UAST
Unidade Acadêmica de Serra Talhada - PE
UFRPE
Desde 2006

Bacharelado em Sistemas de Informação

MPOO

Site: https://sites.google.com/site/profricodemery/mpoo

http://ava.ufrpe.br/

https://sigs.ufrpe.br/sigaa/ava/index.jsf

Disciplina: Modelagem e Programação Orientada a Objetos (MPOO)

Profº: Richarlyson D'Emery

LISTA DE EXERCÍCIOS IV

Leia atentamente as instruções gerais:

- No Eclipse crie um novo **projeto chamado br.edu.mpoo.listaIV.SeuNomeSobrenome**, o qual deverá ter **pastas de pacotes** para cada questão: questao1, questao2, e assim sucessivamente, contendo todas as respostas da lista.
- Quando a questão envolver uma discussão teórica utilize um arquivo .txt (Menu File -> Submenu New -> Opção File), por exemplo, questao1.txt
- A lista envolve questões práticas e conceituais, então deverão ser entregues no AVA tanto os códigos-fonte (projeto completo) quanto às demais respostas. Em caso de imagens e digramas, você poderá salvar o arquivo também na pasta correspondente do projeto.
- A entrega da lista compõe sua frequência e avaliação na disciplina.

Responda:

1) Explique como fazer para:
   1.1) Obrigar uma classe a ter que implementar um método.
   1.2) Permitir que outras classes possam utilizar um método inicialmente definido como `private`
2) Qual a diferença entre sobreposição e sobrecarga de métodos em Java? Qual(is) o(s) conceito(s) da Orientação a Objetos relacionado(s) a essas técnicas de programação?
3) Existe sobrecarga de atributos? Explique e Exemplifique.
4) Quando um método de uma superclasse é inadequado para a subclasse, o programador deve sobrescrever esse método na subclasse. Exemplifique as situações de sobrescrita:
   4.1) toString de uma superclasse (sendo esta a raiz que herda de Object);
   4.2) método concreto herdado de uma superclasse concreta; e
   4.3) método abstrato herdado de uma superclasse abstrata.
5) Porque na geração de código do Eclipse é colocado um `super()` sem parâmetro em uma classe simples? E quando esse `super()` passa a possuir parâmetros?

```
public Usuario(String cpf) {
    super();
    this.cpf = cpf;
}
```

Você Sabia?

Em Java podemos utilizar comentários de documentação para auxiliar o programador a entender a especificação de uma codificação, como, por exemplo, os dados de um método. Esses comentários são conhecidos como "doc comments" ou "Javadoc". Diferentemente de uma comentário simples (`//`) este utiliza a notação `/**` para abrir um doc comments e `*/` para fechá-lo. Em seu corpo é possível utilizar tags javadoc para descrever a codificação. Como ocorre com comentários tradicionais, os de documentação podem abranger múltiplas linhas. Por exemplo:

```
/**
   * Descrição
   * @see especifica outras classes relacionadas que podem ser de interesse para o programador que utiliza
   essa classe
   * @author pode ser utilizado para documentar múltiplos autores
   * @param descreve um parâmetro
   * @throws especifica as exceções lançadas
   * @return descreve um tipo de retorno de um método para ajudar o programador a entender como utilizar o
           valor de retorno do método
 */
```

6) Em **Você Sabia?** foram apresentadas informações sobre documentação para código-fonte Java. Dessa maneira pesquise e descreva quais `tags javadoc` podem ser utilizadas nos comentários de documentação.

7) Importe a codificação disponibiliza no projeto ListaIV e então adicione o doc comments para explicar os construtores e o método definirCampanha disponível nas diferentes classes do pacote campanhaVenda.

Saiba Mais!

