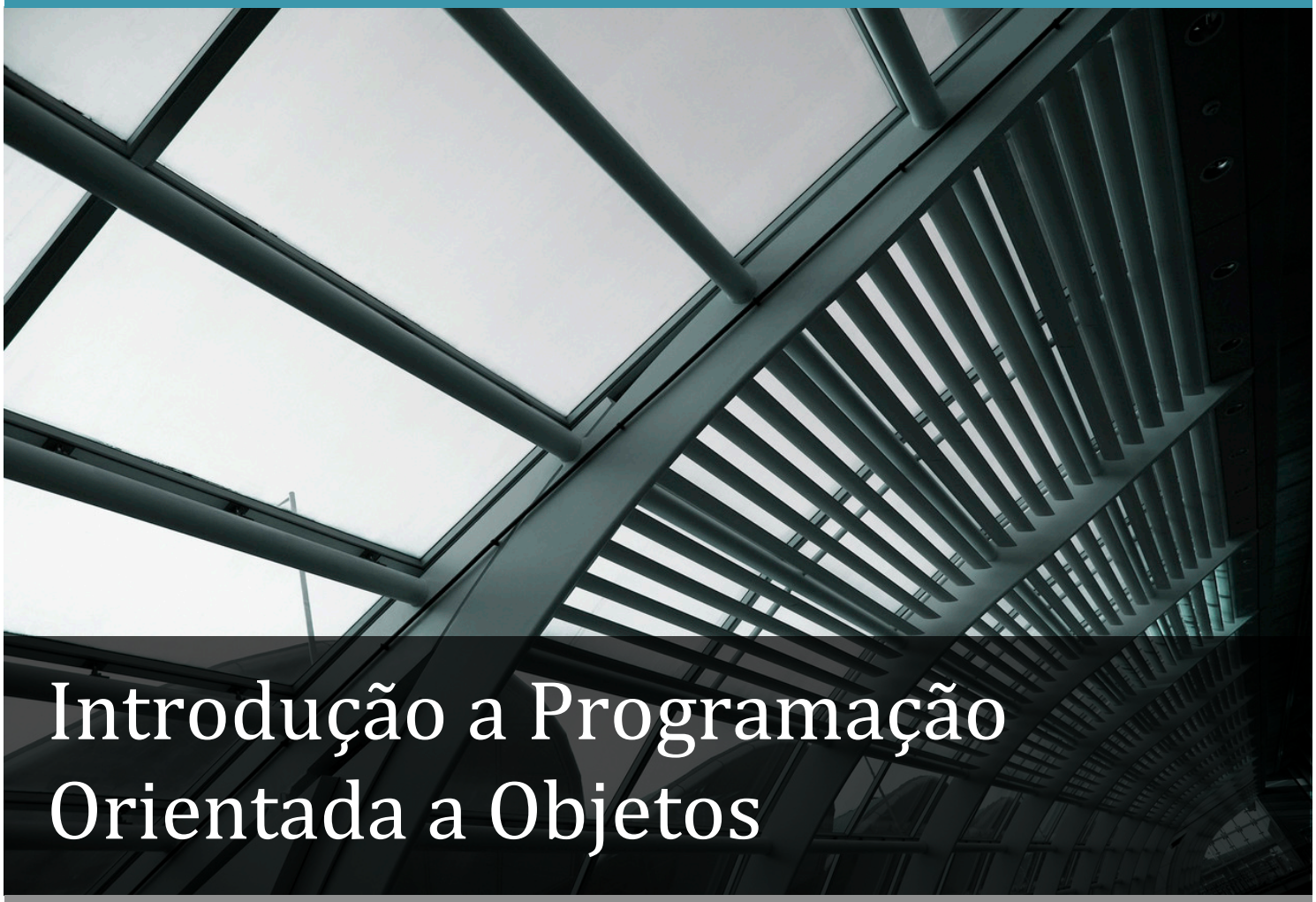


Introdução a Programação Orientada a Objetos

Prof. Igor Takenami

Versão 3.0

www.takenami.com.br
2011

Caro Aluno,

Este módulo visa fixar os conceitos de Orientação a Objetos discutidos em nossas aulas. Seu conteúdo também servirá como material de referência, não só para a sala de aula, mas para ser utilizando quando necessário.

As problematizações apresentadas através dos estudos de caso serão laboratórios para as aplicações dos conceitos e também deve ser guardado como material de apoio e referência.

Lembre que a leitura complementar sobre o tema é obrigatório e ajudará no seu amadurecimento profissional.

O evolução deste material é feito através das interações com os alunos, então, sua contribuição é fundamental. Qualquer dúvida ou sugestão envie um email para: itakenami@gmail.com.

Grande abraço e bom estudo!

Atenciosamente,

Igor Takenami

Índice

AVALIAÇÕES

Avaliação	Data	Peso
Observações		

1. OBJETO

Unidade que utilizamos para representar **abstrações** de um sistema computacional. No mundo real podemos levar em consideração que um objeto é tudo que podemos tocar. A interação entre estes objetos formam grupo de objetos mais complexos que agrupado a outros grupos de objetos complexos dão origem ao sistemas naturais, como por exemplo o funcionamento de um carro. Outro exemplo disto é o corpo humano como descrito por Alan Kay em seu estudo da ***Analogia Biológica*****.** Um objeto é **único** e possui **atributos** que definem **caraterísticas** e/ou **estado** assim como possuem capacidade de realizar **ações** que chamamos de **métodos** ou **funções**. Normalmente se diz que um objeto é uma **instância** de uma **Classe**.

2. CLASSE

Estrutura (**molde**) que define os **atributos** e/ou **estados** de um conjunto de objetos com características similares. Além das **caraterísticas**, uma classe define o comportamento de seus objetos (**ações que o objeto pode fazer**) através de métodos. Uma classe descreve os serviços (**ações**) providos por seus objetos e quais informações eles podem armazenar.

2.1. Atributo

Define as **caraterísticas** e/ou **estado** de uma classe. Após a classe ser **instanciada** em um **objeto** os **atributos** vão receber **valores** (caraterísticas e/ou estados) que definem o **objeto**.

2.2. Métodos

Conjunto de **ações** que um determinado **objeto** pode **executar**. Estas ações definem o que um objeto pode fazer. Os métodos de um objeto são acionados por outros **objetos.** Isto quer dizer que os objetos se comunicam através de métodos. Em O.O. isto é mais conhecido como **troca de mensagens**.

2.3. Construtor

Método especial definido na **classe** e executado no momento que o **objeto** é **instanciado**. O que diferencia o construtor de outro método é que ele não possui retorno e deve ter o mesmo nome da classe. Assim como um método normal o construtor pode receber parâmetros e estes normalmente são utilizados para inicializar os valores dos atributos do objeto.

3. UML

UML (Unified Modeling Language) é uma **linguagem** para representação de **modelos visuais** com um significado **especifico** e **padronizado**. Os modelos são representados através de **diagramas** que possuem **semântica** própria. A UML permite que os programadores visualizem os produtos de trabalho em diagramas padronizados. **UML não é uma linguagem de programação**, muito pelo contrário, é utilizado para fazer todo o **projeto** antes de desenvolver um software.

A UML é composta por diversos diagramas. O diagrama que representa a descrição visual das classes que um sistema possui é o **Diagrama de Classes**. A representação de uma classe no Diagrama de Classes é feita da seguinte forma:

Nome da Classe
Atributos da Classe
Métodos da Classe

4. ESTRUTURA BÁSICA DE UMA CLASSE

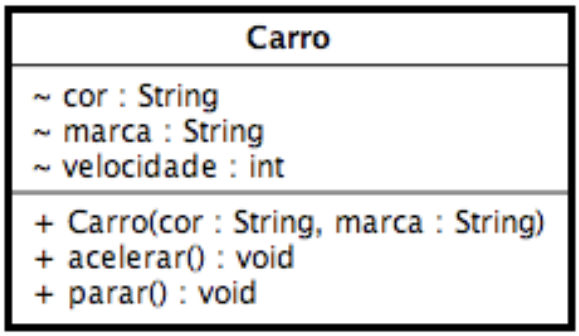
```
public class Carro {

    String cor;
    String marca;
    int velocidade;

    public Carro(String cor, String marca){
        this.cor = cor;
        this.marca = marca;
        velocidade = 0;
    }
    public void acelerar(){
        velocidade++;
    }
    public void parar(){
        velocidade = 0;
    }
}
```

Atributos

Construtor

Métodos

4.1. Representação UML



5. INSTANCIANDO UMA CLASSE

```
public class Main {
    public static void main(String[] args) {
        Carro fusca = new Carro("preto", "volkswagen");
        fusca.acelerar();
        fusca.acelerar();
        fusca.parar();
    }
}
```

6. ABSTRAÇÃO

Representa as **características** que devem conter em uma classe para atender a um determinado problema. Em um sistema O.O. (Orientado a Objetos), se quisermos representar uma classe com todas as características existentes no mundo real poderíamos explorar uma infinidade de informações e muitas vezes as mesmas não necessitam ser levadas em consideração. O conceito de abstração visa à construção de uma classe com somente as **características necessárias** para atender a um determinado **problema**. Esta classe representa um determinado **ponto de vista** ou **abstração** do problema.

Para desenvolver aplicações O.O. é necessário identificar os **objetos na vira real**, extrair a **classe que aquele objeto pertence** e **selecionar os atributos e métodos** que serão necessários levando em consideração o modelo computacional que está sendo desenvolvido.

Lembre que o processo é:

1. Identificar os objetos na vida real;
2. Extrair a classe a qual pertence estes objetos;
3. Selecionar os atributos e métodos que farão parte da classe levando em consideração a abstração do que está sendo desenvolvido.

O modelo O.O. é o conjunto de classes que visam resolver um problema computacional levando em consideração como este problema é abordado no mundo real.

