EAD

EDUCAÇÃO A DISTÂNCIA

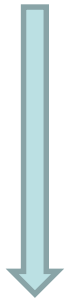
UNIDADE II

Linguagem de Programação Orientada a Objetos

Prof. Me. Ricardo Veras

Promotion e Casting

- A *promotion* é a "elevação" dos tipos numéricos para um tipo que pode representar uma maior grandeza.
- As regras definem que "um valor/uma variável de tipo primitivo de menor capacidade numérica pode ser utilizado no lugar onde seria necessário se utilizar um valor/uma variável de tipo com maior capacidade numérica".
- A promoção não provoca perda de valores. Ela é feita implicitamente (automaticamente) pelo compilador.
- Ordem de *promotion* (do tipo de menor capacidade numérica para o de maior capacidade):



int (32 bits)

long (64 bits)

float (32 bits)

double (64 bits)

Promotion e Casting

Exemplo de *promotion*:

```
...
int f = 4;
long g = 5L + f; //promotion do valor de f para long
float h = 3.4F + g; //promotion do valor de g para float
double i = 2.5 + h; //promotion do valor de h para double
...
```

- Obs.: promove-se o valor da variável, e não a variável.

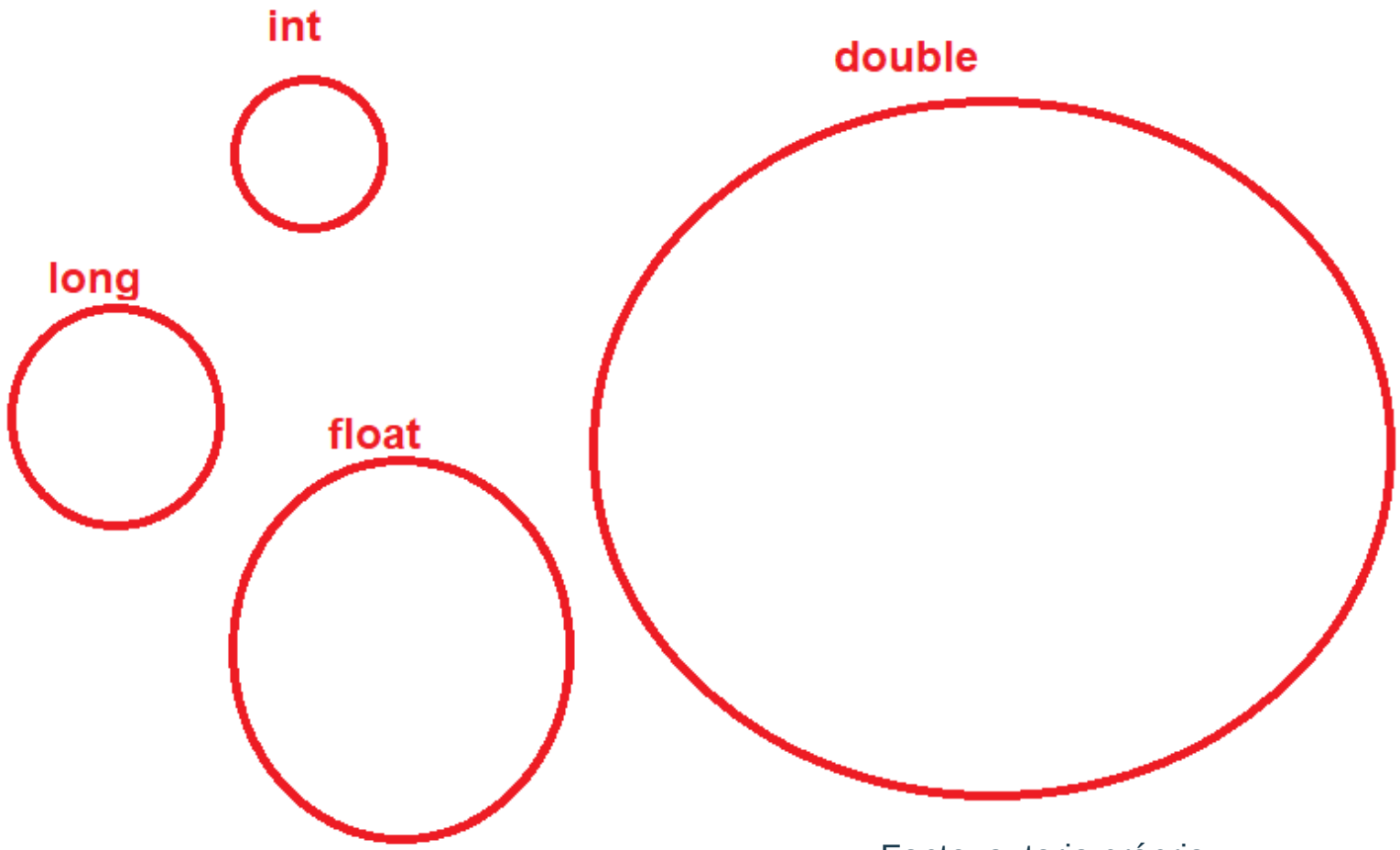
Promotion e Casting

- O *casting* é a mudança de um tipo para outro de forma explícita (forçada) por meio do operador de *casting* (ou seja, o nome do tipo entre parênteses).
- Nesse caso pode haver uma mudança de representação do valor ou não, e nessa conversão poderá haver perda de valor ou de precisão.
- Quando ocorre algum tipo de "erro" nesse sentido, o compilador pode detectar a impossibilidade da operação durante a compilação (não realizando a compilação). Do contrário, não detectando, poderá ocorrer um erro em tempo de execução, se não for possível realizar a conversão do valor.

Promotion e Casting

Representação explicativa dos tipos:

- Imagem apenas representativa para exemplificar “qual tipo caberia no outro tipo”:



Fonte: autoria própria.

Promotion e Casting

Exemplo de *casting*:

```
...
double j = 2.5;
float k = 5.2F + (float)j;
long l = 3L + (long)k;
int m = 7 + (int)l;
short n = (short)m;
byte o = (byte)n;
...
```

- Obs.: do mesmo modo que a promoção, o *casting* ocorre sobre o valor da variável, e não sobre a variável.

