

# Programação Orientada a Objetos - POOS3

1

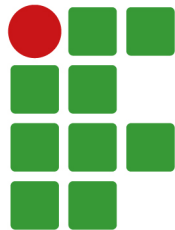
---

Tecnologia em Análise e Desenvolvimento de Sistemas

## Aula 5

Herança, sobrescrita, polimorfismo, classe concreta e classe abstrata

2º semestre de 2018



**INSTITUTO FEDERAL**

São Paulo

Câmpus Araraquara

# Herança

---

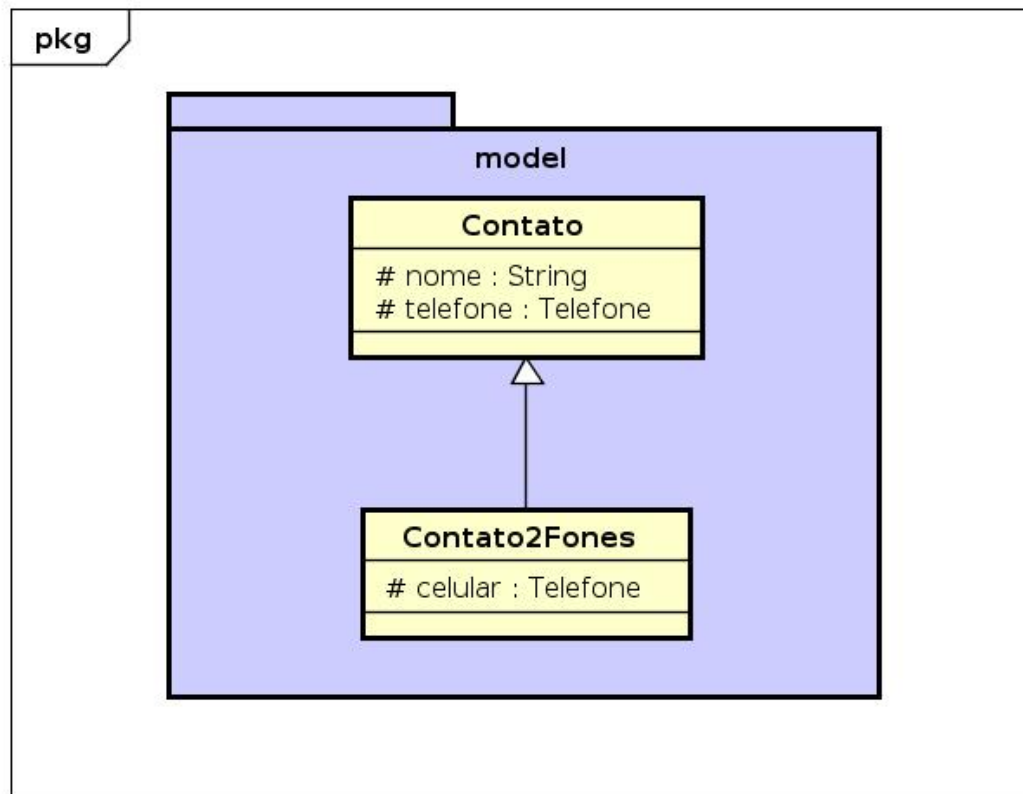
- Herança é um dos três princípios básicos da programação orientada a objetos, porque permite a criação e classificação hierárquica.
  - Com seu uso, pode-se criar uma classe geral que define características comuns a um conjunto de itens relacionados.
- A herança torna possível a reutilização de funcionalidades previamente definidas em uma classe.
- A finalidade é que a **subclasse** inclua o comportamento da **superclasse** e adicione mais funcionalidades.

# Herança: contextualização

---

- Considerando a implementação de uma classe Contato:
  - Essa classe já está em produção, ou seja, clientes estão utilizando o sistema e objetos dessa classe são instanciados.
  - Outro requisito pode surgir no sistema, como por exemplo, a inclusão de um segundo número de telefone.
- Com a herança pode-se ampliar (estender) as características e funcionalidades de uma classe já existente.
  - Também é possível alterar o comportamento de alguns métodos implementados na classe original (superclasse / classe pai).

