

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 VII

Leia atentamente as instruções gerais:

- No Eclipse crie um novo projeto chamado **br.edu.mpoo.listaVII.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. A entrega da lista compõe sua frequência e avaliação na disciplina.

Fique atento!

Método construtor é um método que inicializa os atributos da classe. O nome do método construtor deverá ser o mesmo nome da classe.

```
public class Pessoa{  
    String nome;  
    int rg;  
  
    public Pessoa (String n, int rg){  
        this.nome = n;  
        this.rg = rg;  
    }  
}
```

Em **Herança**, quando uma superclasse define um método construtor, logo o método **default deixa de existir!** Mas se houver a necessidade da existência de um "método construtor default" então se deve declará-lo!

```
public class Pessoa {  
    String nome;  
    int rg;  
  
    public Pessoa() {}  
  
    public Pessoa(String n, int rg) {  
        this.nome = n;  
        this.rg = rg;  
    }  
}
```

Em **Herança**, subclasses podem definir pelo menos um construtor herdado para não ocorrer em erro de sintaxe. Entretanto, sugere-se que subclasses definam construtores para todos os herdados.

```
public class Usuario extends Pessoa {  
    String login;  
    String senha;  
  
    public Usuario() {}  
  
    public Usuario(String login, String senha) {  
        super();  
        this.login = login;  
        this.senha = senha;  
    }  
  
    public Usuario(String nome, int rg, String login, String senha) {  
        super(nome, rg);  
        this.login = login;  
        this.senha = senha;  
    }  
}
```

Diagram illustrating inheritance of constructors:

- A dashed arrow points from the `super();` call in the `Usuario(String login, String senha)` constructor to the `public Pessoa() {}` constructor in the `Pessoa` class.
- A dashed arrow points from the `super(nome, rg);` call in the `Usuario(String nome, int rg, String login, String senha)` constructor to the `public Pessoa(String n, int rg) { this.nome = n; this.rg = rg; }` constructor in the `Pessoa` class.

Responda:

1) Preencha as lacunas:

- 1.1) Se a classe Pessoa herda da classe Animal, a classe Pessoa é chamada de _____ e a classe Animal é chamada de _____.
- 1.2) O conceito de herança permite a _____, que economiza tempo no desenvolvimento e estimula a utilização de programas previamente testados.
- 1.3) Quando uma classe é utilizada com o mecanismo de herança, ela se torna uma superclasse que fornece _____ e _____ para outras classes ou se torna uma subclasse.
- 1.4) O relacionamento “é um” entre as classes representa o conceito de _____, enquanto o relacionamento “tem um” entre classes representa _____.

2) 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;  
}
```

3) Analise o código abaixo e aponte mudanças necessárias. As correções devem ser justificadas.

```
1 package questao3;  
2  
3 public class Conta{  
4     private int num;  
5     private double saldo;  
6  
7     public int Conta (int n, double saldo) {  
8         num = n;  
9     }  
10  
11     public void debito (double valor) {  
12         this.saldo-=valor;  
13     }  
14  
15     public void credito (double valor) {  
16         this.saldo+=valor;  
17     }  
18 }
```