ArrayList e Vector

- ArrayList é uma forma de termos uma variável que contenha vários objetos (um ou mais objetos), a partir de um índice de objetos.

- Os ArrayLists são utilizados para alocar dinamicamente uma coleção de elementos (de objetos), ou seja, é um array de objetos.
- Eles permitem dados duplicados e nulos.
- Não é possível criar ArrayLists de tipos primitivos (int, char etc.).

  - O ArrayList é implementado como um Array, mas que é dimensionado dinamicamente (adiciona-se sempre que for necessário e seu tamanho vai aumentando proporcionalmente).
  - O ArrayList permite que seus elementos sejam: acessados diretamente pelo método get(), adicionados por meio do método add() e removidos pelo método remove().

ArrayList e Vector

- Vector e ArrayList são muito semelhantes na sua utilização.

- A diferença aparece quando trabalhamos com *Threads* (elemento que será visto mais à frente neste curso).
- Nesse caso, vários *Threads* podem operar em um ArrayList ao mesmo tempo (e por isso este é considerado não sincronizado).
- No entanto apenas um único *Thread* pode operar em um Vector por vez (e por isso é considerado sincronizado).

ArrayList e Vector

Exemplo geral:

- Imagine uma classe qualquer, sobre a qual queremos gerar uma série de objetos com essa mesma classe e guardá-los numa matriz de objetos:

```
public class ClasseTipo {
	public int atrib01;
	public String atrib02;
	public double atrib03;
	public ClasseTipo(int i, String s, double d) {
		atrib01 = i;
		atrib02 = s;
		atrib03 = d;
	}
	public void mostraCT() {
		System.out.println(atrib01 + ": " + atrib02
		+ " (" + atrib03 + ")");
	}
}
```

ArrayList e Vector

Exemplo geral (continuação) – com ArrayList:

- Numa outra classe (em seu método main) podemos criar uma matriz com vários objetos do tipo ClasseTipo:

```
ArrayList<ClasseTipo> conjunto = new ArrayList();

ClasseTipo ct;
ct = new ClasseTipo(1, "Primeiro", 25.6);
conjunto.add(ct);
ct = new ClasseTipo(2, "Segundo", 25.7);
conjunto.add(ct);
ct = new ClasseTipo(3, "Terceiro", 25.8);
conjunto.add(ct);
ct = null;

int tam = conjunto.size();

for (int i = 0; i < tam; i++) {
        conjunto.get(i).mostraCT();
}
```

ArrayList e Vector

Exemplo geral (continuação) – com Vector:

- Numa outra classe (em seu método main) podemos criar uma matriz com vários objetos do tipo ClasseTipo:

```
Vector<ClasseTipo> conjunto = new Vector();

ClasseTipo ct;
ct = new ClasseTipo(1, "Primeiro", 25.6);
conjunto.add(ct);
ct = new ClasseTipo(2, "Segundo", 25.7);
conjunto.add(ct);
ct = new ClasseTipo(3, "Terceiro", 25.8);
conjunto.add(ct);
ct = null;

int tam = conjunto.size();

for (int i = 0; i < tam; i++) {
        conjunto.get(i).mostraCT();
}
```

Modificadores de acesso

Em Java, temos quatro modificadores de acesso:

- public
- private
- protected
- (default) – ou sem modificador

Modificadores de acesso

public (público):

- Uma declaração (classes, atributos ou métodos) com o modificador *public* pode ser acessada de qualquer lugar e por qualquer entidade (objeto) que possa visualizar a classe a que esse elemento pertence.

private (privado):

- Uma declaração (atributos ou métodos) com o modificador *private* não pode ser acessado ou usado por nenhuma outra classe, mas apenas por métodos da própria classe. Esses atributos e métodos também não podem ser diretamente visualizados pelas classes herdadas.

Modificadores de acesso

protected (protegido):

- O modificador *protected* torna o elemento (atributos ou métodos) acessível às classes do mesmo pacote ou pela herança. Seus elementos não são acessíveis a outras classes fora do pacote em que foram declarados.

default (padrão):

- Uma declaração (classes, atributos ou métodos) em que não há definição de modificador possui o que chamamos de modificador padrão (*default*), fazendo com que sejam acessíveis somente por classes do mesmo pacote.

Modificadores de acesso

Níveis de acesso:

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`no modifier`	Y	Y	N	N
`private`	Y	N	N	N

Tabela de relação de acesso dos Modificadores (adaptado)

Fonte: Site do Tutorial de Java da Oracle:
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Interatividade

Em um programa existe a seguinte linha de código:

ArrayList<Cliente> clientes = new ArrayList();

O que significa o termo descrito entre os símbolos “<” e “>” desse código?

a) Significa que estamos trabalhando com os clientes dos clientes de uma empresa.
b) Significa que cliente deverá ser gerado a partir de uma lista de nomes de clientes.
c) Significa que a matriz cliente é um ArrayList.
d) Significa que a variável cliente poderá receber todos os clientes da empresa.
e) Significa que esse ArrayList representará uma matriz de objetos que foram gerados a partir da classe cliente, existente para o sistema.

Resposta

Em um programa existe a seguinte linha de código:

ArrayList<Cliente> clientes = new ArrayList();

O que significa o termo descrito entre os símbolos “<” e “>” desse código?

a) Significa que estamos trabalhando com os clientes dos clientes de uma empresa.
b) Significa que cliente deverá ser gerado a partir de uma lista de nomes de clientes.
c) Significa que a matriz cliente é um ArrayList.
d) Significa que a variável cliente poderá receber todos os clientes da empresa.
e) Significa que esse ArrayList representará uma matriz de objetos que foram gerados a partir da classe cliente, existente para o sistema.

Encapsulamento

- Encapsular uma classe é bloquear a possibilidade de, a partir de uma outra classe, alterar os valores dos atributos de forma direta (obj.atrib = valor), ou até de se pegar (ler) os valores também de forma direta (obj.atrib).

- Assim, essas ações devem ficar a cargo dos métodos da própria classe (encapsulada), possibilitando o controle das alterações de seus valores.