# Contato e Contato2Fones



powered by Astah

```
public class Contato {  
    protected String nome;  
    protected Telefone telefone;
```

A classe Contato define dois atributos, o nome do contato e um telefone.

```
    public Contato(String nome, int ddd, int numero) {  
        this.nome = nome.toUpperCase();  
        telefone = new Telefone(ddd, numero);  
    }
```

```
@Override
```

```
    public String toString() {  
        return "Contato: " + nome + " \t Telefone: " + telefone.toString();  
    }
```

```
    public String getNome() {        return nome;    }
```

```
    public void setNome(String nome) {        this.nome = nome;    }
```

```
    public Telefone getTelefone() {        return telefone;    }
```

```
    public void setTelefone(Telefone telefone) {        this.telefone = telefone;    }
```

```
}
```

```
public class Contato2Fones extends Contato{
```

```
    private Telefone fone2;
```

```
    public Contato2Fones(String nome, int ddd, int numero, int ddd2, int numero2) {  
        super(nome, ddd, numero);  
        fone2 = new Telefone(ddd2, numero2);  
    }
```

```
@Override
```

```
    public String toString() {  
        String texto = super.toString();  
        texto += "\t Telefone: " + this.fone2.toString();  
        return texto;  
    }
```

```
    public Telefone getFone2() {  
        return fone2;  
    }
```

```
    public void setFone2(Telefone fone2) {  
        this.fone2 = fone2;  
    }  
}
```

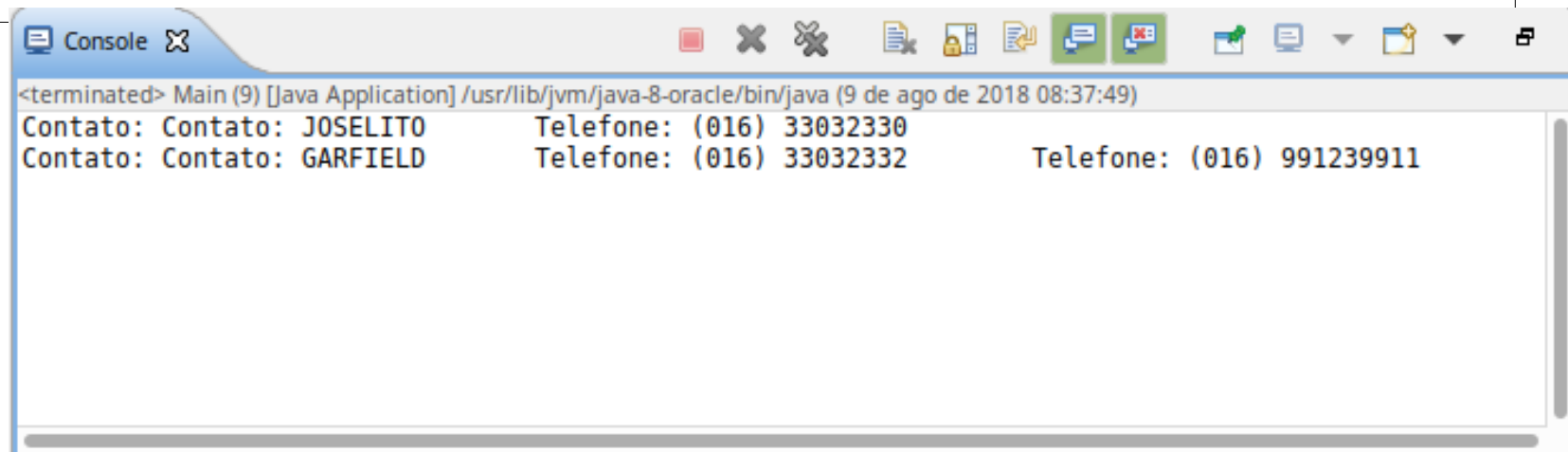
Contato2Fones amplia a classe Contato, para formalizarmos essa ampliação dizemos que a Contato2Fones **estende (extends)** a classe Contato.

Aqui temos que Contato é superclasse (pai) de Contato2Fones.

Além disso, Contato2Fones é um Contato.

O **super** é a indicação da superclasse. No caso, `super()` é a chamada do método construtor da superclasse. Sempre devemos construir a superclasse na subclasse.

```
public class Main {  
  
    public static void main(String[] args) {  
        Contato contato1;  
        Contato2Fones contato2;  
  
        contato1 = new Contato("Joselito", 16, 33032330);  
        contato2 = new Contato2Fones("Garfield", 16, 33032332, 16, 991239911);  
  
        System.out.println("Contato: " + contato1.toString());  
        System.out.println("Contato: " + contato2.toString());  
    }  
}
```



