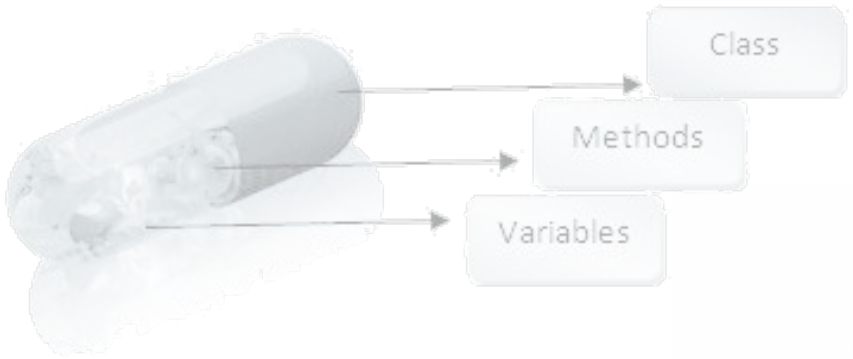


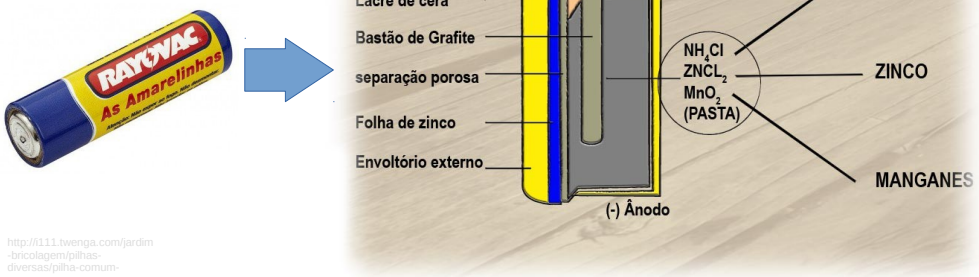
Aula 10

Encapsulamento, Polimorfismo e Classes Abstratas

Rafael Geraldeli Rossi

Encapsulamento

- Essa é uma pilha química



http://i111.twenga.com/jardim-bricolagem/pilhas-diversas/pilha-comum-rayovac-pequena-tp_1452843496957989503f.jpg

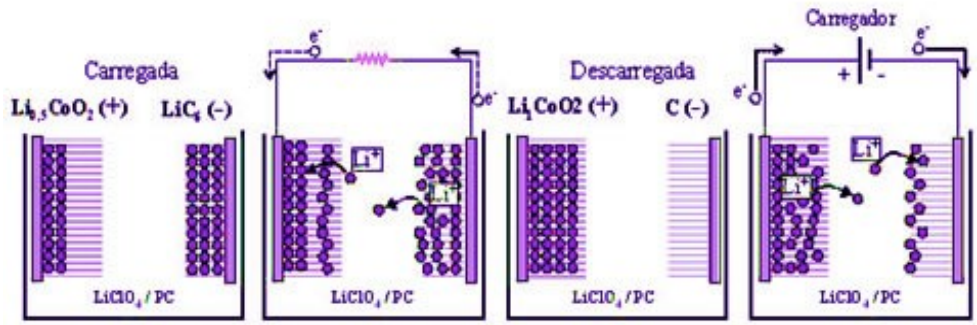
http://www.cec.com.br/images/hotsites/Dicas/construcao/reciclagem_pilhas/pilha4.jpg

- **Funcionamento**: a energia química é transformada em energia elétrica por meio de um sistema apropriado e montado para aproveitar o fluxo de elétrons provenientes de uma reação química de oxirredução.

Encapsulamento

- Essa é uma pilha recarregável

https://www.pedrocinefoto.com.br/media/catalog/product/cache/2/image/9df78eab33525d08d6e5fb8d27136e95/p/i/pilha_recarregav_l_sony_2100mha-1.jpeg



http://mundoeducacao.bol.uol.com.br/upload/conteudo/images/baterias-de-ion-litio.jpg

- **Funcionamento:**
  - Os íons lítio que estão em um solvente não aquoso migram do ânodo para o cátodo
  - Assim, no momento em que se recarrega essa bateria, o processo é inverso e é provocada a migração dos íons lítio da estrutura lamelar do óxido para a grafita

Encapsulamento

Será que seu controle remoto é programado pra funcionar com um tipo específico de pilha?

Se trocarmos um componente químico de uma pilha, o controle vai parar de funcionar?

