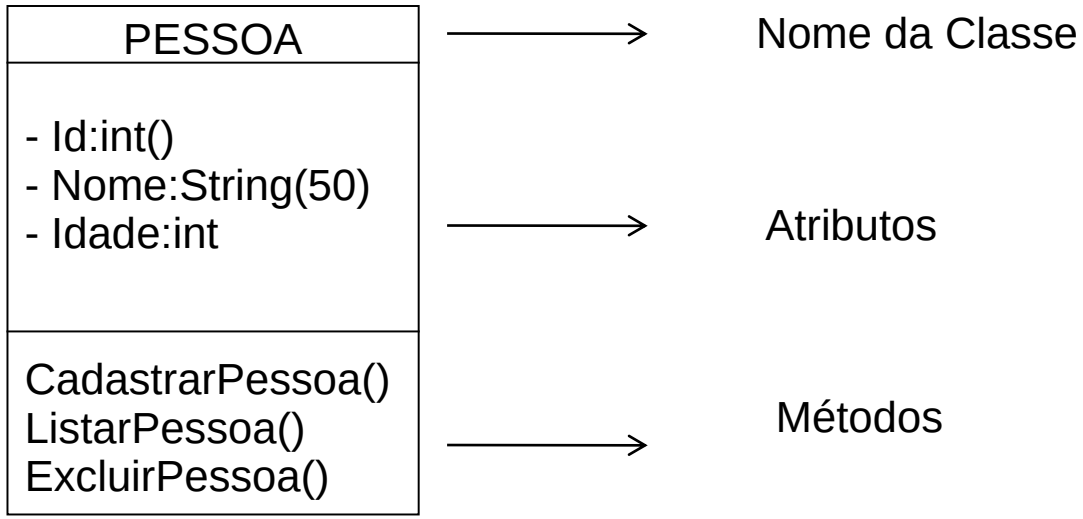
Quantificador	Correspondências
*	Localiza zero ou mais ocorrências do padrão
+	Localiza uma ou mais ocorrências do padrão
?	Localiza zero ou uma ocorrência do padrão
{n}	Localiza exatamente n ocorrências
{n,}	Localiza pelo menos n ocorrências
{n,m}	Localiza entre n e m (inclusive) ocorrências

Classes e Objetos

Classes definem entidades que usualmente representam elementos do mundo real. Eles consistem de um conjunto de valores que **armazenam dados** e de um conjunto de **métodos operam sobre dados**.

Uma instância de uma classe é chamada de objeto e a este objeto é reservado um espaço na memória. Podem existir múltiplas instâncias de uma mesma classe.

Representação de uma Classe



Um objeto é uma instância de uma classe. Uma vez instanciada, objetos possuem seus próprios membros de dados. Objetos distintos da classe são criados (instanciados) usando a palavra-chave **new**

```
Animal gato1 = new Gato( );
Felino gato2 = new Gato( );
```

Classes e Objetos

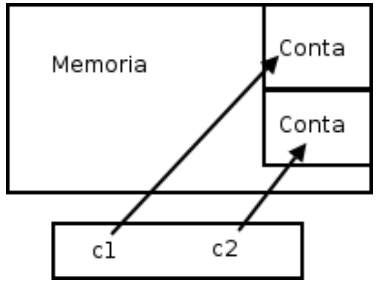
Classe é um tipo de dados definido pelo usuário que tem alguns dados internos e alguns métodos, na forma de procedimentos ou funções, que atuam sobre estes dados.

• A classe é o modelo a partir do qual são criados os objetos.

• Uma classe é um agrupamento de objetos que revelam profundas semelhanças entre si, tanto no aspecto estrutural quanto funcional.

• Classe é uma abstração que descreve as propriedades relevantes de uma “aplicação” em termos de sua estrutura (dados) e de seu comportamento (operações) Programa.

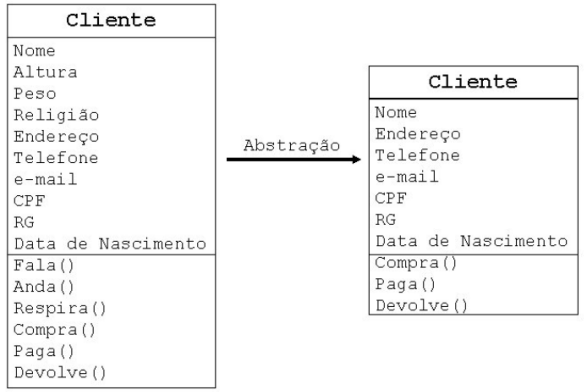
CLASSE CARRO		OBJETO CARRO A	OBJETO CARRO B
Atributos de objeto	Marca	Ford	Mitsubishi
	Modelo	Fiesta	L-200
	Cor	branco	azul royal
	Combustivel	gasolina	diesel
Métodos	ligar		
	acelerar		
	frear		



Classes e Objetos

Operações de Abstração

Em um sistema cuja tarefa principal seja administrar uma base de dados cadastrais de clientes, é evidente que os principais objetos envolvidos são os próprios clientes. O processo de abstração começa com a criação de uma classe para representar esse conjunto de objetos dentro do sistema. As classes não devem incluir todos os atributos dos objetos e tampouco devem incluir todas as suas ações e serviços. Somente devem incluir os atributos e as ações (ou serviços) pertinentes ao papel a ser desempenhado dentro do programa. O processo de abstração compõe-se de um conjunto de operações realizadas com a finalidade de representar objetos do mundo real na forma de classes, de modo que eles possam interagir em um programa para realizar determinadas tarefas. Um primeiro esforço deve ser empreendido no sentido de identificar os atributos e métodos relevantes de cada classe e para eliminar atributos e métodos desnecessários.



Classes e Objetos

O que deve ser ressaltado é que a “Classe” propõe o que é denominado de “modelo conceitual” para os objetos analisados.

- É a representação da ideia (processo de abstração):
- de como os objetos são (características = dados = propriedades = atributos)
- e o que os objetos fazem (comportamento = operações = algoritmos = funções = métodos). Abstração

Classes e Objetos

Métodos, ou operações

Enquanto os atributos permitem armazenar dados associados aos objetos, ou seja, valores que descrevem a aparência ou o estado de um certo objeto, os métodos (methods) ou funções-membro realizam operações sobre os atributos de uma classe ou são capazes de especificar ações ou transformações possíveis para um objeto. Isso significa que os métodos conferem um caráter dinâmico aos objetos, pois permitem que os objetos exibam um comportamento que, em muitos casos, pode mimetizar (imitar) o comportamento de um objeto real ou concreto. Outra maneira de entender o que são os métodos é imaginar que os objetos são capazes de enviar e receber mensagens, de tal forma que possamos construir programas (ou aplicações) nos quais os objetos trocam mensagens, proporcionando o comportamento desejado. A ideia central contida na mensagem que pode ser enviada ao objeto é a mesma quando ligamos um rádio: acionar o botão que liga esse aparelho corresponde a enviar uma mensagem ao rádio: “ligue”. Quando fazemos isso, não é necessário compreender em detalhes como funciona o rádio, mas apenas entender superficialmente como operá-lo Conceito de “Caixa Preta”.

Classes e Objetos

Métodos, ou operações

Método é um serviço que é requisitado a um objeto como parte de seu comportamento em resposta a estímulos (procedimento algorítmico).

Um método é formado por uma interface e sua implementação.

A interface descreve as características externas do método, sua parte visível como: nome, parâmetros e valor retornado.

A implementação contém o código efetivo para a operação, isto é, uma sequência de instruções da linguagem.

Os métodos descrevem o comportamento, como agem e reagem os objetos analisados Funções ou Procedimentos (Algoritmos).

Métodos

A forma genérica para a definição de um método em uma classe é a seguinte:

```
[modificador] tipo nome([parâmetros]) {
    // Corpo do Método
}
```

onde:

- **modificador** (opcional), uma combinação de especifador de acesso (public, protected ou private); abstract ou final e static.
- **tipo** é um indicador do valor de retorno (void quando o método não possuir um valor de retorno = procedimento).
- **nome** do método deve ser um identificador válido.
- **parâmetros** (opcional) são representados por uma lista de parâmetros separados por vírgulas, onde cada parâmetro obedece à forma: tipo nome.

Métodos são essencialmente subrotinas que podem manipular atributos de objetos para os quais o método foi definido. Além dos atributos de objetos, métodos podem definir e manipular variáveis locais; também podem receber parâmetros por valor através da lista de argumentos. Uma boa prática de programação é manter a funcionalidade de um método simples, desempenhando uma única tarefa. O nome do método deve refletir de modo adequado a tarefa realizada.

Métodos

```
// Classe elementar "Matematica.java" implementada para demonstrar
// a definição de atributos e métodos.
public class Matematica
{
  public static int a, b;
  public static int adicao() {
    return (a + b);
  }
  public static int subtracao() {
    return (a - b);
  }
  public static int multiplicacao() {
    return (a * b);
  }
  public static int divisao() {
    return (a / b);
  }
}
```

Declaração da Classe **Matematica**

Atributos, ou variáveis da Classe

"Protocolo" de **Métodos** da Classe

Corpo da Classe

Métodos

Para denotar ou referenciar os atributos ou métodos de uma classe ou objeto deve-se utilizar um operador, denominado seletor, simbolizado por um caractere ponto '.' como segue:

Atributos

NomeDoObjeto, ou NomeDaClasse.nomeDoAtributo

por exemplo:
```
Matematica.a = 10;
System.out.println(Matematica.a);
```

Métodos

NomeDoObjeto, ou NomeDaClasse.nomeDoMétodo([argumentos])

por exemplo:
```
System.out.println(Matematica.adicao());
```

Os parênteses após o nome do método têm duplo propósito: um é diferenciar a construção ou o uso dos métodos da declaração de atributos, e o outro é "permitir" que sejam "especificados valores auxiliares que podem ser enviados" em anexo à mensagem (denominados de argumentos), para informar mais precisamente a forma como a ação deve ser realizada.

Métodos

Para denotar ou referenciar os atributos ou métodos de uma classe ou objeto deve-se utilizar um operador, denominado seletor, simbolizado por um caractere ponto '.' como segue:

Atributos

NomeDoObjeto, ou NomeDaClasse.nomeDoAtributo

por exemplo:
```
Matematica.a = 10;
System.out.println(Matematica.a);
```

Métodos

NomeDoObjeto, ou NomeDaClasse.nomeDoMétodo([argumentos])

por exemplo:
```
System.out.println(Matematica.adicao());
```

Os parênteses após o nome do método têm duplo propósito: um é diferenciar a construção ou o uso dos métodos da declaração de atributos, e o outro é "permitir" que sejam "especificados valores auxiliares que podem ser enviados" em anexo à mensagem (denominados de argumentos), para informar mais precisamente a forma como a ação deve ser realizada.