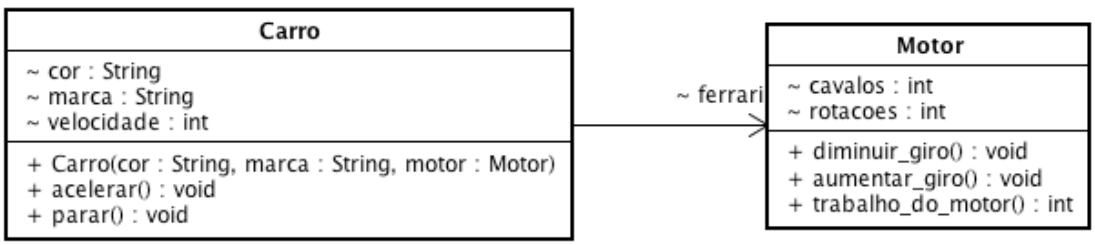
7. ASSOCIAÇÕES

Associação é a forma como uma classe se relaciona com outra classe. Uma classe pode conter atributos que geram instâncias de outra classe, ou seja, uma classe pode conter outra classe como atributo. Quando isto ocorre dizemos que uma classe possui outra classe associada a ela. Observe o exemplo abaixo:

```
public class Carro {

    String cor;
    String marca;
    int velocidade;

    Motor ferrari;

    public Carro(String cor, String marca){

        this.cor = cor;
        this.marca = marca;
        velocidade = 0;

        ferrari = new Motor();
    }

    public void acelerar(){
        ferrari.aumentar_giro();
        velocidade = ferrari.trabalho_do_motor() / 1000;
    }

    public void parar(){
        velocidade = 0;
    }

}
```

```
public class Motor {
    int cavalos;
    int rotacoes;

    public Motor(){
         cavalos = 110;
         rotações = 0;
    }

    public void diminuir_giro(){
        System.out.println("Diminuindo as rotações do motor.");
        rotacoes-=1000;
    }

    public void aumentar_giro(){
        System.out.println("Aumentando as rotações do motor.");
        rotacoes+=1000;
    }

    public int trabalho_do_motor(){
        return rotacoes * cavalos;
    }
}
```

7.1. Representação UML



ESTUDO DE CASO I

1) Utilizando os conhecimentos de O.O. construa um programa utilizando as informações abaixo.

Um carro possui as seguintes características:

- modelo;
- velocidade;
- aceleração;
- marcha.

O carro pode realizar as seguintes ações:

- ligar e desligar;
- acelerar e desacelerar;
- virar a direita e esquerda;
- marcha para cima e para baixo.

2) Utilizando os conhecimentos de O.O. construa um programa utilizando as informações abaixo.

Uma televisão possui as seguintes características:

- tamanho de tela (em polegadas);
- volume: de 1 a 10 iniciando em 5 (somente no construtor);
- marca;
- voltagem (220 e 110);
- canal.

A televisão pode realizar as seguintes ações:

- ligar: ao ligar a televisão deve imprimir seu consumo. O consumo deve ser definido pela voltagem multiplicada pela quantidades de polegadas;
- desligar;
- aumentar e diminuir o volume;
- subir e descer canal.

3) Utilizando os conhecimentos de O.O. construa um programa utilizando as informações abaixo.

As características de um DVD devem ser definidas de acordo com as informações a seguir. Ao ser criado o DVD inicialmente está desligado. Seu volume pode alterar de 1 a 5 sendo que o nível inicial é 2. É possível inserir um filme no DVD. O filme possui as seguintes caraterísticas: nome, categoria e duração. As seguintes operações podem ser realizadas pelo DVD:

- ligar e desligar;
- aumentar e diminuir volume;
- inserir filme;
- play e stop.

O programa deve obedecer as seguintes regras:

- Só é possível fazer qualquer uma destas operações se o DVD estiver ligado;
- Só é possível dar play no DVD se existir algum filme inserido;
- Só é possível dar stop se o DVD estiver em play;
- Ao dar play deve aparecer o nome e a duração do filme que está sendo exibido.

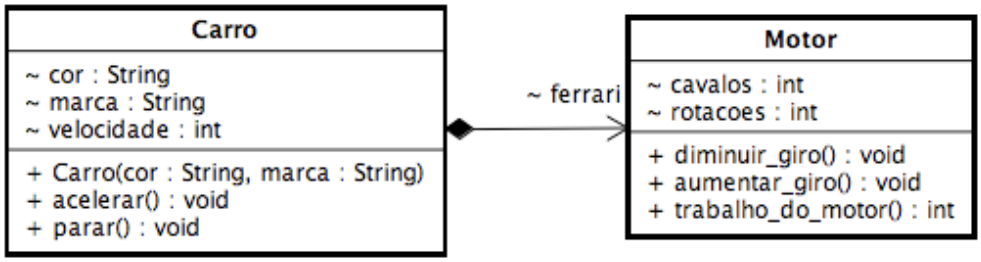
8. AGREGAÇÃO E COMPOSIÇÃO

São tipos específicos de associações entre classes. Em O.O. as associações normalmente podem ter um significado ampliado utilizando os conceitos de Agregação e Composição. Na **Agregação** a classe contida **não é instanciada no escopo da classe principal**, ou seja, não depende da principal para existir e **normalmente é passada por parâmetro.** Isto quer dizer que, caso o objeto da classe principal seja desalocado da memória, os objetos que estão associados a ele serão mantidos em memória. Na **Composição** a classe contida é **instanciada pela classe principal** sendo esta o motivo pelo qual a classe criada existe. Isto quer dizer que, quando uma classe principal é retirada da memória, as outras classes também são.

8.1. Composição

```
public class Carro {

    String cor;
    String marca;
    int velocidade;

    Motor ferrari;

    public Carro(String cor, String marca){

        this.cor = cor;
        this.marca = marca;
        velocidade = 0;

        ferrari = new Motor();
    }

    public void acelerar(){
        ferrari.aumentar_giro();
        velocidade = ferrari.trabalho_do_motor() / 1000;
    }

    public void parar(){
        velocidade = 0;
    }
}
```

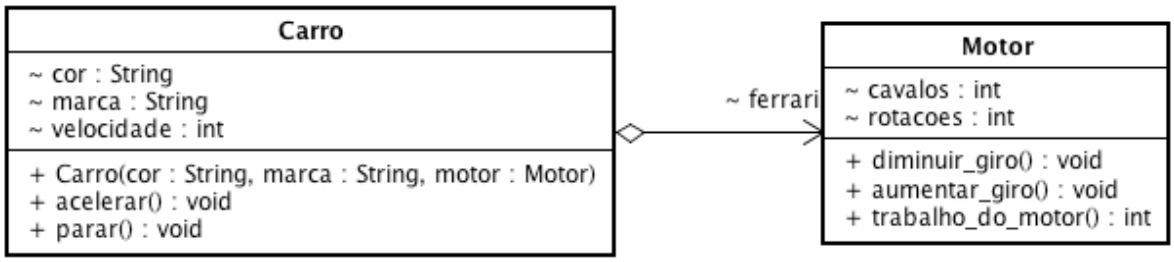
8.2. Representação UML



8.3. Agregação

```
public class Carro {

    String cor;
    String marca;
    int velocidade;

    Motor ferrari;

    public Carro(String cor, String marca, Motor motor){

        this.cor = cor;
        this.marca = marca;
        velocidade = 0;

        this.ferrari = motor;
    }

    public void acelerar(){
        ferrari.aumentar_giro();
        velocidade = ferrari.trabalho_do_motor() / 1000;
    }

    public void parar(){
        velocidade = 0;
    }
}
```

8.4. Representação UML



9. ENCAPSULAMENTO

Separar o programa em partes, tornando cada parte mais isolada possível uma da outra. A idéia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. Permite utilizar o objeto de uma classe sem necessariamente conhecer sua implementação. Uma grande vantagem do encapsulamento é que toda parte encapsulada pode ser modificada sem que os usuários da classe em questão sejam afetados. O encapsulamento também protege o **acesso direto** aos **atributos** de uma **instância** fora da classe onde estes foram criados. Esta proteção é feita através de **modificadores de acesso** restritivos, sobre os atributos definidos na classe.

10. HERANÇA

Capacidade que uma classe tem de herdar as características e comportamentos de outra classe. Na herança a classe pai é chamada de **superclasse** e a filha de **subclasse**. O objetivo da herança é **especializar** o entendimento de uma classe criando novas **características** e **comportamentos** que vão além da **superclasse**. Ao mesmo tempo que a **especialização** amplia o entendimento de uma classe, a **generalização** vai no sentido inverso e define um modelo menos especializado e mais genérico. Em algumas linguagens como o Java só é permitido herdar de uma única classe, ou seja, não permite herança múltipla. Ex:

Superclasse de Morcego - Generalização	**Subclasse de Mamifero - Especialização**

`public class Mamifero {`
`  int altura;`
`  double peso;`
`  public void mamar(){`
`  System.out.println("Mamifero mamando");`
`  }`
`}`
	`public class Morcego extends Mamifero {`
	`  int tamanho_presa;`
	`  public void voar(){`
	`  System.out.println("Morcego Voando");`
	`  }`
	`}`

A classe Morcego possui as seguintes características: **altura**, **peso** e **tamanho_presa** e responde pelos seguintes métodos: **mamar** e **voar**.

Observe que se usarmos os princípios de lógica podemos dizer que **todo morcego é mamífero** porém **NÃO** podemos afirmar que **todo mamífero é morcego**. Com este conceito em mente observe o seguinte código:

```
1 - Mamifero animal_mamifero = new Morcego();

2 - Morcego batman = new Mamifero();
```

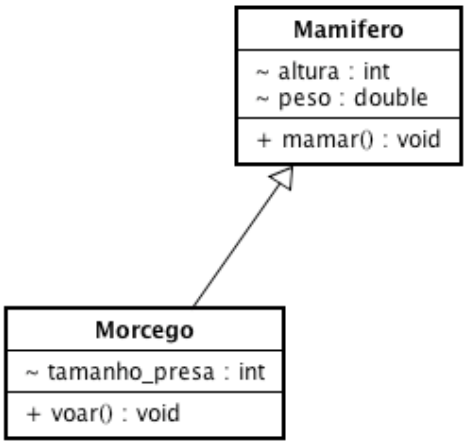
Com base no que foi dito até aqui podemos deduzir que o item 2 deve causar um erro já que não é possível garantir que todo mamífero seja um morcego. Já o item 1 pode parecer estranho, pois a variável é do tipo **Mamifero** e o objeto para o qual a variável se refere é do tipo **Morcego**, porém devemos saber que toda variável pode receber um objeto que seja compartível com o seu tipo e neste caso todo **Morcego CERTAMENTE** é um **Mamifero**. Observe agora este código:

```
Mamifero animal_mamifero = new Morcego();

animal_mamifero.mamar();

animal_mamifero.voar();

Morcego batman = (Morcego) animal_mamifero;

batman.voar();
```

O código apresentado acima funciona? Justifique sua resposta.

