Pergunta 1 (1 ponto)

Questão 8) Vimos, durante nossos estudos, que com a sobrecarga de métodos podemos ter em uma mesma classe diversos métodos de mesmo nome e diferentes assinaturas. A sobrecarga, no entanto, não se resume aos métodos de uma única classe. Quando usamos a herança, métodos de uma superclasse são herdados pelas subclasses, e podemos declarar novos métodos na subclasse que tem o mesmo nome e diferentes assinaturas dos métodos herdados, criando assim a "sobrecarga" na herança. Quando a assinatura do método na subclasse coincide com a assinatura do método da superclasse, no entanto, não temos aqui uma "sobrecarga" e sim uma "sobreposição" de métodos. O código abaixo mostra um exemplo de utilização desses recursos:

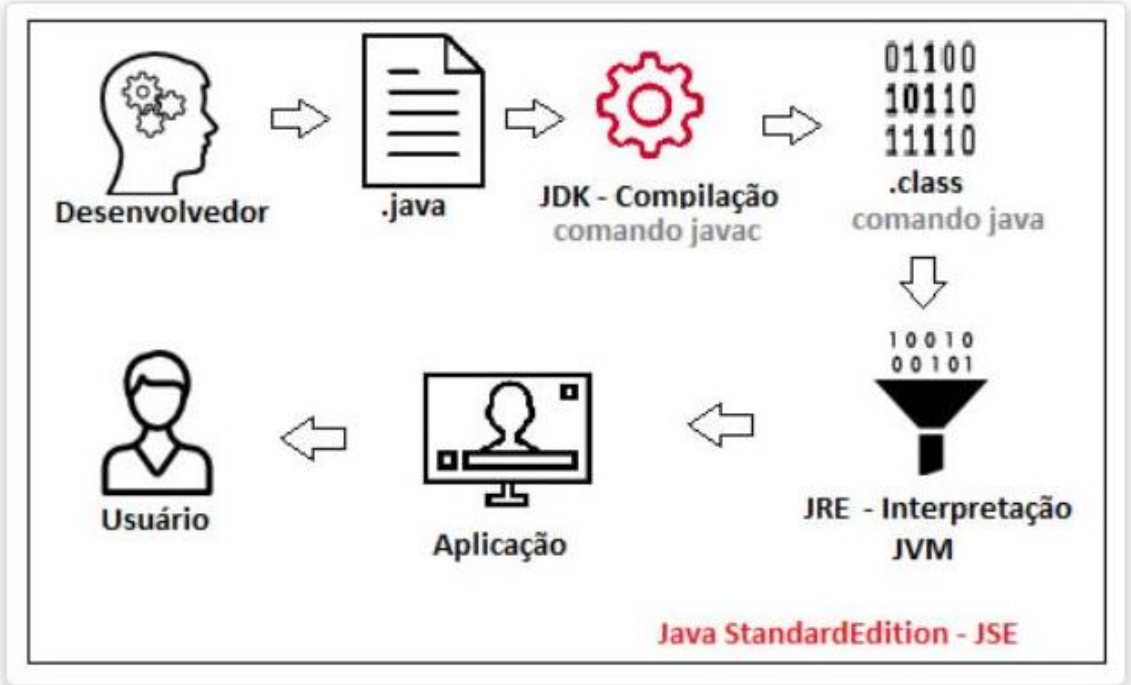
```
1  class ImpressorPai  {
2      public void imprime(String i) {
3          System.out.println("Cadeia do pai: "+ i);
4      }
5      public void imprime(int i) {
6          System.out.println("Inteiro do pai: " + i);
7      }
8  }
9
10
11 class ImpressorFilho extends ImpressorPai {
12     public void imprime(char i){
13         System.out.println("Char do filho: " + i);
14     }
15     public void imprime(int i){
16         System.out.println("Inteiro do filho: " + i);
17     }
18 }
19
20
21 public class Programa {
22     public static void main(String[] args) {
23         ImpressorFilho imp = new ImpressorFilho();
24         imp.imprime("texto");
25         imp.imprime('a');
26         imp.imprime(1);
27     }
28 }
```