Métodos

Métodos em Java têm sua execução encerrada de duas maneiras possíveis:

1. quando um método não tem um valor de retorno (declarado com o tipo de retorno **void**): a execução é encerrada quando o bloco do corpo do método chega ao final;

```
public static void main(String args[]) {
   :
} // fim do corpo do método
```

2. encerra a execução do método através do comando **return**.

```
return;              // sem valor de retorno
return expressão; // retornando o resultado da expressão

public static int soma(int a, int b) {
  return (a + b); // retorna a soma dos parâmetros
}
```

Métodos

Membros de dados e métodos

Membros de dados são também conhecidos como **campos (fields)**, armazenam dados sobre a classe.

[*modificadoresJava] tipo nomeDoMembroDeDados*

Métodos operacionais de uma classe

[*modificadores_java] tipo nomeDoMetodo (listaDeParametros*) [throws *listaDeExcecoesSeparadasPorVirgula*] {
  // Corpo do método
}

Métodos

Membros de dados e métodos

O seguinte exemplo mostra a classe Gato e seus membros e métodos:

```
public class Gato {

  // Membros de dados ou campos
  private String apelido;
  private int idade;

  // Métodos de acesso
  public void setApelido ( String apelido ) { this.apelido = apelido; }
  public String getApelido( ) { return apelido; }

  public void setIdade( int idade ) { this.idade = idade; }
  public int getIdade( ) { return idade; }

} // fim da classe Gato
```

Acessando membros de dados e métodos em objetos
O operador ponto (.) é usado para acessar os membros dados e métodos em um objeto. Não é necessário usar o operador ponto, quando acessando membros de dados internamente, na implementação da classe.

Métodos

Construtores

Construtores são chamados por ocasião da criação do objeto e são usados para inicializar dados no objeto recentemente criado.
Construtores são opcionais, possuem **exatamente o mesmo nome da classe** e ao contrário dos métodos tradicionais, **não possuem tipo de retorno**.
Classes implicitamente possuem um construtor **sem-argumentos (conhecido como construtor padrão)**, isto é, se um construtor explícito não for declarado.
Uma classe pode ter vários construtores (construtores podem ser sobrecarregados). O construtor que será chamado na criação do objeto será aquele correspondente a assinatura.

```
public class Contador {
        int valor;
        Contador(int valor) {
                this.valor = valor;
        }
}
// Cria uma instância de Contador e invoca seu construtor
public class UsaContador  {
        Contador contador = new Contador(10);        }
```

Métodos

Construtores

A criação de novos objetos, ou instanciação, tem a forma:

Nome. Da. Classe nome. Do. Objeto = new Nome. Da. Classe();

O operador new sempre precede uma chamada, semelhante ao acionamento de um método comum, mas no qual o nome desse método é o mesmo que o da classe cujo objeto desejamos instanciar.

Esse método especial é denominado “construtor do objeto” ou apenas construtor (constructor). Os construtores são metodos especiais, acionados pelo sistema no momento da criação de um objeto; assim sendo, o operador new apenas indica que o método especial construtor de uma certa classe será utilizado. O resultado da chamada de um construtor é uma referência à área de memória na qual foi criado o objeto, ou seja, o construtor é o responsável pela alocação de memória e pelo preparo do funcionamento do objeto.

Pelo construtor, podem ser inicializados seus atributos com valores consistentes e adequados ao objeto que está sendo criado e, portanto, também podem ser criados outros objetos necessários, por meio do uso de construtores de outras classes, possibilitando a construção de objetos bastantes sofisticados.

Métodos

Construtores

Os construtores podem, efetivamente, criar objetos concretos, isto é, estruturas de dados e operações que representam uma entidade real, a partir de uma classe (um modelo abstrato de objetos de um certo conjunto). Toda e qualquer classe tem ao menos um construtor. Todos os construtores de uma classe têm, obrigatoriamente, sempre o mesmo nome da classe a que pertencem. Outro ponto importante é que os construtores sempre devem ser especificados "sem valor de retorno" e como públicos (public), uma vez que eles sempre resultam na criação de um novo objeto. Para classes em que não exista um construtor explicitamente definido, a linguagem Java assume a existência de um construtor denominado default, que é um método de mesmo nome que a classe, sem parâmetros, que inicializa automaticamente os atributos existentes na classe, da seguinte forma:

Tipo da variável	Valor *default* de inicialização
byte, short, int, long, char	0
float, double	0
boolean	false
referências para objetos	*null*

Métodos

Construtores

Métodos como

```
public Item( String id, String description, int quantity, double price )
```

são chamados *construtores*. Os construtores inicializam um objeto durante sua criação.

NOVO TERMO *Construtores* são métodos usados para inicializar objetos durante sua instanciação. Você chama a criação de objetos de *instanciação* porque ela cria uma instância do objeto da classe.

NOTA

No construtor e por todo o exemplo `Item`, você pode notar o uso de `this`. `this` é uma referência que aponta para a instância do objeto. Cada objeto tem sua própria referência para si mesmo. A instância usa essa referência para acessar suas próprias variáveis e métodos.

Métodos como `setDiscount()`, `getDescription()` e `getAdjustedTotal()` são todos comportamentos da classe `Item` que retornam ou configuram atributos. Quando um caixa quer totalizar o carrinho, ele simplesmente pega cada item e envia ao objeto a mensagem `getAdjustedTotal()`.

Métodos

Destrutores e a Coleta de Lixo

Os destrutores (destructor) também são métodos especiais que liberam as porções de memória utilizada por um objeto, ou seja, enquanto os construtores são responsáveis pela alocação de memória inicial e pelo preparo do objeto, os destrutores encerram as operações em andamento liberando a memória utilizada e todos os demais recursos alocados do sistema, removendo todos os vestígios da existência prévia do objeto.

A importância dos destrutores reside no fato de que devemos devolver (alocação dinâmica) ao sistema operacional os recursos utilizados, pois, caso contrário, tais recursos podem ficar esgotados, impedindo que esse programa e os demais existentes completem suas tarefas. Um dos maiores problemas do desenvolvimento de aplicações é justamente garantir a correta devolução dos recursos alocados do sistema, concentrando esse problema na devolução da memória ocupada, ou seja, nas operações de eliminação dos objetos que não são mais necessários. Quando um programa não devolve ao sistema a quantidade integral de memória alocada, é como se o sistema estivesse ‘perdendo’ sua memória (memory leakage).

Classes e Objetos

Ciclo de Vida de um Objeto

• (1) declara o objeto a partir da classe, (2) instância ou cria o objeto e (3) perder a referência ao objeto criado (o coletor de lixo marca o objeto para eliminação futura).

• O tempo de vida de um objeto vai desde o tempo em que ele é instanciado ou criado (assim consumindo espaço), até quando sua referência é perdida (garbage collector recupera o espaço).

• Persistência = armazenamento permanente dos objetos em disco.

Classes e Objetos

`public` ou sem especificador (pacote)	acesso, ou visibilidade da classe
`abstract`	a classe não pode ser instanciada
`final`	classe terminal, ou folha, a classe não pode derivar outras classes
`class NomeDaClasse`	nome, ou identificador da classe
`extends Super`	a classe é derivada, ou filha, da superclasse, ou classe pai (**Super**)
`implements Interface`	interfaces implementadas da classe

```
{
        // Corpo da Classe
}
```

Os Pilares da POO – ENCAPSULAMENTO

NOVO TERMO Encapsulamento é a característica da OO de ocultar partes independentes da implementação. O encapsulamento permite que você construa partes ocultas da implementação do software, que atinjam uma funcionalidade e ocultam os detalhes de implementação do mundo exterior.

Encapsulamento

- Proibição do acesso direto ao estado de um objeto, disponibilizando apenas métodos que alterem esses estados na interface pública.

Três características do encapsulamento eficaz:

Abstração
Ocultação da implementação
Divisão de responsabilidades

Os Pilares da POO – ENCAPSULAMENTO

Modificadores

Modificador	Classe	Interface	Construtor	Método	Membros de dados
Modificadores de acesso					
private	Não*	Não*	Sim	Sim	Sim
package-private	Sim	Sim	Sim	Sim	Sim
protected	Não*	Não*	Sim	Sim	Sim
public	Sim	Sim	Sim	Sim	Sim

*** Modificadores permitidos em classes internas (inner) ou classes aninhadas (**

Os Pilares da POO – ENCAPSULAMENTO

•Mod

Os modificadores es, métodos e membros de dad odificador está presente o padrã

.

- Modificadores de acesso e suas características de visibilidade

Modificador	Visibilidade
private	Métodos e membros de dados private são acessíveis apenas dentro da classe que os contém.
package-private	O padrão package-private limita acesso ao interior do pacote que reside.
protected	Os métodos e membros de dados protegidos são acessíveis por dentro de seu pacote e também fora de seu pacote pelas subclasses que os herdam.
public	O modificador public permite acesso a partir de qualquer lugar, inclusise fora do pacote o qual ele foi declarado. Note que interfaces são públicas por padrão.

