

Disciplina: PROGRAMAÇÃO I

Aula 8: Classes abstratas e interfaces

Apresentação

Continuaremos explorando os conceitos da orientação de um objeto através das classes abstratas e interfaces, mas antes demonstraremos a aplicação dos modificadores static e final.

As classes abstratas são usadas como moldes para a criação de outras classes e podem encapsular atributos e comportamentos comuns. Já interface é um recurso muito utilizado em Java.

Uma classe pode implementar várias interfaces. Ao final, você desenvolverá novas aplicações utilizando os conceitos de classes abstratas e interfaces.

Objetivo

- Examinar o conceito de classe abstrata;
- Aplicar o conceito de classe abstrata na construção de programas em Java;
- Identificar e aplicar o conceito de interface.

Modificadores: static e final

Anteriormente, conhecemos os modificadores de acesso ou de visibilidade (Encapsulamento). Agora iremos conhecer mais alguns modificadores que podem ser aplicados sobre classes, no caso do final e sobre membros (Atributos e Métodos) de uma classe. Esses modificadores, como o nome já diz, servem para alterar a forma de uso de classes, métodos e/ou atributos.



Fonte: shutterstock

A referência this

O uso do this em Java é para ajudar na questão das referências (endereçamento) de memória. O this é um ponteiro (variável que armazena endereço de memória) de forma implícita. Java não possui endereçamento direto de memória (endereçamento explícito), apenas o endereçamento indireto (implícito) de memória. A referência this então é uma referência implícita ao endereçamento de memória de um objeto.

A referência this altera a identificação do objeto pelo seu identificador (nome) e pelo endereço de memória do objeto, e só pode referenciar membros da classe, ou seja, somente faz referência a atributos e métodos.

Dessa forma, podemos substituir o nome do objeto pela sua referência, e também separar quando tem o mesmo nome variáveis e atributos:

Classe RefThis (pacote: biblioteca):

```
package biblioteca;
public class RefThis {
    private String nome;
    private int idade;
    private double peso;
    public String getNome() {
        return this.nome;
    }
    // a referência this indica que this.nome
    // faz referência ao atributo nome da classe e
    // sem o this nome identifica o parâmetro nome do método
    public void setNome(String nome) {
        this.nome = nome;
    }
    public int getIdade() {
        return this.idade;
    }
    // a referência this indica que this.idade
    // faz referência ao atributo idade da classe e
    // sem o this idade identifica o parâmetro idade do método
    public void setIdade(int idade) {
        this.idade = idade;
    }
    public double getPeso() {
        return this.peso;
    }
    // a referência this indica que this.peso
    // faz referência ao atributo peso da classe e
    // sem o this peso identifica o parâmetro peso do método
    public void setPeso(double peso) {
        this.peso = peso;
    }
}
```

Nota: O this separa os membros da classe das demais variáveis auxiliares e parâmetros da classe.

O modificador: static

Este modificador pode ser aplicado sobre atributos e métodos e transforma o atributo ou método para a forma “compartilhada”.

O modificador static aplicado a um atributo

Este modificador ao ser aplicado em um atributo de uma classe modifica este atributo de objeto (ou de instância) para um atributo de classe¹.

 Exemplo 1

 Clique no botão acima.

Exemplo 1 - Aplicação com o compartilhamento de um valor: **private static double**

Classe Cotacao (pacote: biblioteca):

```
package biblioteca;
import java.util.Scanner;
public class Cotacao {
    private static double valorDolar;
    public double getValorDolar() {
        return valorDolar;
    }
    public void setValorDolar(double vd) {
        valorDolar = verificarValorDolar(vd);
    }
    private double verificarValorDolar(double vd) {
        if (vd > 0) {
            return vd;
        } else {
            return 0.0;
        }
    }
    public Cotacao() { }
    public Cotacao(double valorDolar) {
        setValorDolar(valorDolar);
    }
    public void cadastrar(double valorDolar) {
        setValorDolar(valorDolar);
    }
    public void imprimir() {
        System.out.println("Valor do dólar :" + getValorDolar());
    }
    public void entrada() {
        Scanner entrada = new Scanner(System.in);
        System.out.println("Qual é o Valor do dólar ?");
        setValorDolar(Double.parseDouble(entrada.nextLine()));
    }
    public double converterRealParaDolar(double real) {
        return real / getValorDolar();
    }
}
```

Classe Exemplo1 (pacote: aplicacao):

```
package aplicacao;
import java.util.Scanner;
import biblioteca.Cotacao;
public class Exemplo1 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        double valorReal;
        Scanner entrada = new Scanner(System.in);
        Cotacao cot_1 = new Cotacao();
        cot_1.entrada();
        System.out.println("Quantos reais para a conversão ?");
        valorReal = Double.parseDouble(entrada.nextLine());
        System.out.println();
        // Conversão com o objeto cot_1 de R$ 1000,00 com dólar R$ 3.82
        cot_1.imprimir();
        System.out.printf("Coversão de Real para dólar : US$ %.2f",
            cot_1.converterRealParaDolar(valorReal));
        System.out.println();
        Cotacao cot_2 = new Cotacao(4.15);
        // Conversão utilizando os objetos cot_1 e cot_2 com o mesmo valor de R$ 1000,00
        cot_1.imprimir();
        System.out.printf("Coversão de Real para dólar : US$ %.2f",
            cot_1.converterRealParaDolar(valorReal));
        System.out.println();
        cot_2.imprimir();
        System.out.printf("Coversão de Real para dólar : US$ %.2f",
            cot_2.converterRealParaDolar(valorReal));
        System.out.println();
    }
}
```

Execução:

```
Qual é o Valor do dólar ?
3.82
Quantos reais para a conversão ?
1000
Valor do dólar :3.82
Coversão de Real para dólar : US$ 261,78
Valor do dólar :4.15
Coversão de Real para dólar : US$ 240,96
Valor do dólar :4.15
Coversão de Real para dólar : US$ 240,96
```

Notas:

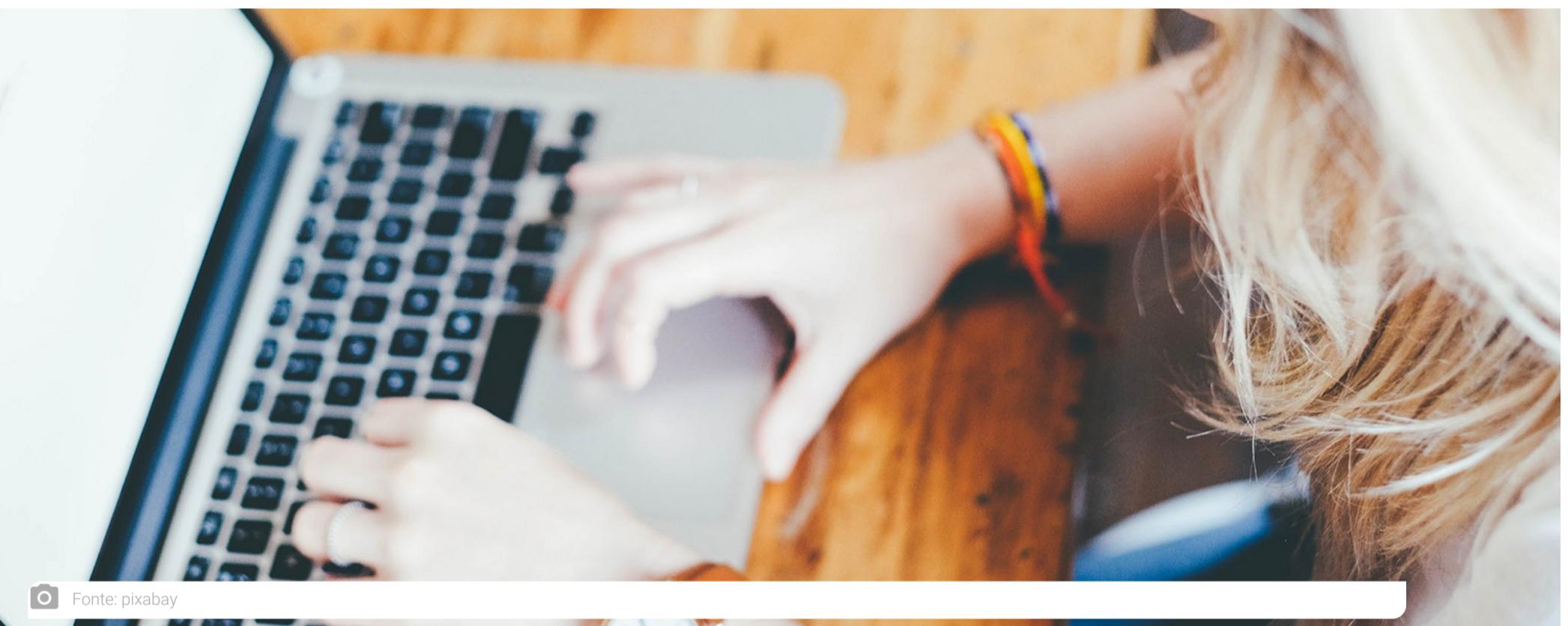
- 1** Primeiramente foi utilizado valor de R\$ 3,82 para a cotação do dólar, com a leitura através do teclado para o objeto cot_1, e foi calculada a conversão do valor de R\$ 1000,00, resultando em: US\$ 261,78;

- 2** Depois, foi criado o objeto cot_2, com o valor da cotação do dólar definido através do método construtor em R\$ 4,15, e foi mantido o mesmo valor de R\$ 1000,00 para a conversão;

- 3** Ambos os objetos deram o mesmo resultado porque o valor do dólar para a conversão era o mesmo, de R\$ 4,15. Isso ocorreu pois, ao alterar o valor da cotação do dólar através do objeto cot_2, o objeto cot_1 também foi afetado, já que o atributo valorDolar é compartilhado: **private static double** ;

Como o atributo valorDolar é compartilhado, o atributo deixa de ser um atributo de objeto (propriedade) com valor próprio a cada objeto, e passa a ser um atributo de classe, ou seja, passa a ser compartilhado por todas as instâncias. Internamente é criado um ponteiro implícito em que todos os atributos compartilhados da classe apontam para o mesmo endereço de memória. Por isso, qualquer objeto que realize uma alteração em um atributo compartilhado afetará todos os demais objetos criados a partir da mesma classe na aplicação.

A referência this não pode ser usada com atributos estáticos (static), apenas atributos não compartilhados, porque a referência dos atributos compartilhados não pertence ao objeto, mas sim à classe.



Fonte: pixabay

Exemplo 2

Clique no botão acima.

Exemplo 2 - Aplicação com um contador de objetos criados: private static int contador

Classe Teste (pacote: biblioteca):

```
package biblioteca;
public class Teste {
    private static int contador;
    public Teste() {
        contador++;
    }
    public int getContador() {
        return contador;
    }
}
```

Classe Exemplo2 (pacote: aplicacao):

```
package aplicacao;
package aplicacao;
import biblioteca.Teste;
public class Exemplo2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Teste t1 = new Teste();
        System.out.println("Contador =" + t1.getContador());
        Teste t2 = new Teste();
        System.out
.println("Contador =" + t2.getContador());

        Teste t3 = new Teste();
        System.out
.println("Contador =" + t3.getContador());

        Teste t4 = new Teste();
        System.out
.println("Contador =" + t4.getContador());

        Teste t5 = new Teste();
        System.out
.println("Contador =" + t5.getContador());

        // Contador de t1 após a criação dos 5 objetos:
        System.out
.println("Contador =" + t1.getContador());
    }
}
```

Execução:

```
Contador =1
Contador =2
Contador =3
Contador =4
Contador =5
Contador =5
```

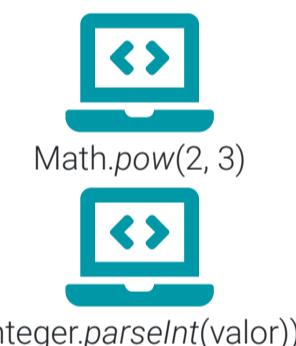
Notas:

- 1 Foram criados cinco objetos, mas em nenhum caso foi feito um acesso direto para o atributo contador; cada objeto, ao ser criado, incrementava o contador;
- 2 Conforme foram sendo criados os objetos, o valor compartilhado do atributo contador ia sendo atualizado para todos os objetos;
- 3 Após a criação do 5º objeto, o objeto t5, o atributo do objeto t1, assim como os demais, compartilhavam o mesmo local de memória para buscar o valor do atributo; por isso, todos os objetos encerraram a aplicação retornando o valor 5 para o atributo contador compartilhado (static).

O modificador static aplicado a um método

Este modificador, ao ser aplicado a um método de uma classe, modifica esse método de forma a torná-lo estático (compartilhado). Um método compartilhado pode ser usado sem que seja necessário criar um objeto para usá-lo.

Você irá perceber que já utilizou muitos métodos compartilhados, tais como:



Perceba que o método `main` deve ser modificado para compartilhado com o modificador `static`; caso contrário, o método não poderia ser executado.

Um método estático ou compartilhado só pode acessar outros membros estáticos (atributos e/ou métodos). Um método estático não pode acessar atributos de objeto (não estáticos) nem outros métodos que não sejam estáticos.

Exemplo 3

 Clique no botão acima.

Exemplo 3 – Uso de métodos estáticos: `public static double valorQuadrado(){ }`

Classe MetodosCompartilhados (pacote: biblioteca):

```
package biblioteca;
public class MetodoCompartilhado {
    private static int valor = 5;
    public static double valorQuadrado() {
        return Math.pow(valor, 2);
    }
    public static int valorDobro() {
        return valor * 2;
    }
}
```

Classe Exemplo3 (pacote: aplicacao):

```
package aplicacao;
import
biblioteca.MetodosCompartilhados;

public class Exemplo3 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("System.out.println é um método estático Compartilhado");
        System.out.println("Uso do método Math.pow para potência:" + Math.pow(2, 3));
        System.out.println("Também utilizamos em nossas conversões:");
        String valor = "1000";
        System.out.println("Double.parseDouble(valor):" + Double.parseDouble(valor));
        System.out.println("Integer.parseInt(valor):" + Integer.parseInt(valor));
        // Agora vamos usar os dois métodos da classe MetodoCompartilhado,
        // sem a criação de objetos:
        System.out.println("Dobro do valor :" +
            MetodoCompartilhados.valorDobro());
        System.out.println("Quadrado do valor :" +
            MetodoCompartilhados.valorQuadrado());
    }
}
```

Execução:

```
System.out.println é um método estático (Compartilhado)
Uso do método Math.pow para potência:8.0
Também utilizamos em nossas conversões:
Double.parseDouble(valor):1000.0
Integer.parseInt(valor):1000
Dobro do valor :10
Quadrado do valor :25.0
```

Notas:

- 1** Foram usados vários métodos estáticos de diferentes classes no exemplo;
- 2** Cada método estático deve ser identificado pela classe anteriormente, ou seja, devemos usar o nome da classe antes do nome do método, e separá-los por um ponto (.);
- 3** O atributo valor na classe MetodosCompartilhados é estático; caso não tivesse sido definido como estático, os métodos estáticos da classe não poderiam fazer menção ao atributo valor;
- 4** Métodos estáticos só podem ter acesso a membros estáticos porque os membros não estáticos necessitam que um objeto seja criado para que possam ser usados.