Encapsulamento

- A resposta para as perguntas anteriores é **NÃO**
- A pilha está envolta em uma **cápsula** (ou seja, está **encapsulada**), e o controle não tem conhecimento sobre o seu funcionamento interno
- O controle só sabe que ao utilizar a pilha, a corrente de elétrons vai do pólo negativo (ânodo) para polo positivo (cátodo) e depois do positivo para o negativo

Encapsulamento

- O mesmo vale para a utilização de controles em geral


https://c2.staticflickr.com/6/5690/21605586346_be6c342b89_o.jpg


https://img1.ibxk.com.br/2011/12/materias/5438347096132138.jpg?w=700

- Não importa o seu funcionamento interno
- O que importa é que ao pressionar um botão, a ação correspondente àquele botão será realizada
- Podemos dizer também que o controle é uma cápsula que oculta seu funcionamento interno do usuário

Encapsulamento

- O **encapsulamento** faz com que seja disponibilizado apenas **o que a classe pode fazer, e não como é feito**
- O encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe
- É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada
- Até agora utilizamos basicamente o nível de acesso mais restritivo, o `private`, para encapsular dados

Encapsulamento

- A boa prática de programação exige que não se permitam acessos públicos aos atributos da classe (limita a flexibilidade em mudar o código), exceto em caso de constantes
- Para se ter acesso a algum atributo ou método que esteja encapsulado com `private` utiliza-se o conceito de métodos `gets` e `sets` ou métodos públicos que possam chamar os métodos privados
- Podemos também utilizar o modificador de acesso `protected`, que faz com que os membros de uma superclasse possam ser acessados pelas subclasses e outras classes do mesmo pacote $\rightarrow$ forma mais "frágil" de encapsulamento

Encapsulamento

- O encapsulamento pode ser feito em **dois níveis**
  - **Nível de classe**: determina o tipo de acesso à classe (geralmente `public` ou `private`)
  - **Nível de membro**: determina o tipo de acesso aos membros da classe (atributos ou métodos) $\rightarrow$ geralmente `public`, `private` ou `protected`

Exemplo do uso de uma classe privada

```
6   public class Aluno extends Pessoa{
7
8       private String rga;
9       private String curso;
10      private ArrayList<Disciplina> disciplinas;
11
12      public Aluno(String nome, String cpf, String rga, String curso){
13          super(nome,cpf);
14          this.rga = rga;
15          this.curso = curso;
16      }
17
18      public String getRga() {...3 linhas }
21
22      public void setRga(String rga) {...3 linhas }
25
26      public String getCurso() {...3 linhas }
29
30      public void setCurso(String curso) {...3 linhas }
33
34      @Override
        public String toString(){...5 linhas }
40
41      private class Disciplina{
            private String nome;
            private double nota;
44
45          public Disciplina(String nome, double nota){
46              this.nome = nome;
47              this.nota = nota;
48          }
49      }
50
51  }
```

Encapsulamento

- O encapsulamento permitirá então utilizar uma pilha, uma conta bancária, um controle, etc., apenas invocando os seus métodos, sem se preocupar com o seu funcionamento

- Durante a disciplina, estamos implementando constantemente as práticas de encapsulamento

Polimorfismo

- Significado de polimorfismo: **várias formas → um objeto de um tipo pode se comportar de várias formas**
- **Exemplos:**
  - **Jogo de xadrez**
    - Todas as peças de xadrez podem ter um método denominado `realizarMovimento(...)`
    - Porem, cada tipo de peça pode realizar um movimento particular
  - **Jogos de ação/RPG**
    - Todos os tipo de personagem podem realizar um ataque por meio de um método `atacar(...)`
    - Porém, o ataque depende do tipo do personagem e da sua arma

Polimorfismo

- **Polimorfismo** é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem **invocar métodos que têm a mesma assinatura mas comportamentos distintos** (métodos sobrescritos)
- O **polimorfismo** permite escrever programas que processam objetos que compartilham a mesma superclasse (direta ou indiretamente) como se todos fossem objetos da superclasse $\rightarrow$ simplificando a programação
- Portanto, o polimorfismo permite "programar no geral" em vez de "programar no específico"

Polimorfismo

- A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução
- Isso é denominado ligação tardia ou *late binding*
- **OBSERVAÇÃO 1:** a sobrecarga é um item fundamental para que isso funcione
- **OBSERVAÇÃO 2:** Podemos fazer uso de heranças e interfaces para fazer com que objetos diferentes sejam tratados da mesma forma na codificação mas ainda apresentem comportamentos distintos $\rightarrow$ chamado de **comportamento polimórfico**