Na prática

Representação de uma Classe

PESSOA
+ Id:int() + Nome:String(50) + End:String(50)
+CadastrarPessoa() +ListarPessoa() +ExcluirPessoa()

No código:

JAVA:

```
public class Pessoa {
 public String Id;
 public String nome;
 public String end;

}
```

JAVA – CAP 2022

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

ENCAPSULAMENTO

No código: O uso do private Determina o conceito de Encapsulamento

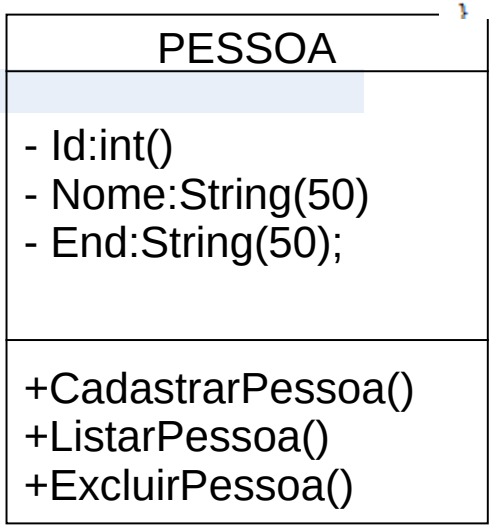
JAVA:

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;

}
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();

    }
}
```



JAVA – CAP 2022

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

Pessoa.Java

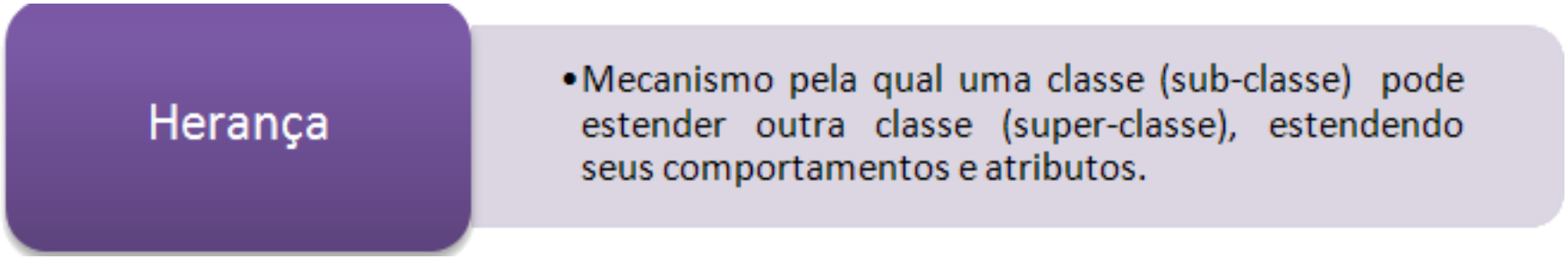
```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```

PESSOA
- Id:int() - Nome:String(50) -End: String(50)
+getId() +setId() +getNome() ...

Os Pilares da POO – HERANÇA



Uma nova classe é criada adquirindo os membros de uma classe existente e, possivelmente, aprimorando-os com capacidades novas ou modificadas. Com a herança, você economiza tempo durante o desenvolvimento de programas baseando novas classes existentes em software testado, depurado e de alta qualidade. Isso também aumenta a probabilidade de que um sistema seja implementado e mantido efetivamente.

Os Pilares da POO – HERANÇA

Ao criar uma classe, em vez de declarar membros completamente novos, você pode designar que a nova classe, chamada de subclasse, deva herdar membros de uma classe existente, que é chamada de superclasse. (Superclasse é a classe básica e a subclasse como a classe derivada.)

Uma subclasse pode tornar-se uma superclasse para futuras subclasses. Uma subclasse pode adicionar seus próprios campos e métodos. Portanto, ela é mais específica que sua superclasse e representa um grupo especializado de objetos. A subclasse exibe os comportamentos da superclasse e pode modificá-los de modo que eles operem adequadamente para a subclasse. É por isso que a herança é às vezes chamada especialização.

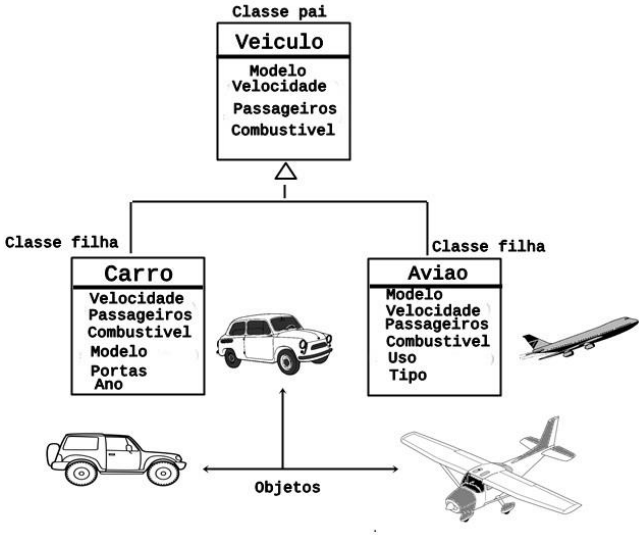
A superclasse direta é aquela a partir da qual a subclasse herda explicitamente. Uma superclasse indireta é qualquer classe acima da superclasse direta na hierarquia de classes, que define os relacionamentos de herança entre classes

No Java, a hierarquia de classes inicia com a classe Object (no pacote java.lang), da qual toda classe em Java direta ou indiretamente estende (ou “herda de”).

O Java só suporta herança única, na qual cada classe é derivada exatamente de uma superclasse direta. Ao contrário de C++, o Java não suporta herança múltipla, que ocorre quando uma classe é derivada de mais de uma superclasse direta.

Os Pilares da POO – HERANÇA

Distinguimos entre o relacionamento é um e o relacionamento tem um, em que é um representa a herança; em um relacionamento é um, um objeto de uma subclasse também pode ser tratado como um objeto da superclasse — por exemplo, um carro é um veículo; por contraste, tem um indica a composição, em um relacionamento tem um, um objeto contém referências como membros a outros objetos — por exemplo, um carro tem um volante (um objeto carro tem uma referência a um objeto volante).



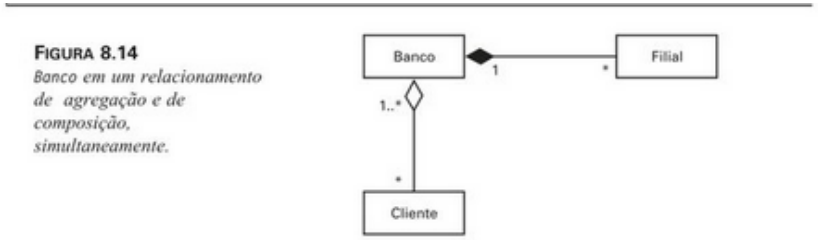


FIGURA 8.14
Banco em um relacionamento de agregação e de composição, simultaneamente.

Os Pilares da POO – HERANÇA

Tipos de Herança:

1 - Para reutilização de implementação
2 - Para diferença
3 - Para substituição de tipo

A Herança para implementação
Em vez de recortar e colar código ou instanciar e usar um componente através de composição, a herança torna o código automaticamente disponível, como parte da nova classe.

A Herança para diferença
A programação pela diferença permite que você programe especificando apenas como uma classe filha difere de sua classe progenitora.
Significa herdar uma classe e adicionar apenas o código que torne a nova classe diferente da classe herdada. Possibilidade de programar através de incrementos.

Para substituição de tipo
Definido como especialização.

Os Pilares da POO – HERANÇA

- Superclasses e Subclasses

Em Java, uma classe (conhecida como subclasse) pode herdar diretamente de uma classe (conhecida como superclasse). A palavra-chave extends indica que uma classe herda membros e métodos de outra classe. Note que subclasses não podem ter acesso direto a membros privados de sua superclasse. Uma **subclasse possui acesso ou membros públicos, protegidos e de pacote da superclasse.**

class A { private int a; protected int b; int c; public int d; }

class B extends A { ... } // possui acesso aos atributos (b, c, d).

Uso das palavras **super** e **this**		
Escopo	Exemplos	Obs:
Construtores	**super( );** **this( ); this(20);**	**Com parêntesis.** Somente em construtores e deve ser a primeira instrução. Representa a assinatura do construtor chamado.
Métodos	**this.**idade; **super.**nome;	**Com ponto.** Indica referência ao objeto da própria classe ou superclasse

Parâmetros	class A {
	public A() { m(**this**); }
	final void m(A a) {...}
	}
		Sozinho.
		Utilizado quando um método recebe como parâmetro um tipo da sua própria classe ou superclasse.

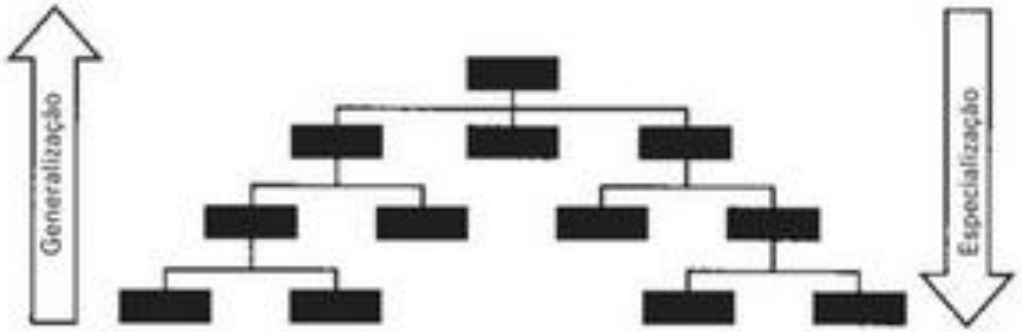
Os Pilares da POO – HERANÇA

ESPECIALIZAÇÃO

Não se confunda com o termo especialização. A especialização permite apenas que você adicione ou redefina os comportamentos e atributos que a filha herda de sua progenitora. A especialização, ao contrário do que o nome possa sugerir, não permite que você remova da filha comportamentos e atributos herdados**. Uma classe não obtém herança seletiva.**

Quando você percorre uma hierarquia para baixo você especializa. Quando você percorre para cima, você generaliza.

FIGURA 4.7
Quando percorre uma hierarquia para cima, você generaliza. Quando percorre uma hierarquia para baixo, você especializa.



Os Pilares da POO – HERANÇA

Resumo Herança

SuperClasse Classe Pai Classe Base Generalização	[e]	SubClasse Classe Filho Classe Derivada Especialização

Reutilização de Código

Os Pilares da POO – HERANÇA

Classe Object

A classe **java.lang.Object** é a superclasse de toda a hierarquia de classes Java, ou é "raiz" a partir da qual todas as classes são definidas. Desse modo, os métodos dessa classe estão disponíveis para objetos de todas as demais classes.