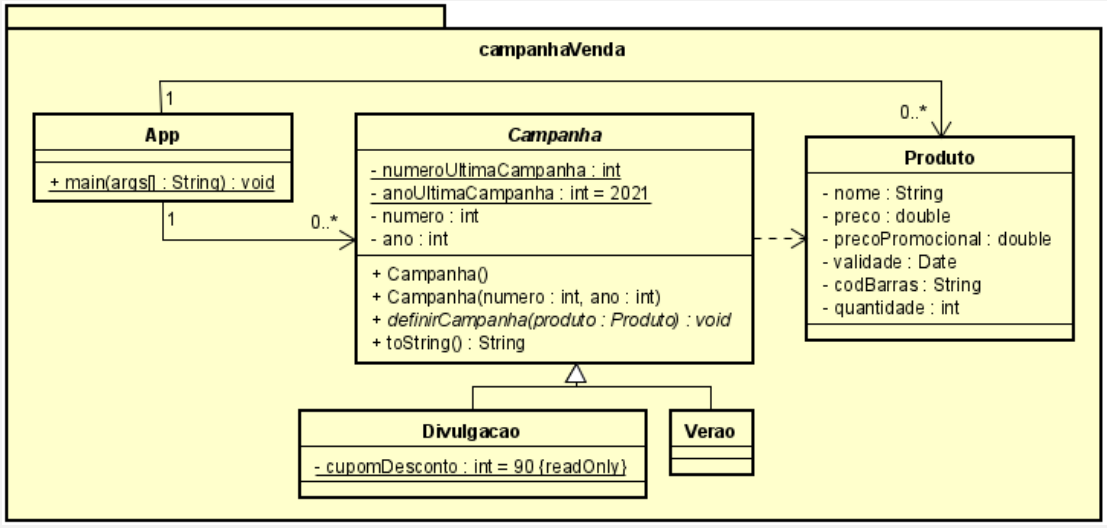
Em POO, o conceito de **Herança** permite que uma subclasse contenha atributos e métodos de uma superclasse, mas o contrário não é verdade. Mas, quando uma subclasse precisa se comportar como a sua superclasse ou implementar comportamentos de maneira específica? Isso é o que chamamos de **comportamento polimórfico**!

O polimorfismo permite 'programar no geral' em vez de 'programar no específico'. Em particular, o polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse em uma hierarquia de classes como se todas fossem objetos da superclasse. Isso é tido como polimorfismo com hierarquias de herança.

Sua utilização evidencia a sobreescrita de métodos, ou seja, quando uma subclasse implementa um comportamento generalizado de forma especializada. É o tipo de polimorfismo também chamado de polimorfismo de objetos ou polimorfismo Universal por Inclusão.

Um exemplo clássico está na redefinição do método toString, na qual a sua chamada é resolvida em tempo de execução (em vez de em tempo de compilação) de acordo com o objeto que o invoca. Esse processo é conhecido como vinculação dinâmica ou vinculação tardia.

Observe o diagrama de classe abaixo:



O diagrama de classes ilustra a vinculação dinâmica da chamada de toString para o polimorfismo entre as especializações Divulgacao e Verao para com a generalização Campanha, as quais demonstram uma ação de venda de uma empresa. A codificação está demonstrada no pacote campanhaVenda do projeto disponibilizado juntamente com esta Lista.

Observe que em App:

```
package campanhaVenda;

public class App {
	public static void main(String[] args) {
		Campanha campanhaDivulgacao = new Divulgacao();
		Campanha campanhaVerao = new Verao();
		System.out.println(campanhaDivulgacao);
		System.out.println(campanhaVerao);
	}
}
```

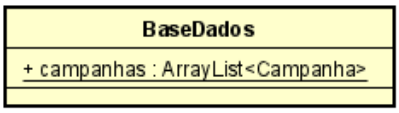
No console há diferentes saídas devido a vinculação dinâmica:

```
Console
<terminated>
Campanha: [1/2022][Cupom Promocional = 90%]
Campanha: [2/2022][Desconto biquine = 50%]
```

8) A partir da codificação e diagrama disponibilizado para a situação Saiba Mais:

(Utilize a estratégia de copiar a pasta pacotes saibaMais para a pasta questao8 de maneira a reaproveitar a codificação disponibilizada)

8.1) Implemente a seguinte classe:



(Não é necessário criar métodos CRUD, a manipulação das campanhas poderá ser feita diretamente pela lista `campanhas`)

8.2) Modifique App de maneira a definir as campanhas utilizando a lista `campanhas`.

Fique atento!

Quando uma superclasse abstrata define um método abstrato, a obrigatoriedade de @Override de métodos abstratos é dada apenas se a subclasse for concreta. Mas, se uma subclasse abstrata que herda o método abstrato de sua superclasse também abstrata é opcional a sopreposição do método abstrato. Sendo assim: (i) se aplicada a sobreposição, os descendentes da subclasse abstrata que herdou da sua superclasse também abstrata, não são obrigados a implementar os métodos abstratos; e (ii) se não aplicada, classes concretas descendentes passam a ter a obrigatoriedade de @Override dos métodos abstratos.

```java
//SuperSuperClass.java
public abstract class SuperSuperClass {
	public abstract void metodo();
}
```

```java
//caso 1:

//SuperClass.java
public abstract class SuperClass extends SuperSuperClass {
	@Override
	public void metodo() {
	  // corpo do método
	}
}

//SubClass.java
public class SubClasse extends SuperClass { }
```

```java
//caso 2: (boa prática de programação)

//SuperClass.java
public abstract class SuperClass extends SuperSuperClass {}

//SubClass.java
public class SubClasse extends SuperClass {
	@Override
	public void metodo() {
	  // corpo do método
	}
}
```

Você Sabia?