Com base no código apresentado podemos concluir que todo Morcego é um Mamífero porem não pode realizar todas as ações de um morcego já que a variável que recebe o objeto é do tipo **Mamifero**. Para que possamos fazer com que o Morcego voe é necessário criar uma nova variável do tipo Morcego e atribuir o objeto que estava na variável animal_mamifero. Este tipo de operação recebe o nome de **TYPE CAST**.

10.1. Representação UML



11. CLASSE ABSTRATA

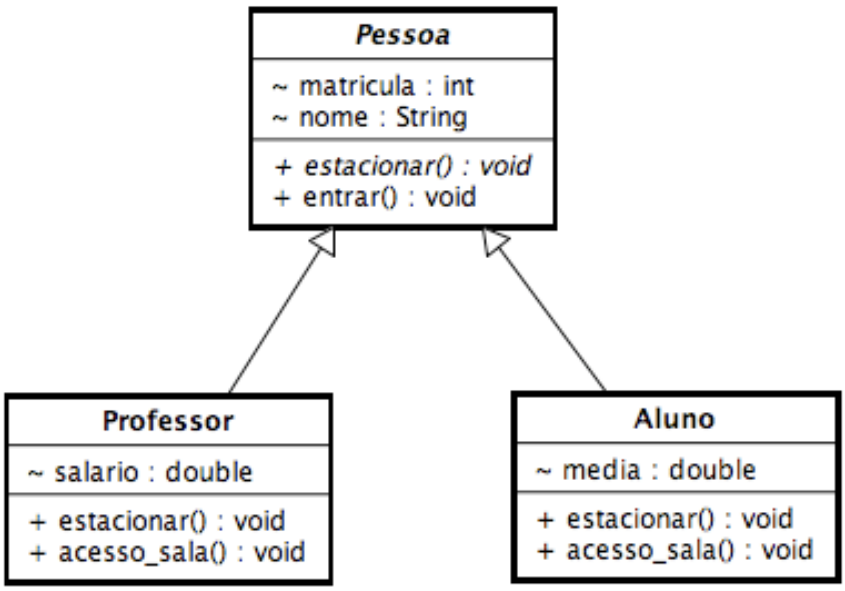
Classe que **não produz instância**, ou seja, não pode ser **instanciada diretamente**. Uma classe abstrata possui características que devem ser implementadas por classes filhas que vão especializar suas características e comportamento, permitindo instanciar o objeto indiretamente através de seu filho. Além de não produzir instância, uma classe abstrata permite a declaração de métodos abstratos. Os métodos abstratos são obrigatoriamente implementados pelas classes filhas concretas, quando a mesma herda de uma classe abstrata. Ex:

```
public abstract class Pessoa {

    int matricula;
    String nome;

    public abstract void estacionar();

    public void entrar(){
        System.out.println("Entrando na Faculdade");
    }

}
```

```
public class Aluno extends Pessoa {

    double media;

    public void estacionar() {
        System.out.println("Estacionando na área para estudante...");
    }

}
```

```
public class Professor extends Pessoa {

    double salario;

    public void estacionar() {
        System.out.println("Estacionando nas vagas de professor");
    }

}
```

11.1. Representação UML



ESTUDO DE CASO II

Passo 1: Criar uma classe abstrata chamada **Operacao** com o atributo **valor** do tipo double e um método abstrato chamado **operar()** que retora um valor do tipo double.

Passo 2: Crie uma classe **Debito** e outra **Credito** que herda as caraterísticas de **Operacao**. O construtor de **Debito** e **Credito** deve receber o **valor** da operação e atribuir este valor a variável definida em **Operacao** (superclasse). Estas classes (**Debito** e **Credito**) devem ter um método **operar()** que deve ser programado de acordo com sua finalidade: **operar()** da classe **Debito** retorna o **valor** negativo pois a operação deve ser um debito enquanto a o método **operar()** de **Credito** retorna o **valor** positivo.

Passo 3: Criar a classe **ContaCorrente** com o atributo **valor** do tipo double que inicia com 0. Esta classe possui um método **executarOperacao(Operacao opr)** que recebe um parâmetro do tipo **Operacao** que vai operar o valor da conta correte (se for debito diminui, se for credito soma). Esta classe também possui o método **getSaldo()** que retorna o valor do saldo atual.

Passo 4: Crie a classe **Correntista** com os seguintes atributos: **nome** (do tipo String) e **conta** (do tipo **ContaCorrente**). O construtor de **Correntista** deve receber seu **nome**. A classe deve ter 2 métodos: **public String getNome()** e **public ContaCorrente getContacorrente()**. Estes métodos retornam o nome e a conta respectivamente.

Passo 5: Crie a classe **Banco** como descrito no código abaixo:

```
public class Banco {

    Correntista c1,c2,c3;

    public Banco(String correntista1,String correntista2,String correntista3) {
        c1 = new Correntista(correntista1);
        c2 = new Correntista(correntista2);
        c3 = new Correntista(correntista3);
    }

    public Correntista getCorrentista(String nome){
        if(c1.getNome().equals(nome)){
            return c1;
        }
        if(c2.getNome().equals(nome)){
            return c2;
        }
        if(c3.getNome().equals(nome)){
            return c3;
        }
        return null;
    }

    public void debitar(String nome_correntista, double valor){
        Debito d = new Debito(valor);
        getCorrentista(nome_correntista).getContacorrente().executarOperacao(d);
    }

    public void creditar(String nome_correntista, double valor){
        Credito c = new Credito(valor);
        getCorrentista(nome_correntista).getContacorrente().executarOperacao(c);
    }

    public double getSaldo(String nome_correntista){
        return getCorrentista(nome_correntista).getContacorrente().getSaldo();
    }

    public void transferir(String nome_correntista_origem, String nome_correntista_destino, double valor){
        debitar(nome_correntista_origem, valor);
        creditar(nome_correntista_destino, valor);
    }
}
```

12. SOBRESCRITA

Substitui o comportamento de uma **subclasse** quando herdada da **superclasse**. Na prática a sobrescrita **reescreve** um **método** que já tinha sido definido na **superclasse**. Ex:

```
public class Mamifero{

  ...

  public void andar(){
    System.out.println("Mamifero andando");
  }

}
```

```
public class Golfinho extends Mamifero{

  ...

  public void andar(){
    System.out.println("Golfinho Nadando");
  }

}
```

13. SOBRECARGA

É a capacidade de definir métodos com o **mesmo nome**, ou seja, que **possuem a mesma funcionalidade**. Para que a sobrecarga aconteça é necessário que a **assinatura** seja diferente. A mudança na assinatura ocorre **alterando a quantidade e/ou tipo de parâmetros** que um método recebe. Ex:

```
public int somar(int v1, int v2){
    return v1 + v2;
}

public int operar(int v1, int v2){
    return somar(v1, v2);
}

public int operar (char op, int v1, int v2){
    switch(op){
        case '+':
            return somar(v1, v2);
            break;
        case '-':
            return subtrair(v1, v2);
    }
}
```

ESTUDO DE CASO III

MODELO O.O.

Um semestre possui 4 avaliações (estas avaliações devem ser informados no construtor da classe Semestre). As avaliações podem ser: Prova, Trabalho ou Interdisciplinar. As avaliações podem ter pesos diferentes sendo que a soma dos pesos deve dar 7 (a definição dos pesos é feito na classe).

A classe Semestre deve possuir a capacidade de informar se o peso das avaliações é válido (=7), a média e o resultado (APROVADO ou REPROVADO). Uma Avaliação deve ter a capacidade de informar a nota, peso e a a quantidade de pontos (peso*nota).

O peso de cada avaliação deve ser informado na definição da classe sendo que uma avaliação também pode ser composta por diversas notas, que geram uma ÚNICA nota para a avaliação. Ex: Um trabalho pode ser formada por 3 notas sendo que a nota final do trabalho é a soma destas notas dividido por 3.

PROBLEMA

Para avaliar seus alunos a disciplina de Programação Orientada a Objetos utiliza os seguinte critérios de avaliação:

- 2 provas de peso 2;
- 1 trabalho de peso 1, composto por 3 notas;
- 1 interdisciplinar de peso 2, composto por 4 notas (3 notas de peso 2 e 1 nota de peso 4).

Construa um programa seguindo o modelo O.O. citado para resolver o problema apresentado.