- protected Object clone()
  criar e retorna um "novo" objeto com o mesmo conteúdo do objeto existente

- public boolean equals(Object obj)
  permite comparar objetos por seus conteúdos

- protected void finalize()
  método destrutor, faz o objeto perder a "referência"

- public String toString()
  permite converter uma representação interna do objeto em uma *string* que pode ser apresentada ao usuário

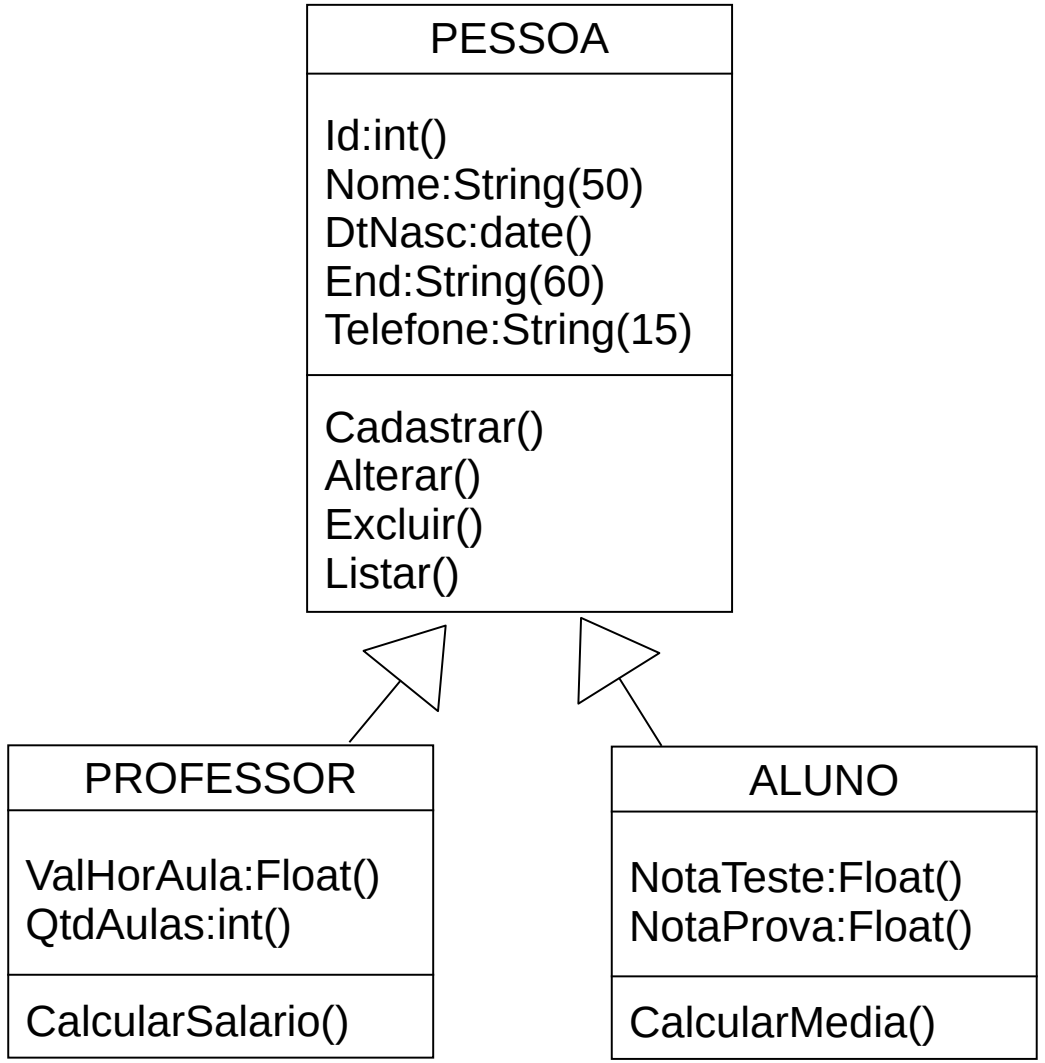
- public final Class getClass()
  retorna um objeto que representa a classe à qual o objeto pertence.

  A partir do objeto retornado, da classe **java.lang.Class**, é possível obter:
  - o nome da classe usando o método getName(); e
  - o nome da superclasse usando o método getSuperclass();

  estes métodos retornam uma String com o nome da classe ou da superclasse, respectivamente.

Na prática

HERANÇA



Código em JAVA:

Pessoa.Java

```java
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

Professor.Java

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;


    public double getValorHoraAula() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }


}
```

Os Pilares da POO – HERANÇA

- Classes e métodos abstratos
- **Classes abstratas**

É um tipo de classe especial que não pode ser instanciada, apenas herdada. Sendo assim, uma classe abstrata não pode ter um objeto criado a partir de sua instanciação. Essas classes são muito importantes quando não queremos criar um objeto a partir de uma classe "geral", apenas de suas "subclasses".

Uma classe abstrata é tipicamente usada como uma classe base e não pode ser instanciada. Ela pode conter métodos abstratos e não-abstratos, e pode ser uma subclasse de uma classe abstrata ou uma não-abstrata. Todos os métodos abstratos precisam ser defindos pela classe que herda (extende) ao menos que a subclasse seja também abstrata.

```
public abstract class Alarme {
  public void reset() {...} // método concreto possui corpo.
 public abstract void processarAlarme();// método abstrato apenas definição.
}
```

- **Métodos abstratos**

Um método abstrato contém apenas a definição do método, o qual precisa ser implementado por qualquer classe não-abstrata que herdá-lo.

```
public class DisplayAlarme extends Alarme {
 public void processarAlarme () {
   System.out.println("Active alarm.");      } }
```

Pode-se dizer que as **classes abstratas** servem como "modelo" para outras classes que dela herdem, não podendo ser instanciada por si só. Para ter um objeto de uma classe abstrata é necessário criar uma classe mais especializada herdando dela e então instanciar essa nova classe. Os métodos da classe abstrata devem então serem sobrescritos nas classes filhas.

Por exemplo, é definido que a classe "Animal" seja herdada pelas subclasses "Gato", "Cachorro", "Cavalo", mas ela mesma nunca pode ser instanciada.

```
1  abstract class Conta {
2
3      private double saldo;
4
5      public void setSaldo(double saldo) {
6          this.saldo = saldo;
7      }
8
9      public double getSaldo() {
10         return saldo;
11     }
12
13     public abstract void imprimeExtrato();
14
15 }
```

No exemplo da Listagem 5, o método "`imprimeExtrato()`" tem uma annotation conhecida como `@Override`, significando que estamos sobrescrevendo o método da superclasse. Entende-se em que na classe abstrata "Conta" os métodos que são abstratos têm um comportamento diferente, por isso não possuem corpo. Ou seja, as subclasses que estão herdando precisam desse método mas não de forma genérica, aonde permite inserir as particularidades de cada subclasse.

```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3
4  public class ContaPoupanca extends Conta {
5
6      @Override
7      public void imprimeExtrato() {
8          System.out.println("### Extrato da Conta ###");
9
10         SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/aaaa HH:mm:ss");
11         Date date = new Date();
12
13         System.out.println("Saldo: "+this.getSaldo());
14         System.out.println("Data: "+sdf.format(date));
15
16     }
17 }
```

Listagem 5. Classe ContaPoupanca

```java
public class TestaConta {
    public static void main(String[] args) {
        Conta cp = new ContaPoupanca();
        cp.setSaldo(2121);
        cp.imprimeExtrato();

    }
}
```

Listagem 6. Classe Testadora para classes abstratas

Os Pilares da POO – HERANÇA

- **Interfaces**

As interfaces oferecem um conjunto de métodos públicos que não possuem corpo. Uma classe que implementa uma interface deve prover implementações concretas de todos os métodos definidos pela interface, ou ela deve ser declarada como abstrata.
Uma interface é declarada usando-se a palavra-chave interface, seguida do nome da interface e de um conjunto de declarações de métodos.

```
public interface Cantor  {
    void cantar();
}
```

Uma classe que implementa uma interface utiliza a palavra-chave implements na definição da classe.

```
class RobertoCarlos  implements Cantor {
  public void cantar() {
    System.out.println("Você meu amigo de fé...");
  }
}
```

As classes podem implementar múltiplas interfaces, e as interfaces podem estender múltiplas interfaces.

Os Pilares da POO – HERANÇA

- **Interfaces**
- **As interfaces são padrões definidos através de contratos ou especificações. Um contrato define um determinado conjunto de métodos que serão implementados nas classes que assinarem esse contrato. Uma interface é 100% abstrata, ou seja, os seus métodos são definidos como abstract, e as variáveis por padrão são sempre constantes (static final).**
- **Uma interface é definida através da palavra reservada "interface". Para uma classe implementar uma interface é usada a palavra "implements", descrita na Listagem 8.**
- **Como a linguagem Java não tem herança múltipla, as interfaces ajudam nessa questão, pois bem se sabe que uma classe pode ser herdada apenas uma vez, mas pode implementar inúmeras interfaces. As classes que forem implementar uma interface terão de adicionar todos os métodos da interface ou se transformar em uma classe abstrata, veja nos exemplos abaixo.**
- **Na Listagem 7 a interface Conta tem seus métodos sem corpo, apenas com os parâmetros e o tipo de retorno.**

```
1  interface Conta{
2      void depositar(double valor);
3      void sacar(double valor);
4      double getSaldo();
5  }
```

Listagem 7. Declaração de uma interface

Neste exemplo da Listagem 8 os métodos são sobrepostos através da interface Conta.

```
1  public class ContaCorrente implements Conta {
2      private double saldo;
3      private double taxaOperacao = 0.45;
4
5      @Override
6      public void deposita(double valor) {
7          this.saldo += valor - taxaOperacao;
8      }
9
10     @Override
11     public double getSaldo() {
12         return this.saldo;
13     }
14
15     @Override
16     public void sacar(double valor) {
17         this.saldo -= valor + taxaOperacao;
18     }
19
20 }
```

Listagem 8. Classe Conta Corrente

```java
public class ContaPoupanca implements Conta {
    private double saldo;

    @Override
    public void deposita(double valor) {
        this.saldo += valor;
    }

    @Override
    public double getSaldo() {
        return this.saldo;
    }

    @Override
    public void sacar(double valor) {
        this.saldo -= valor;

    }

}
```

Listagem 9. Classe ContaPoupanca com os métodos sobrepostos da interface Conta

Os Pilares da POO – HERANÇA

O método “geradorConta”, da Listagem 10, mostra a entrada de um parâmetro do tipo Conta, essa função será útil para a saída de um resultado.

```
1  public class GeradorExtratos {
2
3      public void geradorConta(Conta conta){
4          System.out.println("Saldo Atual: "+conta.getSaldo());
5      }
6
7  }
```

Listagem 10. Classe GeradorExtratos

Na Listagem 11 são instanciadas as classes e o gerador de extratos. Na classe "GeradorExtratos" é invocado o método que aceita como parâmetro um tipo de "Conta".

```
1  public class TestaContas {
2
3      public static void main(String[] args) {
4
5          ContaCorrente cc = new ContaCorrente();
6          cc.deposita(1200.20);
7          cc.sacar(300);
8
9          ContaPoupanca cp = new ContaPoupanca();
10         cp.deposita(500.50);
11         cp.sacar(25);
12
13
14         GeradorExtratos gerador = new GeradorExtratos();
15         gerador.geradorConta(cc);
16         gerador.geradorConta(cp);
17     }
18
19 }
```

Listagem 11. Classe TestaContas

Tipos Enum

O tipo enum básico que define um conjunto de constantes representadas como identificadores únicos.

```
enum DisplayButton {ROUND, SQUARE}
DisplayButton round = DisplayButton.ROUND;
```

Do ponto de vista além dos termos simples, um enumerado é uma classe do tipo **enum**. Classes do tipo enumerado podem ter métodos, construtores e membros de dados.

```
enum DisplayButton {
   // Tamanho em polegadas
   ROUND (.50f),
  SQUARE (.40f);
   private final float size;
   DisplayButton(float size) {this.size = size;}
   private float size()  { return size; }
}
```

O método **values()** retorna um array da lista ordenada de objetos defindos pelo enum.

```
for (DisplayButton b : DisplayButton.values())
  System.out.println("Button: " + b.size());
```

Os Pilares da POO – POLIMORFISMO

Polimorfismo

- Princípio pelo qual as instâncias de duas classes ou mais classes derivadas de uma mesma super-classe podem invocar métodos com a mesma assinatura, mas com comportamentos distintos.