Quadriláteros

- Se a classe Retângulo for derivada da classe Quadrilátero, então um objeto `Retângulo` é uma versão mais específica de um `Quadrilátero`
- Qualquer operação (por exemplo, calcular o perímetro ou a área) que pode ser realizada em um `Quadrilátero` também pode ser realizada em um `Retângulo`
- Essas operações também podem ser realizadas em outros `Quadriláteros`, como `Quadrados`, `Paralelogramos` e `Trapezoides`

Quadrilátero

```
11
12  public class Quadrilatero {
13
14      protected double[] lados;
15
16      Quadrilatero(){
17          lados = new double[4];
18      }
19
20      Quadrilatero(double[] lados){
21          this.lados = lados;
22      }
23
24      public double calculaPerimetro(){
25          double perimetro = 0;
26          for(int i=0;i<lados.length;i++){
27              perimetro += lados[i];
28          }
29          return perimetro;
30      }
31
32      public double calculaArea(){
33          return -1;
34      }
35
36  }
```

Quadrilátero

```
12  public class Quadrado extends Quadrilatero{
13
14
15      public Quadrado(double[] lados){
16          super(lados);
17      }
18
19      @Override
        public double calculaArea(){
21          double area = lados[0] * lados[1];
22          return area;
23      }
24
25  }
```

Quadrilátero

```
12  public class Retangulo extends Quadrilatero{
13
14      public Retangulo(double[] lados){
15          super(lados);
16      }
17
18      @Override
        public double calculaArea(){
20          double area = 0;
21          double lado1 = lados[0];
22          for(int i=1;i<lados.length;i++){
23              double lado2 = lados[i];
24              if(lado2 != lado1){
25                  area = lado1 * lado2;
26                  return area;
27              }
28          }
29          return area;
30      }
31
32  }
```

Quadrilátero

```
12  public class Trapezio extends Quadrilatero{
13
14      private double baseMaior;
15      private double baseMenor;
16      private double altura;
17
18      public Trapezio(double lados[], double baseMaior, double baseMenor, double altura){
19          super(lados);
20          this.baseMaior = baseMaior;
21          this.baseMenor = baseMenor;
22          this.altura = altura;
23      }
24
25      @Override
26      public double calculaArea(){
27          double area = ((baseMaior + baseMenor) * altura) / 2;
28          return area;
29      }
30
31  }
```

Quadrilátero

```
        ArrayList<Quadrilatero> quadrilateros = new ArrayList<Quadrilatero>();
19
20      double[] lados1 = {2,2,2,2};
21      Quadrado quadrado = new Quadrado(lados1);
22
23      double[] lados2 = {2,4,2,4};
24      Retangulo retangulo = new Retangulo(lados2);
25
26      double[] lados3 = {5,2,4,2};
27      Trapezio trapezio = new Trapezio(lados3, 5, 4, 1.8);
28
29      quadrilateros.add(quadrado);
30      quadrilateros.add(retangulo);
31      quadrilateros.add(trapezio);
32
33      for(int i=0;i<quadrilateros.size();i++){
34          System.out.println("Área do Quadrilatero " + (i+1) + " (" +
                quadrilateros.get(i).getClass().getName() + "): "
36              + quadrilateros.get(i).calculaArea());
37      }
```

Quadrilatero.Teste > main > for (int i = 0; i < quadrilateros.size(); i++) >

Saída - Teste (run) ×

```
run:
Área do Quadrilatero 1 (Quadrilatero.Quadrado): 4.0
Área do Quadrilatero 2 (Quadrilatero.Retangulo): 8.0
Área do Quadrilatero 3 (Quadrilatero.Trapezio): 8.1
```

Objetos espaciais em um videogame

- Suponha um jogo (algo do tipo invasão espacial) que precise manipular objetos das classes `Marciano`, `Venusiano`, `Plutoniano`, `NaveEspacial` e `CanhaoDeLaser`
- Considere que cada classe herda da superclasse `ObjetoEspacial`, que contém o método `desenhar()`, e que cada subclasse implementa esse método
- Um programa de gerenciamento de tela mantém um `Array` ou um `ArrayList` contendo objetos das várias classes acima
- Para atualizar, o gerenciador de tela envia periodicamente a mesma mensagem a cada objeto (`desenhar()`)

Objetos espaciais em um videogame

- Cada objeto responde de maneira única → a renderização do marciano é diferente do venusiano, que é diferente da nave espacial ...
- Um gerenciador de tela poderia utilizar o polimorfismo para facilitar a adição de novas classes a um sistema com modificações mínimas no código do sistema → novos objetos só teriam que estender a classe `ObjetoEspacial` e implementar a classe `desenhar()`

Objetos espaciais em um videogame

```
public class ObjetoEspacial {

    public void desenhar(){}

}
```