14. INTERFACE

O objetivo de uma interface em Orientação a Objetos é definir um conjunto de **comportamentos** (**métodos**) que devem obrigatoriamente ser implementados pelas classes que utilizam a interface. Diferente da herança uma classe Java pode implementar **n** interfaces. Ex:

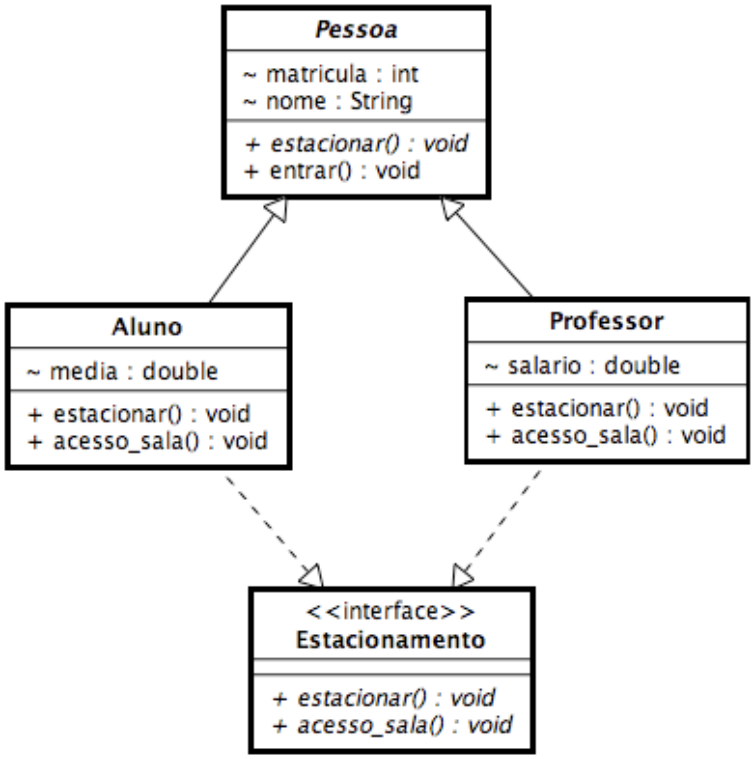
```
public abstract class Pessoa {

    int matricula;
    String nome;

    public void entrar(){
        System.out.println("Entrando na Faculdade");
    }
}
```

```
public interface Estacionamento {
    public void estacionar();
    public void acesso_sala();
}
```

```
public class Aluno extends Pessoa implements Estacionamento {

    double media;

    public void estacionar() {
        System.out.println("Estacionando na área para estudante...");
    }

    public void acesso_sala() {
        System.out.println("Acesso a sala dos ALUNOS");
    }

}
```

```
public class Professor extends Pessoa implements Estacionamento {

    double salario;

    public void estacionar() {
        System.out.println("Estacionando nas vagas de professor");
    }

    public void acesso_sala() {
        System.out.println("Acesso a sala dos professores");
    }

}
```

14.1. Representação UML



ESTUDO DE CASO IV

Você foi contratado para desenvolver um sistema de cadastro de projetos. Para isto sua empresa adquiriu um pacote de componentes em Java. Basicamente este pacote é composto por 1 classe chamada **TabelaDados** e 1 Interface chamada **IRegistro**.

Nome da Classe: TabelaDados	
Métodos	**Descrição**
public void salvar(IRegistro reg)	Salva o objeto na tabela
public void excluir(IRegistro reg)	Exclui o objeto da tabela
public IRegistro buscar(String conteudo)	Busca na tabela por um objeto que atende ao critério de consulta definido
Nome da Interface: IRegistro	
Métodos	**Descrição**
public Object getIdentificador()	Define um identificador para o objeto que será salvo. O identificador deve ser único.
public String getConteudoBusca()	Define qual a informação que será comparada quando for feito uma busca em TabelaDados.
public void imprimeDados()	Imprime no console os dados referente ao objeto.

A classe **TabelaDados** é utilizada para armazenar um objeto qualquer e não é necessário saber como ela implementa isto.

1) Qual o conceito de OO associado a esta afirmativa? Explique este conceito.
2) Observe que a classe **TabelaDados** pode **salvar**, **excluir** ou **buscar** recebendo como parâmetro uma interface (**IRegistro**). O que isto quer dizer?

O sistema deve ser implementado de acordo com as seguintes informações: Um **Projeto** possui as seguintes características: código, nome, descrição, data inicio, data fim, responsável que é um **Analista** e 2 participantes que são **Programadores**. Todos os integrantes do projeto (1 **Analista** e 2 **Programadores**) são **Membros** do projeto e possuem as seguintes características em comum: matricula, nome e salario. O salario do **Analista** é de R$ 5.000,00 enquanto do **Programador** é de R$ 4.000,00.

3) Qual o conceito de O.O. envolve as classes: **Membro**, **Analista** e **Programador**? Qual o tipo de classe de **Membro** ? Quais as suas características ?
4) Implemente o modelo O.O. E o diagrama de classes para o problema. A aplicação deve funcionar de acordo com o seguinte programa de inicialização:

```
public class App {

    public static void main(String[] args) {

        //Cria o objeto que vai salvar os dados
        TabelaDados tab = new TabelaDados();

        //Define os partiicpantes do projeto
        Analista analista1 = new Analista(1, "Joao");
        Programador prog1 = new Programador(2, "Maria");
        Programador prog2 = new Programador(3, "Carlos");


        //Cadastra o projeto
        Projeto p = new Projeto();
        p.setCodigo(1);
        p.setNome("FJA");
        p.setDescricao("Descricao do Projeto");
        p.setResponsavel(analista1);
        p.setProgramador1(prog1);
        p.setProgramador2(prog2);
        tab.salvar(p);

        //Busca o projeto e imprime as informações
        Projeto projeto_buscado = (Projeto)tab.buscar("FJA");
        projeto_buscado.imprimeDados();
    }
}
```

15. MODIFICADORES DE ACESSO

Como o próprio nome diz, os modificadores de acesso modificam a visibilidade dos **atributos** e **métodos** de uma classe. Os principais modificadores de acesso são:

- **public** – pode ser acessado por qualquer objeto;
- **private** – só permite acesso da própria classe;
- **protected** – classes no mesmo pacote ou subclasses da classe onde o elemento foi definido, é importante notar que para os elementos sejam acessados na subclasse não é preciso estar no mesmo pacote da classe pai;
- **default** – modificador aplicado quando nada é informado. Da acesso a classes do mesmo pacote.

16. MÉTODOS E VARIÁVEIS ESTÁTICAS

Em programação O.O. podemos definir métodos e atributos como estáticos. Um atributo estático é um atributo que **pertence a classe e NÃO ao objeto**. Sendo assim, quando alteramos um valor em uma variável estática esta alteração é refletida em todas as instâncias de objetos. Um método estático, assim como o atributo, **pertence a classe** e pode ser chamado sem a necessidade de se criar uma instância. Quando criamos um método estático toda as chamadas são **realizadas pela classe** e **NÃO** pelo objeto.

Atributo estático:

```
public static String nome = "JOAO";
```

Método estático:

```
public static boolean validarCPF(){...}
```

17. IMUTABILIDADE

Imutabilidade é quando algo não pode ser modificado. Quando aplicamos este conceito em O.O. a uma variável, por exemplo, temos o que conhecemos como **constante**. A imutabilidade em Java é aplicada através da palavra reservada **final**. Podemos utilizar **final** em:

Classes – Faz com que uma classe não possa ser herdada.

Ex: `public final class PessoaFisica{...}`

Métodos – Faz com que o método não possa ser reescrito.

Ex: `public final void mudarValor(Object o){...}`

Atributos – Faz com que o atributo só receba valor uma única fez (constante).

Ex: `private final String nome = "João"`

ESTUDO DE CASO V

Você foi contratado para desenvolver um programa O.O. para manipular um celular que será desenvolvido especialmente para os funcionários da empresa XPTO. Para isto, sua empresa fechou um acordo com uma operadora que forneceu a classe **Operadora**:

Nome da Classe: Operadora	
Métodos	**Descrição**
public void conectar(String numero)	Conecta o celular na antena da operadora e efetua a ligação na rede móvel.
public void desconectar()	Desconecta o celular da rede móvel.

A classe **Operadora** também aciona o método **atender,** do celular, passando como parâmetro o número de quem está ligando para seu aparelho. Além disto, outra divisão da empresa construiu uma classe chamada **Agenda**. Esta classe contem o nome e o telefone de todos os funcionários da XPTO:

Nome da Classe: Agenda	
Métodos	**Descrição**
public static String getNumero(String nome)	Retorna o número do funcionários a partir do nome.
public static String getNome(String numero)	Retorna o nome do funcionário a partir do número.

O Engenheiro de Software da empresa projetou o celular de acordo com a folha seguinte.

1) Cite e explique o conceito O.O. de **Abstração** por traz da classe **Agenda**.

2) Observe as linhas que vão de **07** a **11** da classe **Celular** e explique: O que é este método? Qual o motivo dele ser publico? O que acontece quando instanciamos os objetos nas linhas **08** e **09**.

3) Descreva as caraterísticas de uma classe abstrata e uma interface?

4) Codifique os métodos da classe **Celular** e desenvolva as classes **Display3Linhas** (Tela com somente 3 linhas de informação), que implementa a interface **Display**, e **TecladoQwerty** (Teclado com letras e números) que herda a classe **Teclado**. Utilize as seguintes especificações:

a) O Método desligar do celular deve desconectar da rede da operadora, limpar a tela e o valor digitado no Teclado;
b) Para fazer uma ligação o celular deve obter o número que será informado pelo teclado e exibir uma mensagem na tela. A mensagem de 3 linhas conterá: Informação de discando, nome da pessoa chamada e o número discado. Após isto deve-se conectar na rede móvel;
c) Ao atender uma ligação o celular deve exibir nas 3 linhas: Informação de recebendo ligação, nome e o número de quem está discando;
d) A classe TecladoQwerty só digita texto, ou seja, o usuário digita o nome e através da classe Agenda teve-se buscar o número para o celular fazer a ligação.