O modificador: final

Esse modificador pode ser aplicado sobre classes, atributos e métodos e transforma a classe, atributo ou método para a forma constante (final).

O modificador final aplicado a um atributo

Esse modificador, ao ser aplicado a um atributo, altera o atributo de forma a torná-lo constante. É a forma na linguagem Java de criarmos dados com um valor fixo que não poderá ser alterado durante a execução do sistema, tornando-o um atributo constante¹.

Exemplo 4

 Clique no botão acima.

Exemplo 4 – uso de atributo final (constante)

Classe MetodosCompartilhados (pacote: biblioteca):

```
package biblioteca;
public class Constante {
    private final int valor=5;
    /*
    public void setValor(int v) {
        valor = v;
    }
    */
    public int getValor() {
        return valor;
    }
}
```

Classe Exemplo4 (pacote: aplicacao):

```
package aplicacao;
import biblioteca.Constante;
public class Exemplo4 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Constante c = new Constante();
        System.out.println("Valor c = " + c.getValor());
        c.valor = 10;// já existe um aviso de erro
        System.out.println("Valor c = " + c.getValor());
    }
}
```

Execução:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  The final field Constante.valor cannot be assigned
  at aplicacao.Exemplo4.main(Exemplo4.java:11)
```

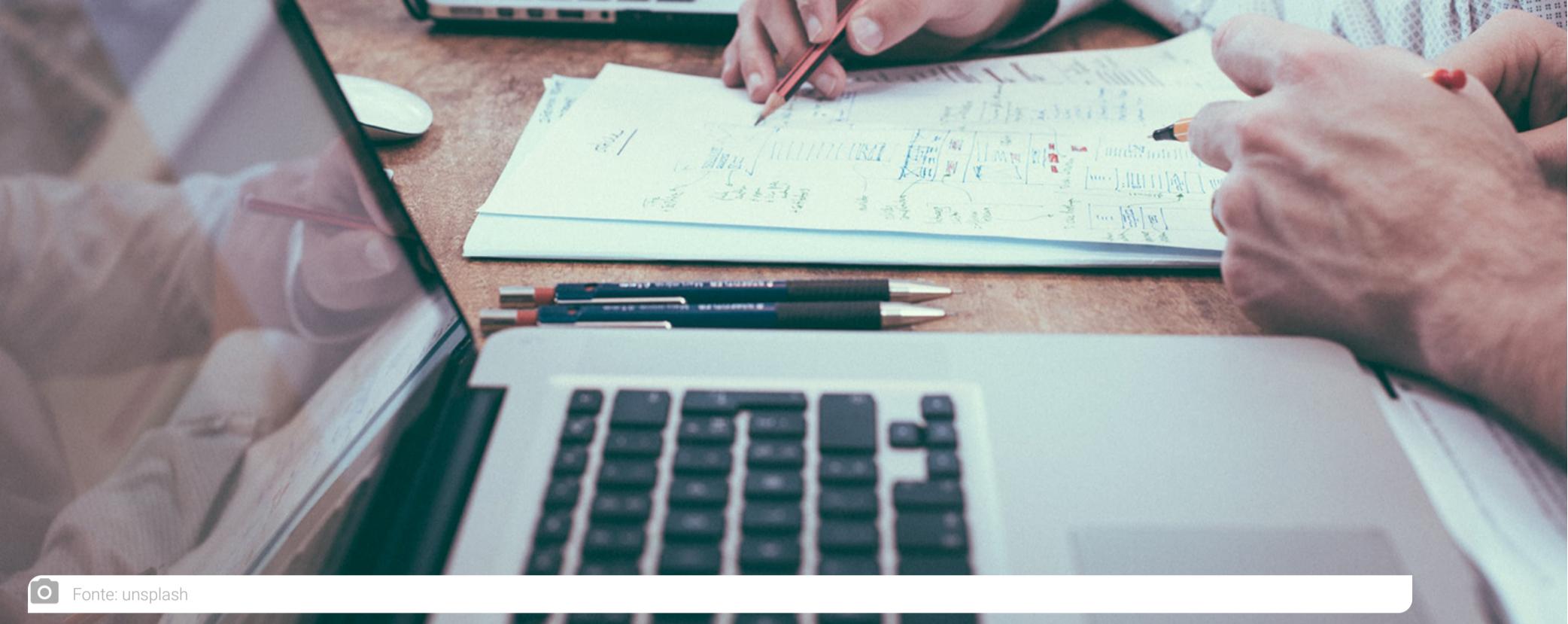
Notas:

- 1** Na classe Constante, o compilador acusou um erro de compilação na atribuição: `valor = v;`

- 2** Não se deve criar métodos Setters para atributos modificados com final por dois motivos: o primeiro é que não devemos tentar alterar um valor constante; e o segundo é que o compilador não deixará você realizar essa operação, e não compilará o seu código;

- 3** Na aplicação, a tentativa de alteração do valor constante gera um erro de compilação, evitando que a operação possa ser realizada.

Um atributo, ou variável auxiliar, modificado com final não poderá ter o valor alterado após ter sido definido.



Fonte: unsplash

O modificador final aplicado a um método

Esse modificador, ao ser aplicado a um método, fará com que ele se torne constante; um método constante é um método que não pode ser alterado, ou seja, em uma situação de herança, esse **método não poderá ser sobreescrito**.

O modificador final, ao ser utilizado em um método, só deve ser aplicado em situações de herança porque sua função é impedir que um método possa ser sobreescrito em alguma subclasse.

Exemplo 5

Clique no botão acima.

Exemplo 5 – uso de método final (constante)

Classe SuperClasse (pacote: biblioteca):

```
package biblioteca;
public class SuperClasse {
    protected static int valor = 5;
    public int getValor() {
        return valor;
    }
    public void setValor(int valor) {
        // Não pode usar a referência this
        // porque valor foi modificado com static
        SuperClasse.valor = valor;
    }
    public final double valorQuadrado() {
        return Math.pow(valor, 2);
    }
    public final int valorDobro() {
        return valor * 2;
    }
}
```

Classe SubClasse (pacote: biblioteca):

