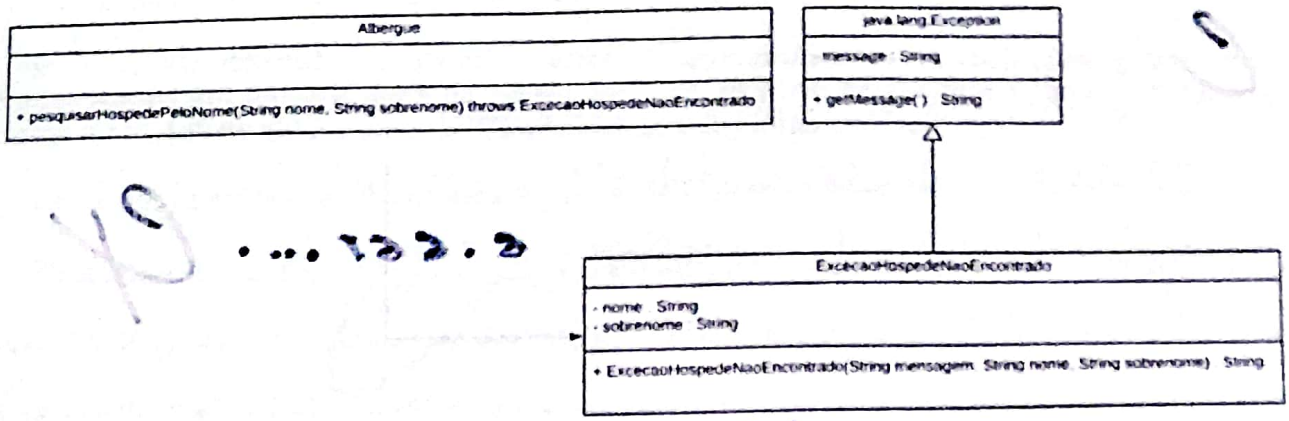
1. Sendo fornecida o diagrama de classes e trechos de código fonte abaixo, preencha as lacunas de acordo com as perguntas 1.1 a 1.6 (enunciado após o código. Reticências indicam código corretamente implementado):




```
class ExcecaoHospedeNaoEncontrado _____ { // 1.1
   _____ // 1.2
   _____

   public ExcecaoHospedeNaoEncontrado(String mensagem, String nome,
                                      String sobrenome) {
      _____ // 1.3
      ...
      ...
   }
}

class Albergue {
   ...
   public Customer pesquisarHospedPeloNome(String nome, String sobrenome)
                                    throws ExcecaoHospedeNaoEncontrado {
      for (Hospede hospede: hospedes) {
        if (hospede.getNome().equals(nome) &&
                          hospede.getSobrenome().contains(sobrenome)){
          return hospede;
        }
      }
```

```
    }
}

class TesteAlbergue {
    ...
    ...

    public static void main(String[] args) {
      Customer c = null;

                                                    // 1.5
        c = hl.pesquisarHospedPeloNome("Florentino", "Ariza");

                                                    // 1.6
      }
}
```

1.1 Qual código deverá ser inserido aqui?

1.2 Qual código deverá ser inserido aqui?

1.3 O parâmetro `mensagem` dever ser usado para se iniciar a variável `message` presente na classe `java.lang.Exception` conforme diagrama de classes.

1.4 Qual código deverá ser inserido aqui caso o hóspede não seja encontrado (a sua resposta deverá incluir a mensagem "`Hóspede não encontrado`" e deverá usar os parâmetros `nome` e `sobrenome`).

1.5 Ao ser invocado, o método `pesquisarHospedePeloNome` pode gerar uma exceção.

1.6 Caso uma exceção seja gerada, precisamos tratá-la.

2. Sendo fornecida a classe `Albergue` abaixo, preencha as lacunas de acordo com as perguntas 2.1 a 2.2 (enunciado após o código):

```
public class Albergue {
     private String nome;

                                                    ; // 2.1

     public boolean adicionarReserva(String dataReserva) {
         Reserva reserva = new Reserva( );
         reserva.setDataReserva(dataReseva);
                                                    ; // 2.2
```

1.9

```
        return true;
    }
```

2.1 Um `Albergue` possui 0, 1 ou mais objetos do tipo `Reserva` (Declare e instancie a classe adequada da API Collection, lembrando de usar *generics* para parametrizar o tipo de dados e o operador diamante - *diamond*).