CÓDIGO FORNECIDO

```
01 public class Celular {
02
03    private Display tela;
04    private Teclado botoes;
05    private Antena antena;
06
07    public Celular(){
08        tela = new Display3Linhas();
09        botoes = new TecladoQwerty();
10        antena = new Antena();
11    }
12    public Display getDisplay(){
13        //Implemente aqui
14    }
15    public Teclado getTeclado(){
16        //Implemente aqui
17    }
18    public void desligar(){
19        //Implemente aqui
20    }
21    public void ligar(){
22        //Implemente aqui
23    }
24    public void atender(String numero){
25        //Implemente aqui
26    }
27
28 }
```

```
01 public abstract class Teclado {
02
03    private String valor_digitado;
04
05    public void limpar(){
06        valor_digitado = "";
07    }
08    public void voltar(){
09        valor_digitado = valor_digitado.substring(0,
valor_digitado.length()-1);
10    }
11
12    public void digitar(char tecla){
13        valor_digitado = valor_digitado + tecla;
14    }
15
16    public String getValorDigitado(){
17        return valor_digitado;
18    }
19
20    public abstract String getNumeroDiscar();
21 }
```

```
01 public interface Display {
02     public void exibirTelaDiscagem(String[] info);
03     public void exibirTelaAtende(String[] info);
04     public void limpar();
05 }
```

```
01 public class Antena {
02    private Operadora op;
03    public Antena(){
04       op = new Operadora();
05    }
06    public Operadora getSinalOperadora(){
07       return op;
08    }
09 }
```

```
public class App {

    public static void main(String[] args) {

        Celular xpto = new Celular();

        xpto.getTeclado().digitar('I');
        xpto.getTeclado().digitar('G');
        xpto.getTeclado().digitar('O');
        xpto.getTeclado().digitar('R');

        xpto.ligar();
        xpto.desligar();

    }

}
```

18. PACOTES

Forma de organizar classes dentro de uma estrutura de árvores. Podemos entender a estrutura de árvores como os diretórios do sistema operacional. O nome completo de uma classe é definido pelo seu pacote e o nome. Isto permite criar classes de mesmo nome que estejam em pacotes diferentes. Para que não seja necessário informar o nome completo de uma classe para utiliza-lá é possível importar o conteúdo de um determinado pacote utilizando somente o nome da classe para criar novas instâncias.

```
package animais.verterbrados;

public class Mamifero {

    int altura;
    double peso;

    public void mamar(){
        System.out.println("Mamifero mamando");
    }

}
```

```
import animais.verterbrados.Mamifero;
public class App{
      public static void main(String args[]){
            Mamifero m = new Mamifero();
      }
}
```

19. POLIMORFISMO

É a capacidade que um mesmo objeto possui de **responder a um mesmo método de forma diferente**. A esta altura você já deve ter utilizado diversas vezes o conceito de **Polimorfismo** sem saber. Aplicar o Polimorfismo só é possível quando utilizados os conceitos de classe abstrata (**através de herança**) ou interfaces.

ESTUDO DE CASO VI

Uma empresa Chinesa resolveu entrar no mercado Brasileiro para vender seu Mp3 Player. O dispositivo possui tela touch screen e é composto por 4 módulos com as seguintes funcionalidades: ouvir música, assistir filme, ler texto e jogar. O hardware e o software principal serão produzidos na China porém a gigante Chinesa terceirizou o desenvolvimento, de cada um dos módulos, por diferentes empresas no Brasil.

Para facilitar o desenvolvimento do módulo foi liberado parte do código-fonte do software principal. Basicamente o Player é implementado pelas classes **Mp3Player**, **Modulo**, **Tela** e **Botao**. As classes **Mp3Player** e **Modulo** foram liberadas pela empresa. O restante das classes estão descritas nos quadros:

```
public class Mp3Player {

  //Vetor contendo os módulos do player
  private Modulo modulos[];
  //Variável que indica a quantidade de módulos incluídos no player
  private int qtd_modulos;
  //Variável que aponta para o módulo selecionado
  private Modulo modulo_selecionado;
  //Índice do vetor que contém o módulo selecionado
  private int idx_modulo_selecionado;
  //Tela (touch screen) do player
  private Tela tela;

  //Construtor
  public Mp3Player() {

    modulos = new Modulo[4]; //Define quantos módulos o player terá (4)
    qtd_modulos = 0; //Informa que não existe nenhum módulo adicionado
    idx_modulo_selecionado = -1; //Informa que nenhum módulo foi
selecionado

    tela = new Tela(); //Cria a tela
  }

  //Adiciona um módulo ao player
  public void addModulo(Modulo md) {
    if (qtd_modulos < 4) { //Só poderá adicionar no máximo 4 módulos
      modulos[qtd_modulos] = md;
      qtd_modulos++;
    }
  }

  //Muda o módulo selecionado
  public void mudarModulo() {
    //Caso esteja no quarto módulo volta para o primeiro
    if (idx_modulo_selecionado == 3) {
      idx_modulo_selecionado = 0;
    } else {
      idx_modulo_selecionado++;
    }
    modulo_selecionado = modulos[idx_modulo_selecionado];

    //Adiciona na tela os botões definidos no módulo
    tela.addBotoes(modulo_selecionado.getBotoes());
  }

  //Método disparado quando o usuário toca nos botões da tela
  public void toqueTela() {
    //Informa ao médulo selecionado qual botão foi apertado
    modulo_selecionado.pressBotao(tela.getBotao());

    //Exibe na tela o retorno informado pelo módulo
    tela.showDisplay(modulo_selecionado.getDisplay());
  }

  //Liga o player
  public void ligar() {
    this.mudarModulo();
  }
}
```

```
public abstract class Modulo {

  //Retorna um vetor com os botões que serão utilizados pelo módulo
  public abstract Botao[] getBotoes();

  //Método que recebe o botão disparado através da interface touch
  public abstract void pressBotao(Botao b);

  //Retorna um String com as informações que serão exibidas na tela
  public abstract String getDisplay();

  //Metodo que retorna um inteiro que identifica o Botão no módulo
  public int getBotaoValor(Botao b) {
    return b.getValor();
  }
}
```

Tela – Classe responsável por exibir as informações na tela além de controlar o touch screen do Player.	
public **Botao** getBotao()	Retorna o Botão que o usuário tocou na tela.
public void addBotoes(Botao[] botoes)	Método utilizado para adicionar os botões **de um módulo** na Tela.
void showDisplay(String display)	Metodo utilizado para exibir informações na tela.

Botao – Classe que representa ações na tela(touch screen) e que o usuário pode apertar para operar o Player.	
public Botao(int valor,String img)	Construtor de Botão. Recebe como parâmetro um valor inteiro que será utilizado para identificar o botão e o nome do arquivo com a imagem que vai aparecer na tela.
public int getValor()	Retorna o valor inteiro utilizado para identificar o botão.
public String getImagem()	Retorna o nome do arquivo com a imagem que vai aparecer n tela.

Modulo – Classe abstrata que define as características comuns a cada módulo	
public abstract **Botao[]** getBotoes()	Retorna um vetor com os botões que serão utilizados pelo módulo.
public abstract void pressBotao(**Botao b**);	Método que recebe o botão disparado através da interface.
public abstract String getDisplay();	Método que retorna um String contendo as informações que serão exibidas no display da tela.
public int getBotaoValor(**Botao** b)	Método que retorna um valor inteiro que identifica o Botão no módulo.

Sua empresa foi contratada para desenvolver somente o módulo de música e para isto deve implementar a classe **Musica**. O módulo deve utilizar um chip especifico, que já vem pronto da fábrica, com todas as funcionalidades que devem ser implementadas. Para enviar instruções para o chip você deve utilizar a classe **ChipMusica** fornecida pelos Chineses e descrita no quadro abaixo:

ChipMusica – Classe que controla o chip com capacidade de executar músicas.	
public void play()	Inicia a música.
public void stop()	Para a música.
public void avancar()	Avança uma música.
public void voltar()	Volta para a música anterior.
public String getNomeMusica()	Retorna o nome da música atual.

O módulo de música terá 4 botões. O display da tela touch screen deve exibir a ação que está sendo realizada e a música atual. Também foi fornecido alguns arquivos com as seguintes imagens: **play.gif**, **stop.gif**, **avancar.gif**, e **voltar.gif**.

A classe que inicializa os objetos do modelo O.O. está descrita abaixo:

```
1   public class App {
2
3       public static void main(String[] args) {
4
5           Mp3Player player = new Player;
6
7           Modulo som = new Musica();
8           player.addModulo(som);
9
10          Modulo video = new Filme();
11          player.addModulo(video);
12
13          Modulo texto = new Leitura();
14          player.addModulo(texto);
15
16          Modulo game  = new Jogo();
17          player.addModulo(game);
18
19          player.ligar();
20      }
21
22  }
```

1) Faça uma comparação entre Classe Abstrata e Interface? No contexto do problema explique o motivo de **Modulo** ser uma classe abstrata e não uma interface.

2) Observe o trecho de código **Modulo som = new Musica();** descrito na linha 7 da classe App. Responda: A variável som **recebe uma instância da classe Modulo**?? Explique como é possível e análise o que ocorre quando fazemos isto.

3) Desenvolva o módulo pedido implementando a classe Musica que vai manipular o chip fornecido.

- Lembre dos conceitos de O.O. para entender o modelo O.O.;
- Desenhar o diagrama de classes pode ajudar bastante no entendimento do problema;
- O desenho dos objetos envolvido no problema também pode ajudar no entendimento;
- Procure entender o modelo O.O. para responder as perguntas e implementar a classe;
- Lembre que a classe a ser implementada é Musica e a mesma deve funcionar no Mp3 Player.

20. INSTÂNCIA DINÂMICA

O código abaixo apresenta a forma que utilizamos para instanciar uma classe.

```
Morcego batman = new Morcego();
```

O Java oferece outras alternativas, como o seguinte código:

```
Morcego batman = Class.forName("Morcego").newInstance();
```

A diferença de instanciar uma classe desta forma é que o nome da classe é passado como um **parâmetro do tipo String**. Isto quer dizer que a classe instanciada, pode ser definida em **tempo de execução** e não mais em **tempo de codificação**.

21. COLLECTIONS

Coleções são objetos que possuem a capacidade de armazenar um **conjunto de objetos**. As Collections são semelhantes aos vetores (arrays), a diferença é que a quantidade de elementos não precisa ser predefinida, ou seja, seu tamanho é **variável**.

21.1. Collections Framework

É o conjunto de interfaces, implementações e algoritmos para se trabalhar com coleções. O conjunto de classe que define a Collections Framework segue os princípios de O.O. e as principais classes estão descritas abaixo:

Collection – Interface que define um comportamento comum á todas as classes da Collections Framework. É o topo na hierarquia das coleções.

List – Interface que herda as características de Collection. Define uma coleção ordenada podendo ter elementos duplicados. A manipulação do elemento é feito pelo índice que inicia de 0. Possui as seguintes implementações concretas: **ArrayList**, **LinkedList** e **Vector**.