- Assim, no encapsulamento, os atributos das classes devem ficar privados (*private*) e devem ser gerados os métodos *Setters* e *Getters*.

Encapsulamento

- Exemplo de um encapsulamento:

Suponha a seguinte classe ainda não encapsulada:

```
public class Pessoa {
    public String nome;
    public int idade;
    public double altura;
}
```

Assim, para se realizar o encapsulamento dessa classe, pode-se seguir os seguintes passos:

- Deixar os modificadores de acesso privados;
- Criar os métodos *Setters* e *Getters.*

Encapsulamento

- Após o encapsulamento, tem-se a seguinte classe:

```
public class Pessoa {
	private String nome;
	private int idade;
	private double altura;
	//.....................................
	public void setNome(String nome) {
	    this.nome = nome;
	}
	public String getNome() {
	    return nome;
	}
	public void setIdade(int idade) {
	    this.idade = idade;
	}
	public int getIdade() {
	    return idade;
	}
	public void setAltura(double altura) {
	    this.altura = altura;
	}
	public double getAltura() {
	    return altura;
	}
}
```

Encapsulamento

- Nesse caso, poderíamos por exemplo controlar (por meio de lógica de programação) o valor que é inserido no atributo idade:

```
public class Pessoa {
      ...
      private int idade;
      ...
      public void setIdade(int idd) {
         if (idd < 0 || idd > 150) {
            this.idade = 0;
         } else {
            this.idade = idd;
         }
      /* Nesta lógica, foi determinado que não se poderá
      inserir nenhum valor menor que 0(zero) e nenhum
      valor maior que 150 (zerando o atributo)*/
      }
      ...
}
```

Sobrecarga de métodos

A assinatura de um método:

- A assinatura de um método é dada por:

  Nome_do_Método + (parâmetros)

- Assim, na declaração de métodos, temos:

```
modificador(es) tipoRetorno nomeMetodo(Parâmetros) {
...                          assinatura do método
}
```

  - Não pode haver 2 métodos com mesma assinatura numa mesma classe.

Sobrecarga de métodos

- Assim, quando temos, numa mesma classe, dois ou mais métodos com o mesmo nome, todos devem estar diferenciados em seus parâmetros (já que não pode existir dois métodos com mesma assinatura numa mesma classe).
- Tem-se com isso caracterizada a sobrecarga desse método.

Sobrecarga de métodos (*overloading*):

- É a existência de mais de um método com mesmo nome em uma mesma classe (ou em classes herdadas), diferenciados em seus parâmetros (na quantidade ou nos tipos dos parâmetros).

Sobrecarga de métodos

Exemplo de sobrecarga de método:

```
public class Calculadora {
	...
	public int somar(int a, int b) {
	    ...
	}
	public double somar(double a, double b) {
	    ...
	}
	public int somar(int a, int b, int c) {
	    ...
	}
	public int somar(int a, int b, double c) {
	    ...
	}
}
```

Métodos construtores

- Um Método Construtor (MC) é um método que é acionado (apenas) no momento em que a classe está sendo instanciada.
- Esse método é responsável por "construir" a classe em memória, ou seja, por gerar o objeto (que irá representar a classe) em memória para que possa ser utilizado.
- Características dos MCs:
  - Possuem (exatamente) o mesmo nome da classe;
  - São públicos (*public*);
  - Não possuem indicação alguma de retorno de valor;
    - Sua sintaxe é: `public NomeDaClasse(parâmetros) {...}`;
    - Uma vez determinado numa classe um ou mais MCs, para que se possa instanciar essa classe, deve-se utilizar um dos MCs determinados;
    - Uma classe pode possuir mais de um MC criado (caracterizando, nesse caso, uma sobrecarga de métodos construtores).

Métodos construtores

- Exemplo de Métodos Construtores (MCs):

```
public class Pessoa {

      private String nome;
      private int idade;
      private double altura;
      //...
      public Pessoa(String s) {
          nome = s;
      }
      public Pessoa(String s, int i, double d) {
          nome = s;
          idade = i;
          altura = d;
      }
      //... e mais os métodos Setters e Getters
      //...
}
```

MC_1

MC_2

Métodos construtores

- Caso tenhamos uma classe externa instanciando a classe pessoa, então:

```
public class Teste {

      public static void main (String[] args) {
          Pessoa p1 = new Pessoa("Amauri");
          //... esta instância (p1) já será criada
          //... em memória com o nome Amauri

          //:: Desta forma, a seguinte instância não pode
          //:: ...ser feita (portanto gera um erro):
          //Pessoa p2 = new Pessoa();
          //:: ...já que não existe este método
          //:: construtor na Classe
      }

}
```

Interatividade

Sabendo-se que a classe carro é uma classe encapsulada, que contém o atributo “velocidade” (do tipo double) e que a mesma foi instanciada com o objeto “c2”, qual das opções abaixo mostra o comando que insere corretamente um valor naquele atributo da classe?

a) velocidade = 80.5;

b) c2.velocidade = 80.5;

c) c2.setVelocidade(80.5);

d) setVelocidade = 80.5;

e) c2.getVelocidade(80.5);

Resposta

Sabendo-se que a classe carro é uma classe encapsulada, que contém o atributo “velocidade” (do tipo double) e que a mesma foi instanciada com o objeto “c2”, qual das opções abaixo mostra o comando que insere corretamente um valor naquele atributo da classe?

a) velocidade = 80.5;

b) c2.velocidade = 80.5;

c) c2.setVelocidade(80.5);

d) setVelocidade = 80.5;

e) c2.getVelocidade(80.5);

Herança

- A orientação a objetos é baseada em alguns conceitos elementares e característicos (princípios) desse paradigma (os três pilares da O.O. – herança, polimorfismo e encapsulamento).

Um desses conceitos é o de herança, que permite:

- Reaproveitamento de código ou de características.
- Generalização e especificação de classes.
- Criação de novas classes a partir de outras.
- Criação de uma hierarquia de classes.
- Extensão da definição de uma classe.