Uso de **downcast** em POO: Uma solução ao comportamento de Herança.

```java
// SuperClasse.java
public class SuperClasse {
    private int atr_SuperClasse;

	public int getAtr_SuperClasse() { return atr_SuperClasse; }

	public void setAtr_SuperClasse(int atr_SuperClasse) {
		this.atr_SuperClasse = atr_SuperClasse;
	}
}
// SubClasse.java
public class SubClasse extends SuperClasse{
	private int atr_SubClasse;

	public int getAtr_SubClasse() { return atr_SubClasse; }

	public void setAtr_SubClasse(int atr_SubClasse) {
		this.atr_SubClasse = atr_SubClasse;
	}
}
//continua...
```

```java
// App.java
public class App {
	public static void main(String[] args) {
		SubClasse subClasse = new SubClasse();
		System.out.println(subClasse.getAtr_SuperClasse());
		System.out.println(subClasse.getAtr_SubClasse());
/*
* Questionamento: e se subClasse fosse do tipo SuperClasse? como acessar o atr_SubClasse?
* Solução: Usar downcast.
*/
		SuperClasse subClasse2 = new SubClasse();
		System.out.println(subClasse2.getAtr_SuperClasse());
		System.out.println(((SubClasse)subClasse2).getAtr_SubClasse());//solução
	}
}
```

9) Responda V se verdadeiro ou F se Falso. Justifique se falso.

9.1) ( ) O objeto de uma subclasse pode ser tratado como um objeto de sua superclasse, mas o contrário não é verdadeiro.

9.2) ( )Quando um método de uma superclasse é inadequado para a subclasse, o programador deve sobrescrever esse método na subclasse.

9.3) ( ) Uma superclasse representa um número maior de membros que sua subclasse

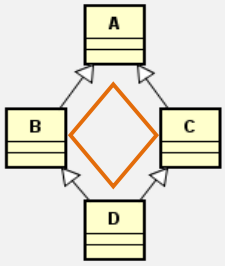
9.4) ( ) O objeto de uma subclasse também é um objeto da superclasse dessa subclasse.

10) O que é toString()? E qual sua relação com Object?

11) Por que na codificação da classe Divulgacao disponibilizada no pacote campanhaVenda de saibaMais não há o método set para *cupomPromocional*?

Você Sabia?

Em POO, o conceito de Herança em Java só permite a herança simples, diferentemente de outras linguagens como, por exemplo, Python que permite que uma classe herde de mais de uma classe, ou seja, permite a implementação de herança múltipla.



Mas, uma classe ao herdar de várias classes naõ apenas podem herdar propriedades completamente diferentes, complicando-se quando superclasses possuem mesmos métodos ou atributos. Essa ambiguidade é conhecida como o problema do diamante (ou problema do losango), e diferentes linguagens resolvem esse problema de maneiras diferentes. O Python segue uma ordem específica para percorrer a hierarquia de classes, chamada de Ordem de Resolução de Métodos (MRO, do inglês Method Resolution Order), fazendo com que a escolha pelo método ou atributo seja dada a partir da ordem da explicitação da generalização, ou seja, a ordem será sempre da esquerda para direita:

```
class Subclasse(ClasseOrdem1,ClasseOrdem2,..., Classe OrdemN)
```

12) Sabemos que em Java não há herança múltipla, mas em outras linguagens sim. Vejamos um exemplo em Python:

```python
#pai.py
class Pai:
    def __init__(self, nome, sobrenome='DEmery'):
        self.nome = nome
        self.sobrenome = sobrenome

    def metodo(self):
        print ('pai')
```

```python
#mae.py
class Mae:
    def __init__(self, nome, sobrenome='A lves'):
        self.nome = nome
        self.sobrenome = sobrenome

    def metodo(self):
        print ('mae')
```

```python
#filho.py

from pai import Pai
from mae import Mae

class Filho(Pai,Mae):
    def __init__(self, cpf, nome):
        super().__init__(nome)
        self.cpf=cpf
```

```python
#main.py
from filho import Filho
from pai import Pai
from mae import Mae

filho = Filho('111.111.111-11', nome='Rico')
pai = Pai(nome='Emerson')
mae = Mae(nome='Sueli')
```

Responda:

12.1) Em `#main.py`, qual a saída para `print(filho.sobrenome)`?