Set – Também herda as características de Collection. Não garante ordem entre os elementos. Armazena os elementos em qualquer posição em uma tabela hash. **Não** obtém os objetos pelo índice. Possui as seguintes implementações concretas: **HashSet** e **TreeSet**.

Map – Interface para mapeamento de chave e valores. Não permite chaves duplicadas sendo assim cada chave leva a somente 1 elemento. Possui as seguintes classe concretas: **HashMap** e **TreeMap**.

21.2. Iterando Objetos de Collections

As coleções que acessam seus objetos pelo índice podem ser listadas através de uma estrutura de repetição que incremente o índice e obtenha o objeto. Ex:

```
Collection col = new ArrayList();
col.add("a");
col.add("b");
col.add("c");
for(int x=0;x<col.size();x++){
      System.out.println((String)col.get(x));
}
```

As coleções que armazenam objetos através de **chaves** e **valores** não são acessadas pelo índice e por este motivo precisam ser percorridas de outra forma. Podemos percorrer através do objeto **Iterator**:

```
Iterator it = col.iterator();
while(it.hasNext()){
      System.out.println((String)it.next());
}
```

Mesmo uma coleção que acessa seus objetos pelo índice também pode ser **iterada**.

21.3. Generics

Como já vimos, uma classe do tipo Coleção pode armazenar objetos de qualquer outra classe e para recuperar o objeto é necessário fazer um CAST. Isto e necessário para garantir o retorno de um tipo definido. Quando usamos Generics podemos definir coleções para um tipo especifico. Ex:

```
ArrayList<Usuario> usuarios = new ArrayList<Usuarios>();
Usuario u1 = new Usuario();
Usuario u2 = new Usuario();
Usuario u3 = new Usuario();
usuario.add(u1);
usuario.add(u2);
usuario.add(u3);
for(int x=0;x<usuario.size();x++){
      Usuario u = usuarios.get(x);
}
```

Observe que definimos uma coleção especifica para o tipo **Usuario**. Temos um **ArrayList** do tipo **Usuario** e o objeto **usuarios** que por sua vez só aceita receber o tipo **Usuario**. Por este motivo não é necessário fazer o CAST no momento de obter o objeto da coleção. Em Java podemos utilizar Generics para diversas outras finalidades porém neste momento vamos nos ater a este.

22. SERIALIZAÇÃO

Transforma o objeto em uma **conjunto de bytes** que podem ser gravados em um arquivo ou transmitido pela rede. Para serializar um objeto é necessário marca-lo e para isto implementa-se a interface `Serializable` na classe que será serializada.

Para serializar e armazenar objetos em arquivos fazemos:

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("arquivo.dat"));
Morcego batman = new Morcego();
out.writeObject(batman);
out.close();
```

Para recuperar o objeto serializado fazemos:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("arquivo.dat"));
Morcego batman = (Morcego)in.readObject();
in.close();
```

23. SWING

Códigos úteis:

Painel com imagem:

```
public class Carta extends JPanel {
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        Image img = (new ImageIcon("c:/carta1.png")).getImage();
        g.drawImage(img, 0, 0, this);
    }
}
```

Acesso a configuração:

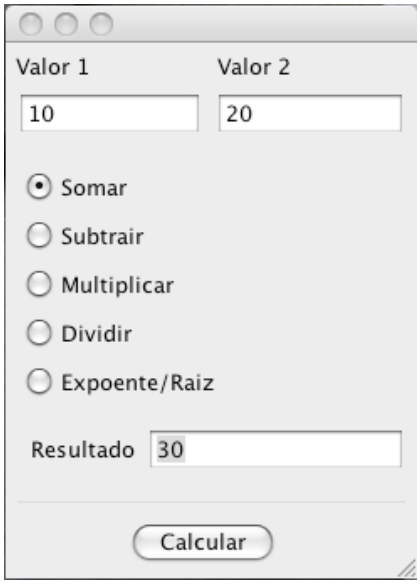
```
import java.util.ResourceBundle;
public class Main {
    public static void main(String[] args) {
        ResourceBundle rb = ResourceBundle.getBundle("config");
        System.out.println(rb.getString("nome"));
    }
}
```

ESTUDO DE CASO VII

1) Utilizado os conhecimentos de interface gráfica Swing (Conceitos de O.O.) crie um programa para fazer cálculos de acordo com a tela informada e as especificações a seguir.

O usuário deverá informar 2 números e selecionar uma das operações (Soma, Subtração, Multiplicação, Divisão e Expoente/Raiz). A operação de Expoente/Raiz possui 2 comportamentos diferentes dependendo da quantidade de valores informados pelo usuário:

a) Se o usuário informar somente o **Valor 1**, deixando o **Valor 2** em branco o sistema deve informar a Raiz;
b) Caso o usuário preencha os 2 Valores o programa deve informar o **Valor 1** elevado ao **Valor 2**.



OBS: Para este projeto utilize somente a classe que possui a interface gráfica.

2) Utilizado os conhecimentos de Modelo O.O. e a interface gráfica Swing modifique o jogo criado para utilizar classes especificas para os cálculos.

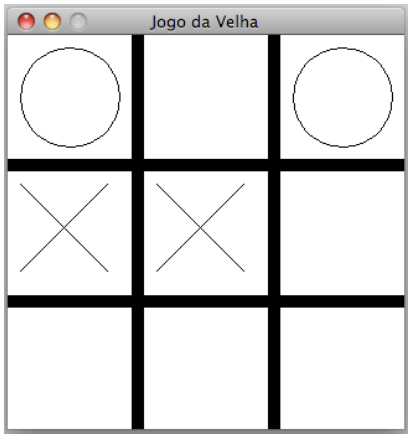
ESTUDO DE CASO VIII

Utilizado os conhecimentos de Modelo O.O. e a interface gráfica Swing crie um Jogo da Velha com as seguintes características:

O Jogo será jogado por 2 pessoas em um mesmo computador. O programa deve informar o vencedor (bola ou cruz). O Jogo será controlado pela classe Tabuleiro. Um Tabuleiro possui um vetor de String (casas[][]) e dois jogadores. O tabuleiro também possui a informação do jogador atual a executar a jogada. A classe Tabuleiro e Jogador estão descritas abaixo:

```
public class Tabuleiro {

    String[][] casas;
    Jogador jogador1,jogador2;
    Jogador jogador_vez;

    public Tabuleiro(Jogador j1, Jogador j2){
        //Implemente aqui
    }

    public void iniciarJogada(){
        if(jogador_vez==jogador1){
            jogador_vez=jogador2;
        }else{
            jogador_vez=jogador1;
        }
    }

    public void jogar(int x, int y){
        casas[x][y] = jogador_vez.peca;
    }

    public boolean existeVencedor(){
        Jogador player1=jogador_vez;
        //Implemente Aqui
    }

}
```

```
public class Jogador {

    String nome;
    String peca;

    public Jogador(String n, String p){
        //Implemente Aqui
    }

}
```



Tela Principal do Sistema

O jogo é operado através da interface gráfica (classe Tela) composta por um jFrame de fundo preto e 9 jPanel's na cor branca. A classe Tela está descrita abaixo:

```
public class Tela extends javax.swing.JFrame {

    Jogador igor;
    Jogador maria;
    Tabuleiro t;

    public Tela() {
        initComponents();
        this.setSize(320, 340);
        //Implemente Aqui
    }
    ...

    private void click00(java.awt.event.MouseEvent evt) {
        jogar(0, 0, jPanel1);
    }

    private void click01(java.awt.event.MouseEvent evt) {
        jogar(0,1, jPanel2);
    }

    private void click02(java.awt.event.MouseEvent evt) {
        jogar(0,2, jPanel3);
    }

    ...

    public void iniciarJogo(){
        igor = new Jogador("Igor","O");
        maria = new Jogador("Maria","X");
        t = new Tabuleiro(igor,maria);
    }

    public void desenharBola(JPanel p){
        p.getGraphics().drawOval(10, 10, 80, 80);
    }

    public void desenharCruz(JPanel p){
        p.getGraphics().drawLine(10, 10, 80, 80);
        p.getGraphics().drawLine(10, 80, 80, 10);
    }

    public void jogar(int x, int y, JPanel p){
            //Implemente Aqui
    }
    ...
}
```

a) Entenda como o modelo O.O. foi implementado;

b) Finalize a implementação do sistema;

c) Utilize o NetBeans para desenvolver este programa.

24. TRATAMENTO DE ERROS

O tratamento de erros em Java é feito em **tempo de execução** através do **tratamento de exceção**. Exceções são classes que seguem o modelo O.O. e são lançadas quando o sistema encontra um problema. Além de tratar erros as exceções também são utilizadas para validar regras de negócio principalmente quando o método não suporta um retorno compatível com um resultado anômalo. Ao criar um método dividir como podemos sinalizar (valor do retorno) para a aplicação quando executamos uma divisão por 0?

Observe que a função retorna um valor numérico e o fato de passar 0 como parâmetro acarretaria em um problema cujo retorno seria incompatível com o erro. Neste caso a aplicação poderia lançar uma **exceção de aplicação** levantando um problema que deve ser tratado. Outro tipo de exceção é a de **sistema.** Estas são geradas internamente **e capturadas pelo programa**.