2.2 Um objeto do tipo `Reserva` deve ser inserido na coleção de reservas do `Albergue`.

3. No conteúdo cobrado na prova anterior, implementávamos associações 1 para muitos representadas em um diagrama de classe em UML usando *Arrays* (por exemplo, 1 `Albergue` possui `N` objetos do tipo `Reserva`). Responda as perguntas abaixo:

3.1 Qual a desvantagem no uso do *array*?

Para usar array você deve ter um tamanho prévio do array na declaração, assim se o array tiver tamanho 10 e eu tentar inserir 11 elementos irá ocorrer um erro.

3.2. Qual a vantagem no uso de uma coleção de `Object` (sem uso de *generics*) sobre o uso de *arrays*?

Você não precisa definir um Tamanho específico para a quantidade de elementos da coleção, ou seja pode ir adicionando vários elementos sem um tamanho pré estabelecido.

3.3. Qual a vantagem no uso de uma coleção parametrizada com `Generics` sobre uma coleção de `Object`?

Evita o uso de conversão desnecessária, fazendo com que o código fique mais limpo para esse tipo de situação.

4. Responda as questões de múltipla escolha abaixo:

4.1 Selecione a alternativa INCORRETA sobre genéricos (*generics*)

a. *Generics* não podem ser tipos primitivos.
b. Com *Generics*, checagem de tipo é feita em tempo de compilação.
c. *Generics* eliminam conversões (castings) desnecessários.
X A definição do tipo genérico é mantida em tempo de execução.
e. Nenhuma das demais alternativas.

4.2 Assinale a alternativa INCORRETA:

a. *Hashcodes* tipicamente são usados para aumentar a performance de grandes coleções.
b. Um *hashcode* não é um identificador único de um objeto.
c. As interfaces `Set`, `List` e `Queue` da API Collection herdam da interface `Collection`.
X Um `HashSet` é com ordem de iteração (*unordered*) enquanto um `LinkedHashSet` é
de iteração (*ordered*) mantendo a ordem de inserção dos objetos.
e. Quando uma coleção é ordenada por iteração (*ordered*), significa que você pode iterar pela
em uma ordem específica, não-aleatória.

4.3 Assinale a alternativa INCORRETA:

a. O bloco `finally` sempre executa, com exceção quando houver uma instrução `return` no bloco
`try`.
b. De maneira geral, um objeto da casse `Exception` (que não seja `RuntimeException`) na
representa um erro de programação, mas sim um erro indicando que algum recurso não está
disponível ou alguma condição necessária para correta execução não está presente.
c. As exceções que um método pode lançar devem ser declaradas (a menos que elas sejam
subclasses de `RuntimeException` ou `Error`)
d. Cada método deve ou tratar todas as exceções verificadas (*checked*) fornecendo uma cláusula
`catch` ou listar cada exceção verificada e não tratada por meio da cláusula `throws` na assinatura do
método.
X Exceções verificadas não precisam ser especificadas na assinatura dos métodos nem mesmo
tratadas.

4.4 Assinale a alternativa INCORRETA:

X Um `ArrayList` não mantém a ordem de iteração através do índice do elemento.
b. Uma coleção ordenada (*sorted*) significa que a ordem da coleção é determinada por regras naturais
(a < b, 3 < 4, etc..) ou por regras customizadas pelo desenvolvedor.
c. A ordem de iteração de um `HashSet` não é garantida.
d. Uma `LinkedHashSet` é uma versão ordenada por iteração de um `HashSet` que mantém uma lista
duplamente encadeada entre os elementos.
e. Se você tentar adicionar um elemento em um `Set` que já exista, o elemento duplicado não será
adicionado e o método `add()` retornará `false`.