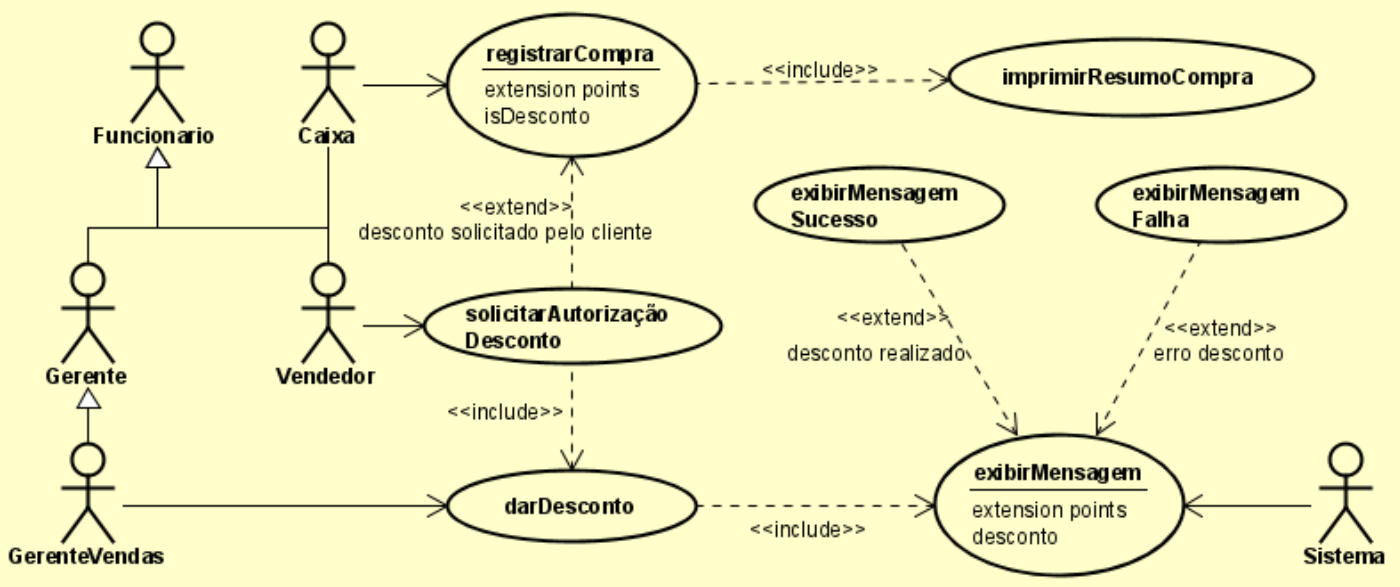
12.2) Em `#main.py`, qual a saída para `filho.metodo()`?

12.3) Análise e explique o que acontece na herança implementada.

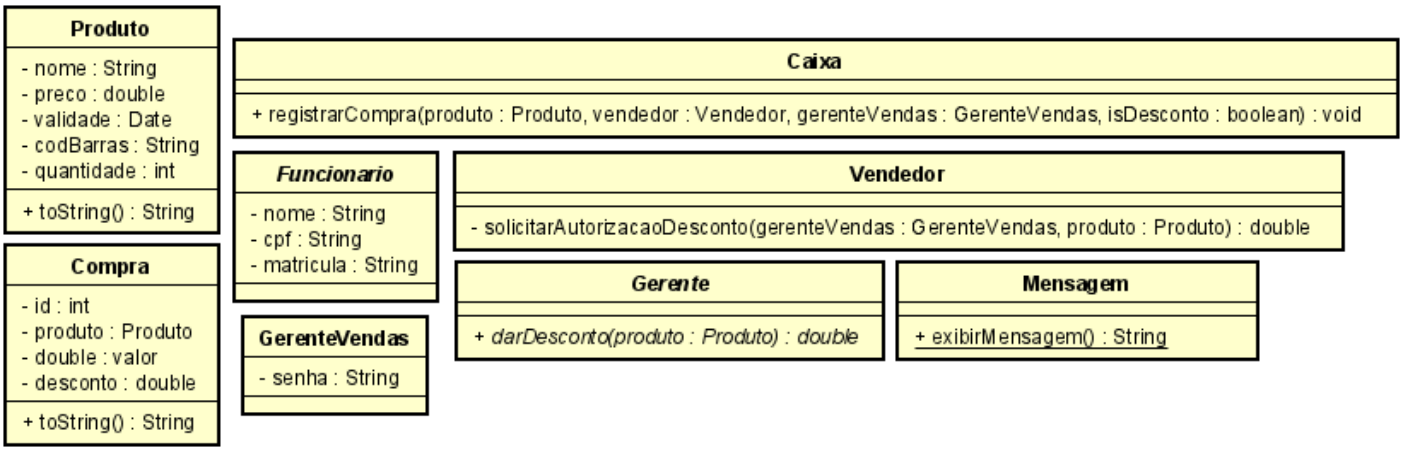
Desafio

Você, aluno de MPOO, está experienciando situações-problemas do universo de desenvolvimento de software e começará a ser desafiado a solucionar problemas a partir de conhecimentos de Programação e Orientação a Objetos.