Tendo como referência esse código, assinale a alternativa INCORRETA:

- (●) Ao chamarmos o método "imprime", na linha 26, será chamado o método da classe "ImpressorPai", definido na linha 5.
- ( ) Podemos observar, nas linhas 2 e 5, uma sobrecarga do método "imprime" na classe "ImpressorPai". Nesse caso, temos métodos com o mesmo nome e diferentes tipos de parâmetros.
- ( ) Ao chamarmos o método "imprime", na linha 26, será chamado o método da classe "ImpressorFilho", definido na linha 15.
- ( ) Na linha 15 estamos sobrepondo, na classe "ImpressorFilho", o método definido na linha 5 da classe "ImpressorPai", dando a ele uma nova implementação.
- ( ) A classe "ImpressorFilho" herda os métodos da classe "ImpressorPai". Assim, os métodos definidos nas linhas 2 e 5, definidos na classe "ImpressorPai", passam a fazer parte da classe "ImpressorFilho". Assim, quando declaramos o método "imprime", na linha 12, estamos fazendo uma sobrecarga dos métodos herdados.

Pergunta 2 (1 ponto)

Questão 1) A figura abaixo mostra como um programa em Java é compilado e executado.



Tendo como referência essa figura, assinale a alternativa INCORRETA.

- ( ) Como os arquivos ".class" são executados por uma máquina virtual Java (também conhecida como JVM), podemos levar esses arquivos para qualquer ambiente que tenha uma máquina virtual Java compatível, seja ele em Windows, Linux ou Mac OS, e conseguiremos assim executar nosso programa Java. Essa característica do Java corresponde à ideia da independência de plataforma provida pela linguagem.

- (•) A máquina virtual Java (também conhecida como JVM) é a mesma para Windows, Linux e Mac OS. Assim como os arquivos ".class" elas são independentes de plataforma, e por isso existe um único instalador para todas essas plataformas.

- ( ) Para executar arquivos ".class" precisamos de uma máquina virtual Java (também conhecida como JVM). A máquina virtual Java pega os bytecodes que estão nos arquivos ".class" e entende eles como instruções que deverão ser executadas para execução de um programa em Java.

- ( ) Escrevemos nossos programas, em Java, em arquivos com a extensão ".java". Esses arquivos, quando compilados pelo programa "javac", são transformados em arquivos com a extensão ".class".

- ( ) Os arquivos com extensão ".class" contém bytecodes, que são instruções para uma máquina virtual Java (também conhecida como JVM). Esses arquivos não podem ser diretamente executados pelo sistema operacional, como os arquivos ".exe" do Windows.

Pergunta 3 (1 ponto)

Questão 2) Vetores são variáveis que armazenam vários valores, sendo que todos esses valores tem o mesmo tipo de dados (int, double, String, etc). Os dados armazenados em um vetor são referenciados pela sua posição dentro do vetor, e essa posição é também conhecida como "índice" do vetor. O código abaixo mostra um exemplo de utilização de vetores em Java:

```
1
2  import java.util.Arrays;
3  import java.util.Scanner;
4
5  public class Programa {
6      public static void main(String[] args) {
7          int vetor[] = new int[10] ;
8
9          // Solicita elementos do vetor
10         Scanner in = new Scanner(System.in) ;
11         for ( int i = 0; i < vetor.length; i++ ) {
12             System.out.printf("Entre com vetor[ %d ] => ", i) ;
13             vetor[ i ]  = in.nextInt() ;
14         }
15
16         // Determina o maior elemento do vetor
17         int maior = vetor[ 0 ] ;
18         for ( int i = 0; i < 10; i++ ) {
19             if ( vetor[i] > maior ) {
20                 maior = vetor[ i ] ;
21             }
22         }

23         System.out.printf("Vetor lido: %s \n", Arrays.toString(vetor)) ;
24         System.out.printf("O maior valor = %d ", maior) ;
25     }
26 }
```

Tendo como referência esse exemplo, assinale a alternativa INCORRETA:

- O primeiro elemento do vetor está armazenado na posição zero. Por isso, nos dois loops que aparecem no código (linhas 11 e 18), usamos uma variável "i" iniciando em zero para fazer referência ao primeiro elemento do vetor.
- Todo vetor, em Java, possui um atributo "length" que guarda o número de elementos do vetor. Na linha 11 estamos usando esse atributo na condição do comando "for".
- Na linha 13 do código estamos guardando em uma posição do vetor, dada pela variável "i", um valor informado pelo usuário.
- Todo vetor, em Java, é um objeto. Por isso, assim como fazemos com objetos, usamos o operador "new" para criar o vetor. Isso é feito na linha 7 do código.

◉ Nesse exemplo, a condição do comando "if" da linha 19 é verdadeira quando o valor armazenado na posição "i" do vetor for maior ou igual ao valor armazenado na variável "maior".

Pergunta 4 (1 ponto)

Questão 4) Durante nossos estudos aprendemos que uma classe pode conter atributos e métodos. Os construtores, segundo alguns autores, podem ser considerados como um tipo especial de método, porque embora não tenham um retorno explícito (como os métodos comuns) são chamados para criar objetos (ou instâncias) de uma classe. O código abaixo mostra um exemplo de classe contendo atributos, métodos e um construtor:

```
1  class Produto {
2
3      private int id ;
4      private String nome ;
5
6      public Produto(int id, String nome) {
7          this.id = id;
8          this.nome = nome;
9      }
10
11     public int getId() {
12         return id;
13     }
14
15     public void setId(int id) {
16         this.id = id;
17     }
18
19     public String getNome() {
20         return nome;
21     }
22
23     public void setNome(String nome) {
24         this.nome = nome;
25     }
26 }
27
```

```
28
29 public class Programa {
30     public static void main(String[] args) {
31         Produto p = new Produto(1, "Core Java") ;
32         System.out.printf("Id do produto: %d, nome do produto: %s", p.getId(), p.getNome());
33     }
34 }
```

Tendo como referência esse código, assinale a alternativa INCORRETA:

- Nas linhas 7 e 8 guardamos nos atributos da classe "Produto" os valores informados para o construtor. A palavra reservada "this", nesse caso, serve para diferenciar um atributo da classe de um parâmetro do construtor, visto que os dois tem o mesmo nome.

- Nas linhas 11 e 19 declaramos métodos "get". Esses métodos são usados para recuperar o valor armazenado nos atributos da classe, e seguindo a regra do encapsulamento são declarados com visibilidade "pública".

- Na linha 6 declaramos um construtor para a classe "Produto". Esse construtor nos informa que, para criarmos um objeto do tipo "Produto" é necessário informar para o construtor da classe o id e o nome do produto. A criação de um objeto do tipo "Produto" é feita na linha 31 do código.

- (●) Nas linhas 3 e 4 declaramos os atributos da classe "Produto". Esses atributos são declarados com visibilidade "privada". No entanto, para estar de acordo a regra do encapsulamento, eles deveriam ter sido declarados com visibilidade "pública".

- Nas linhas 15 e 23 declaramos métodos "set". Esses métodos são usados para informar novos valores para os atributos da classe, e seguindo a regra do encapsulamento são declarados com visibilidade "pública".

Pergunta 5 (1 ponto)

Questão 5) Um interessante recurso, oferecido pela linguagem Java, é a "sobrecarga" de métodos em uma classe. Com a sobrecarga de métodos podemos incluir em uma classe vários métodos de mesmo nome. O código abaixo mostra um exemplo de utilização desse recurso:

```
1
2  public class Impressor {
3
4      public void imprimir( String s ) {
5          System.out.println( "Cadeia: " + s );
6      }
7
8      public void imprimir( int i ) {
9          System.out.println( "Inteiro: " + i );
10     }
11
12     public void imprimir( double d ) {
13         System.out.println( "Real: " + d );
14     }
15
16     public static void main(String[] args) {
17         Impressor imp;
18         imp = new Impressor();
19         imp.imprimir( "Um texto" );
20         imp.imprimir( 1 );
21         imp.imprimir( 3.0f );
22     }
23 }
```

endo como referência esse código, assinale a alternativa INCORRETA:

- ○ Podemos observar a sobrecarga de métodos, nesse exemplo, nas declarações que aparecem nas linhas 4, 8 e 12 do código.
- ○ Na sobrecarga desse exemplo temos três métodos que tem o mesmo nome, "imprimir", e diferentes assinaturas (diferentes parâmetros).
- ○ Quando o método "imprimir" for chamado, na linha 19, será chamado o método declarado na linha 4.
- ◉ Quando o método "imprimir" for chamado, na linha 20, será chamado o método declarado na linha 12.
- ○ Quando o método "imprimir" for chamado, na linha 21, será chamado o método declarado na linha 12.

Pergunta 6 (1 ponto)

Questão 6) Por meio da herança conseguimos aproveitar funcionalidades que já foram implementadas em uma classe, para incluí-las em uma nova classe. Nesse caso, as funcionalidades "herdadas" passam a existir na nova classe como se tivessem sido ali definidas. O código abaixo mostra um exemplo de utilização da herança.

```
1  import java.math.BigDecimal;
2  import java.time.LocalDateTime;
3
4  class Pagamento {
5      // Atributos
6      private int idPagamento ;
7      private LocalDateTime dataPagamento ;
8      private BigDecimal valorPagamento ;
9
10     // Métodos get/set
11     public int getIdPagamento() {
12         return idPagamento;
13     }
14     public void setIdPagamento(int idPagamento) {
15         this.idPagamento = idPagamento;
16     }
17     public LocalDateTime getDataPagamento() {
18         return dataPagamento;
19     }
20     public void setDataPagamento(LocalDateTime dataPagamento) {
21         this.dataPagamento = dataPagamento;
22     }
23     public BigDecimal getValorPagamento() {
```

```
24          return valorPagamento;
25      }
26      public void setValorPagamento(BigDecimal valorPagamento) {
27          this.valorPagamento = valorPagamento;
28      }
29  }
30
31  class PagamentoComCartao extends Pagamento {
32      // Atributos
33      private String numeroCartao ;
34
35      // Métodos get/set
36      public String getNumeroCartao() {
37          return numeroCartao;
38      }
39
40      public void setNumeroCartao(String numeroCartao) {
41          this.numeroCartao = numeroCartao;
42      }
43  }
44
45  class PagamentoComBoleto extends Pagamento {
46      // Atributos
47      private String numeroBoleto ;
48
49      // Métodos get/set
50      public String getNumeroBoleto() {
51          return numeroBoleto;
52      }
53
54      public void setNumeroBoleto(String numeroBoleto) {
55          this.numeroBoleto = numeroBoleto;
56      }
57  }
```

Tendo como referência esse código, assinale a alternativa INCORRETA:

- ( ) Como os atributos de "Pagamento" são privados, eles não são herdados pelas subclasses "PagamentoComCartao" e "PagamentoComBoleto". Assim, caso sejam adicionados nessas subclasses novos métodos que precisem acessar os atributos da superclasse, eles deverão fazer uso dos métodos públicos "get" e "set" correspondentes.

- ( ) Se trocarmos a visibilidade dos atributos da classe "Pagamento" de "private" para "protected", eles poderão ser diretamente acessados nas subclasses "PagamentoComCartao" e "PagamentoComBoleto". Assim, caso sejam adicionados nessas subclasses novos métodos que precisem acessar os atributos da superclasse, eles poderão acessá-los diretamente, sem precisar recorrer aos métodos "get" e "set" correspondentes.

- (•) Podemos dizer que, pela herança, o atributo "numeroCartao" e os métodos get/set da classe "PagamentoComCartao" são herdados pela classe "Pagamento".

- ( ) Podemos dizer que, pela herança, existe entre "PagamentoComCartao" e "Pagamento" um relacionamento do tipo "é um". Ou seja, nesse caso, "PagamentoComCartao" é um "Pagamento".