Objetos espaciais em um videogame

```
public class Marciano extends ObjetoEspacial{

    @Override
    public void desenhar(){
        System.out.println("   /]_/ ");
        System.out.println("  |\\/|.--/'-. ");
        System.out.println("  \\|/:o /  /\\    ._, ");
        System.out.println("    \\_/_.'0/    _|_ ");
        System.out.println("     \\____]] (>[___]=]]]=== ");
        System.out.println("     /    \\___/P{] ");
        System.out.println("   __//    /----\\/ ");
        System.out.println("  (_[-'\\__/_ ");
        System.out.println("     / | | \\ ");
        System.out.println("    '=='='==' ");
        System.out.println("    ____||||___ ");
        System.out.println("  (_\"\"_/ \\_\"\"_) ");
    }

}
```

Objetos espaciais em um videogame

```
public class NaveEspacial extends ObjetoEspacial{

    @Override
    public void desenhar(){
        System.out.println("                                    `. ___ ");
        System.out.println("                                __,' __`.                _..----....____");
        System.out.println("                    __...--.'``;.   ,.   ;``--..__     .'    ,-._    _.-'");
        System.out.println("  _..-''-------'   `'   `'   `'     O ``-''._   (,;') _,'");
        System.out.println(",'_________________                          \\`-._`-','");
        System.out.println(" `._              `````````````------...___   '-.._'-:");
        System.out.println("    ```--.._      ,.                     ````--...__\\-.");
        System.out.println("            `.--. `-`                       ____    |  |`");
        System.out.println("              `. `.                       ,'`````.  ;  ;`");
        System.out.println("                `._`.        _________    `.      \\\\ __/`");
        System.out.println("                   `-:._____/______/___/____`.     \\\\  `");
        System.out.println("                              |           `._    `.   \\\\");
        System.out.println("                              `._________`-.   `.    `.___");
        System.out.println("                                            `------'`");
    }

}
```

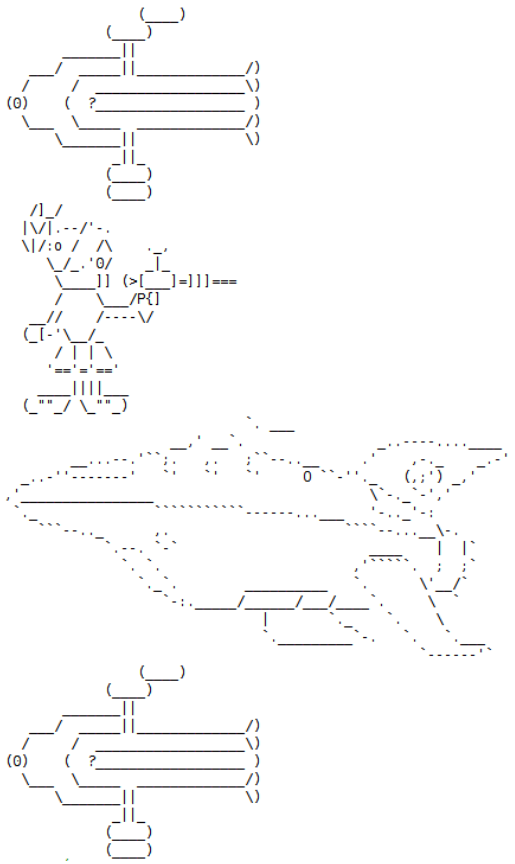
Objetos espaciais em um videogame

```
public class CanhaoLaser extends ObjetoEspacial{

    @Override
    public void desenhar(){
        System.out.println("                (____)");
        System.out.println("           (____)");
        System.out.println("      _____||");
        System.out.println("  ___/  _____||_____________/)");
        System.out.println(" /     /  ____________________\\)");
        System.out.println("(0)    (  ?____________________ )");
        System.out.println("  \\___  \\_____  _____________/)");
        System.out.println("      \\_______||              \\)");
        System.out.println("              _||_");
        System.out.println("           (____)");
        System.out.println("           (____)");
    }

}
```

Objetos espaciais em um videogame

```
public class Principal {

    public static void main(String[] args) {
        ArrayList<ObjetoEspacial> objetos = new ArrayList<ObjetoEspacial>();

        objetos.add(new CanhaoLaser());
        objetos.add(new Marciano());
        objetos.add(new NaveEspacial());
        objetos.add(new CanhaoLaser());

        desenhar(objetos);
    }

    public static void desenhar(ArrayList<ObjetoEspacial> objetos){
        for(ObjetoEspacial obj : objetos){
            obj.desenhar();
        }
    }

}
```