Herança

- A herança é um princípio de O.O. que permite a criação de novas classes a partir de outras previamente criadas.
- Essas novas classes são chamadas de subclasses (são as classes filhas).
- As classes já existentes (que deram origem às subclasses) são chamadas de superclasses (são as classes mãe).
- Deste modo, gera-se uma hierarquia de classes, criando assim classes mais genéricas (superclasse) e classes mais específicas (subclasse).
- A subclasse herdará todos os métodos e todos os atributos da superclasse.

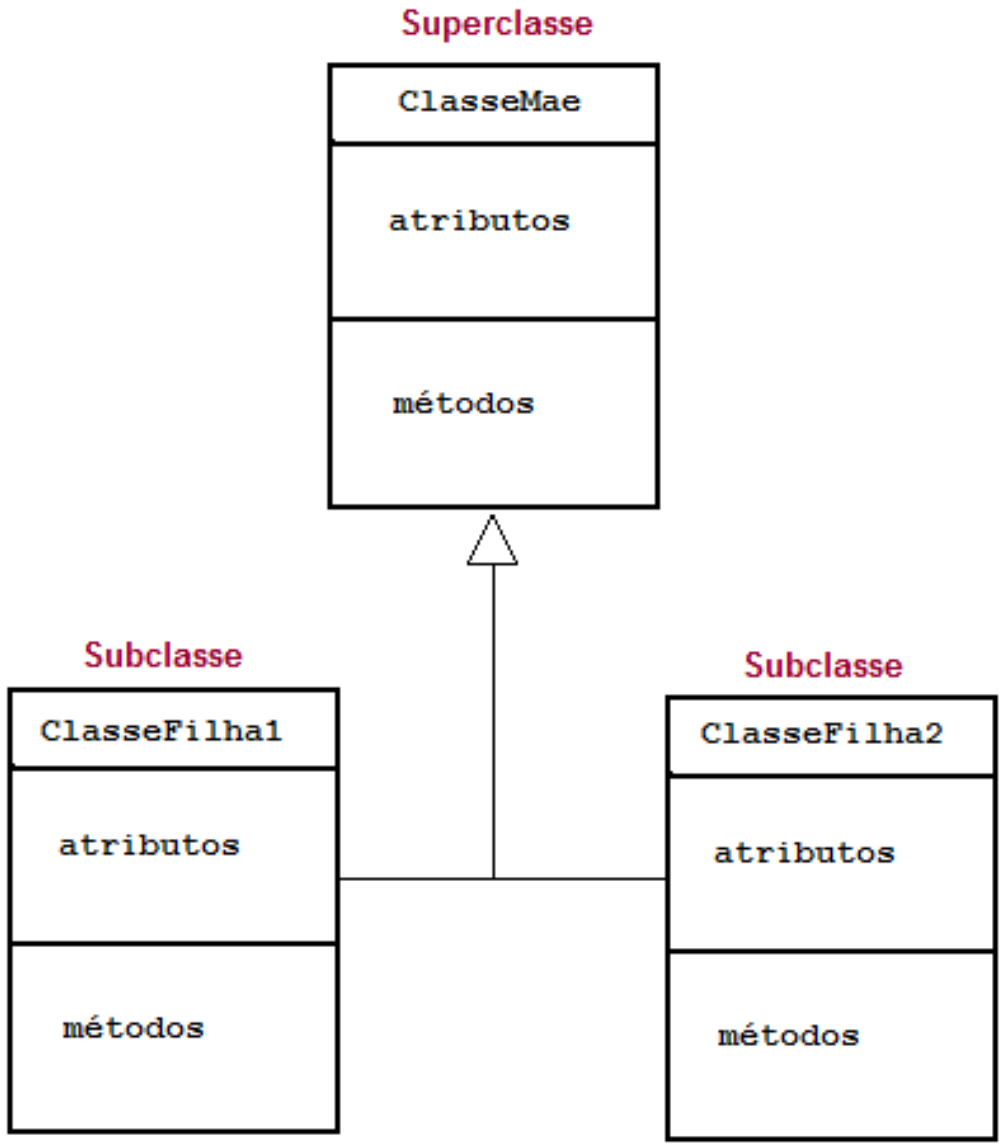
Herança

Características da herança:

- Uma classe filha pode herdar apenas uma classe mãe (já que em Java não há herança múltipla).
- A classe filha possuirá todos os atributos e métodos da classe mãe (independentemente do modificador de acesso que cada um deles tiver).
- Para que uma classe herde outra, inserimos na declaração da classe a palavra *extends* e o nome da classe que ela está herdando (classe mãe).
- Conceito: quando uma classe herda outra classe, ela passa a **ser** esta outra classe (“a classe filha **é** a classe mãe”).

Herança

Representação (diagrama de classes – UML):



Fonte: autoria própria.

Herança

Exemplo de herança:

- Imagine a classe pessoa encapsulada do exemplo dado anteriormente:

```
public class Pessoa {

	private String nome;
	private int idade;
	private double altura;

	public Pessoa(String s) {
	    nome = s;
	}

	/*
	... e mais os métodos Setters e Getters
	*/
}
```

Herança

Exemplo de herança (continuação):

- Vamos criar agora a classe aluno e, ao invés de repetirmos os dados da classe pessoa, vamos fazer com que ela herde (“extenda”) a classe pessoa (pelo conceito: aluno é pessoa).

```
public class Aluno extends Pessoa {

	private String ra;
	private String curso;
	private String turma;

	public Aluno(String s) {
	    //this.nome = s  //... neste caso não pode
	    this.setNome(s);
	}
	//... Setters e Getters dos atributos de Aluno
}
```

Herança

Exemplo de herança (continuação):

- Agora vamos instanciar a classe aluno e popular (inserir valores em seus atributos):

```
public class TesteAluno {

    public static void main(String[] args) {
       //.. Instanciando a Classe Aluno (já com um nome)
       //.. utilizando seu MC
       Aluno a1 = new Aluno("José");
       a1.setIdade(20);
       a1.setAltura(1.73);
       a1.setRa("C26H32-8");
       a1.setCurso("Ciência da Computação");
       a1.setTurma("1A");
    }

}
```