```
package biblioteca;
public class SubClasse extends SuperClasse {

    Multiple markers at this line
    - Cannot override the final method from SuperClasse
    - overrides biblioteca.SuperClasse.valorQuadrado

    public double valorQuadrado() {
        return Math.pow(valor, 2);
    }

    Multiple markers at this line
    - overrides biblioteca.SuperClasse.valorDobro
    - Cannot override the final method from SuperClasse

    public int valorDobro() {
        return valor * 2;
    }
}
```

Notas:

- 1** Na classe SuperClasse, os métodos valorQuadrado e ValorDobro foram modificados para final e, dessa forma, impediram que a classe SubClasse pudesse sobrescrever esses métodos;
- 2** Na classe SubClasse, os métodos não puderam ser sobrescritos, impedindo a compilação da SubClasse.

O modificador final aplicado a uma classe

Esse modificador, ao ser aplicado a uma classe, fará com que ela se torne constante; uma classe constante pode ser estendida (possui subclasses), ou seja, **essa classe não poderá ser estendida com subclasses**.

O modificador final, ao ser utilizado em uma classe, impede que ela possa se tornar uma SuperClasse.

Exemplo 6

 Clique no botão acima.

Exemplo 6 – uso de classe final (constante)

Classe SuperClasse (pacote: biblioteca):

```
package biblioteca;
public final class SuperClasse {
    protected static int valor = 5;
    public int getValor() {
        return valor;
    }
    public void setValor(int valor) {
        // Não pode usar a referência this
        // porque valor foi modificado com static
        SuperClasse.valor = valor;
    }
    public double valorQuadrado() {
        return Math.pow(valor, 2);
    }
    public int valorDobro() {
        return valor * 2;
    }
}
```

Classe SubClasse (pacote: biblioteca):

```
package biblioteca;
public class SubClasse extends SuperClasse {
}
```

Notas:

- 1 A classe SuperClasse foi modificada para final, impedindo que possam ser criadas subclasses dela;

- 2 A classe SubClasse, ao estender a classe SuperClasse, gerou um erro de compilação, uma vez que a SuperClasse está impedida de gerar subclasses.

O uso conjunto dos modificadores static e final aplicados a um atributo

Podemos utilizar em conjunto os modificadores static e final a um atributo; tornando-o compartilhado (static) e constante (final).

Exemplo

```
private static final String nome="Acme";
private static final private int idade=25;
private static final double peso=78.5;
```

Observação: os atributos deverão ter os seus valores iniciais definidos na declaração.

O uso conjunto dos modificadores static e final aplicados a um método

Podemos utilizar em conjunto os modificadores static e final a um método, tornando-o compartilhado (static), podendo ser usado sem a necessidade de instanciação de um objeto e constante (final), não podendo ser sobreescrito em nenhuma subclasse.

Exemplo

```
public static final double valorQuadrado() {
    return Math.pow(valor, 2);
}
public static final int valorDobro() {
    return valor * 2;
}
```



Fonte: unsplash

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Classes abstratas

Introdução

Quando trabalhamos com herança, em muitos casos nos deparamos com métodos comuns a várias classes, mas com implementações diferentes em cada uma. O método aquecer comum a Fogão e Micro-ondas possuem diferentes implementações. O mesmo ocorre com vários tipos de transporte: frear, virarDireita, virarEsquerda, acelerar são comuns a vários veículos, mas nem sempre realizam a ação da mesma forma. Frear um caminhão é bem diferente de frear um navio, ou automóvel, ou uma moto. Dessa forma, temos um problema:

Como definir esses métodos na Superclasse se eles são comuns, mas possuem diferentes implementações?

Outra situação é a segurança: imagine “esquecer” algum desses métodos em algum veículo. A segurança do mesmo estará comprometida.

Classes abstratas

Uma classe abstrata determina um conjunto de métodos (regras de negócio, comuns na análise de sistemas) que deverão ser implementados. Esse conjunto de regras obriga que uma Subclasse implemente esses métodos, que, inicialmente, foram declarados, mas não implementados.

O método abstrato

Um método abstrato, ao contrário dos métodos concretos (possuem a implementação), são apenas declarados, não têm a sua implementação codificada.

Exemplo

```
public abstract int calculaINSS( double salario);
public abstract void frear();
public abstract double acelerar( double potencia, int marcha);
```

Os métodos definidos acima não possuem a implementação, apenas as suas declarações. Por isso, devem ser modificados para abstract.

Quando uma classe passa a ter ao menos um método abstrato, ela se torna uma classe abstrata.

Exemplo

```
package biblioteca;
public abstract class Abstrata {
    public abstract int calculaINSS( double salario);
    public abstract void frear();
    public abstract double acelerar( double potencia, int marcha);
}
```

Uma classe abstrata não pode ser instanciada diretamente com seus construtores.

Exemplo

```
package aplicacao;  
import biblioteca.Astrata;  
public class Exemplo7 {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Cannot instantiate the type Astrata  
  
        Astrata abs = new Astrata();  
    }  
}
```

Para atender aos métodos abstratos da classe Astrata, é necessário que utilizemos a herança, uma vez que o uso de classes abstratas exige o emprego dela. Será necessário criar uma ou mais Subclasses desta classe Astrata de forma a atender aos requisitos dos métodos abstratos definidos.

Exemplo 7

 Clique no botão acima.

Exemplo 7 – uso de atributo final (constante)

Classe Abstrata (pacote: biblioteca):

```
package biblioteca;
public abstract class Abstrata {
    private String nome;
    private int idade;
    private double peso, altura;
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public double getPeso() {
        >return peso;
    }
    public void setPeso(double peso) {
        this.peso = peso;
    }
    public double getAltura() {
        >return altura;
    }
    public void setAltura(double altura) {
        this.altura = altura;
    }
    public int getIdade() {
        >return idade;
    }
    public void setIdade(int idade) {
        this.idade = idade;
    }
    public abstract double calculaImc(double peso, double altura);
    public abstract boolean maiorIdade();
    public abstract void imprimir();
}
```

Classe SubAbstrata (pacote: biblioteca):

Versão 1: package biblioteca;

- The type SubAbstrata must implement the inherited abstract method Abstrata.maioridade()
- The type SubAbstrata must implement the inherited abstract method Abstrata.calculaImc(double, double)
- The type SubAbstrata must implement the inherited abstract method Abstrata.imprimir()

```
public class SubAbstrata extends Abstrata{  
}
```

Versão 2:

```
public class SubAbstrata extends Abstrata{  
    public double calculaImc(double peso, double altura) {  
        double imc = peso / Math.pow(altura, 2);  
        return imc;  
    }  
    public boolean maiorIdade() {  
        if(this.getIdade()>=18) {  
            return true;  
        }  
        else {  
            return false;  
        }  
    }  
    public void imprimir() {  
        System.out.println("Nome :" + super.getNome());  
        System.out.println("Idade :" + super.getIdade());  
        System.out.println("Peso :" + super.getPeso());  
        System.out.println("Altura :" + super.getAltura());  
    }  
}
```

Classe Exemplo7 (pacote: aplicacao):

```
package aplicacao;  
import biblioteca.Abstrata;  
import biblioteca.SubAbstrata;  
public class Exemplo7 {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        SubAbstrata sabs = new SubAbstrata();  
        sabs.setNome("André");  
        sabs.setIdade(25);  
        sabs.setAltura(1.75);  
        if(sabs.maiorIdade()) {  
            System.out.println("Sim, maior de idade.");  
        }  
        else{  
            System.out.println("Não, não é de maior idade.");  
        }  
        System.out.println("IMC = "+ sabs.calculaImc(90, 1.78));  
        sabs.imprimir();  
    }  
}
```

Execução:

```
Sim, maior de idade.  
IMC = 28.40550435551067  
Nome :André  
Idade :25  
Peso :85.5  
Altura :1.75
```

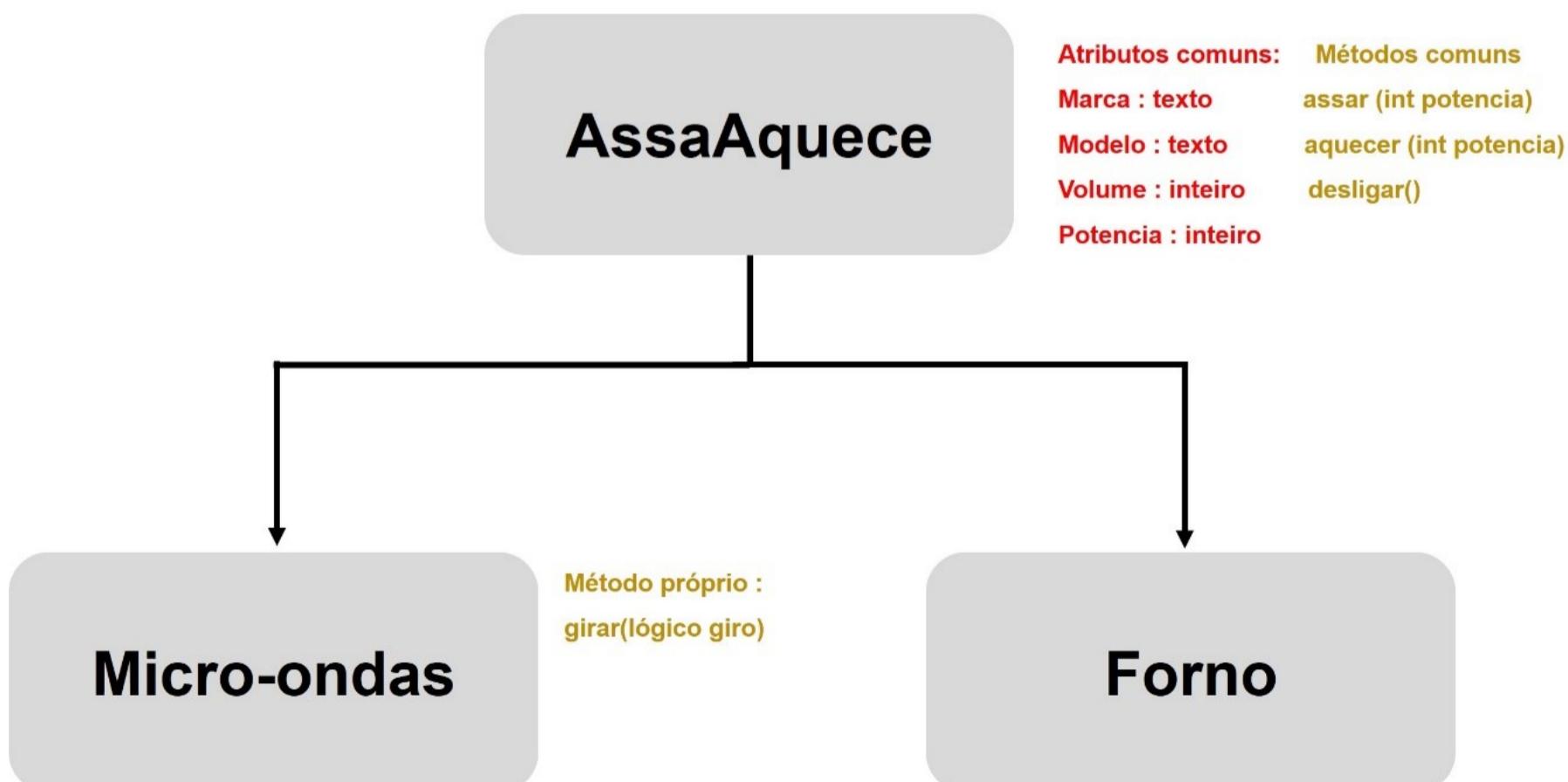
Notas:

- 1** A classe Abstrata possui métodos abstratos que devem ser implementados pela(s) subclasse(s);
- 2** As subclasses só compilarão após atenderem aos requisitos e implementarem, através de métodos concretos, todos os métodos abstratos definidos na superclasse;
- 3** A aplicação criou o objeto a partir da subclasse, uma vez que não pôde criar a partir da superclasse com seu construtor padrão.

Na prática, as classes Forno e Micro-ondas podem aquecer ou assar, sendo que o Micro-ondas necessita girar o prato e o Forno não. Ambos possuem uma potência para realizar as ações, mas apenas o Micro-ondas necessita da ação extra girar(). Dessa forma, ambos os métodos são comuns às duas classes e, por isso, devem ser definidos na Superclasse. Entretanto, as implementações são distintas, o que faz com que devamos usar o conceito de classe abstrata.

Vejamos a distribuição de classes a seguir, levando em conta que os métodos de acesso (Setters e Getters) - imprimir, cadastrar, entrada e construtores - devem ser implementados em todas as classes.

Classes Abstratas



 Solução

 Clique no botão acima.

Superclasse AssaAquece (pacote: assadores):

```
package assadores;
import java.util.Scanner;
public abstract class AssaAquece {
    private String marca, modelo;
    private int volume, potencia;
    public String getMarca() {
        return marca;
    }
    public void setMarca( String marca ) {
        this.marca = marca;
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo( String modelo ) {
        this.modelo = modelo;
    }
    public int getVolume() {
        return volume;
    }
    public void setVolume( int volume ) {
        this.volume = volume;
    }
    public int getPotencia() {
        return potencia;
    }
    public void setPotencia( int potencia ) {
        this.potencia = potencia;
    }
    public AssaAquece( ) { }
    public AssaAquece( String marca, String modelo ) {
        this.marca = marca;
        this.modelo = modelo;
    }
    public AssaAquece( int volume, int potencia ) {
        this.volume = volume;
        this.potencia = potencia;
    }
    public AssaAquece( String marca, String modelo, int volume, int potencia ) {
        this.marca = marca;
        this.modelo = modelo;
        this.volume = volume;
        this.potencia = potencia;
    }
    public void cadastrar( String marca, String modelo, int volume, int potencia ) {
        this.marca = marca;
        this.modelo = modelo;
        this.volume = volume;
        this.potencia = potencia;
    }
    public void imprimir() {
        System.out.println("Marca :" + getMarca());
        System.out.println("Modelo :" + getModelo());
        System.out.println("Volume :" + getVolume());
        System.out.println("Potência:" + getVolume());
    }
    public void entrada() {
        Scanner ent = new Scanner(System.in);
        System.out.println("Marca :");
        setMarca(ent.nextLine());
    }
}
```

```
System.out.println("Modelo :");
setModelo(ent.nextLine());
System.out.println("Volume :");
setVolume(Integer.parseInt(ent.nextLine()));
System.out.println("Potência:");
setVolume(Integer.parseInt(ent.nextLine()));

}

public abstract void assar(int potencia);
public abstract void aquecer(int potencia);
public abstract void desligar();
}
```

Subclasse Micorondas (pacote: assadores):