Polimorfismo significa muitas formas. Em termos de programação, o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático. Assim, um nome pode assumir muitas formas e como pode representar código diferente o mesmo nome pode representar muitos comportamentos diferentes.

Os Pilares da POO – POLIMORFISMO

O polimorfismo permite que classes abstratas consigam receber comportamentos através de classes concretas. Por exemplo, um dispositivo USB, podemos considerar que o USB seria uma classe abstrata enquanto os dispositivos (Pen Driver, Ipad, Câmeras, etc) seriam as classes concretas. Ou seja, o USB é uma especificação que pode ter várias implementações com características diferentes.

A figura a seguir ilustra alguns exemplos de tipos de polimorfismo.

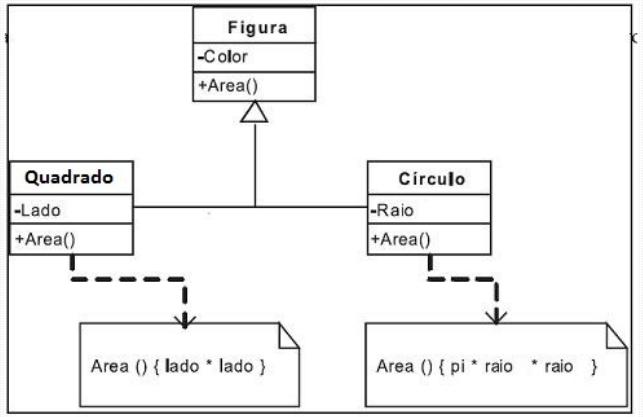


Figura 1. Exemplo de Polimorfismo

JAVA – CAP 2022

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRESCRITA**

JAVA:

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;

    public String getNome() {
        System.out.println("Olá professor: "+nome);
        return nome;
    }

    public double getValorHora() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }
```

```java
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {
        System.out.println("Olá aluno: "+nome);
        return nome;
    }
    public double getNota1() {
        return nota1;
    }

    public void setNota1(double nota1) {
        this.nota1 = nota1;
    }

    public double getNota2() {
        return nota2;
    }

    public void setNota2(double nota2) {
        this.nota2 = nota2;
    }
```

Na prática

Pessoa.Java

```java
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

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

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```java
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```

JAVA – CAP 2022

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


    public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

    }

    public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
    }
```

JAVA – CAP 2022

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
10
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir();
24
25
26      }
27
```

Sem nenhum parâmetro!!!

```
Saída -
Nome: Matheus
Endereço: Rua suvaco da minhoca
Idade: 19
Nota1: 7.9
Nota2: 10.0
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

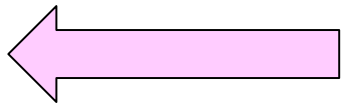
JAVA – CAP 2022

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

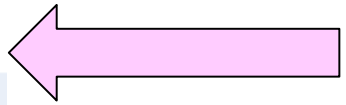
```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


   public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

   }

   public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
   }
```



Na prática: **SOBRECARGA**

JAVA:

```
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir(a.getNota1(), a.getNota2());
24
25
26      }
27
```



Com 2 parâmetros!!!

```
Saída -
run:
A média é:8.95
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

MULTHREADING

O que é uma thread

Uma thread age como um programa individual. Threads, em Java, são objetos.
Individualmente, cada thread faz de conta que tem total poder sobre a CPU

O sistema garante que, de alguma forma, cada thread tenha acesso à CPU de acordo com:
Cotas de tempo (**quantum** ou fração de tempo)
Prioridades
Programador pode controlar parcialmente a forma de agendamento dos threads
Há dependência de plataforma no agendamento

Por que usar múltiplos threads?
Todo programa tem pelo menos uma thread, chamado de Main Thread.
O método main() roda no Main Thread
Em alguns tipos de aplicações, threads adicionais são essenciais. Em outras, podem melhorar o bastante o desempenho, como por exemplo em interfaces gráficas (GUI).
Essencial para ter uma interface do usuário que responda enquanto outra tarefa está sendo executada:
- Rede
- Essencial para que servidor possa continuar a esperar por outros clientes enquanto lida com as requisições de cliente conectado.

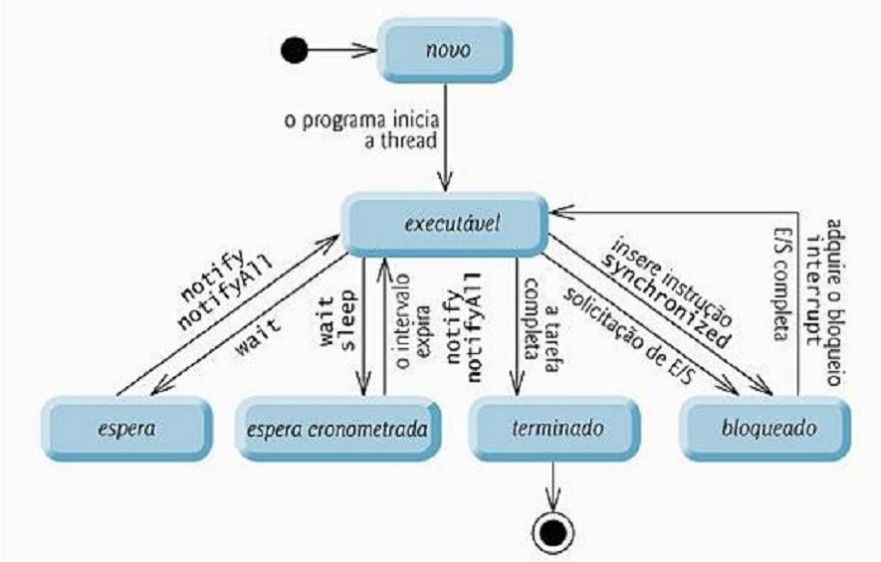
Estratégias para criação de Threads

Herdar da classe java.lang.Thread
Implementar a interface java.lang.Runnable

MULTHREADING

Ciclo de Vida

Uma nova thread inicia seu ciclo de vida no estado *novo*. Ela permanece nesse estado até o programa iniciar a thread, o que a coloca no estado *executável*.



MULTHREADING

Métodos importantes da classe Thread

run() – é o "main" do Thread. Deve ser implementado na thread ou no objeto Runnable passado a thread.

start() - sinaliza à JVM que a thread pode ser executada, mas saiba que essa execução não é garantida quando esse método é chamado, e isso pode depender da JVM.

isAlive() - retorna true se a thread está sendo executada e ainda não terminou.

sleep() - suspende a execução da thread por um tempo determinado.

yield() – faz com que o thread atual pare e permita que outros que estão na sua fila de prioridades executem.

currentThread() - é um método estático da classe Thread que retorna qual a thread que está sendo executada.

getName() - retorna o nome da Thread, você pode especificar o nome de uma Thread com o método setName() ou na construção da mesma, pois existe os construtores sobrecarregados.

MULTHREADING

O Java fornece utilitários de alto nível para a concorrência a fim de ocultar boa parte dessa complexidade e tornar a programação multithread menos propensa a erros. As prioridades de thread são utilizadas nos bastidores para interagir com o sistema operacional, mas a maioria dos programadores que utiliza o multithreading do Java não se preocupará com a configuração e o ajuste de prioridades de thread.

Sincronização

Recursos podem ser compartilhados por vários threads simultaneamente

Cada objeto tem um bloqueio que pode ser acionado pelo método que o modifica para evitar corrupção de dados
Dados podem ser corrompidos se uma thread deixar um objeto em um estado incompleto e outro thread assumir a CPU

Recursos compartilhados devem ser protegidos

A palavra-reservada synchronized permite que blocos sensíveis ao acesso simultâneo sejam protegidos de corrupção impedindo que objetos os utilizem ao mesmo tempo.

Synchronized deve limitar-se aos trechos críticos (performance!)

Métodos inteiros podem ser synchronized

Se um recurso crítico está sendo usado, só uma thread tem acesso. É preciso que:
Os outros esperem até que o recurso esteja livre
O thread que usa o recurso avise aos outros que o liberou

Esse controle é possível através de métodos da classe Object, que só podem ser usados em blocos synchronized
wait(): faz com que a thread sobre o qual é chamado espere por um tempo indeterminado, até receber um...
notify(): notifica a próxima thread que o recurso bloqueado foi liberado. Se há mais threads interessados, deve-se usar o
notifyAll(): avisa a todos os threads.

Genéricos

A motivação de estudar Generics em Java é de poupar o desenvolvedor de códigos redundantes, como é o caso de casting excessivo. Este foi introduzido desde o Java SE 5.0.

Vimos sobre a coleção ArrayList genérica — uma estrutura de dados dinamicamente redimensionável do tipo array, que armazena referências a objetos de um tipo que você especifica ao criar o ArrayList. Para entendimento sobre genéricos em java, vamos entender primeiramente o framework collection do Java, que contém muitas outras estruturas de dados genéricos predefinidas.

Genéricos

Uma coleção é uma estrutura de dados — na realidade, um objeto — que pode armazenar referências a outros objetos. Normalmente, coleções contêm referências a objetos de qualquer tipo que tem o relacionamento é um com o tipo armazenado na coleção. As interfaces de estrutura de coleções declaram as operações a ser realizadas genericamente em vários tipos de coleções. A Figura 16.1 lista algumas das interfaces da estrutura das coleções. Várias implementações dessas interfaces são fornecidas dentro da estrutura. Você também pode fornecer suas próprias implementações.

Interface	Descrição
`Collection`	A interface-raiz na hierarquia de coleções a partir da qual as interfaces `Set`, `Queue` e `List` são derivadas.
`Set`	Uma coleção que *não* contém duplicatas.
`List`	Uma coleção ordenada que *pode* conter elementos duplicados.
`Map`	Uma coleção que associa chaves a valores e que *não pode* conter chaves duplicadas. `Map` não deriva de `Collection`.
`Fila`	Em geral, uma coleção *primeiro a entrar, primeiro a sair* que modela uma *fila de espera*; outras ordens podem ser especificadas.