Herança

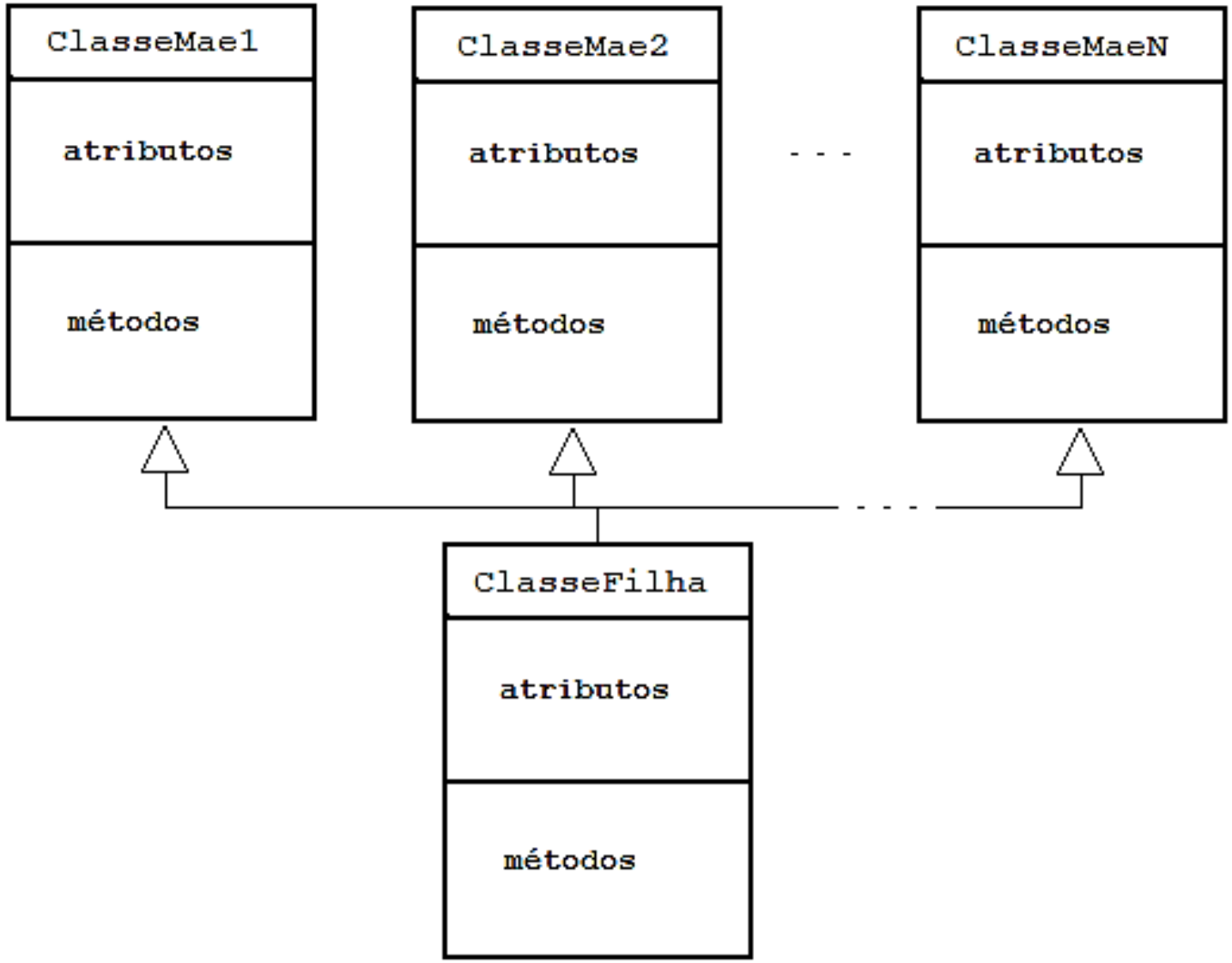
- A linguagem Java não permite herança múltipla (como permitem outras linguagens que seguem o mesmo paradigma – O.O. – : C++ e Python).

- No entanto, o Java permite o que chamamos de herança encadeada, em que uma classe herda outra classe, que por sua vez herda outra classe (...e assim por diante).

- Obs.: se a ClasseA herda a ClasseB, e a ClasseB herda a ClasseC, então por encadeamento de heranças, a ClasseA herdará também a ClasseC.

Herança

Herança múltipla (C++ ou Python):



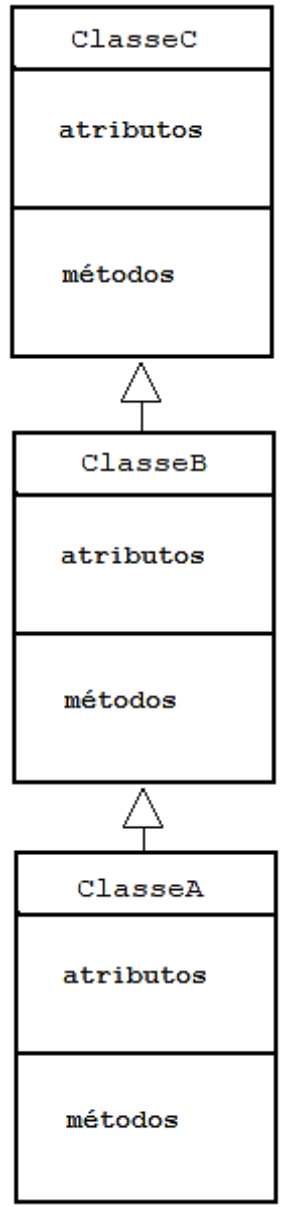
Fonte: autoria própria.

Herança

Herança encadeada (Java):

- No exemplo ao lado, a ClasseA herda a ClasseB, que herda a ClasseC.
- Assim, a ClasseA terá todas as características de B, e todas as de C (e das classes superiores à ClasseC, caso existam).

- Obs.: no Java, a "*classe mãe de todas as classes*" é a classe "Object". Quando criamos uma classe em Java, mesmo que não explicitemos uma herança (com a palavra *extends*), ela automaticamente irá herdar a classe Object.



Fonte: autoria própria.