```
package assadores;
import java.util.Scanner;
public class Microondas extends AssaAquece{
    public Microondas( ) {
        super();
    }
    public Microondas( String marca, String modelo ) {
        super(marca, modelo);
    }
    public Microondas( int volume, int potencia ) {
        super(volume, potencia);
    }
    public Microondas( String marca, String modelo, int volume, int potencia ) {
        super(marca, modelo, volume, potencia);
    }
    public void cadastrar( String marca, String modelo, int volume, int potencia ) {
        super.cadastrar(marca, modelo, volume, potencia);
    }
    public void imprimir() {
        super.imprimir();
    }
    public void entrada() {
        super.entrada();
    }
    private void girar(boolean giro) {
        if(giro){
            System.out.println("Girando o prato!");
        } else {
            System.out.println("Prato parado!");
        }
    }
    public void assar(int potencia) {
        super.setPotencia(potencia);
        girar(true);
        System.out.println("Microondas assando com potencia=" + super.getPotencia());
    }
    public void aquecer(int potencia) {
        super.setPotencia(potencia);
        girar(true);
        System.out.println("Microondas aquecendo com potencia=" +
                           super.getPotencia());
    }
    public void desligar() {
        girar(false);
        System.out.println("Desligar Microondas");
    }
}
```

Subclasse Forno (pacote: assadores):

```
package assadores;
public class Forno extends AssaAquece{
    public Forno( ) {
        super();
    }
    public Forno( String marca, String modelo ) {
        super(marca, modelo);
    }
    public Forno( int volume, int potencia ) {
        super(volume, potencia);
    }
    public Forno( String marca, String modelo, int volume, int potencia ) {
        super(marca, modelo, volume, potencia);
    }
    public void cadastrar( String marca, String modelo, int volume, int potencia ) {
        super.cadastrar(marca, modelo, volume, potencia);
    }
    public void imprimir() {
        super.imprimir();
    }
    public void entrada() {
        super.entrada();
    }
    public void assar(int potencia) {
        super.setPotencia(potencia);
        System.out.println("Forno assando com potencia=" + super.getPotencia());
    }
    public void aquecer(int potencia) {
        super.setPotencia(potencia);
        System.out.println("Forno aquecendo com potencia=" + super.getPotencia());
    }
    public void desligar() {
        System.out.println("Desligar Forno.");
    }
}
```

Classe Comida (pacote: aplicacao):

```
package aplicacao;
// importará todas as classes do pacote
import assadores.*;
public class Comida {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Forno f = new Forno("GE", "f505", 60, 120);
        Microondas m = new Microondas("Brastemp", "M328", 28, 250);
        f.assar(200);
        f.desligar();
        m.aquecer(150);
        m.desligar();
    }
}
```

Execução:

```
Forno assando com potencia=200
Desligar Forno.
Girando o prato!
Microondas aquecendo com potencia=150
Prato parado!
Desligar Microondas
```

Notas:

- 1** As Subclasses atendem aos requisitos da Superclasse Abstrata, ao implementarem todos os métodos abstratos da Superclasse;
- 2** As implementações das classes Forno e Micro-ondas se diferem pela necessidade do Micro-ondas fazer o prato girar;
- 3** Na aplicação, os objetos usam o mesmo construtor e métodos, mas realizam operações diferentes em função de suas necessidades próprias.

Podemos instanciar uma classe abstrata ao usarmos na instanciação um construtor da Subclasse. Isso fará com que o objeto criado a partir da classe abstrata use os métodos concretos da Subclasse que os implementou.

Exemplo:

Classe Comida (pacote: aplicacao): Atualizada com versão 2.

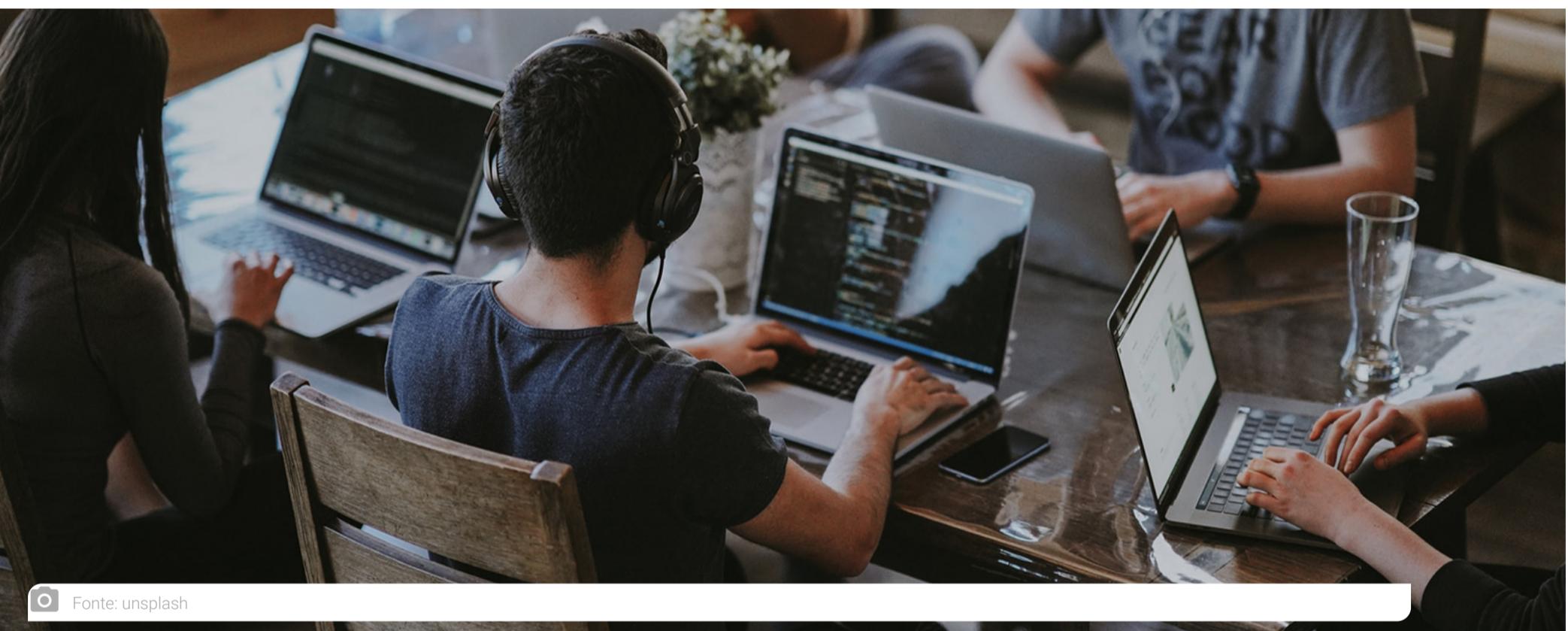
```
package aplicacao;
// importará todas as classes do pacote
import assadores.*;
public class Comida {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // objetos Forno e Microondas
        Forno f = new Forno("GE", "f505", 60, 120);
        Microondas m = new Microondas("Brastemp", "M328", 28, 250);
        f.assar(200);
        f.desligar();
        m.aquecer(150);
        m.desligar();
        System.out.println();
        // objeto AssaAquece com construtor da classe Forno
        AssaAquece af = new Forno("Bratemp", "br105", 55, 200);
        af.aquecer(300);
        af.desligar();
        System.out.println();
        // objeto AssaAquece com construtor da classe Microondas
        AssaAquece am = new Microondas("GE", "ge10", 35, 180);
        am.aquecer(120);
        am.desligar();
        System.out.println();
    }
}
```

Execução:

```
Forno assando com potencia=200
Desligar Forno.
Girando o prato!
Microondas aquecendo com potencia=150
Prato parado!
Desligar Microondas
Forno aquecendo com potencia=300
Desligar Forno.
Girando o prato!
Microondas aquecendo com potencia=120
Prato parado!
Desligar Microondas
```

Notas:

o comportamento do objeto criado a partir da Superclasse AssaAquece será o mesmo da Subclasse ao qual o objeto utilizou o construtor.



Fonte: unsplash

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Introdução

Em muitos projetos devemos atender a diferentes requisitos, que podem ser definidos a partir de “regras de negócio”. Uma interface se assemelha a uma classe abstrata, com a exceção de que uma interface só possuir métodos abstratos.

Enquanto uma classe abstrata pode conter métodos concretos, uma interface apenas define “regras” a serem implementadas pelas classes que implementarem a interface. Uma interface então possui apenas atributos especiais (static e final) e métodos abstratos. Ao implementar uma interface, a classe obrigatoriamente contará com seus atributos, e terá que implementar os métodos nela definidos. Ao contrário das classes abstratas, a interface não precisa ser usada em situações de herança, apesar de poderem ser usadas nesse contexto.

Para criar uma interface, devemos incluir um arquivo próprio para isso. Acompanhe na imagem a seguir:

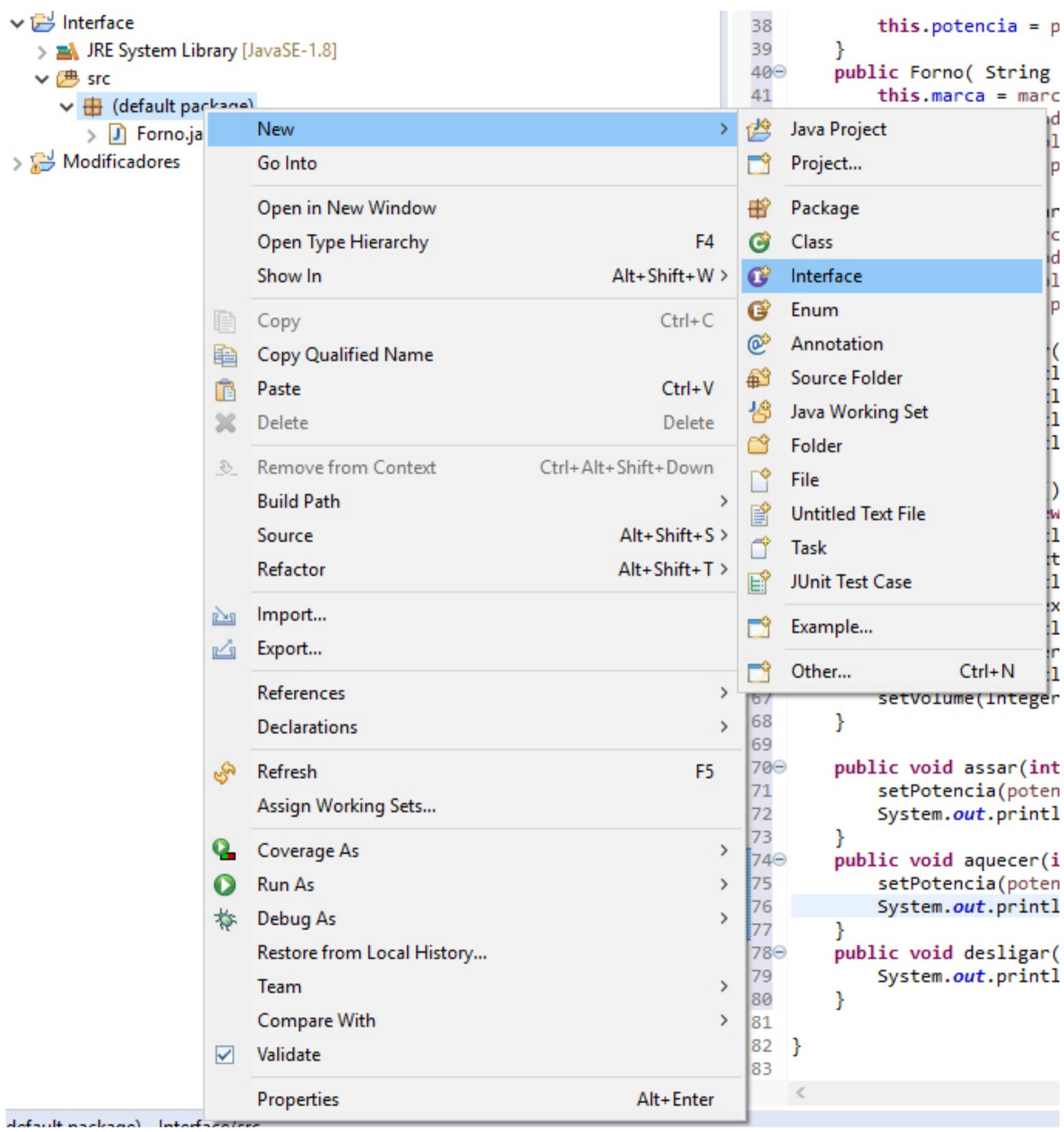


Imagen 1: Incluir um arquivo para uma interface.

Como exemplo, vamos usar apenas a classe Forno e determinar uma interface para os métodos abstratos.

Interface Controle (pacote: biblioteca): Define as regras de implementação das classes.

```
package biblioteca;
public interface Controle {
    // Todos os atributos definidos em uma interface
    // são implicitamente static e final
    int POTENCIAMAXIMA = 350; // static e final
    int POTENCIAMINIMA = 120; // static e final
    // Somente os métodos Getters
    // como os atributos em uma interface são final
    // não poderão ser alterados, ou seja, não teremos
    // os métodos Setters
    public int getPotenciaMaxima();
    public int getPotenciaMinima();
    // Todos os métodos definidos em uma interface
    // são implicitamente public e abstract
    void assar(int potencia); // public e abstract
    void aquecer(int potencia); // public e abstract
    void desligar(); // public e abstract
}
```

Classe Forno (pacote: biblioteca):

```
package biblioteca;
import java.util.Scanner;
public class Forno implements Controle {
    private String marca, modelo;
    private int volume, potencia;
    public String getMarca() {
        return marca;
    }
    public void setMarca( String marca ) {
        this.marca = marca;
    }
    public String getModelo() {
        return modelo;
    }
    public void setModelo( String modelo ) {
        this.modelo = modelo;
    }
    public int getVolume() {
        return volume;
    }
    public void setVolume( int volume ) {
        this.volume = volume;
    }
    public int getPotencia() {
        return potencia;
    }
    public void setPotencia( int potencia ) {
        this.potencia = potencia;
    }
    public Forno( ) { }
    public Forno( String marca, String modelo ) {
        this.marca = marca;
        this.modelo = modelo;
    }
    public Forno( int volume, int potencia ) {
        this.volume = volume;
        this.potencia = potencia;
    }
    public Forno( String marca, String modelo, int volume, int potencia ) {
        this.marca = marca;
        this.modelo = modelo;
        this.volume = volume;
        this.potencia = potencia;
    }
    public void cadastrar( String marca, String modelo, int volume, int potencia ) {
        this.marca = marca;
        this.modelo = modelo;
        this.volume = volume;
        this.potencia = potencia;
    }
    public void imprimir() {
        System.out.println("Marca :" + getMarca());
        System.out.println("Modelo :" + getModelo());
        System.out.println("Volume :" + getVolume());
        System.out.println("Potência:" + getVolume());
    }
    public void entrada() {
        Scanner ent = new Scanner(System.in);
        System.out.println("Marca :");
        setMarca(ent.nextLine());
        System.out.println("Modelo :");
        setModelo(ent.nextLine());
    }
}
```