Figura 16.1 | Algumas interfaces da estrutura de coleções.

Genéricos

Coleções baseadas em Object

As classes e interfaces da estrutura das coleções são membros do pacote java.util.
Nas primeiras versões do Java, as classes da estrutura das coleções armazenavam e manipulavam somente referências Object, permitindo armazenar qualquer objeto em uma coleção, porque todas as classes direta ou indiretamente derivam da classe Object.
Programas normalmente precisam processar tipos específicos de objetos. Como resultado, as referências Object obtidas de uma coleção em geral precisam sofrer downcast em um tipo apropriado para permitirem que o programa processe os objetos corretamente.

Coleções genéricas
Para eliminar esse problema, o framework das coleções foi aprimorado com as capacidades de genéricos introduzidas com ArrayLists genéricas.

Genéricos permitem especificar o tipo exato que será armazenado em uma coleção e fornecem os benefícios da verificação de tipo em tempo de compilação — o compilador emite mensagens de erro se você usar tipos inadequados nas coleções. Depois de especificar o tipo armazenado em uma coleção genérica, qualquer referência que você recupera da coleção terá esse tipo. Isso elimina a necessidade de coerções de tipo explícitas que podem lançar ClassCastExceptions se o objeto referenciado não for do tipo apropriado. Além disso, as coleções genéricas são retrocompatíveis com o código Java que foi escrito antes que genéricos tenham sido introduzidos.

Genéricos

Escolhendo uma coleção

A documentação para cada coleção discute os requisitos de memória e as características de desempenho dos métodos para operações como adição e remoção de elementos, pesquisa de elementos, classificação de elementos etc. Antes de escolher uma coleção, revise a documentação on-line para a categoria da coleção que você está considerando (Set, List, Map, Queue etc.), então selecione a implementação que melhor atende às necessidades de seu aplicativo.

Classes empacotadoras de tipo Todo tipo primitivo tem uma classe empacotadora de tipo correspondente (no pacote java.lang). Essas classes chamam-se Boolean, Byte, Character, Double, Float, Integer, Long e Short. Elas permitem manipular valores de tipo primitivo como objetos.

O Java fornece conversões *boxing* e *unboxing* que convertem automaticamente entre valores de tipo primitivo e objetos empacotadores de tipo. Uma **conversão boxing** converte um valor de um tipo primitivo em um objeto da classe empacotadora de tipo correspondente. Uma **conversão unboxing** converte um objeto de uma classe empacotadora de tipo em um valor do tipo primitivo correspondente. Essas conversões são executadas automaticamente — o que é chamado de **autoboxing** e **auto-unboxing**. Considere as seguintes instruções:

```
Integer[] integerArray = new Integer[5]; // cria integerArray
integerArray[0] = 10; // atribui Integer 10 a integerArray[0]
int value = integerArray[0]; // obtém valor int de Integer
```

Genéricos

Listas

Uma List (às vezes chamada de sequência) é uma Collection ordenada que pode conter elementos duplicados. Como os arrays, índices de List são baseados em zero (isto é, o índice do primeiro elemento é zero). Além dos métodos herdados de Collection, List fornece métodos para manipular elementos por meio de seus índices, manipular um intervalo especificado de elementos, procurar elementos e obter um ListIterator para acessar os elementos.

A interface List é implementada por várias classes, inclusive as classes ArrayList, LinkedList e Vector. O autoboxing ocorre quando você adiciona valores de tipo primitivo a objetos dessas classes, porque eles armazenam apenas referências a objetos.

As classes ArrayList e Vector são implementações de arrays redimensionáveis de List. Inserir um elemento entre os elementos existentes de um ArrayList ou Vector é uma operação ineficiente — todos os elementos depois do novo devem ser removidos, o que pode ser uma operação cara em uma coleção com um grande número de elementos. Uma LinkedList permite a inserção (ou remoção) eficiente dos elementos no meio de uma coleção, mas é muito menos eficiente que um ArrayList para pular para um elemento específico na coleção.

Genéricos

ArrayList e Vector têm comportamentos praticamente idênticos.

Operações em Vectors são sincronizadas por padrão, enquanto aquelas em ArrayLists não o são. Além disso, a classe Vector é do Java 1.0, antes de a estrutura de coleções ser adicionada ao Java. Assim, Vector tem alguns métodos que não fazem parte da interface List e não são implementados na classe ArrayList. Por exemplo, os métodos Vector addElement e add acrescentam um elemento a um Vector, mas somente o método add é especificado na interface List e implementado por ArrayList. As coleções não sincronizadas fornecem melhor desempenho que as sincronizadas. Por essa razão, ArrayList em geral é preferida a Vector em programas que não compartilham uma coleção entre threads.

Dica de desempenho:
ArrayLists comportam-se como Vectors sem sincronização e, portanto, executam mais rápido que Vectors, porque ArrayLists não têm o overhead de sincronização de thread.

Genéricos

16.7 Métodos de coleções

A classe `Collections` fornece vários algoritmos de alto desempenho para manipular elementos de coleção. Os algoritmos (Figura 16.5) são implementados como métodos `static`. Os métodos `sort`, `binarySearch`, `reverse`, `shuffle`, `fill` e `copy` operam em `Lists`. Os métodos `min`, `max`, `addAll`, `frequency` e `disjoint` operam em `Collections`.

Método	Descrição
`sort`	Classifica os elementos de uma `List`.
`binarySearch`	Localiza um objeto em uma `List` usando o algoritmo de pesquisa binária de alto desempenho introduzido na Seção 7.15 e discutido em detalhes na Seção 19.4.
`reverse`	Inverte os elementos de uma `List`.
`shuffle`	Ordena aleatoriamente os elementos de uma `List`.
`fill`	Configura todo elemento `List` para referir-se a um objeto especificado.
`copy`	Copia referências de uma `List` em outra.
`min`	Retorna o menor elemento em uma `Collection`.
`max`	Retorna o maior elemento em uma `Collection`.
`addAll`	Acrescenta todos os elementos em um array a uma `Collection`.

Método	Descrição
`frequency`	Calcula quantos elementos da coleção são iguais ao elemento especificado.
`disjoint`	Determina se duas coleções não têm nenhum elemento em comum.

Genéricos

```
1  // Figura 16.9: Sort3.java
2  // Método sort de Collections com um objeto Comparator personalizado.
3  import java.util.List;
4  import java.util.ArrayList;
5  import java.util.Collections;
6
7  public class Sort3
8  {
9     public static void main(String[] args)
10    {
11       List<Time2> list = new ArrayList<>(); // cria List
12
13       list.add(new Time2(6, 24, 34));
14       list.add(new Time2(18, 14, 58));
15       list.add(new Time2(6, 05, 34));
16       list.add(new Time2(12, 14, 58));
17       list.add(new Time2(6, 24, 22));
18
19       // gera saída de elementos List
20       System.out.printf("Unsorted array elements:%n%s%n", list);
21
22       // classifica em ordem utilizando um comparador
23       Collections.sort(list, new TimeComparator());
24
25       // gera saída de elementos List
26       System.out.printf("Sorted list elements:%n%s%n", list);
27    }
28 } // fim da classe Sort3
```

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Figura 16.9 | O método `Collections sort` com um objeto `Comparator` personalizado.

Genéricos

16.13 Coleções sincronizadas

No Capítulo 23, discutimos *multithreading*. Exceto para `Vector` e `Hashtable`, as coleções na estrutura das coleções são *dessincronizadas* por padrão, assim elas podem funcionar eficientemente quando *multithreading* não for necessário. Mas como elas são *dessincronizadas*, o acesso concorrente a uma `Collection` por múltiplas threads pode provocar resultados indeterminados ou erros fatais — como demonstrado no Capítulo 23. Para evitar potenciais problemas de threading, **empacotadores de sincronização** são usados para as coleções que podem ser acessadas por múltiplas threads. Um objeto **empacotador** (wrapper) recebe chamadas de método, adiciona sincronização de thread (para evitar acesso simultâneo à coleção) e *delega* as chamadas para o objeto de coleção empacotado. A API `Collections` fornece um conjunto de métodos `static` para empacotar coleções como versões sincronizadas. Os cabeçalhos de método para os empacotadores de sincronização estão listados na Figura 16.20. Os detalhes desses métodos estão disponíveis em `http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html`. Todos esses métodos aceitam um tipo genérico e retornam uma *visualização sincronizada* do tipo genérico. Por exemplo, o seguinte código cria uma `List` sincronizada (`list2`) que armazena objetos `String`:

```
List<String> list1 = new ArrayList<>();
List<String> list2 = Collections.synchronizedList(list1);
```

Cabeçalhos do método `public static`
`<T> Collection<T> synchronizedCollection(Collection<T> c)`
`<T> List<T> synchronizedList(List<T> aList)`
`<T> Set<T> synchronizedSet(Set<T> s)`
`<T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)`
`<K, V> Map<K, V> synchronizedMap(Map<K, V> m)`
`<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> m)`

Figura 16.20 | Métodos empacotadores de sincronização.

Classes e métodos genéricos

Seria conveniente se pudéssemos escrever um único método sort para classificar os elementos em um array de Integer, em um array de String ou em um array de qualquer tipo que suporte ordenamento (isto é, seus elementos podem ser comparados). Também seria conveniente se pudéssemos escrever uma única classe Stack que seria utilizada como uma Stack de inteiros, uma Stack de números de ponto flutuante, uma Stack de Strings ou uma Stack de qualquer outro tipo. Seria ainda mais conveniente se pudéssemos detectar não correspondências de tipos em tempo de compilação — conhecida como segurança de tipos em tempo de compilação. Por exemplo, se uma Stack deve armazenar somente números inteiros, uma tentativa de colocar uma String nessa Stack deve produzir um erro de compilação.

Especificamente métodos genéricos e classes genéricas fornecem os meios para criar os modelos gerais seguros para os tipos mencionados.

Classes e métodos genéricos

Características

ESTRUTURAIS

- Não aceita tipos primitivos como parâmetros de tipo, ou seja, apenas tipos referência
- Métodos e construtores genéricos podem ser sobrecarregados por métodos ou construtores genéricos ou não genéricos respeitando as regras de assinatura e o erasure efetuado pelo compilador.
- Uma classe genérica pode ser derivada de uma classe não-genérica. Por exemplo, a classe Object é uma superclasse direta ou indireta de cada classe genérica.
- Uma classe genérica pode ser derivada de outra classe genérica
- Uma classe não genérica pode ser derivada de uma classe genérica.
- Um método genérico em uma subclasse pode sobrescrever um método genérico em uma superclasse se os dois métodos tiverem a mesma assinatura.