- ( ) Podemos dizer que, pela herança, "PagamentoComBoleto" tem tudo que "Pagamento" tem, e ainda adiciona novos atributos e métodos.

Pergunta 7 (1 ponto)

Questão 3) Quando começamos a falar de classes, vimos que é possível declarar diversas "variáveis" em uma classe, para manter em um mesmo lugar dados que estão relacionados. Essas "variáveis" são os atributos de uma classe. O código abaixo mostra um exemplo disso:

```java
import java.math.BigDecimal;

class Produto  {
    int id ;
    String nome ;
    BigDecimal preco ;
}


public class Programa {
    public static void main(String[] args) {
        // Cria uma nova instância de Produto
        Produto produto = new Produto();
        produto.id = 10 ;
        produto.nome = "Livro Core Java" ;
        produto.preco = BigDecimal.valueOf(49.90) ;

        // Mostra os dados do produto
        System.out.printf("Id do produto: %d \n", produto.id);
        System.out.printf("Nome do produto: %s \n", produto.nome);
        System.out.printf("Preço do produto: %.2f \n", produto.preco);
```

Tendo como referência esse código, assinale a alternativa INCORRETA:

- Para usar a classe "Produto" declaramos uma variável do tipo "Produto", na linha 13, e criamos um novo objeto do tipo "Produto" com o operador "new".

- Se os atributos da classe "Produto", declarados nas linhas 4, 5 e 6, fossem definidos com o modificador "private" o código não iria mais funcionar. Isso porque, nesse caso, os atributos não estariam mais "visíveis" para a classe "Programa", e por isso eles não poderiam ser mais diretamente acessados nas linhas 14, 15, 16, 19, 20 e 21.

- (●) Se os atributos da classe "Produto", declarados nas linhas 4, 5 e 6, fossem definidos com o modificador "public" o código não iria mais funcionar. Isso porque, nesse caso, os atributos não estariam mais "visíveis" para a classe "Programa", e por isso eles não poderiam mais ser diretamente acessados nas linhas 14, 15, 16, 19, 20 e 21.

- ( ) Nesse exemplo usamos uma classe "Produto" para manter em um mesmo lugar os dados de um produto. Esses dados correspondem aos atributos "id", "nome" e "preco", declarados nas linhas 4, 5 e 6 do código.

- ( ) Nesse exemplo não definimos a visibilidade dos atributos da classe "Produto". Mas esse atributos podem ser diretamente acessados pela classe "Programa", nas linhas 14, 15, 16, 19, 20 e 21, porque as duas classes foram definidas em um mesmo arquivo, e portanto, pertencem ao mesmo "pacote".

Pergunta 8 (1 ponto)

Questão 10) As exceções, em Java, são utilizadas para indicar que ocorreu algo inesperado durante a execução do programa. Um exemplo típico desse cenário acontece, por exemplo, quando tentamos chamar um método usando uma variável do tipo classe que guarda null, ao invés da referência a um objeto. Nesse caso, como não existe um objeto, é levantada uma exceção do tipo "NullPointerException". Podemos, então, adicionar no nosso código um bloco "try-catch" para tratar exceções, caso essas sejam levantadas durante a execução do programa. O código abaixo mostra essa ideia.

```
1  public class Programa {
2      public static void main(String[] args) {
3          try {
4              int vetor[] = { 1, 2, 3, 4, 5 } ;
5              for(int i= 0; i < 6; i++) {
6                  System.out.println(vetor[i]);
7              }
8
9          } catch(Exception e) {
10             System.out.println("Foi levantada uma exceção!");
11         }
12     }
13 }
```

Tendo como referência esse código, assinale a alternativa INCORRETA:

- Esse código, quando executado, levanta uma exceção e imprime na tela a mensagem "Foi levantada uma exceção".

- A classe "Exception" tem como subclasse "RuntimeException", que corresponde às exceções não verificadas, como "NullPointerException". Outras subclasses de "Exception", diferentes de "RuntimeException", correspondem a exceções verificadas. As exceções verificadas necessariamente devem ser tratadas, ou seja, o código capaz de levantar esse tipo de exceção deve ser cercado por um bloco "try-catch". As exceções não verificadas, por sua vez, não precisam necessariamente serem tratadas por um bloco "try-catch".