```

        System.out.println("Volume :");
        setVolume(Integer.parseInt(ent.nextLine()));
        System.out.println("Potência:");
        setVolume(Integer.parseInt(ent.nextLine()));

    }

    public int getPotenciaMaxima() {
        return POTENCIAMAXIMA;
    }

    public int getPotenciaMinima() {
        return POTENCIAMINIMA;
    }

    public void assar(int potencia) {
        setPotencia(potencia);
        System.out.println("Forno assando com potencia=" + getPotencia());
    }

    public void aquecer(int potencia) {
        setPotencia(potencia);
        System.out.println("Forno aquecendo com potencia=" + getPotencia());
    }

    public void desligar() {
        System.out.println("Desligar Forno.");
    }

}

```

Classe AppForno (pacote: aplicacao):

```

package aplicacao;
import biblioteca.Forno;
public class AppForno {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // objetos Forno e Microondas
        Forno f = new Forno("GE", "f505", 60, 120);
        f.assar(200);
        f.desligar();
        System.out.println();
    }
}

```

Execução:

```

Forno assando com potencia=200
Desligar Forno.

```

Notas:

- 1** A interface Controle (deve ser identificada como uma classe, ou seja, seu identificador deve começar por letras maiúsculas) determina as regras de negócio a serem implementadas pelas classes que implementarem a interface;

- 2** A classe Forno implementa a interface Controle, isso faz com tenha acesso aos atributos da interface (static e final) e deve implementar todos os métodos definidos na interface Controle;

- 3** A aplicação utiliza normalmente a classe Forno, independentemente se esta utiliza classes Abstratas ou Interfaces.

- Uma classe pode implementar mais de uma interface:

```
public class Forno implements Controle, Regras, Descontos { .. }
```

A classe Forno deverá obrigatoriamente implementar todas as funcionalidades determinadas pelas três interfaces.

- Uma interface pode estender outra (herança entre Interfaces):

```
public interface Controle extends Regras {
```

- Podemos utilizar em conjunto a Herança, Agregação, Classes Abstratas e Interfaces:

```
public class SubClasse extends Abstrata implements Regras, Controle
```

Acompanhe o exemplo a seguir:

Classe tampa (pacote: biblioteca):

```
package biblioteca;
public class Tampa {
    // ...
}
```

Superclasse Abstrata (pacote: biblioteca):

```
package biblioteca;
public abstract class Abstrata {
    public Tampa tampa = new Tampa();
    public abstract void metodoX(int n);
}
```

Subclasse SubClasse (pacote: biblioteca):

```
package biblioteca;
public class SubClasse extends Abstrata implements Regras, Controle {
    public void imprimir() {
        // ...
    }
    public void entrada() {
        // ...
    }
    public int getPotenciaMaxima() {
        // ...
        return 0;
    }
    public int getPotenciaMinima() {
        // ...
        return 0;
    }
    public void assar(int potencia) {
        // ...
    }
    public void aquecer(int potencia) {
        // ...
    }
    public void desligar() {
        // ...
    }
    public void metodoX(int n) {
        // ...
    }
}
```

Interface Regras (pacote: biblioteca):

```
package biblioteca;
public interface Regras {
    void imprimir();
    void entrada();
}
```

Interface Controle (pacote: biblioteca):

```
package biblioteca;
public interface Controle extends Regras {
    // Todos os atributos definidos em uma interface
    // são implicitamente static e final
    int POTENCIAMAXIMA = 350; // static e final
    int POTENCIAMINIMA = 120; // static e final
    // Somente os métodos Getters
    // como os atributos em uma interface são final
    // não poderão ser alterados, ou seja, não teremos
    // os métodos Setters
    public int getPotenciaMaxima();
    public int getPotenciaMinima();
    // Todos os métodos definidos em uma interface
    // são implicitamente public e abstract
    void assar(int potencia); // public e abstract
    void aquecer(int potencia); // public e abstract
    void desligar(); // public e abstract
}
```

Notas:

1 Foram criadas duas interfaces, que foram implementadas pela SubClasse;

2 A superclasse Abstrata é uma classe abstrata que agrega a classe Tampa;

3 A subclasse SubClasse estende a superclasse abstrata (extends) e implementa (implements) a funcionalidade do método metodoX(), além de implementar todas as funcionalidades das interfaces Regras e Controle. Não esquecendo que essa subclasse herdou também o objeto tampa (Tampa) da superclasse.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Atividades

1) Classes abstratas têm uma função importante na orientação a objeto em Java, pois incentivam o polimorfismo e dão flexibilidade à modelagem de classes, favorecendo a abstração. A respeito de classes abstratas, considere as seguintes afirmativas:

1. Uma classe abstrata pode ser instanciada através da palavra chave new.
2. Para que uma classe seja abstrata, ela precisa ter pelo menos um método abstrato.
3. Uma classe abstrata tem como objetivo fornecer uma superclasse apropriada a partir da qual outras classes podem herdar e, assim, compartilhar um design comum.
4. Para criarmos uma classe ou método abstrato, usamos a palavra-chave abstract.

Assinale a alternativa correta:

- a) Somente a afirmativa 1 é verdadeira.
 - b) Somente a afirmativa 2 é verdadeira.
 - c) Somente as afirmativas 2 e 3 são verdadeiras.
 - d) Somente as afirmativas 1 e 4 são verdadeiras.
 - e) Somente as afirmativas 2, 3 e 4 são verdadeiras.
-

2) "A interface é um recurso muito utilizado em Java, bem como na maioria das linguagens orientadas a objeto, para 'obrigar' a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente." [ROBSON, 2012 – Devmedia]

Sobre interfaces na linguagem Java, é correto afirmar que:

- I. Interfaces podem ter métodos construtores.
- II. Uma Interface pode estender outra interface (com extends) para criar uma hierarquia de interfaces.
- III. Interfaces podem ter membros públicos, private e protected.
- IV. Interfaces não podem implementar métodos, somente defini-los.

Assinale a alternativa correta:

- a) Somente as afirmativas 1 e 2 são verdadeiras.
 - b) Somente as afirmativas 2 e 3 são verdadeiras.
 - c) Somente as afirmativas 1 e 3 são verdadeiras.
 - d) Somente as afirmativas 2 e 4 são verdadeiras.
 - e) Somente as afirmativas 3 e 4 são verdadeiras.
-

Notas

Atributo constante¹

Um atributo é dito final quando seu valor é constante, e não poderá ser modificado durante a execução da aplicação.

Referências

DEITEL, Paul. **Java: como programar** (Biblioteca Virtual). 10a ed. São Paulo: Pearson, 2017.

- Tratamento de exceções.

Explore mais

Pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto.

Em caso de dúvidas, converse com seu professor online por meio dos recursos disponíveis no ambiente de aprendizagem.