VANTAGENS

- Facilita o reuso
- Auxilia na manutenção
- Mantém a segurança de tipos em tempo de compilação.

Exemplo: Se uma pilha armazenasse somente tipos inteiros, tentar inserir uma String nessa pilha irá emitir um erro em tempo de compilação.

```
List<Integer> intStack = new ArrayList<Integer>();
String str = "Adonai";
intStack.add(str); // Erro em tempo de compilação
```

Classes e métodos genéricos

Erasure

Quando o compilador traduz o método genérico em bytecodes, ele remove a seção de parâmetro de tipo e substitui os parâmetros de tipo por tipos reais. Esse processo é conhecido como erasure.

Tipos genéricos

Todas as declarações de métodos genéricos têm uma seção de parâmetro de tipo delimitada por colchetes angulares (<E>) que precedem o tipo de retorno do método.

Cada seção de parâmetro de tipo contém um ou mais parâmetros de tipos (também chamados parâmetros de tipo formais), separados por vírgulas.

Um parâmetro de tipo, também conhecido como uma **variável de tipo** é um identificador que específica um nome genérico do tipo.

Os parâmetros de tipo podem ser utilizados para declarar o tipo de retorno, tipos de parâmetros e tipos de variáveis locais em uma declaração de método genérico e atuam como marcadores de lugar para os tipos dos argumentos passados ao método genérico, conhecidos como **argumentos de tipos reais**.

Convenção de nomes para parâmetros de tipo:

E - Element
K - Key
N - Number
T - Type
V - Value

Classes e métodos genéricos

Métodos genéricos

```
public class Box2<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <E> void verifica(E e){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("E: " + e.getClass().getName());
    }

    public static void main(String[] args) {
        Box2<Integer> integerBox = new Box2<Integer>();
        integerBox.add(new Integer(10));
        integerBox.verifica("alguma string");
    }
}
```

Classes e métodos genéricos

Métodos genéricos

```
public class Box2<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <E> void verifica(E e){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("E: " + e.getClass().getName());
    }

    public static void main(String[] args) {
        Box2<Integer> integerBox = new Box2<Integer>();
        integerBox.add(new Integer(10));
        integerBox.verifica("alguma string");
    }
}
```

Classes e métodos genéricos

Parâmetros de Tipo Limitados

Parâmetros de Tipo podem especificar um limite superior (extends) ou um limite inferior (super), que irá limitar os tipos que podem ser passados.

```
import java.util.*;

public class Box3<T extends Number> {

   private List<T> lista;

public Box3() {
          lista = new ArrayList<T>();
}

   public void add(T t) {
     lista.add(t);
   }

public List<T> get() {
          return lista;
}

   public static void main(String[] args) {

      Box3<Number> numeros = new Box3<Number>();

      numeros.add(10);
 numeros.add(5.2);
 numeros.add(new Long(128));

 //numeros.add("uma string"); // erro ao adicionar string

  for(Number n : numeros.get()) {
       System.out.printf("%s - %s %n",
       n.getClass().getName(), n);

   }
     }
}
```

Classes e métodos genéricos

Tipos Brutos

Observe esta nova versão do código Box2.java, agora denominado Box4.java

```
public class Box4<T> {

   private T t;

   public Box4() {}

   public Box4(T t) { this.t = t; }

   public void add(T t) {
      this.t = t;
   }

   public T get() {
            return t;
   }

   public <E> void verifica(E e){
      System.out.println("T: " + t.getClass().getName());
      System.out.println("E: " + e.getClass().getName());
   }

   public static void main(String[] args) {
      Box4<Integer> integerBox = new Box4<Integer>();
      Box4 box = new Box4(10.5);
            integerBox.add(new Integer(10));
      integerBox.verifica("alguma string");
  box.verifica(box.get());
   }
}
```

Nesse caso, dizse que o "box" tem um tipo bruto, o que significa que o compilador utiliza implicitamente o tipo Object por toda a classe genérica para cada argumento de tipo. Assim, a instrução precedente cria uma Box4 que pode armazenar objetos de qualquer tipo.

Isso é importante para retrocompatibilidade com versões anteriores do Java. Por exemplo, todas as estruturas de dados do Java Collections Framework armazenavam referências a Objects, mas agora são implementadas como tipos genéricos.

Embora essa atribuição seja permitida, é perigosa porque uma Box4 do tipo bruto armazenaria outros tipos além de Integer. Nesse caso, o compilador emite uma mensagem de alerta que indica a atribuição insegura.

```
Box4.java:24: warning: [unchecked] unchecked call to
Box4(T) as a member of the raw type Box4

------------------------------------------------

Box4 box = new Box4(10.5);
                    ^
Box4.java:27: warning: [unchecked] unchecked call to
<E>verifica(E) as a member of the raw type Box4
               box.verifica(box.get());
                            ^
2 warnings
```

Classes e métodos genéricos

Curingas

Usado quando trabalhamos com subtipos desconhecidos

```
import java.util.*;

public class Box5<T extends Number> {

   private List<T> lista;

public Box5() {
            lista = new ArrayList<T>();
}

   public void add(T t) {
      lista.add(t);
   }

public List<T> get() {
            return lista;
}

public static double soma(List<Number> lista) {
            double cont = 0;
            for(Number n : lista)
                        cont += n.doubleValue();
            return cont;
}

public static double somaWildCard(List<? extends Number> lista) {
            double cont = 0;
            for(Number n : lista)
                        cont += n.doubleValue();
            return cont;
}

   public static void main(String[] args) {
      Box5<Number> numeros = new Box5<Number>();
            Box5<Integer> inteiros = new Box5<Integer>();
     numeros.add(10);
            numeros.add(5.2);
            numeros.add(new Long(128));
            inteiros.add(10);
            inteiros.add(20);
            inteiros.add(30);
            // Uma lista de Number não é o mesmo que uma lista
            // de Integer pois são subtipos indiretos
            //List<Integer> inteiros = numeros.get(); //
            System.out.println(soma(numeros.get()));
            System.out.println(somaWildCard(inteiros.get()));
      }
}
```

Classes e métodos genéricos

Para iniciarmos vamos a um exemplo muito comum, que mostra como ficaria uma Lista de Objetos com Generics e outra sem Generics.

```java
/* COM GENERICS */
List<Apple> box = ...;
Apple apple = box.get(0);

/* SEM GENERICS */
List box = ...;

/*
Se o objeto retornado de box.get(0) não puder
ser convertido para Apple, só saberemos disso em tempo
de execução
*/
Apple apple = (Apple) box.get(0);
```

Classes e métodos genéricos

De inicio já podemos notar 2 problemas básicos que são encontrados quando optamos por não utilizar Generics:

→ Teremos que fazer um cast para o objeto do tipo Apple toda vez que capturarmos algo da List box;

–> Caso algum erro de cast ocorra, só veremos em tempo de execução, pois este cast só será feito assim que este determinado trecho do código for executado. Diferente do Generics, que o erro é em tempo de compilação, ou seja, já nos deparamos com o erro antes mesmo de tentar executar o projeto, o próprio compilador nos avisará que não é possível atribuir um objeto box ao Apple pois estes são de tipos diferentes, veja um exemplo abaixo que já apresenta erro em tempo de compilação.

```
1  List<Orange> box = ...;
2
3  /*
4  Erro em tempo de compilação pois
5  uma lista de Orange não pode ser atribuido a um
6  objeto do tipo Apple. Isso porque ao fazer "box.get(0)"
7  estamos retornando um Orange e não um Apple.
8  */
9  Apple apple = box.get(0);
```

Classes e métodos genéricos

Generic em Classes e Interfaces

Podemos também utilizar os Generics em Classes ou Interfaces. Estes servem como parâmetro para nossa Classe, assim poderemos utilizar esta “variável” em todo escopo de nossa classe.

```
1 | public interface List<T> extends Collection<T> {
2 | ...
3 | }
```

Esta é a interface da List em Java, perceba que podemos fazer: List pois a interface nos permite isso. A vantagem de fazer isso é o retorno do Objeto quando fazemos um “get”.

```
1 | T get(int index);
```

O método acima irá retornar um objeto do tipo “T” dado determinado index, e quem é T ? Em princípio não sabemos, só vamos descobrir ao implementar a interface. Na listagem 1 o nosso T = Apple.

Sendo assim, o mesmo poderá ser utilizado durante todo desenvolvimento da interface para evitar o uso de castings excessivos. São inúmeras as possibilidades que temos ao se trabalhar com Generics em Classes e Interfaces, muitos problemas que antes seriam resolvidos com horas e até dias de código “sujo”, podem ser resolvidos em apenas algumas linhas, basta ter a habilidade necessária para utilizar tal ferramenta.

Classes e métodos genéricos

Generic em Métodos e Construtores

Usamos um método Genérics em Java:

```
1 | public static <T> getFirst(List<T> list)
```

o método getFirst aceita uma lista do tipo T e retorna um objeto do tipo T.
A iteração em Java (Iterator) também possui Generic, assim um Iterator pode facilmente ser retornado para um novo objeto sem a necessidade do Cast explícito.

```
1 | for (Iterator<String> iter = str.iterator(); iter.hasNext();) {
2 | String s = iter.next();
3 | System.out.print(s);
```

O que você percebe acima é o uso do Generic para transformar o Iterator em iterações apenas de String, assim sempre que fizermos “iter.next()” estamos automaticamente retornando uma String sem precisar fazer “String.valueOf(iter.next())”.

O código também pode ser convertido para um foreach, assim aproveitamos também o recurso de generic que nós é oferecido.

```
1 | for (String s: str) {
2 | System.out.print(s);
3 | }
```

Classes e métodos genéricos

Generic em Métodos e Construtores

Usamos um método Genérics em Java:

```
1 | public static <T> getFirst(List<T> list)
```

o método getFirst aceita uma lista do tipo T e retorna um objeto do tipo T.
A iteração em Java (Iterator) também possui Generic, assim um Iterator pode facilmente ser retornado para um novo objeto sem a necessidade do Cast explícito.

```
1 | for (Iterator<String> iter = str.iterator(); iter.hasNext();) {
2 | String s = iter.next();
3 | System.out.print(s);
```

O que você percebe acima é o uso do Generic para transformar o Iterator em iterações apenas de String, assim sempre que fizermos “iter.next()” estamos automaticamente retornando uma String sem precisar fazer “String.valueOf(iter.next())”.