Objetos espaciais em um videogame



Observações quanto ao polimorfismo

- Para se fazer uso do polimorfismo, temos que tratar as subclasses como superclasses, isso é, atribuindo identificadores de objetos de subclasses à identificadores de objetos de superclasses
- Entretanto, fazendo isso, não é possível chamar um método específico de uma subclasse por meio do identificador de uma superclasse
- Caso seja necessário, em algum momento, invocar um método específico de uma subclasse, é necessário fazer um *casting* (nesse caso denominado *downcasting*) para a subclasse

Drivers em um SO

```
public class DriverDeVideo {

    public void renderiza(int x, int y, int r, int g, int b){ }

}
```

Drivers em um SO

```java
public class DriverDeVideoNvidia extends DriverDeVideo{

    //Atributos do Driver da NVidia

    @Override
    public void renderiza(int x, int y, int r, int g, int b){
        //Código de renderização da NVidia
    }

    //Outros métodos do Driver da NVidia

}
```

Drivers em um SO

```
public class DriverDeVideoRadeon extends DriverDeVideo{

    //Atributos do Driver da Radeon

    @Override
    public void renderiza(int x, int y, int r, int g, int b){
        //Código de renderização da Radeon
    }

    //Outros métodos do Driver da Radeon

}
```

Drivers em um SO

```java
public class SistemaOperacional {

    DriverDeVideo driverVideo;
    DriverDeSom driverSom;
    DriverDeRede driverRede;
    //Demais drivers...

    SistemaOperacional(){
        driverVideo = new DriverDeVideoNvidia();// SO com driver da NVidia
        //Inicialização dos demais drivers
    }


    public void renderizaTela(int x, int y, int r, int g, int b){
        driverVideo.renderiza(x, y, r, g, b);
    }

}
```

Drivers em um SO

```java
public class SistemaOperacional {

    DriverDeVideo driverVideo;
    DriverDeSom driverSom;
    DriverDeRede driverRede;
    //Demais drivers...

    SistemaOperacional(){
        driverVideo = new DriverDeVideoRadeon();// SO com driver da NVidia
        //Inicialização dos demais drivers
    }


    public void renderizaTela(int x, int y, int r, int g, int b){
        driverVideo.renderiza(x, y, r, g, b);
    }

}
```

Conceitos

- Quando pensamos em um tipo de classe, supomos que os programas criarão objetos desse tipo
- Porém, muitas vezes é inadmissível que o programa instancie objetos de um determinado tipo de classe
- Considerando os exemplos anteriores, classe `Quadrilatero`, `ObjetoEspacial` e `DriverDeVideo`, é inútil criar um objeto de ambas as classes uma vez que a função `calculaArea()` só retorna -1, a função `desenhar()` está vazia e a função `renderiza()` também está vazia

Conceitos

- Além disso, se o programador que criar uma classe que herda de `Quadrilatero`, `ObjetoEspacial` ou `DriverDeVideo` não sobrescrever os métodos `calculaArea()`, `desenhar()` ou `renderizar()`, os objetos das classes herdadas não funcionarão corretamente
- Porém, os métodos foram implementados dessa forma para que pudéssemos derivar objetos dessas classes e sobrescrever esses métodos de forma a habilitar o uso do polimorfismo

Há alguma forma de proibir a criação de um objeto de uma classe e obrigar a implementação de um determinado método herdado?

Classes e Métodos Abstratos

- **SIIIIIIIIIIIM** $\rightarrow$ com **CLASSES ABSTRATAS**
- O propósito de uma classe abstrata é **fornecer uma superclasse apropriada** a partir da qual outras classes possam herdar e assim **compartilhar um design comum**
- Essas classes **não podem ser usadas para instanciar objetos** porque essas classes são **classes incompletas**
- As subclasses devem **implementar as "partes ausentes"** para tornarem-se classes "concretas", e, a partir daí, **pode-se instanciar os objetos das subclasses**

Classes e Métodos Abstratos

- No exemplos de polimorfismos apresentados anteriormente, se não implementarmos um determinado método na subclasse, o compilador irá utilizar o método da superclasse (conceito básico da herança)
  - No caso de calcular a área dos quadriláteros, a resposta padrão da área de um quadrilátero que não implementa o método `calculaArea()` seria -1
  - No caso dos objetos espaciais, a resposta padrão seria um desenho em branco

Classes e Métodos Abstratos

- Nos exemplos anteriores, ambas as respostas são indesejadas, certo??
- Com o uso de **métodos abstratos**, podemos **forçar uma subclasse a implementar um método**
- **Uma classe que contém métodos abstratos deve ser declarada como uma classe abstrata** mesmo se essa classe contiver alguns métodos concretos (não abstratos)