Sobrescrita de métodos

- No Java, quando uma classe herda outra classe, ela herda seus atributos e seus métodos. Podemos alterar o comportamento desses métodos herdados na classe filha, criando-os na classe filha com a mesma assinatura, deixando-os mais específicos (ou seja, mais de acordo com as características da classe filha em que se localizam).

- Podemos então reescrever (sobrescrever) esse método, de modo que se o invocarmos, estaremos invocando o método reescrito (e não o método inicial descrito na classe mãe).

    - Depois de reescrito, não é mais possível chamar diretamente, para um objeto gerado da classe filha, o método antigo (que pertence à classe mãe).

    - Obs.: é possível invocar o método da classe mãe, mas apenas de dentro da classe filha (utilizando a palavra *super*).

Sobrescrita de métodos

- Sobrescrita de métodos (*overriding*) acontece quando temos dois métodos com exatamente a mesma assinatura.

- Isso não pode ocorrer em uma mesma classe, mas ocorre em herança, entre classe mãe e classe filha, quando a classe filha herda um método da classe mãe, porém necessita de um comportamento mais adequado às suas características.

Sobrescrita de métodos

Exemplo de sobrescrita de métodos:

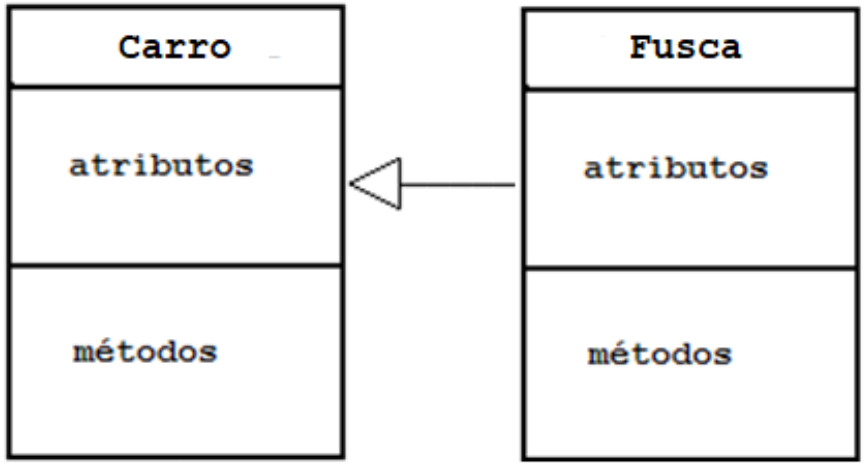
```
public class Mae {
	public void escreve() {
	    System.out.println("texto1");
	}
}
```

```
public class Filha extends Mae {
	public void escreve() {
	    System.out.println("texto2 da Classe Filha");
	}
}
```

```
public class Teste {
	public static void main(String[] args) {
	    Mae m = new Mae();
	    Filha f = new Filha();
	    m.escreve(); // ... texto1
	    f.escreve(); // ... texto2 da Classe Filha
	}
}
```

Interatividade

De acordo com a imagem abaixo, qual das classes é a subclasse e qual deve ser a declaração dessa classe?



Fonte: autoria própria.

a) A subclasse é a classe Fusca e sua declaração é: public class Fusca extends Carro {...}

b) A subclasse é a classe Carro e sua declaração é: public class Carro extends Fusca {...}

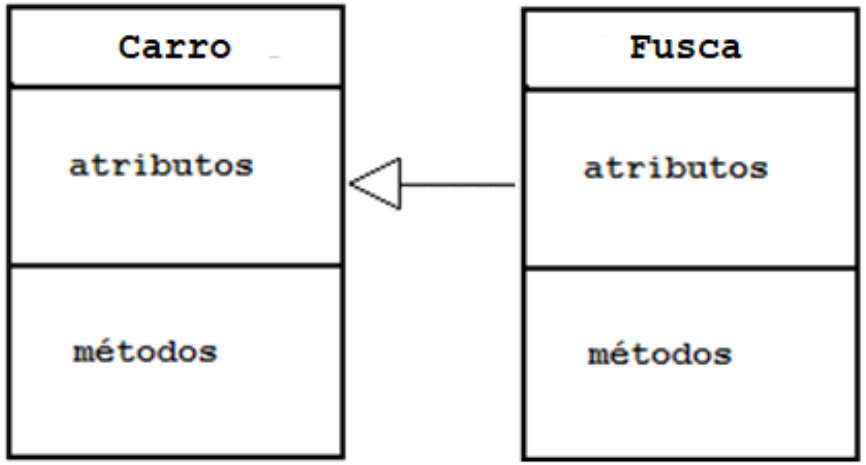
c) Não há subclasse e, portanto, não se pode declarar.

d) A subclasse é a classe Fusca e sua declaração é: public class Fusca getCarro {...}

e) A subclasse é a classe Carro e sua declaração é: public class Carro getFusca {...}

Resposta

De acordo com a imagem abaixo, qual das classes é a subclasse e qual deve ser a declaração dessa classe?



Fonte: autoria própria.

a) A subclasse é a classe Fusca e sua declaração é: public class Fusca extends Carro {...}

b) A subclasse é a classe Carro e sua declaração é: public class Carro extends Fusca {...}

c) Não há subclasse e, portanto, não se pode declarar.

d) A subclasse é a classe Fusca e sua declaração é: public class Fusca getCarro {...}

e) A subclasse é a classe Carro e sua declaração é: public class Carro getFusca {...}

Trabalhando com *Strings*

- A classe *String* representa os tipos de dados alfanuméricos como palavras, frases e textos em geral. Na linguagem Java, esse tipo de dado não é um tipo primitivo, de forma que essa classe contém uma série de métodos que permitem a manipulação desse conjunto de caracteres de diversas formas.
- Podemos, a partir dos métodos que pertencem à classe *String*:
  - verificar seu tamanho (a quantidade de caracteres que possui);
  - dividi-la em partes;
  - alterar uma parte dela substituindo por outra;
  - verificar se existe um texto específico dentro dela;
  - descobrir em que posição está;
  - entre outras ações.

