



Classes e métodos abstratos

Classes e métodos abstratos

- ▶ Servem apenas como **modelos** para classes concretas;
- ▶ **Não podem ser instanciadas** diretamente com o *new*, devem ser herdadas por classes concretas.
- ▶ **Podem conter ou não métodos abstratos**, ou seja, pode implementar ou não um método.
 - ▶ **Um método abstrato não tem corpo.**
 - ▶ Os métodos abstratos definidos em uma classe abstrata **devem** obrigatoriamente ser implementados em uma classe concreta.
 - ▶ Se uma classe abstrata herdar outra classe abstrata, a classe que herda **não precisa** implementar os métodos abstratos.

Exemplo

```
public abstract class Eletrodomestico {
    private boolean ligado;
    private int voltagem;
    // métodos abstratos não possuem corpo */
    public abstract void ligar();
    public abstract void desligar();
    // método construtor - Classes Abstratas também podem ter métodos
    // construtores, mas não podem ser usados para instanciar um objeto
    public Eletrodomestico(boolean ligado, int voltagem) {
        this.ligado = ligado;
        this.voltagem = voltagem;
    }
    // métodos concretos - Uma classe abstrata pode possuir métodos concretos
    public void setVoltagem(int voltagem) {
        this.voltagem = voltagem; }
    public int getVoltagem() {
        return this.voltagem; }
    public void setLigado(boolean ligado) {
        this.ligado = ligado; }
    public boolean isLigado() {
        return ligado; }
}
```

Exemplo

```
public class TV extends Eletrodomestico {
    private int tamanho;
    private int canal;
    private int volume;
    public TV(int tamanho, int voltagem) {
        super (false, voltagem); // construtor classe abstrata
        this.tamanho = tamanho;
        this.canal = 0;
        this.volume = 0;
    }
    /* implementação dos métodos abstratos */
    public void desligar() {
        super.setLigado(false);
        setCanal(0);
        setVolume(0);
    }
    public void ligar() {
        super.setLigado(true);
        setCanal(3);
        setVolume(25);
    }
    // abaixo teríamos todos os métodos construtores get e set...
}
```

Exemplo

```
public class Radio extends Eletrodomestico {
    public static final short AM = 1;
    public static final short FM = 2;
    private int banda;
    private float sintonia;
    private int volume;
    public Radio(int voltagem) {
        super(false, voltagem);
        setBanda(Radio.FM);
        setSintonia(0);
        setVolume(0);
    }
    /* implementação dos métodos abstratos */
    public void desligar() {
        super.setLigado(false);
        setSintonia(0);
        setVolume(0);
    }
    public void ligar() {
        super.setLigado(true);
        setSintonia(88.1f);
        setVolume(25);
    }
    // abaixo teríamos todos os métodos construtores get e set... }
```

Exemplo

```
public class Main {  
    public static void main(String[] args) {  
        TV tv1 = new TV(29, 110);  
        Radio radiol = new Radio(110);  
        /**  
        chamando os métodos abstratos implementados  
        * dentro de cada classe (TV e Radio)  
        */  
        tv1.ligar();  
        radiol.ligar();  
        System.out.print("Neste momento a TV está ");  
        System.out.println(tv1.isLigado() ? "ligada" : "desligada");  
        System.out.print("e o Rádio está ");  
        System.out.println(radiol.isLigado() ? "ligado." : "desligado.");  
    }  
}
```

Concluindo...

- ▶ As classes abstratas servem de **base para codificação de uma classe inteira**, diferentemente das interfaces que são apenas assinaturas dos métodos.
- ▶ Sumarizando, quando temos que definir variáveis, constantes, regras, e pequenas ações definidas devemos usar classes abstratas. Mas, se formos apenas criar a forma como objetos devem realizar determinadas ações (**métodos**) devemos optar por **interfaces**.



Interfaces

Por que padronizar?

▶ Padronização

- ▶ No dia-a-dia lidamos com diversos aparelhos elétricos e as empresas fabricam aparelhos elétricos com plugues;
- ▶ E se cada empresa decidisse por conta própria o formato dos plugues ou tomadas que fabricará?
- ▶ Essa **falta de padrão** pode gerar **problemas de segurança**, aumentando o risco de uma pessoa levar um choque...
- ▶ O governo estabelece padrões para plugues e tomadas, facilitando a utilização para os consumidores e aumentando a segurança.
- ▶ **Padronizar pode trazer grandes benefícios, inclusive no desenvolvimento de aplicações.**

Por que padronizar?