4.5 Selecione a alternativa incorreta sobre melhorias introduzida no Java SE 7.

a. A classe `Files` introduzida no Java SE 7 possui uma série de métodos estáticos para manipulação
de arquivos, dentre eles os métodos `isDirectory`, `createFile`, `deleteIfExists`, `copy` e `move`.
b. Nem sempre foi possível usar `Strings` em instruções `switch` em Java.
c. O recurso de *multicatch* permite tratar mais de um tipo de exceção, caso as exceções demandem
a mesma lógica de tratamento.
d. A partir do Java SE 7 podemos representar literais numéricos representando grandes valores
usando o caracter sublinhado. Por exemplo, `8_000_000.00` significa 8 milhões.
X A API `WatchService` permite monitorarmos diretórios utilizando o conceito de dividir para
conquistar e o *design pattern* `Adapter`.

Selecione a alternativa incorreta sobre manipulação de datas em Java:

a. Requisitos comuns implementados em Java para manipulação de datas e horas: (i) representar a data atual, horário e informações de fuso horário. (ii) exibir data atual, horário e informações de fuso horário. (iii) criar uma data específica. (iv) executar `aritmética` com datas.
b. Antes do JavaSE 8 era comum usarmos a classe `Date` para criação da data corrente do sistema e `GregorianCalendar` para criação de datas específicas.
c. `DateFormat` é uma classe utilizada antes do Java SE 8 para formatação de datas antes de sua exibição no console, por exemplo.
d. Antes do Java SE 8, o método `add` de `GregorianCalendar` era usado para adicionar, por exemplo, um dia à data atual.
X A classe `Calendar` herda de `GregorianCalendar` fornecendo métodos específicos para manipulação de datas e horas que incluem informações de fuso horário.

4.7 Selecione a alternativa incorreta sobre manipulação de datas e horas em Java a partir do Java SE 8

a. Após o Java SE 8, uma API mais simples para manipulação de datas e horas foi introduzida: `LocalDateTime`, `LocalTime` e `LocalDate`.
b. Usamos o método estático `now` de `LocalDateTime` para criarmos uma data e horário atuais.
X Usamos o método estático `of` de `LocalDate` para criarmos uma data específica.
d. Usamos o método `add` de `LocalDate` para somarmos um número `x` de dias a uma data específica.
e. A partir de Java SE 8, usamos a classe `DateTimeFormatter` para formatação de datas (antes do Java SE 8 usávamos `DateFormat`).

4.8 Todas afirmativas sobre o artefato *Modelo de Domínio* do Processo Unificado são verdadeiras, exceto:

a. Normalmente não é necessária a inclusão de métodos em um *Modelo de Domínio*.
b. O *Modelo de Domínio* mostra classes conceituais de uma situação real, não classes de *software*.
c. Classes conceituais em um *Modelo de Domínio* normalmente são derivadas de substantivos presentes em um idioma.
d. Usamos os cenários de um caso de uso como entrada para a criação do *Modelo de Domínio* do Processo Unificado.
X Diretriz: se pensarmos em uma classe conceitual `X` como sendo um número ou texto no mundo real, `X` provavelmente é uma classe conceitual, não um atributo.

4.9 Todas as alternativas abaixo sobre análise e *design* orientado a objetos estão corretas, exceto:

a. Análise orientada a objetos tem como ênfase encontrar e descrever os conceitos no domínio do problema.
b. *Design* (projeto) tem como ênfase a definição dos objetos de *software* e como eles colaboram para a satisfação dos requisitos. Enfatiza uma solução conceitual em *software* ou *hardware* que satisfaça os requisitos
c. Um *Modelo de Domínio* engloba conceitos que descrevem objetos de *software*.
d. UML é uma linguagem gráfica para especificar, construir e documentar os artefatos dos sistemas.
X É mais importante saber projetar objetos do que entender UML.

Boa Prova a tod*s !!!

Rodrigo Martins Pagliares