  Esses métodos são úteis quando vamos trabalhar com organização ou transferência de informações, mineração de dados, inteligência artificial, aprendizado de máquina (*machine learning*), análise de sentimentos e muito mais.

Trabalhando com *Strings*

O método length()

- Esse método retorna um valor do tipo *int* que representa o tamanho de uma string, ou seja, a quantidade de caracteres que a *string* possui, lembrando que, por caracteres, estão compreendidos números, letras maiúsculas, letras minúsculas, símbolos (@, !, #, $ ...) e o espaço.

Sua sintaxe é:

```
<nome_da_String>.length()
```

Exemplo:

```
...
String nome = "Fulano de Tal";
int tam = nome.length(); // no caso: tam = 13
...
```

- Nesse exemplo, a variável “tam” receberá o tamanho da variável “nome” (que é 13), já que o objeto “nome” representa uma String (é um objeto do tipo String).

Trabalhando com *Strings*

Os métodos toUpperCase() e toLowerCase()

toUpperCase(): retorna o texto inteiramente em maiúsculo (CAIXA-ALTA);

toLowerCase(): retorna o texto inteiramente em minúsculo (caixa-baixa).

É importante saber que esses dois métodos não alteram o valor da variável, mas sim retornam o valor equivalente à sua característica. Sua sintaxe é:

```
<nome_da_String>.toUpperCase()
<nome_da_String>.toLowerCase()
```

Exemplo:

```
String nome = "Fulano de Tal";
String txt1 = nome.toUpperCase();
// no caso, txt1 recebe: FULANO DE TAL
String txt2 = nome.toLowerCase();
// no caso, txt2 recebe: fulano de tal
// Importante: a variável "nome" continua com o
// seu valor inicial: Fulano de Tal
```

Trabalhando com *Strings*

O método trim()

- Esse método retorna o texto original (um valor do tipo string) no qual são removidos todos os espaços em branco que aparecem no início e no final do texto original (esse método não altera o valor da variável original, mas apenas retorna o valor alterado).
- Obs.: os espaços internos, mesmo que duplicados, continuarão intactos (sem alteração).

Sua sintaxe é: `<nome_da_String>.trim()`

Exemplo:

```
String texto = "   um    texto qualquer    ";
String txt = texto.trim();
System.out.println("[" + txt + "]");
// imprimirá: [um    texto qualquer]
System.out.println("[" + texto + "]");
// imprimirá: [   um    texto qualquer    ]
```

- Obs.: o texto inicial contém mais de um espaço no início e no fim. Na variável txt, esses espaços iniciais e finais foram retirados, mas não alterou os espaços internos.

Trabalhando com *Strings*

O método charAt(...)

- Como na linguagem Java, uma string não é um array de caracteres. Assim, para acessarmos um determinado caractere no interior da string, utiliza-se o método charAt(...) que permite que acessemos um caractere numa determinada posição do interior de uma string. A posição desejada é um valor numérico inteiro (começando com o valor 0) e que deve ser indicado no parâmetro do método. Esse método retornará um valor do tipo char que representará o caractere existente na posição indicada no parâmetro.

Sua sintaxe é: `<nome_da_String>.charAt(<num_posicao>)`

Exemplo:

```
String texto = "palavra";
char c1 = texto.charAt(0); // c1 recebe 'p'
char c2 = texto.charAt(6); // c2 recebe 'a' (o último 'a')
char c3 = texto.charAt(8);
// este último resulta em ERRO pois é uma posição
// inexistente, gerando uma exceção do tipo
// "StringIndexOutOfBoundsException"
```

Trabalhando com *Strings*

O método indexOf(...)

- Utilizamos esse método quando precisamos procurar um trecho de String (uma substring) ou até uma letra dentro da String. O método retornará um número inteiro equivalente ao valor da posição inicial da substring procurada. É importante ressaltar que caso haja repetição da substring, ou da letra, no texto original, o método indexOf() retornará a posição apenas do primeiro caso encontrado do trecho a partir da posição inicial (que pode ser o próprio valor dessa posição, dependendo do caso).

Sua sintaxe é: <nome_da_String>.indexOf(<substring>)

ou:

<nome_da_String>.indexOf(<substring>, <posic_inicial>)

Trabalhando com *Strings*

O método indexOf(...) – Exemplo

- O exemplo a seguir mostra a posição de todas as letras “a” que aparecem no texto inicial, utilizando-se de um laço de repetição, sempre renovando a posição inicial de busca da substring (no caso, a posição da letra a‘):

```
String texto = "aa batata";
int pos = texto.indexOf("a");
while (pos >= 0) {
    System.out.print(pos + "\t");
    pos = texto.indexOf("a", pos + 1);
}
// imprimirá: 0   1     4     6     8
// Porém:
System.out.print(texto.indexOf("w"));
// Imprimirá: -1
// pois esta String "w" não existe no texto original
```

Trabalhando com *Strings*

O método replace(...)

- Esse método retorna um valor do tipo String, de modo que toda ocorrência encontrada com o trecho indicado no parâmetro (substring procurada) será substituído pela nova substring. A variável original não sofre a alteração, ou seja, o método replace() retorna um valor alterado sem alterar o valor da variável original. Caso o método não encontre a substring procurada, ele não resulta em erro, simplesmente retornando o valor original sem alterações.

Sua sintaxe é: `<nome_da_String>.replace(<substr_procurada>, <nova_substr>)`

Exemplo:

```
String texto = "palavra";
String txt = texto.replace("a", "x");
// txt receberá: pxlxvrx
// onde TODA ocorrência da substring "a" foi alterada,
// de modo que o valor da variável "texto"
// continua sendo "palavra".
txt = texto.replace("k", "x");
// neste caso txt receberá: palavra
// já que a substring "k" não existe na String original.
```

Trabalhando com *Strings*

O método substring(...)

- Esse método retorna um valor do tipo string equivalente a um trecho da string original. Os parâmetros desse método devem receber um número inteiro equivalente ao índice relativo à posição inicial e à posição final+1 (este último, opcional).