Precisamos saber que erros **podem ser tratados** em cada método ou **passados adiante**. Métodos podem capturar ou deixar passar exceções que ocorrerem em seu corpo, mas para isto é obrigatório que o método declare a sua decisão. É o programador que implementa o código que define que um determinado método pode gerar algum tipo de exceção. Para repassar o tratamento de erro para quem chama o método utilizamos o **throws**. Uma declaração **throws** é obrigatória em métodos e construtores que deixam de capturar uma ou mais exceções que podem ocorrer. Ex:

```
public void validar() throws Excecao1, Excecao2 {…}
```

`throws` declara que o método pode provocar exceções do tipo declarado (ou de qualquer subtipo). Para tratar a exceção no método utilizamos o **`try/catch`**. Ex:

```
try{
     //Codigo verificado
}catch(TipoExcecao1 ex1){
     //Captura uma exceção TipoExcecao1
}catch(TipoExcecao2 ex2){
     //Captura uma exceção TipoExcecao2
}
```

24.1 Criando uma Exceção

A implementação de exceções seguem os conceitos de O.O.. Para se criar uma exceção basta herdar a classe **`Exception`**. Ex:

```
public class MinhaException extends Exception {…}
```

O método que lança a Exceção deve utilizar o **throws**:

```
public double dividir(double v1, double v2) throws MinhaException {...}
```

Para lançar a Exceção utilizamos o **throw** e criamos o objeto **Exception**. Ex:

```
if(v2==0){
     throw new MinhaException("Divisão por ZERO");
}
```

ESTUDO DE CASO IX

1) Utilizado os conhecimentos em tratamento de exceção crie uma classe Calculadora somente com o método dividir. Este método deve receber dois valores (parâmetros) do tipo double e retornar o resultado da divisão (double). Para que o método funcione corretamente você deve implementar os seguintes requisitos:

- Nenhum dos dois valores recebidos como parâmetros podem ser negativos. Caso algum dos valores sejam negativo o método deve levantar uma exceção personalizada;

- Faça o tratamento da operação através do try/catch e levante uma exceção personalizada caso ocorra algum problema (por exemplo divisão por 0);

- Crie outra classe com o método **public static void main**. No método principal instancie a classe Calculadora e chame o método dividir passando os seguintes parâmetros:

  a) 4, 0
  b) 2,-1
  c) 120,6
  d) 3,9

Informe quais foram o resultado de cada umas das divisões acima.

25. THREADS

Threads são linhas de execuções que realizam tarefas simultâneas (caso tenhamos mais de 1 processador) ou de forma a compartilhar o processamento. Cada thread é como um programa individual que tem total poder sobre a CPU. Existem duas formas para criar um thread: **Extendendo a classe Thread** ou **implementando a interface Runnable**.

Nos dois casos é necessário sobrescrever o método **run()** que é o "**main()**" do thread. O método run deve conter a execução que irá rodar pelo tempo de vida do thread. Quando o método terminar, o thread morre.

Para iniciar o thread é necessário chamar o método **start()**. Observe que o método implementado é o **run(),** mas para iniciar o thread chamamos o método **start()**. É a maquina virtual quem controla a execução e o ciclo de vida do thread.

Herdando a classe Thread:

```
public class ThreadA extends Thread {
     public void run(){
          for(int x=0;x<1000000;x++){
               System.out.println("A: "+x);
          }
     }
}
```

Implementando a interface Runnable:

```
public class ThreadB implements Runnable {
     public void run(){
          for(int x=0;x<1000000;x++){
               System.out.println("B: "+x);
          }
     }
}
```

Para iniciar o Thread utilizamos:

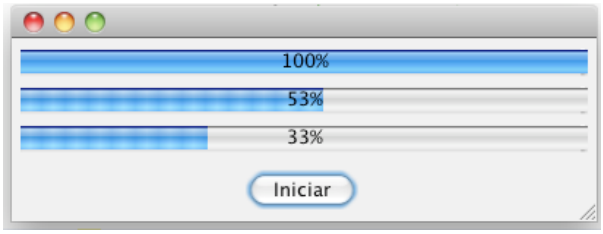
```
public class TesteThread{
     public static void main(String[] args){
          ThreadA ta = new ThreadA();
          Thread tb = new Thread(new ThreadB());
          ta.start();
          tb.start();
     }
}
```

Recursos compartilhados devem ser protegidos e para isto o Java utiliza a palavra-reservada **synchronized**. Esta permite que blocos sensíveis ao acesso simultâneo sejam protegidos de corrupção, impedindo que objetos os utilizem ao mesmo tempo. Se um recurso crítico está sendo usado, só um thread tem acesso. É preciso que os outros esperem até que o recurso esteja livre.

ESTUDO DE CASO X

1) Utilizado os conhecimentos sobre swing e thread crie um programa que vai disparar 3 thread's para preenchem 3 barras de progresso.

Cada thread é responsável por uma barra de progresso. Configure a propriedade **maximum** da barra de progresso para 10000000. A propriedade que define o aumento da barra de progresso é: **setValue**(valor do progresso na barra). O programa deve parecer como a figura abaixo:



Siga as instruções e responda os seguintes questionamentos:

a) Qual o motivo de termos que configurar a propriedade **maximum** da barra de progresso para um valor tão grande??

b) Dispare os 3 thread's para preencher as barras de progresso. Qual o resultado do experimento?

c) Defina a prioridade 10, 5 e 1 respectivamente para cada um dos thread's. Execute novamente o programa. Qual o resultado do experimento?

d) Faça com que todos os thread's utilizem um mesmo método para incrementar a barra de progresso e depois defina que este método é do tipo **synchronized**. Qual o resultado do experimento?

ESTUDO DE CASO XI

Por causa de um acidente de trabalho, Teobaldo (programador da Yoopa Informática) terá que afastar-se de suas atividades por 3 meses. Ele estava desenvolvendo um programa O.O. para importar um arquivo texto com dados dos clientes de uma Lan House. O sistema foi desenvolvido em JAVA utilizando interface gráfica Swing. A tela principal do sistema é apresentado na **Figura 1**.

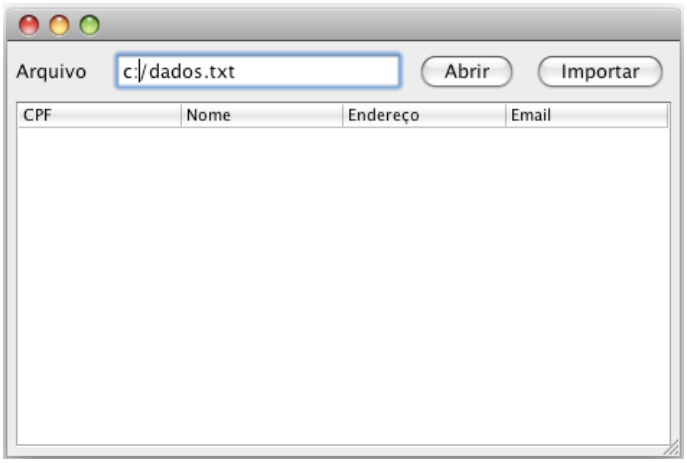


Figura 1 – Tela Principal do Sistema

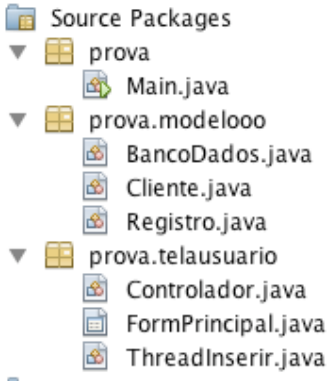


Figura 2 – Pacotes e Classes do Projeto

```
123,João,Av. Teste1,joao@gmail.com
456,Maria,Av. Teste2,maria@gmail.com
789,Pedro,Av. Teste3,pedro@gmail.com
...
```

Figura 3 – Conteúdo do Arquivo de dados

O sistema opera da seguinte forma: O usuário informa, na caixa de texto, o caminho do arquivo que contém os dados apresentados na **Figura 3**. Dando um click no botão **Abrir**, os dados são apresentados no grid. Dando um click no botão **Importar** os dados serão importados. As classes do projeto e seus respectivos pacotes são apresentados na **Figura 2**. Para importar os dados o sistema deve dividir o conteúdo do arquivo em 2 partes e delegar cada parte a um Thread diferente. Na folha em anexo contém a descrição das classes e o código deixado por Teobaldo. Você deve trabalhar na finalização do programa desenvolvendo os métodos não implementados.

Dicas:

- Procure entender o modelo O.O. para implementar os métodos;
- Os atributos da classe Cliente devem ser públicos;
- O atributo cpf deve ser inteiro;
- O método que obtem um valor de jTextField é: getText();
- O ThreadInserir processa um ArrayList contendo todos os dados que devem ser processados por ele. A divisão do ArrayList com o total de Clientes em 2 outros ArrayList's (Cada ArrayList terá metade) é feito no Controlador.

Classe	Descrição
FormPrincipal	Formulário principal da aplicação. Possui a classe Controlador por composição. Além disto possui os métodos: • **getDadosArquivo** – Retorna um ArrayList de Cliente com o conteúdo do arquivo que será importado • **preencherGrid** – Recebe um ArrayList e preenche o grid com o seu conteúdo • **abrir** – Abre um arquivo no computador e retorna o seu conteúdo
Controlador	Contém as operações que podem ser realizadas pelo modelo O.O. e serão chamadas pela classe FormPrincipal. Esta classe possui um método que divide um ArrayList (com os dados de Clientes que serão importados) em 2 outros ArrayLists que serão processadas pelos Threads.
BancoDados	Classe responsável por armazenar os dados da aplicação. A classe utiliza um contrato definido pela classe abstrata Registro para armazenar os objetos (Ex: objetos da classe Cliente). Implementa o Padrão de Projeto Singleton e é utilizada na classe ThreadInserir.
ThreadInserir	Thread que executa a inserção dos dados. Esta classe deve utilizar a classe BancoDados para realizar as operações.
Registro	Classe abstrata que define um contrato para armazenar objetos na classe BancoDados
Cliente	Classe que possui os atributos do cliente
Main	Classe principal do projeto que contém o método `public static void main`