The screenshot shows a Java application window titled "Console". The command prompt shows the execution of the Java program. The output displays the details of two contact objects: "JOSELITO" and "GARFIELD", including their IDs and phone numbers.

```
<terminated> Main (9) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (9 de ago de 2018 08:37:49)  
Contato: Contato: JOSELITO      Telefone: (016) 33032330  
Contato: Contato: GARFIELD      Telefone: (016) 33032332      Telefone: (016) 991239911
```

```
public class Contato {  
    protected String nome;  
    protected Telefone telefone;
```

```
    public Contato(String nome, int ddd, int numero) {  
        this.nome = nome.toUpperCase();  
        telefone = new Telefone(ddd, numero);  
    }
```

Estudamos a pouco o conceito de **overload** (sobrecarga), agora iremos estudar o conceito de **override** (sobrescrita).

```
@Override  
    public String toString() {  
        return "Contato: " + nome + " \t Telefone: " + telefone.toString();  
    }
```

```
    public String getNome() {        return nome;    }
```

```
    public void setNome(String nome) {        this.nome = nome;    }
```

```
    public Telefone getTelefone() {        return telefone;    }
```

```
    public void setTelefone(Telefone telefone) {        this.telefone = telefone;    }
```

```
}
```



# Sobrescrita

---

- Essa técnica consiste na definição de métodos com a mesma assinatura (nome e lista de argumentos) entre superclasse e subclasse, e com corpo diferente.
- Ou seja, o método da classe pai tem um comportamento e o mesmo método (sobrescrito) na classe filha tem outro comportamento.

```
public class Funcionario {  
    private String nome;  
    private double salario;  
  
    public Funcionario(double salario, String nome) {  
        super();  
        this.salario = salario;  
        this.nome = nome;  
    }  
  
    public String getNome() {        return nome;    }  
  
    public void setNome(String nome) {        this.nome = nome;    }  
  
    public double getSalario() {        return salario; }  
  
    public void setSalario(double salario) {        this.salario = salario;    }  
  
    public double getComissao() {  
        return getSalario() * 50 / 100;  
    }  
}
```

```
public class Gerente extends Funcionario{

    public Gerente(double salario, String nome) {
        super(salario, nome);
    }

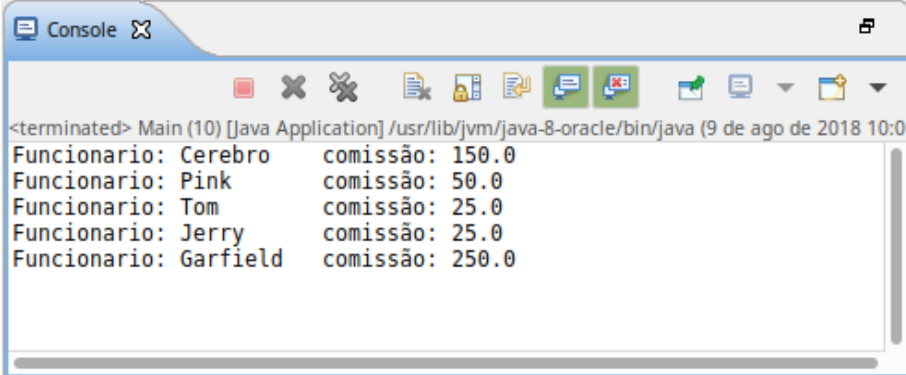
    @Override
    public double getComissao() {
        return getSalario() * 15 / 100;
    }
}
```

```
public class Operador extends Funcionario{

    public Operador(double salario, String nome) {
        super(salario, nome);
    }

    @Override
    public double getComissao() {
        return getSalario() * 5 / 100;
    }
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Gerente f1;  
        Operador f2, f3, f4;  
        Funcionario f5;  
  
        f1 = new Gerente(1000, "Cerebro");  
        f2 = new Operador(1000, "Pink");  
        f3 = new Operador(500, "Tom");  
        f4 = new Operador(500, "Jerry");  
        f5 = new Funcionario(500, "Garfield");  
  
        System.out.println("Funcionario: " + f1.getNome() +  
            " \tcomissão: " + f1.getComissao());  
        System.out.println("Funcionario: " + f2.getNome() +  
            " \tcomissão: " + f2.getComissao());  
        System.out.println("Funcionario: " + f3.getNome() +  
            " \tcomissão: " + f3.getComissao());  
        System.out.println("Funcionario: " + f4.getNome() +  
            " \tcomissão: " + f4.getComissao());  
        System.out.println("Funcionario: " + f5.getNome() +  
            " \tcomissão: " + f5.getComissao());  
    }  
}
```