► Padrões de plugues



Contratos

- ▶ Podemos dizer que os objetos se “encaixam” através dos **métodos públicos** assim como um plugue se encaixa em uma tomada através dos pinos.
 - ▶ Para os objetos de uma aplicação “conversarem” entre si mais facilmente é importante padronizar o conjunto de métodos oferecidos por eles.
- ▶ Um padrão é definido através de especificações ou contratos.
 - ▶ Em orientação a objetos, um **contrato é chamado de interface**
 - ▶ Um **interface** é composta basicamente por **métodos abstratos**

Interfaces

- ▶ Os **métodos** de uma interface **não possuem corpo** (implementação) pois serão implementados nas classes vinculadas a essa interface.
 - ▶ São abstratos
- ▶ Todos os métodos de uma interface devem ser públicos e abstratos
 - ▶ Os modificadores *public* e *abstract* são opcionais
- ▶ As classes concretas que implementam uma interface são **obrigadas** a possuir uma implementação para cada método declarado na interface

Interfaces

► Vantagens:

- **Padronizar** as assinaturas dos métodos oferecidos por um determinado conjunto de classes
- **Garantir** que determinadas classes implementem certos métodos

► Exemplo:

```
1 interface Conta {  
2     void deposita(double valor);  
3     void saca(double valor);  
4 }
```

```
1 class ContaPoupanca implements Conta {  
2     public void deposita(double valor) {  
3         // implementacao  
4     }  
5     public void saca(double valor) {  
6         // implementacao  
7     }  
8 }
```

```
1 class ContaCorrente implements Conta {  
2     public void deposita(double valor) {  
3         // implementacao  
4     }  
5     public void saca(double valor) {  
6         // implementacao  
7     }  
8 }
```

Modificadores *static* e *final*

Modificador *static*

- ▶ Modifica o escopo de um método ou atributo, pois estes passam a **pertencer à classe** e não à instância do objeto.
- ▶ É usado para a criação de uma variável que poderá ser acessada por todas as instâncias de objetos desta classe como uma variável comum, ou seja, a variável criada será a mesma em todas as instâncias e quando seu conteúdo é modificado numa das instâncias, a modificação ocorre em todas as demais.

Métodos *static*

- ▶ Às vezes, um método realiza uma tarefa que não depende de um objeto. Esse método se aplica à classe em que é declarado como um todo e é conhecido como método *static* ou método de classe.
- ▶ Os métodos *static* podem ser chamados sem uma instância pois ajudam no acesso direto à classe.
- ▶ Desta forma, não é necessário instanciar um objeto para acessar o método.

`NomeDaClasse.nomeDoMétodo(argumentos)`

Atributos *static*

- ▶ Os atributos *static* possuem o mesmo valor para todas as instâncias de um objeto;
- ▶ Exemplo:

```
public class MinhaClasse {  
    static int valorGlobal = 1;  
    public static int getValorGlobal() {  
        return valorGlobal;  
    }  
}  
  
// classe principal  
MinhaClasse c1 = new MinhaClasse();  
MinhaClasse c2 = new MinhaClasse();  
MinhaClasse.valorGlobal = 2;  
System.out.println(c1.getValorGlobal()); //imprime 2  
System.out.println(c2.getValorGlobal()); //imprime 2
```

Classes *final*

- ▶ Uma classe com este modificador **não pode ser estendida**, isto é, não pode ter classes que herdam dela.
- ▶ Usa-se para garantir que uma determinada implementação não tenha seu comportamento modificado (imutabilidade).
- ▶ Exemplo:

```
public final class minhaClasse  
{ ... }
```

Métodos *final*

- ▶ É um método que **não pode ser sobrescrito** nas subclasses.
- ▶ Usa-se para garantir que um determinado algoritmo não possa ser modificado pelas subclasses.
- ▶ Exemplo:

```
public class Xadrez{  
    int jogador;  
    final int getJogador() {  
        return jogador;  
    }  
    ...  
}
```

Atributos *final*

- ▶ É um atributo que só **pode ter seu valor atribuído uma única vez, seja na própria declaração ou no construtor.**
- ▶ Usa-se para garantir que um atributo de objeto não vai mudar.
- ▶ Exemplo:

```
public class MinhaClasse {  
    final int x = 1;  
    final int y;  
    public MinhaClasse(int y) {  
        this.y = y;  
    }  
}
```

Variáveis *final*

- ▶ É uma variável que só **pode ter seu valor atribuído uma única vez.**
- ▶ Use para garantir que você não está modificando o valor indevidamente.
- ▶ Exemplo:

```
final boolean a = lerInputUsuario();  
final boolean b = lerInputUsuario();
```

Resumindo o *final*...

- ▶ Quando é aplicado na classe, não permite estendê-la;
- ▶ Quando é aplicado nos métodos, impede que o mesmo seja sobrescrito (*overriding*) na subclasse;
- ▶ Quando é aplicado nos atributos ou valores de variáveis, não pode ser alterado depois que já tenha sido atribuído um valor.

static e *final* juntos

- ▶ Utilizados juntos para constantes, pois indicam que o mesmo valor vai ser visto para todas as instâncias da classe (*static*) e nunca vai poder ser modificado depois de inicializado (*final*).