Criando uma Classe Abstrata

- Cria-se uma classe abstrata utilizando a palavra-chave `abstract`

Declarando uma classe abstrata Quadrilatero

```
public abstract class Quadrilatero {
   ...
}
```

Criando um Método Abstrato

- Uma classe abstrata normalmente contém um ou mais métodos abstratos
- Um método abstrato não tem corpo (sem implementação)
- Para declarar um método abstrato deve-se utilizar a palavra-chave `abstract`

Declaração de um método `abstract desenhar()`

```
public abstract void desenhar();
```

Classe Abstrata Quadrilatero

```
13
14      protected double[] lados;
15
16      Quadrilatero(){
17          lados = new double[4];
18      }
19
20      Quadrilatero(double[] lados){
21          this.lados = lados;
22      }
23
24      public abstract double calculaArea();
25
26      public double calculaPerimetro(){
27          double perimetro = 0;
28          for(int i=0;i<lados.length;i++){
29              perimetro += lados[i];
30          }
31          return perimetro;
32      }
33
34  }
```

Classe Abstrata Quadrilatero

- Como o método `calculaArea` é abstrato, ao estender a classe `Quadrilatero` torna-se obrigatória a sua implementação

```
8  /**
9   *
10  * @author rafael
11  */
public class Quadrado extends Quadrilatero{
13
14 }
```

Quadrado is not abstract and does not override abstract method calculaArea() in Quadrilatero
---
(Alt-Enter mostra dicas)

Classe Abstrata Quadrilatero

- Como o método `calculaArea` é abstrato, ao estender a classe `Quadrilatero` torna-se obrigatória a sua implementação

```
12  public class Quadrado extends Quadrilatero{
13
14      public Quadrado(double[] lados){
15          super(lados);
16      }
17
18      @Override
19      public double calculaArea(){
20          double area = lados[0] * lados[1];
21          return area;
22      }
23
24  }
```

Observação

- **OBSERVAÇÃO:** os construtores e métodos `static` nao podem ser declarados `abstract`

Operador instanceof

- O operador `instanceof` pode ser utilizado para verificar o tipo de um objeto

```
        ArrayList<Quadrilatero> quadrilateros = new ArrayList<Quadrilatero>();
19
20      double[] lados1 = {2,2,2,2};
21      Quadrado quadrado = new Quadrado(lados1);
22
23      double[] lados2 = {2,4,2,4};
24      Retangulo retangulo = new Retangulo(lados2);
25
26      double[] lados3 = {5,2,4,2};
27      Trapezio trapezio = new Trapezio(lados3, 5, 4, 1.8);
28
29      quadrilateros.add(quadrado);
30      quadrilateros.add(retangulo);
31      quadrilateros.add(trapezio);
32
33      for(int i=0;i<quadrilateros.size();i++){
34          if(quadrilateros.get(i) instanceof Quadrado){
35              System.out.println("A forma é um Quadrado");
36          }else if(quadrilateros.get(i) instanceof Retangulo){
37              System.out.println("A forma é um Retangulo");
38          }else if(quadrilateros.get(i) instanceof Trapezio){
39              System.out.println("A forma é um Trapezio");
40          }else{
41              System.out.println("Nenhuma das formas anteriores");
42          }
43      }
44  }
```

Quadrilatero.Teste > main > for (int i = 0; i < quadrilateros.size(); i++) > if (quadrilateros.get(i) i

Saída - Teste (run)

```
run:
A forma é um Quadrado
A forma é um Retangulo
A forma é um Trapezio
```

Métodos e Classes `final`

- Vimos anteriormente nesta disciplina que as variáveis podem ser declaradas como `final` para indicar que elas não podem ser modificadas depois de serem inicializadas
- Também é possível declarar métodos, parâmetros de método e classes com o modificador `final`
- Um **método `final` em uma superclasse não pode ser sobrescrito em uma subclasse**
- Os métodos que são declarados `private` são implicitamente `final`, uma vez que não é possível sobrescrevê-los em uma subclasse

Métodos e Classes `final`

- Os métodos que são declarados `static` também são implicitamente `final`
- Uma declaração do método `final` nunca pode mudar → assim todas as subclasses utilizam a mesma implementação do método
- Portanto, chamadas a métodos `final` são resolvidas em tempo de compilação → **vinculação estática**

Exemplo de declaração de um método final para a classe desenhar()

```
public final void desenhar(){  ...  }
```