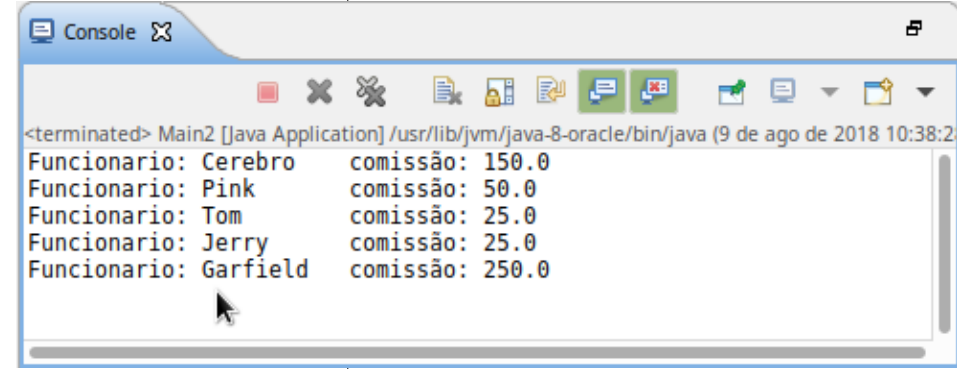


```
<terminated> Main (10) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (9 de ago de 2018 10:0  
Funcionario: Cerebro      comissão: 150.0  
Funcionario: Pink        comissão: 50.0  
Funcionario: Tom         comissão: 25.0  
Funcionario: Jerry       comissão: 25.0  
Funcionario: Garfield    comissão: 250.0
```

Qual a diferença  
de criar três  
classes diferentes?  
Não vi vantagem  
nessa herança!!!



```
public class Main2 {  
  
    public static void main(String[] args) {  
  
        Funcionario f1, f2, f3, f4, f5;  
  
        f1 = new Gerente(1000, "Cerebro");  
        f2 = new Operador(1000, "Pink");  
        f3 = new Operador(500, "Tom");  
        f4 = new Operador(500, "Jerry");  
        f5 = new Funcionario(500, "Garfield");  
  
        System.out.println("Funcionario: " + f1.getNome() +  
            " \tcomissão: " + f1.getComissao());  
        System.out.println("Funcionario: " + f2.getNome() +  
            " \tcomissão: " + f2.getComissao());  
        System.out.println("Funcionario: " + f3.getNome() +  
            " \tcomissão: " + f3.getComissao());  
        System.out.println("Funcionario: " + f4.getNome() +  
            " \tcomissão: " + f4.getComissao());  
        System.out.println("Funcionario: " + f5.getNome() +  
            " \tcomissão: " + f5.getComissao());  
    }  
}
```



```
<terminated> Main2 [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (9 de ago de 2018 10:38:2  
Funcionario: Cerebro      comissão: 150.0  
Funcionario: Pink        comissão: 50.0  
Funcionario: Tom         comissão: 25.0  
Funcionario: Jerry       comissão: 25.0  
Funcionario: Garfield    comissão: 250.0
```

Só diminuiu a  
quantidade de definições  
de variáveis, ainda não  
vejo vantagem!





Começou a fazer sentido, mas como isso funciona? Não são objetos diferentes?

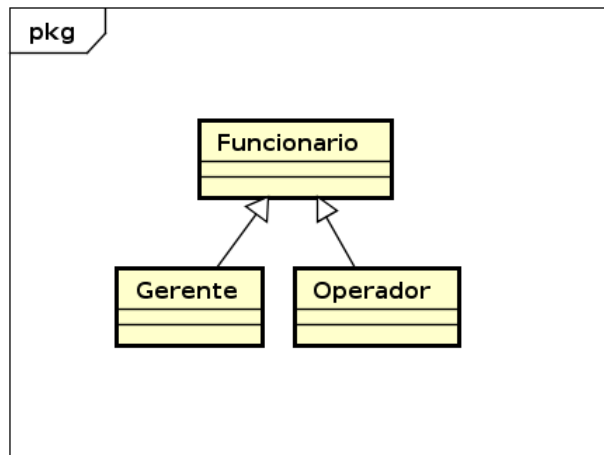
Polimorfismo!



```
public class Main3 {  
  
    public static void main(String[] args) {  
        Funcionario[] funcs = new Funcionario[5];  
  
        funcs[0] = new Gerente(1000, "Cerebro");  
        funcs[1] = new Operador(1000, "Pink");  
        funcs[2] = new Operador(500, "Tom");  
        funcs[3] = new Operador(500, "Jerry");  
        funcs[4] = new Funcionario(500, "Garfield");  
  
        for(int i=0; i<5; i++) {  
            System.out.println("Funcionario: " + funcs[i].getNome() +  
                               " \tcomissão: " + funcs[i].getComissao());  
        }  
    }  
}
```