Sua sintaxe é:

```
<nome_da_String>.substring(<posic_inicial>)
ou:
<nome_da_String>.substring(<posic_inicial>, <posic_final>)
```

- No primeiro caso, só com o valor da posição inicial, o texto resultante equivale ao trecho que inicia no índice indicado e termina no final da string original.
    - No segundo caso, o valor da posição inicial equivalerá ao índice exato do caractere que fará parte da substring resultante, e a posição final indica que a substring resultante terá todos os caracteres até um índice anterior ao valor indicado (vide exemplo a seguir).

    Obs.: a primeira posição é a posição 0(zero).

Trabalhando com *Strings*

O método substring(...)

- É importante salientar que caso um dos valores de posição (seja o inicial ou o final) não exista no texto original (lembrando que o valor final pode sempre ser um valor acima do valor do último índice), esse comando resultará em um erro, interrompendo a execução do código e provocando o que chamamos de exceção (do tipo StringIndexOutOfBoundsException).

Exemplo:

```
String texto = "um texto qualquer";
txt = texto.substring(3);
// txt receberá: "texto qualquer"
// já que na posição 3 está a
// primeira letra t da palavra "texto"
txt = texto.substring(3, 10);
// txt receberá: "texto q"
// já que no índice 9 (um antes do valor 10) temos
// a primeira letra "q" da palavra "qualquer".
txt = texto.substring(3, 30);
// Este comando resulta em ERRO,
// já que a posição 30 não existe na String original.
```

Trabalhando com *Strings*

O método split(...)

- Esse método é muito utilizado em leitura de arquivos de texto “externos ao programa”, a fim de identificar partes de seu texto que geralmente contêm dados específicos sobre configurações ou de banco de dados.
- Esse método retorna um array de strings (uma matriz de strings) que contém, em cada uma de suas posições, trechos (substrings) do texto original "recortado em partes". Esse recorte é realizado segundo uma referência, ou, ainda, uma substring, indicada no parâmetro do método.
- O fato de retornar um array facilita sua manipulação, já que podemos ler um array utilizando uma estrutura for (vide exemplo a seguir).

Sua sintaxe é: `<nome_da_String>.split(<referencia>)`

Trabalhando com *Strings*

O método split(...)

Quanto à substring de referência, podemos ter três situações:

- a) caso essa substring seja encontrada: nesse caso, ele gerará um array com todas as substring separadas pela referência, na ordem que elas aparecem (observação: a string de referência é retirada ao gerar as substrings – entende-se que esse valor é apenas uma referência para a identificação de separação das substrings);
- b) caso essa substring não seja encontrada: nesse caso, ele gerará um array com apenas um elemento, que é o próprio texto completo da string original;
- c) caso seja uma String vazia (""): nesse caso, retornará um array de strings em que cada um dos caracteres da String original ocupará uma posição no array resultante, de modo que o tamanho do array será igual ao tamanho da string original.

Trabalhando com *Strings*

- Exemplo do método split(...)

```
String texto = "um texto qualquer";
String[] matStr = texto.split("");
for (int x = 0; x < matStr.length; x++) {
     System.out.println(matStr[x]);
}
// O laço acima irá imprimir letra a letra
// do texto original, uma abaixo da outra:
matStr = texto.split(" ");
for (int x = 0; x < matStr.length; x++) {
     System.out.println(matStr[x]);
}
// O laço acima imprime palavra por palavra
matStr = texto.split("w");
for (int x = 0; x < matStr.length; x++) {
     System.out.println(matStr[x]);
}
// O laço acima imprime o texto original
// ...já que a referência não existe no texto original.
```

Trabalhando com *Strings*

O método equals(...)

- Quando uma string for instanciada como se fosse uma classe, as comparações realizadas com o operador == não funcionarão como funcionam com as strings criadas na forma literal.

```
// criando uma String na forma literal (txt1)
String txt1 = "qualquer coisa";
// criando uma String na forma de Classe (txt2 e txt3)
String txt2 = new String("qualquer coisa");
String txt3 = txt2.substring(2);
```

Trabalhando com *Strings*

O método equals(...)

Exemplo:

```
String txt1 = "qualquer coisa";
String txt2 = "qualquer coisa";
if (txt1 == txt2) {
  System.out.println("São iguais");
} else {
  System.out.println("São diferentes");
}
// Neste caso, como ambas foram criadas de forma literal
// a estrutura condicional aceitará a comparação com o
// operador ==, e imprimirá que "São iguais".
// Obs.: Isto também funcionará para a situação
// de comparação com o método equals(...).
```

Trabalhando com *Strings*

O método equals(...)

Exemplo:

```
String txt1 = "qualquer coisa";
String txt2 = "qualquer coisa";
String txt3 = new String("qualquer coisa");
String txt4 = new String("qualquer coisa");
if (txt1 == txt2) System.out.println("Iguais – comp. 1");
if (txt1.equals(txt2)) System.out.println("Iguais – comp. 2");
if (txt1 == txt3) System.out.println("Iguais – comp. 3");
if (txt3 == txt4) System.out.println("Iguais – comp. 4");
if (txt1.equals(txt3)) System.out.println("Iguais – comp. 5");
if (txt3.equals(txt4)) System.out.println("Iguais – comp. 6");
// Nestes casos acima, as únicas saídas serão:
// Iguais – comp. 1
// Iguais – comp. 2
// Iguais – comp. 5
// Iguais – comp. 6
// As comparações com o operador == só funcionarão se AS DUAS variáveis
tiverem sido criadas na sua forma literal.
```

Interatividade

O que será impresso na tela da console, após a execução do código abaixo:

```
String txt0 = "Exercicio de String com java";
String[] matTxt = txt0.split(" ");
String txt1 = matTxt[2].substring(1, 4);
System.out.println(txt1);
```

a) Exercicio com
b) de S
c) de java
d) tri
e) Stri

Resposta

O que será impresso na tela da console, após a execução do código abaixo:

```
String txt0 = "Exercicio de String com java";
String[] matTxt = txt0.split(" ");
String txt1 = matTxt[2].substring(1, 4);
System.out.println(txt1);
```

a) Exercicio com
b) de S
c) de java
d) tri
e) Stri

ATÉ A PRÓXIMA!