13) **(Desafio)** Uma Empresa solicitou a um de seus programadores (de codinome *mustela putórius furo* – "O Furão") que resolvesse uma demanda de funcionalidades de seu sistema criada pelo setor de venda da empresa. Cada ator detém comportamento(s) específico(s) conforme sua função ilustrada no diagrama de use case abaixo:



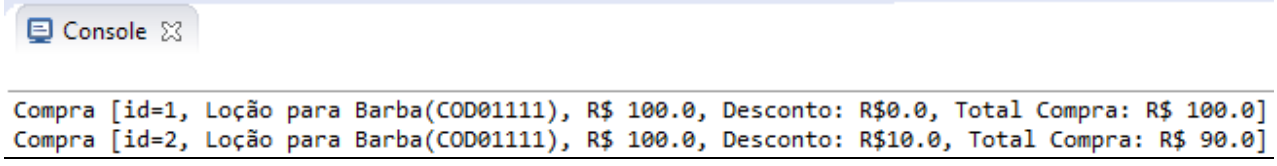
Esboço de diagrama do projeto sem relacionamentos:



Sabe-se que o sistema da empresa deve ser codificado em Java e também deve incluir um diagrama de classes com seus devidos relacionamentos. Antes de responder, analise as situações. Em todas essas, deve-se respeitar as seguintes regras de negócios:

- RN01 – um funcionário só pode executar os comportamentos definidos;
- RN02 – a codificação deve aproveitar comportamentos já definidos, evitando a duplicidade de programação;
- RN03 – uma compra tem identificação autoincrementável;
- RN04 – um desconto só é atribuído a uma compra se for solicitado verbalmente por um cliente no momento do registro de uma compra;
- RN05 – Um desconto ou solicitação de autorização de desconto só poderá ser executado por um funcionário válido; e
- RN06 – Em nenhuma hipótese deve-se alterar o valor de um produto.

Ilustre em uma aplicação: (i) a compra de um produto em que o cliente solicita verbalmente um desconto; e (ii) a compra de um produto em que o cliente não solicita um desconto. Em ambos os casos deve-se exibir em console o resumo da compra:

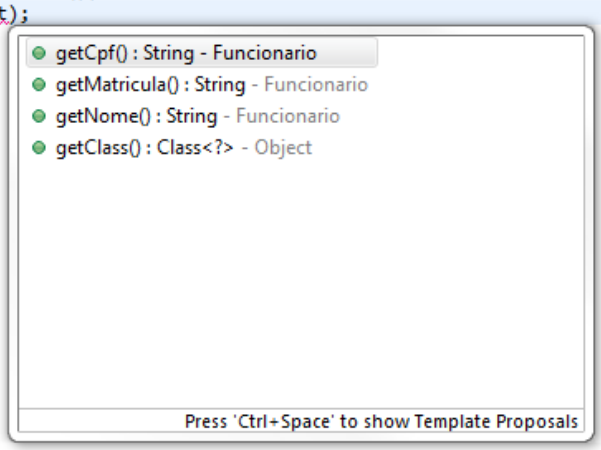


14) O programador “O Furão” resolveu criar uma aplicação para ilustrar o acesso aos atributos de um gerente de vendas:

```
public class App {

    public static void main(String[] args) {

        GerenteVendas gerente1Venda = new GerenteVendas("Ermenegildo Silva", "111.111.111-11", "GERVEN001", "aAbBcCdD");
        Funcionario gerente2Venda = new GerenteVendas("Pregentino Santos", "222.222.222-22", "GERVEN002", "UrsoPAndA");
        System.out.println(gerente1Venda.getSenha());
        System.out.println(gerente2Venda.get);
    }
}
```



Entretanto, não soube explicar o que aconteceu quando tentou acessar o método `getSenha()` para exibir a senha do gerente Pregentino Santos. A partir dos conceitos de Orientação a Objetos:

14.1)Explique o que aconteceu.

14.2)Sem redefinir a definição para a instância de `gerente2Venda`, como “O Furão” poderia acessar o método `getSenha()`?