- Se trocarmos a condição do loop para "i < vetor.length", quando o programa for executado nenhuma exceção será levantada.

- Nesse exemplo, os comandos que aparecem entre as linhas 4 e 7 estão dentro de um "try". Assim, se uma exceção do tipo "Exception" for levantada o comando que aparece dentro do "catch" será executado.

- (●) Esse código, quando executado, imprime na tela os elementos do vetor. Nenhuma exceção é levantada.

Pergunta 9 (1 ponto)

Questão 7) Por meio da herança conseguimos aproveitar funcionalidades que já foram implementadas em uma classe, para incluí-las em uma nova classe. Nesse caso, as funcionalidades "herdadas" passam a existir na nova classe como se tivessem sido ali definidas. O código abaixo mostra um exemplo de utilização da herança.

```java
import java.math.BigDecimal;
import java.time.LocalDateTime;

class Pagamento {
    // Atributos
    private int idPagamento ;
    private LocalDateTime dataPagamento ;
    private BigDecimal valorPagamento ;

    // Métodos get/set
    public int getIdPagamento() {
        return idPagamento;
    }
    public void setIdPagamento(int idPagamento) {
        this.idPagamento = idPagamento;
    }
    public LocalDateTime getDataPagamento() {
        return dataPagamento;
    }
    public void setDataPagamento(LocalDateTime dataPagamento) {
        this.dataPagamento = dataPagamento;
    }
    public BigDecimal getValorPagamento() {

        return valorPagamento;
    }
    public void setValorPagamento(BigDecimal valorPagamento) {
        this.valorPagamento = valorPagamento;
    }
}

class PagamentoComCartao extends Pagamento {
    // Atributos
    private String numeroCartao ;

    // Métodos get/set
    public String getNumeroCartao() {
        return numeroCartao;
    }

    public void setNumeroCartao(String numeroCartao) {
        this.numeroCartao = numeroCartao;
    }
}

class PagamentoComBoleto extends Pagamento {
    // Atributos
    private String numeroBoleto ;
```

```
48
49      // Métodos get/set
50      public String getNumeroBoleto() {
51          return numeroBoleto;
52      }
53
54      public void setNumeroBoleto(String numeroBoleto) {
55          this.numeroBoleto = numeroBoleto;
56      }
57  }
58
59
60  public class Programa {
61      public static void main(String[] args) {
62          // Cria pagamento com cartao
63          PagamentoComCartao pagamento = new PagamentoComCartao() ;
64          pagamento.setIdPagamento(1);
65          pagamento.setDataPagamento(LocalDateTime.of(2021, 6, 22, 15, 19));
66          pagamento.setValorPagamento(BigDecimal.valueOf(59.90));
67          pagamento.setNumeroCartao("123-456");
68
69          // Mostra dados
70          System.out.println(pagamento.getIdPagamento());
71          System.out.println(pagamento.getDataPagamento());
72          System.out.println(pagamento.getValorPagamento());
73          System.out.println(pagamento.getNumeroCartao());
74      }
75  }
76
```

Tendo como referência esse código, assinale a alternativa INCORRETA:

- ( ) Nesse exemplo, podemos observar que as classes "PagamentoComCartao" e "PagamentoComBoleto", declaradas nas linhas 31 e 45, herdam da classe "Pagamento".
- (•) De acordo com a terminologia da Programação Orientada a Objetos, podemos dizer que "PagamentoComBoleto" é uma generalização de "Pagamento".
- ( ) Usamos a palavra-chave "extends" para dizer que uma classe "herda" de outra classe.

◯ Por causa da herança, podemos dizer que "PagamentoComCartao" passa a ter todos os métodos que foram definidos na classe "Pagamento". Por isso, ao declarar uma variável do tipo "PagamentoComCartao", na linha 63, podemos chamar nas linhas 64, 65 e 66 os métodos "set" que foram definidos na classe "Pagamento", e nas linhas 70, 71 e 72 os métodos "get" que também foram definidos nessa classe.