Métodos e Classes `final`

- **Uma classe que é declarada como `final` não pode ser uma superclasse, ou seja, uma classe não pode estender uma classe `final`**
- Todos os métodos em uma classe `final` são implicitamente `final`
- **OBSERVAÇÃO 1:** a classe `String` é uma classe `final`
- **OBSERVAÇÃO 2:** tornar a classe `final` também impede que programadores criem subclasses que poderiam driblar as restrições de segurança

Caixa de Diálogo com Opções Personalizadas

- Para exibir uma "caixa de confirmação" personalizada, ou seja, sem as opções predefinidas no método `showConfirmDialog` como SIM ou NÃO, OK e CANCELAR, etc., pode-se utilizar o método estático `showOptionDialog` da classe `JOptionPane`

```
public static int showOptionDialog(Component parentComponent,
                    Object message,
                    String title,
                    int optionType,
                    int messageType,
                    Icon icon,
                    Object[] options,
                    Object initialValue)
```

Caixa de Diálogo com Opções Personalizadas

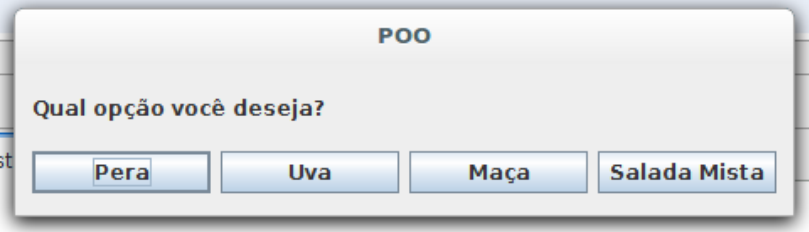
```
14  public class CaixaDialogo {
15
16      public static void main(String[] args) {
17
18          String[] options = { "Pera", "Uva", "Maça", "Salada Mista" };
19          int op = JOptionPane.showOptionDialog(null, "Qual opção você deseja?", "POO",
20          JOptionPane.DEFAULT_OPTION, JOptionPane.DEFAULT_OPTION,
21          null, options, options[0]);
22
23          System.out.println("Opção escolhida foi: "+ options[op]);
24
25      }
```

CaixaDialogo > main >

Saída ×

Console do Depurador × | Teste (run) × | Test

```
run:
Opção escolhida foi: Pera
```



Exercício

- O exercício consiste em alterar as classes do Projeto Banco da seguinte forma:
  - Iremos implementar a classe `ContaBancaria`
    - Deve-se ter os mesmos atributos da classe `Conta`
    - Deve-se implementar o método não abstrato `depositar(double valor)`: responsável por acrescer um valor no saldo de conta
    - Deve-se definir o método abstratos `sacar(double valor)`: será responsável por subtrair de uma determinada forma um valor do saldo da conta
    - Deve-se implementar o método abstrato `atualizar(double taxa)`: será responsável por incrementar o saldo de uma conta de acordo com uma determinada taxa de juros e de acordo com as características da conta
    - Os demais métodos devem ser os mesmos da classe `Conta`

Exercício

- Deve-se derivar duas classes da classe `ContaBancaria`: `ContaCorrente` e `ContaPoupanca`

- Classe `ContaPoupanca`
  - No método `sacar(...)`, só realizar a operação de saque se houver saldo, i.e, $saldo - valor \geq 0$

  - No método `atualizar(...)`, atualizar o saldo com duas vezes a taxa de juros

Exercício

- Classe `ContaCorrente`
  - Deverão ser acrescidos os campos `limite(double)` e `mensalidade (double)`
  - No método sacar(...), pode-se realizar um saque mesmo que não haja saldo mas desde que não extrapole o limite estabelecido pela conta
  - No método atualizar(...). deve-se atualizar o saldo da conta de acordo com a taxa de juros e subtrair o valor da mensalidade
  - Implementar um método para realizar a transferência $\rightarrow$, portanto, só contas bancárias correntes podem fazer transferências

Exercício

- Classe `ContaPoupanca` e `ContaCorrente`
  - Incluir a operação de atualização no extrato da conta
  - Incluir a informação do tipo de conta quando exibir o extrato;
- Classe `Banco`
  - Acrescentar a opção `Atualizar Contas` na `Área do Gerente`: deve-se percorrer todas as contas e atualizá-las
  - Alterar o cadastro de novas contas de acordo com os novos campos das classes
- Apagar a classe `Conta`

Material Complementar

- Polimorfismo

  https://www.caelum.com.br/apostila-java-orientacao-objetos/heranca-reescrita-e-polimorfismo/#7-4-polimorfismo