```
1 package questao3;  
2  
3 public class Poupanca extends Conta{  
4     public Poupanca (int num){  
5         super (num, saldo);  
6     }  
7  
8     public void debito (double valor) {  
9         this.saldo-=valor;  
10    }  
11  
12     public void rendeJuros(){  
13         this.saldo+=saldo*taxa/100;  
14    }  
15 }
```

4) Crie uma classe Data que forneça a data em múltiplos formatos. Use construtores sobrecarregados para criar objetos Data inicializados com datas em diferentes formatos de apresentação.

//continua...

- 5) A partir da descrição do problema abaixo, implemente em Java as definições para os **atores** do sistema. Não é preciso codificar o corpo dos comportamentos, apenas suas definições. Faça o devido uso de Herança.

[RF01] - Devolver produto	
Ator Principal	Cliente
Atores Secundários	Funcionário e Caixa
Resumo	Este caso de uso descreve as etapas necessárias para que um cliente devolva um produto comprado.
Pré-condições	É necessário existir um produto.
Pós-condições	Escolher opção de devolução
Fluxo Principal	
Ações do Ator	Ações do sistema
1. Solicitar a devolução de um produto apresentando o produto.	
	2. Executar a Devolução de um produto
	3. Executar Caso de uso Atualizar estoque
	4. Efetuar troca ou ressarcir o valor do produto
Restrições /validações	
1. O produto só poderá ser devolvido pelo cliente que realizou a compra	
2. O produto só poderá ser devolvido se não apresentar avaria.	
Fluxo Alternativo I – devolução	
Ações do Ator	Ações do sistema
	1. Executar caso de uso Efetuar Troca
Fluxo Alternativo II – ressarcimento	
Ações do Ator	Ações do sistema
	1. Ressarcir ao cliente o valor pago pelo produto (não é considerado taxas de envio)

[RF02] – Atualizar estoque	
Ator Principal	Vendedor
Resumo	Este caso de uso descreve as etapas necessárias para atualizar o estoque
Pré-condições	É necessário haver uma demanda de atualização
Pós-condições	Verificar situação do produto
Fluxo Principal	
Ações do Ator	Ações do sistema
1. Atualizar a base com um produto	
	2. Executar a atualização de estoque
Restrições /validações	
1. Existência de produto requisitado para troca	

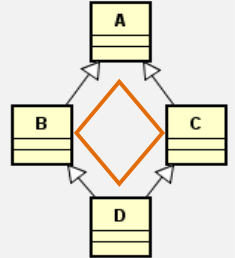
[RF03] – Efetuar Troca	
Ator Principal	Vendedor
Resumo	Este caso de uso descreve as etapas necessárias para efetuar a troca de um produto
Pré-condições	É necessário existir uma solicitação de troca de produto
Pós-condições	É necessário dar baixa do produto trocado no estoque
Fluxo Principal	
Ações do Ator	Ações do sistema
1. Efetuar a troca de um produto por outro	
	2. Executar o caso de uso Baixar Estoque
Restrições /validações	
1. Existência de produto requisitado para troca	

[RF04] – Ressarcir cliente	
Ator Principal	Caixa
Resumo	Este caso de uso descreve o ressarcimento de um cliente
Pré-condições	É necessário existir uma solicitação de ressarcimento
Fluxo Principal	
Ações do Ator	Ações do sistema
1. Realizar a devolução do ressarcimento de valores a cliente.	
	2. Executar o ressarcimento de valor a cliente
Restrições /validações	
1. Existência de uma conta bancária	
Fluxo Alternativo I – ressarcimento em conta	
Ações do Ator	Ações do sistema
	1. Executar a transferência de valor em conta bancária de um cliente

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 não 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)
```

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

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

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

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

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

```
#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
```

```
#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:

- 6.1) Em #main.py, qual a saída para `print (filho.sobrenome)`?
- 6.2) Em #main.py, qual a saída para `filho.metodo()`?
- 6.3) Análise e explique o que acontece na herança implementada.

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 sobreposição 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.

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

//caso 1:

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

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

//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
    }
}
```

Saiba Mais!