◯ De acordo com a terminologia da Programação Orientada a Objetos, podemos dizer que "PagamentoComCartao" é uma especialização de "Pagamento".

Pergunta 10 (1 ponto)

Questão 9) De acordo com nossos estudos, vimos que o polimorfismo consiste em usar uma referência genérica para realizar uma chamada especializada (método sobrescrito) em tempo de execução (ligação tardia ou dinâmica). Ou seja, a ideia do polimorfismo, basicamente, consiste em definir um método em uma superclasse e sobrescrevê-lo nas suas subclasses. Usando uma variável do tipo "superclasse" podemos, então, fazer referência a um objeto do tipo "subclasse" e chamar o método correto no objeto que está sendo referenciado. O código abaixo mostra essa ideia.

```
1  abstract class ImpostoGasolina {
2      private double valorCalculo ;
3
4      public ImpostoGasolina(double valorCalculo) {
5          this.valorCalculo = valorCalculo ;
6      }
7
8      public double getValorCalculo() {
9          return valorCalculo;
10     }
11
12     public abstract double getImposto() ;
13 }
14
15 class ImpostoGasolinaSP extends ImpostoGasolina {
16
17     public ImpostoGasolinaSP(double valorCalculo) {
18         super(valorCalculo) ;
19     }
20
```

```
21      @Override
22      public double getImposto() {
23          return getValorCalculo() * 0.25 ;
24      }
25  }
26
27  class ImpostoGasolinaRJ extends ImpostoGasolina {
28
29      public ImpostoGasolinaRJ(double valorCalculo) {
30          super(valorCalculo) ;
31      }
32
33      @Override
34      public double getImposto() {
35          return getValorCalculo() * 0.34 ;
36      }
37  }
38
39  public class Programa {
40      public static void main(String[] args) {
41          ImpostoGasolina imposto ;
42
43          imposto = new ImpostoGasolinaSP(100) ;
44          System.out.printf("Imposto em SP: %.2f \n", imposto.getImposto());
45
46          imposto = new ImpostoGasolinaRJ(100) ;
47          System.out.printf("Imposto no RJ: %.2f \n", imposto.getImposto());
48      }
49  }
50
```

Tendo como referência esse exemplo, assinale a alternativa INCORRETA:

- (●) Nesse exemplo, mesmo sendo a classe "ImpostoGasolina" abstrata, poderíamos criar um objeto do tipo "ImpostoGasolina" usando o comando: "imposto = new ImpostoGasolina(100) ;"
- ( ) Sabemos que o construtor de uma subclasse deve chamar algum construtor da sua superclasse. Nesse exemplo, os construtores de "ImpostoGasolinaSP" e "ImpostoGasolinaRJ" estão chamando o construtor de "ImpostoGasolina" por meio da chamada a "super" nas linhas 18 e 30. Na chamada, os construtores das subclasses estão repassando para o construtor da superclasse o valor que será utilizado no cálculo do imposto.

- Podemos observar, nas linhas 43 e 46, que objetos do tipo "ImpostoGasolinaSP" e "ImpostoGasolinaRJ" estão sendo tratados como se fossem objetos do tipo "ImpostoGasolina". Isso é possível por causa da herança que existe entre essas classes, e também por causa do polimorfismo.

- Na linha 41 é declarada uma variável do tipo "ImpostoGasolina". Na linha 43, essa variável recebe a referência a um objeto do tipo "ImpostoGasolinaSP". Quando o método "getImposto" é chamado, na linha 44, será chamado o método definido na linha 22, que corresponde ao método de "ImpostoGasolina" que foi sobreposto na subclasse "ImpostoGasolinaSP".

- Podemos observar, nesse exemplo, que "getImposto" é um método abstrato na classe "ImpostoGasolina" (linha 12). Declaramos um método como abstrato, em uma superclasse, para que ele possa ser sobrescrito nas suas subclasses, provendo para o método uma implementação. É o que acontece nas linhas 22 e 34 do código.