- Universidade XTI - JAVA - 050 - Polimorfismo, Classes abstract

  https://www.youtube.com/watch?v=egpR68ILGjs

- Classes Abstratas

  https://www.caelum.com.br/apostila-java-orientacao-objetos/classes-abstratas/#9-4-aumentando-o-exemplo

Material Complementar

- Curso POO Teoria #06a - Pilares da POO: Encapsulamento

  `https://www.youtube.com/watch?v=1wYRGFXpVlg&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY&index=11`

- Curso POO Java #06b - Encapsulamento

  `https://www.youtube.com/watch?v=x4JfzV0Wb5w&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY&index=12`

- Curso POO Teoria #12a - Conceito Polimorfismo (Parte 1)

  `https://www.youtube.com/watch?v=9-3-RMEMcq4&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY&index=23`

Material Complementar

- Curso POO Java #12b - Polimorfismo em Java (Parte 1)

  https://www.youtube.com/watch?v=NctjqlfKC0U&index=24&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY

- Curso POO Teoria #13a - Conceito Polimorfismo (Parte 2)

  https://www.youtube.com/watch?v=hYek1xqWzgs&index=25&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY

- Curso POO Java #13b - Polimorfismo Sobrecarga (Parte 2)

  https://www.youtube.com/watch?v=b7xGYh3NHZU&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY&index=26

Material Complementar

- showOptionDialog

  `https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html#showOptionDialog(java.awt.Component,%20java.lang.Object,%20java.lang.String,%20int,%20int,%20javax.swing.Icon,%20java.lang.Object[],%20java.lang.Object)`

- Aprendendo sobre Classes Internas

  `https://www.devmedia.com.br/aprendendo-sobre-classes-internas/15581`

Imagem do dia

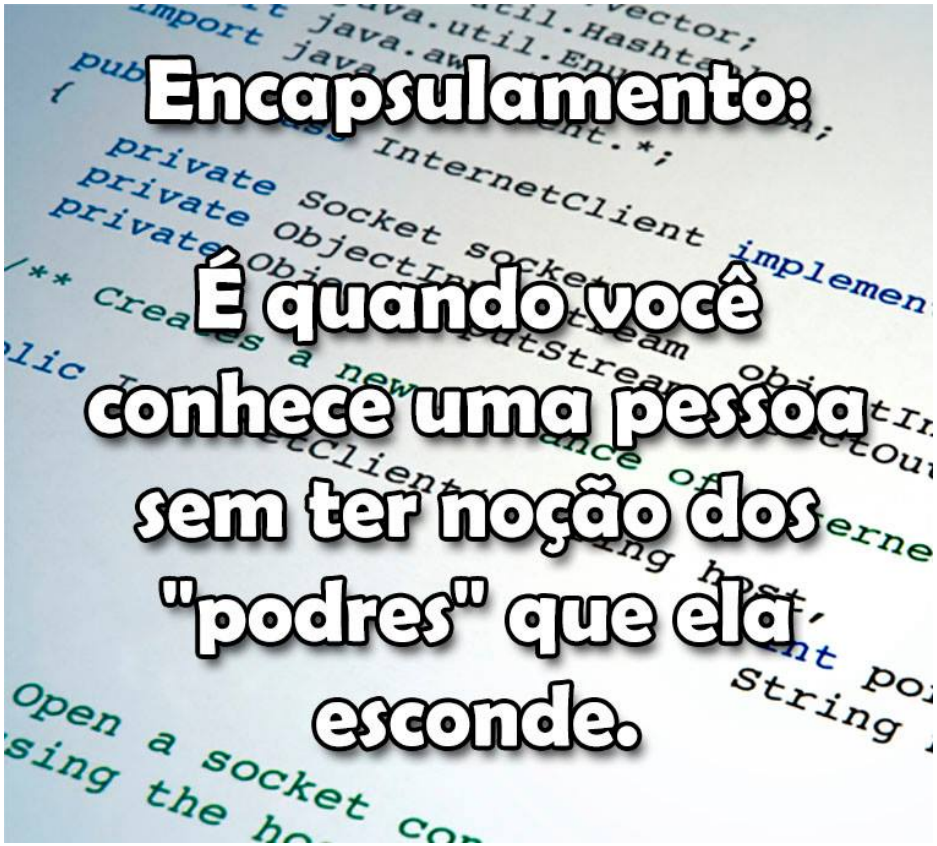


Imagem do dia

Programação Orientada a Objetos
http://lives.ufms.br/moodle/

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br

Slides baseados em [Deitel and Deitel, 2010]

Referências Bibliográficas I

Deitel, P. and Deitel, H. (2010).
Java: How to Program.
How to program series. Pearson Prentice Hall, 8th edition.