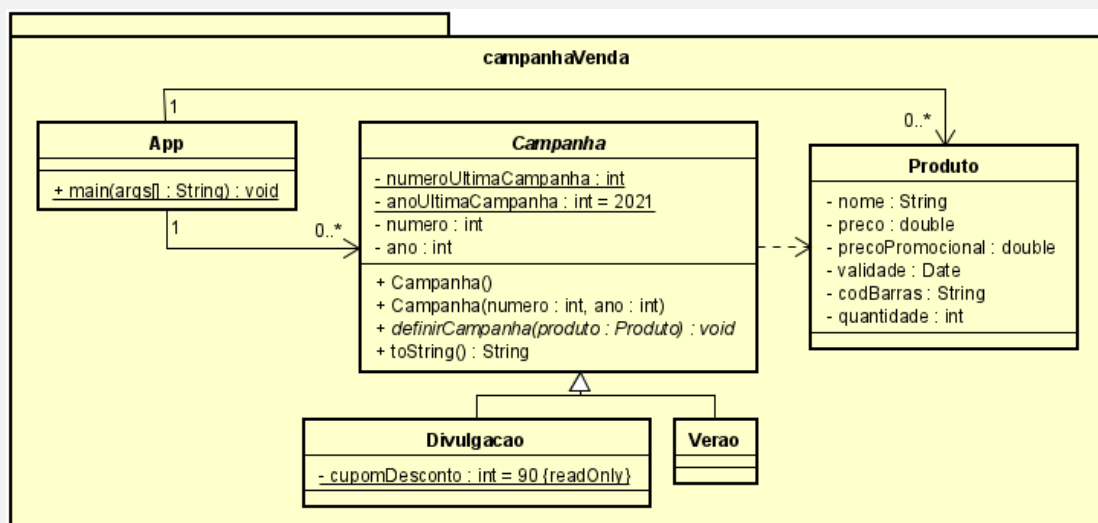
Em POO, o conceito de **Herança** permite que uma subclasse contenha atributos e métodos de uma superclasse, mas o contrário não é verdade. Mas, quando uma subclasse precisa se comportar como a sua superclasse ou implementar comportamentos de maneira específica? Isso é o que chamamos de **comportamento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



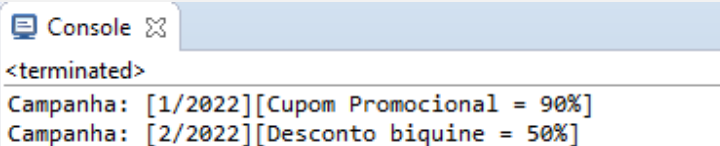
O diagrama de classes ilustra a vinculação dinâmica da chamada de `toString` para o polimorfismo entre as especializações `Divulgacao` e `Verao` para com a generalização `Campanha`, as quais demonstram uma ação de venda de uma empresa. A codificação está demonstrada no pacote `campanhaVenda` do projeto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

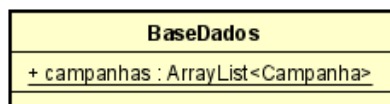


```
<terminated>
Campanha: [1/2022][Cupom Promocional = 90%]
Campanha: [2/2022][Desconto biquine = 50%]
```

- 7) 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?
- 8) Existe sobrecarga de atributos? Explique e Exemplifique.
- 9) Explique como fazer para:
 - 9.1) Obrigar uma classe a ter que implementar um método.
 - 9.2) Permitir que outras classes possam utilizar um método inicialmente definido como `private`
- 10) O que é `toString()`? E qual sua relação com `Object`?
- 11) Quando um método de uma superclasse é inadequado para a subclasse, o programador deve sobrescrever esse método na subclasse. Exemplifique as situações de sobrescrita:
 - 11.1) `toString` de uma superclasse (sendo esta a raiz que herda de `Object`);
 - 11.2) método concreto herdado de uma superclasse concreta; e
 - 11.3) método abstrato herdado de uma superclasse abstrata.
- 12) Por que na codificação da classe `Divulgacao` disponibilizada no pacote `campanhaVenda` de `saibaMais` não há o método `set` para `cupomPromocional`?
- 13) A partir da codificação e diagrama disponibilizado para a situação `Saiba Mais`:

(Utilize a estratégia de copiar a pasta `pacotes` `saibaMais` para a pasta `questao13` de maneira a reaproveitar a codificação disponibilizada)

 - 13.1) Implemente a seguinte classe:



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

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