Código-fonte deixado

```
package prova.telausuario;

import java.io.FileReader;
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;
import prova.modelooo.Cliente;
import javax.swing.table.DefaultTableModel;

public class FormPrincipal extends javax.swing.JFrame {

    Controlador ctr;

    public FormPrincipal() {
        ...
        ctr = new Controlador();
    }

    private void clickAbrir(java.awt.event.ActionEvent evt) {
        //Implemente Aqui
    }

    private void clickImportar(java.awt.event.ActionEvent evt) {
        //Implemente Aqui
    }

    public ArrayList getDadosArquivo() throws Exception{
        ArrayList col = new ArrayList();
        String info = abrir(jTextField1.getText());
        String dados[] = info.split("\n");
        for(int x=0;x<dados.length;x++){
            //Implemente Aqui
        }
        return col;
    }

    public void preencherGrid(ArrayList c){
        jTable1.setModel(new DefaultTableModel(new Object[][]{},new String[] {"CPF","Nome","E
        DefaultTableModel aModel = (DefaultTableModel) jTable1.getModel();
        for(int x=0;x<c.size();x++){
            String[] objects = new String[4];
            Cliente cli = (Cliente) c.get(x);
            objects[0] = String.valueOf(cli.cpf);
            objects[1] = cli.nome;
            objects[2] = cli.endereco;
            objects[3] = cli.email;
            aModel.addRow(objects);
        }
    }

    public static String abrir(String arquivo) throws Exception{
        FileReader in = new FileReader(arquivo);
        String texto="";
        int i;
        while((i=in.read())!=-1){
          texto = texto + (char)i;
        }
        return texto;
    }

    ...
    private javax.swing.JTextField jTextField1;
}
```

```
package prova.modelooo;

public abstract class Registro {
    public abstract int getIdentificador();
}
```

```
package prova.modelooo;

public class Cliente extends Registro {

    //Implemente Aqui

}
```

```
package prova.telausuario;

import java.util.Collection;
import java.util.Iterator;
import prova.modelooo.BancoDados;
import prova.modelooo.Cliente;

public class ThreadInserir extends Thread {
```

```
package prova.telausuario;

import java.util.ArrayList;
import prova.modelooo.Cliente;

public class Controlador {

    public void importar(ArrayList col){

        ArrayList cl[] = new ArrayList[2]; //Coleções que serão passadas para as Threads
        cl[0] = new ArrayList(); // Instacia a 1a Coleção
        cl[1] = new ArrayList(); // Instacia a 2a Coleção

        for(int x=0;x<col.size();x++){
            Cliente c = (Cliente) col.get(x); //Retira o cliente da coleção principal
            if(x%2==0){
                cl[0].add(c); //Adiciona na coleção par.
            }else{
                cl[1].add(c); //Adiciona na coleção impar.
            }
        }

        ThreadInserir t1 = new ThreadInserir(cl[0]); //Cria o primeiro Thread
        ThreadInserir t2 = new ThreadInserir(cl[1]); //Cria o segundo Thread
        //Inicia os threads
        t1.start();
        t2.start();
    }

}
```

```
package prova.modelooo;

import java.util.HashMap;

public class BancoDados {

    private static BancoDados instance;
    private HashMap h;

    private BancoDados(){
        h = new HashMap();
    }

    public synchronized static BancoDados getInstance(){
        if(instance==null){
            instance = new BancoDados();
        }
        return instance;
    }

    public void inserir(Registro r){
        h.put(r.getIdentificador(), r);
    }

}
```

```
package prova;

import prova.telausuario.FormPrincipal;

public class Main {

    public static void main(String[] args) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new FormPrincipal().setVisible(true);
            }
        });
    }

}
```

ESTUDO DE CASO XII

Um programador da Yoopa Informática precisa tirar férias e você foi selecionado para cobrir um de seus projetos. Ele estava desenvolvendo um programa O.O. para enviar E-mails. O sistema foi desenvolvido em JAVA utilizando interface gráfica Swing. A tela principal do sistema é apresentado na **Figura 1**.

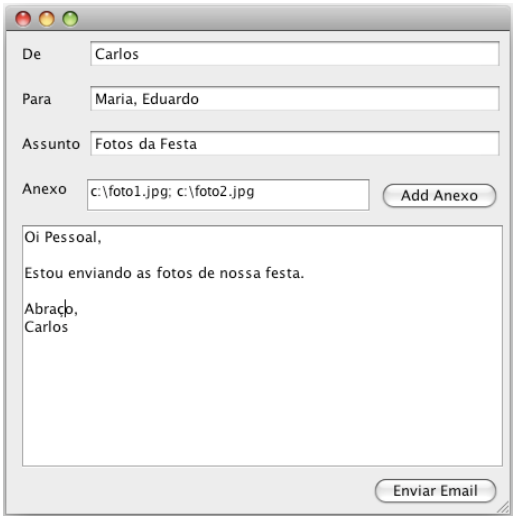


Figura 1 – Tela Principal do Sistema

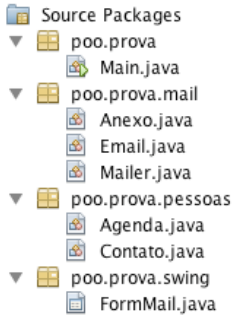


Figura 2 – Pacotes e Classes do Projeto

O sistema opera basicamente com o usuário preenchendo os campos: de, para, assunto, anexo e escrevendo a mensagem. O usuário pode digitar mais de um destinatário (para) separado por virgula. Porem o sistema só aceita o nome dos destinatários e para obter o email é preciso utilizar a classe Agenda. Dando um click no botão **Add Anexo** é possível adicionar vários anexos a mensagem. O botão **Enviar Email** deve utilizar os conceitos de O.O e as classes desenvolvidas para enviar o email aos destinatários. As classes do projeto e seus respectivos pacotes são apresentados na **Figura 2**. Na folha em anexo contém a descrição das classes e o código deixado pelo programador. Você deve trabalhar na finalização do programa desenvolvendo os métodos não implementados.

Dicas:

- Lembre dos conceitos de O.O. para entender o modelo O.O.;
- Procure entender o modelo O.O. para implementar os métodos;
- O método que obtem um valor de jTextField ou jTextArea é: getText();

Descrição das Classes do Projeto

Classe	Descrição
FormMail	Formulário principal da aplicação. Possui o método enviarClick que através da classe Mailer e Email envia emails. Além deste possui os seguintes métodos: • **getVetorContatos** – Recebe um String com uma lista de nomes, separados por virgula, e retorna um vetor com estes nomes • **getVetorAnexos** – Recebe um String com uma lista de anexos, separados por porto e virgula, e retorna um vetor com estes anexos
Email	Classe que define um Email. Possui as características de um email além dos métodos: Email (Construtor), addContato (Adiciona um destinatário) e addAnexo (Adiciona um anexo).
Mailer	Classe responsável por processar e enviar os emails aos seus destinatários. Implementa o Padrão de Projeto Singleton e é utilizada na classe FormMail.
Contato	Classe que define as características de um Contato. Um contato pode ser um remetente ou destinatário.
Agenda	Classe que possui um método estático (getEmail) onde é fornecido um nome e retorna o email referente ao nome fornecido.
Anexo	Classe que define as características de um anexo.
Main	Classe principal do projeto que contém o método public static void main

```
package poo.prova.swing;

import poo.prova.mail.*;
import poo.prova.pessoas.*;

public class FormMail extends javax.swing.JFrame {
    ...

    public String[] getVetorContatos(String nomes){
        return nomes.split(",");
    }

    public String[] getVetorAnexos(String arquivo){
        return arquivo.split(";");
    }

    /*
    ======================================================
    Para percorer um vetor de String:
    ======================================================
    String frutas = "laranja, mamão, limão";
    Syting lista[] = frutas.split(",");
    for(int x=0;x<lista.length;x++){
      System.out.println("A fruta atual é: " + lista[x]);
    }
    ======================================================
    */

    private void enviarClick(java.awt.event.ActionEvent evt) {
        //Implemente o Metodo Enviar
    }

    ...
    private javax.swing.JTextField jTextField1;
    private javax.swing.JTextField jTextField2;
    private javax.swing.JTextField jTextField3;
    private javax.swing.JTextArea jTextArea1;
}
```

```
package poo.prova.pessoas;

public class Contato {

    String nome;
    String email;

    public Contato(String nome, String email){
        this.nome = nome;
        this.email = email;
    }
}
```

```
package poo.prova.mail;

public class Anexo {

    public String arquivo;

    public Anexo(String arquivo){
        this.arquivo = arquivo;
    }
}
```

```
package poo.prova.pessoas;

public class Agenda {
    public static String getEmail(String nome){
        ...
    }
```

```
package poo.prova.mail;

import poo.prova.pessoas.Contato;
import java.util.ArrayList;


public class Email {

    public Contato de;
    public ArrayList<Contato> para;
    public ArrayList<Anexo> anexos;
    public String assunto;
    public String mensagem;

    //Implemente o Construtor

    //Implemente o Metodo addContato

    //Implemente o Metodo addAnexo

}
```

```
package poo.prova.mail;

public class Mailer {

    public static Mailer instance;

    private Mailer(){}

    public static Mailer getInstance(){
        if(instance==null){
            instance = new Mailer();
        }
        return instance;
    }

    public void enviar(Email e){
        ...
    }

}
```

```
package poo.prova;

import poo.prova.swing.FormMail;

public class Main {

    public static void main(String[] args) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new FormMail().setVisible(true);
            }
        });
    }

}
```

Desafio

Do ponto de vista da Arquitetura da Aplicação e do Modelo O.O., descreva o que poderia ser melhorado e explique o motivo.

QUESTIONÁRIO I

1) Qual a diferença entre classe e objeto?

2) O que ocorre quando instanciamos uma classe concreta em um variável de classe abstrata ou interface?

3) O que são pacotes e qual o seu objetivo?

4) Explique o conceito de sobrecarga de métodos.

5) Explique o conceito de Polimorfismo e como podemos implementa-lo?

6) Qual a diferença entre Agregação e Composição?

7) Como podemos fazer o tratamento de exceção em Java?

8) Explique como funciona uma Thread em Java.

9) Qual o objetivo do modificador final em um(a): variável, método e classe?

10) O que é um Siglteon e qual a sua finalidade?

QUESTIONÁRIO II

1) Qual a relação existente entre classe e objeto?

2) O que ocorre quando recebemos como parâmetro de uma função, uma interface?

3) Como podemos identificar, no código, as relações de Agregação e Composição?

4) Explique o conceito de sobrescrita de método.

5) Explique o funcionamento de Generics nas coleções

6) Qual o significado dos métodos e variáveis estáticas?

7) O que é necessário para criar uma nova Exceção e como fazemos para lançar esta Exceção.

8) Descreva como é criada e inciada uma Thread citando os métodos que devem