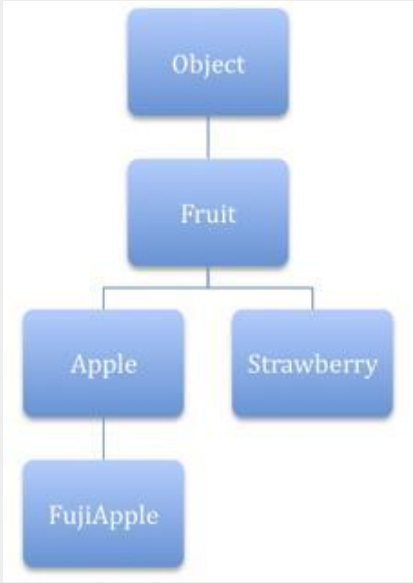
O código também pode ser convertido para um foreach, assim aproveitamos também o recurso de generic que nós é oferecido.

```
1 | for (String s: str) {
2 | System.out.print(s);
3 | }
```

Classes e métodos genéricos

Subtipos Genéricos

Usamos um método Generics em Java:



Significa que um Apple é um Fruit e um Fruit é um Object, então um Apple é um Object também. Analogamente, se A é filho B e B é filho de C, então A é filho de C.

```
1  Apple a = ...;
2  Fruit f = a;
```

```
1  List<Apple> apples = ...;
2  List<Fruit> fruits = apples;
```

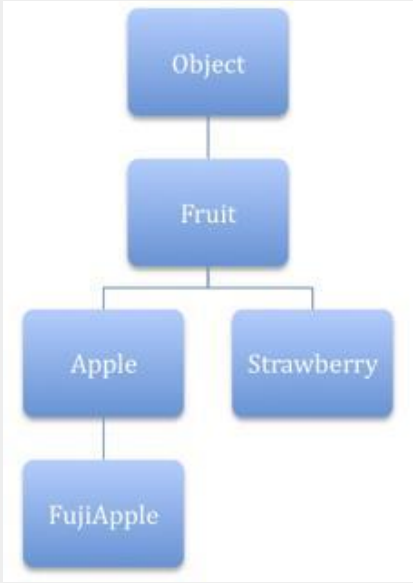
Se seguirmos o raciocínio, o código acima será compilado normalmente sem nenhum erro. Porém isso não é possível no Generics, pois em algumas situações poderia ocorrer uma quebra a consistência da linguagem.
Imagine que você tem o código acima funcionando normalmente (o que não é o caso). Temos então uma caixa de maças, e transformamos nossa caixa de maças em uma caixa de frutas, pois toda maça é fruta, certo ?

Agora já que temos uma caixa de frutas (que sabemos na verdade que ela é uma caixa de maças), o que nos impede de colocar morangos ? Nada.

Classes e métodos genéricos

Subtipos Genéricos

Usamos um método Generics em Java:




```
1 List<Apple> apples = ...;
2 List<Fruit> fruits = apples;
3 fruits.add(new Strawberry());
```

O código acima está logicamente errado, pois antes havíamos atributo a caixa de maças a caixa de frutas, ou seja, nossa atual caixa de frutas na verdade é uma caixa de maças, mas desta forma nada nos impede de colocar um moranga em uma caixa de frutas, isso porque, um morango também é uma fruta. É por este motivo que o Generic não permite esse tipo de atribuição. Podemos resumir isso tudo a apenas uma só frase: Generics são invariantes.

Classes e métodos genéricos

Subtipos Genéricos

Usamos um método Generics em Java:

Não é possível atribuir uma lista de “Apples” a uma lista de “Fruits” mesmo o tipo Apple sendo filho de Fruit, isso porque estaríamos quebrando o “contrato” que diz que Apple só aceita Apple, pois se colocarmos este como Fruit, poderíamos aceitar Morangos, laranjas e etc.

Para isso os Wildcards solucionamos este problema.

Existem 3 tipos de Wildcards em Generics:

- Unknown Wildcard, ou seja, Wildcard desconhecido.
- Extends Wildcard
- Super wildcard

Classes e métodos genéricos

- Unknown Wildcard, ou seja, Wildcard desconhecido.

Como você não sabe o tipo do objeto, você deve tratá-lo da forma mais genérica possível.

```
public void processElements(List<?> elements){
        for(Object o : elements){
           System.out.println(o);
        }
    }

/* Podemos atribuir um list de qualquer tipo a nosso
método, pois ele tem um tipo desconhecido/genérico */
List<A> listA = new ArrayList<A>();

   processElements(listA);
```

Classes e métodos genéricos

- Extends Wildcard

Podemos utilizar este tipo de Wildcard para possibilitar o uso de vários tipos que se relacionam entre si, ou seja, podemos dizer que o nosso método processElements aceita uma lista de qualquer tipo de Frutas, seja moranga, maça ou etc.

```
public void processElements(List<? extends Fruit> elements){
        for(Fruit a : elements){
          System.out.println(a.getValue());
        }
      }

/* Podemos agorar passar nossas frutas diversas ao método processElements */
List<Apple> listApple = new ArrayList<Apple>();
   processElements(listApple);

   List<Orange> listOrange = new ArrayList<Orange>();
   processElements(listOrange);

   List<Strawberry> listStrawberry = new ArrayList<Strawberry>();
   processElements(listStrawberry);
```

Classes e métodos genéricos

- Super Wildcard

AO contrário do extends, o wildcard super permite que elementos Fruit e Object sejam utilizados, isso significa que apenas são permitidos de “Fruit para cima”. Se fizermos um List estamos permitindo todos os Apples, Fruits e Objects.

```
public class Box<T> {

    private T t;

    public void add(T t) {
      this.t = t;
    }

    public T get() {
      return t;
    }

    public static void main(String[] args) {
       Box<Integer> integerBox = new Box<Integer>();
       Box<String> stringBox = new Box<String>();

       integerBox.add(new Integer(10));
       stringBox.add(new String("Hello World"));

       System.out.printf("Integer Value :%d\n\n", integerBox.get());
       System.out.printf("String Value :%s\n", stringBox.get());
    }
  }
```

Integer Value :10

String Value :Hello

Classes e métodos genéricos

- Super Wildcard

AO contrário do extends, o wildcard super permite que elementos Fruit e Object sejam utilizados, isso significa que apenas são permitidos de “Fruit para cima”. Se fizermos um List estamos permitindo todos os Apples, Fruits e Objects.

```
public class Box<T> {

    private T t;

    public void add(T t) {
      this.t = t;
    }

    public T get() {
      return t;
    }

    public static void main(String[] args) {
       Box<Integer> integerBox = new Box<Integer>();
       Box<String> stringBox = new Box<String>();

       integerBox.add(new Integer(10));
       stringBox.add(new String("Hello World"));

       System.out.printf("Integer Value :%d\n\n", integerBox.get());
       System.out.printf("String Value :%s\n", stringBox.get());
    }
  }
```

Classes e métodos genéricos

Motivação para métodos genéricos

Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados. Para motivar os métodos genéricos, começaremos com um exemplo (Figura 20.1) contendo métodos printArray sobrecarregados (linhas 22 a 29, 32 a 39 e 42 a 49) que imprimem as representações String dos elementos de um array Integer, um array Double e um array Character, respectivamente. Poderíamos ter utilizado arrays dos tipos primitivos int, double e char. Utilizamos arrays das classes empacotadoras de tipo para definir nosso exemplo de método genérico, porque somente tipos por referência podem ser usados para especificar os tipos genéricos em métodos e classes genéricos.

```
1  // Figura 20.1: OverloadedMethods.java
2  // Imprimindo elementos do array com métodos sobrecarregados.
3
4  public class OverloadedMethods
5  {
6     public static void main(String[] args)
7     {
8        // cria arrays de Integer, Double e Character
9        Integer[] integerArray = {1, 2, 3, 4, 5, 6};
10       Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
11       Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};
12
13       System.out.printf("Array integerArray contains:%n");
14       printArray(integerArray); // passa um array de Integer
15       System.out.printf("%nArray doubleArray contains:%n");
16       printArray(doubleArray); // passa um array Double
17       System.out.printg("%nArray characterArray contains:%n");
18       printArray(characterArray); // passa um array de Character
19    }
20
21    // método printArray para imprimir um array de Integer
22    public static void printArray(Integer[] inputArray)
23    {
24       // exibe elementos do array
25       for (Integer element : inputArray)
26          System.out.printf("%s ", element);
27
28       System.out.println();
29    }
30
31    // método printArray para imprimir um array de Double
32    public static void printArray(Double[] inputArray)
33    {
34       // exibe elementos do array
35       for (Double element : inputArray)
36          System.out.printf("%s ", element);
37
38       System.out.println();
39    }
40
41    // método printArray para imprimir um array de Character
42    public static void printArray(Character[] inputArray)
43    {
44       // exibe elementos do array
45       for (Character element : inputArray)
46          System.out.printf("%s ", element);
47
48       System.out.println();
49    }
50 } // fim da classe OverloadedMethods
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Figura 20.1 | Imprimindo elementos do array com métodos sobrecarregados.

Classes e métodos genéricos

Motivação para métodos genéricos

```
1  public static void printArray(T [] inputArray)
2  {
3     // exibe elementos do array
4     for (T element : inputArray)
5        System.out.printf("%s ", element);
6
7     System.out.println();
8  }
```

Figura 20.2 | O método `printArray` em que os nomes dos tipos reais são substituídos por um nome de tipo genérico (nesse caso, `T`).

Classes e métodos genéricos

Motivação para métodos genéricos

```
1  // Figura 20.3: GenericMethodTest.java
2  // Imprimindo elementos do array com o método genérico printArray.
3
4  public class GenericMethodTest
5  {
6     public static void main(String[] args)
7     {
8        // cria arrays de Integer, Double e Character
9        Integer[] intArray = {1, 2, 3, 4, 5};
10       Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
11       Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
12
13       System.out.printf("Array integerArray contains:%n");
14       printArray(integerArray); // passa um array de Integers
15       System.out.printf("%nArray doubleArray contains:%n");
16       printArray(doubleArray); // passa um array Doubles
17       System.out.printf("%nArray characterArray contains:%n");
18       printArray(characterArray); // passa um array de Characters
19    }
20
21    // método genérico printArray
22    public static <T> void printArray(T[] inputArray)
23    {
24       // exibe elementos do array
25       for (T element : inputArray)
26          System.out.printf("%s ", element);
27
28       System.out.println();
29    }
30 } // fim da classe GenericMethodTest
```

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O
```

Figura 20.3 | Imprimindo elementos do array com o método genérico `printArray`.