# Polimorfismo

- Existem duas formas de polimorfismo:
  - De métodos
    - Falaremos mais adiante
  - De compatibilidade de tipos
    - Classes filhas são, também, do mesmo tipo da classe pai



Observe que Gerente e Operador são filhas de Funcionario, desta forma, pode-se dizer:

- Gerente **é um** Funcionário
- Operador **é um** Funcionário

E com isso, temos que as classes filhas podem ser tratadas como se fossem a classe pai.

# Problema

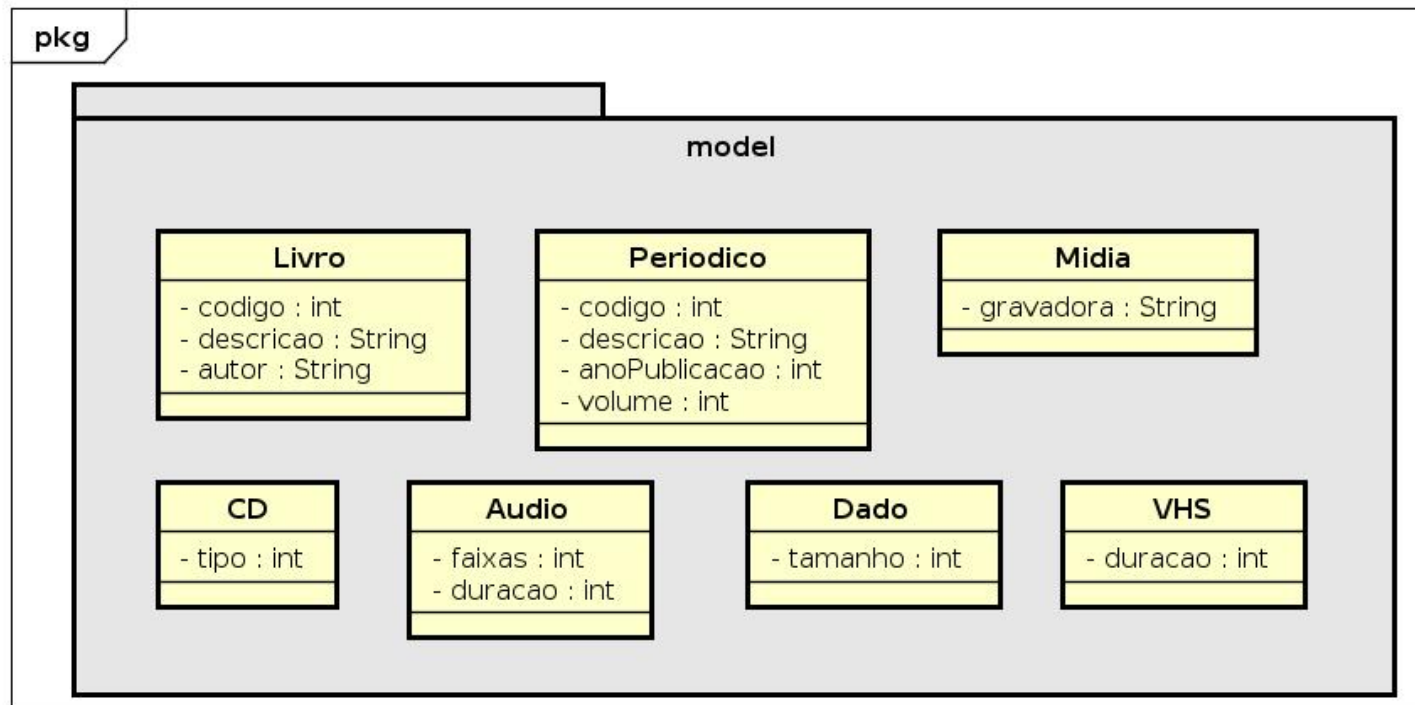
- Uma biblioteca possui vários itens em seu acervo, são eles:
  - Livro;
  - Periódico;
  - Mídia:
    - CD:
      - Áudio
      - Dados
    - VHS





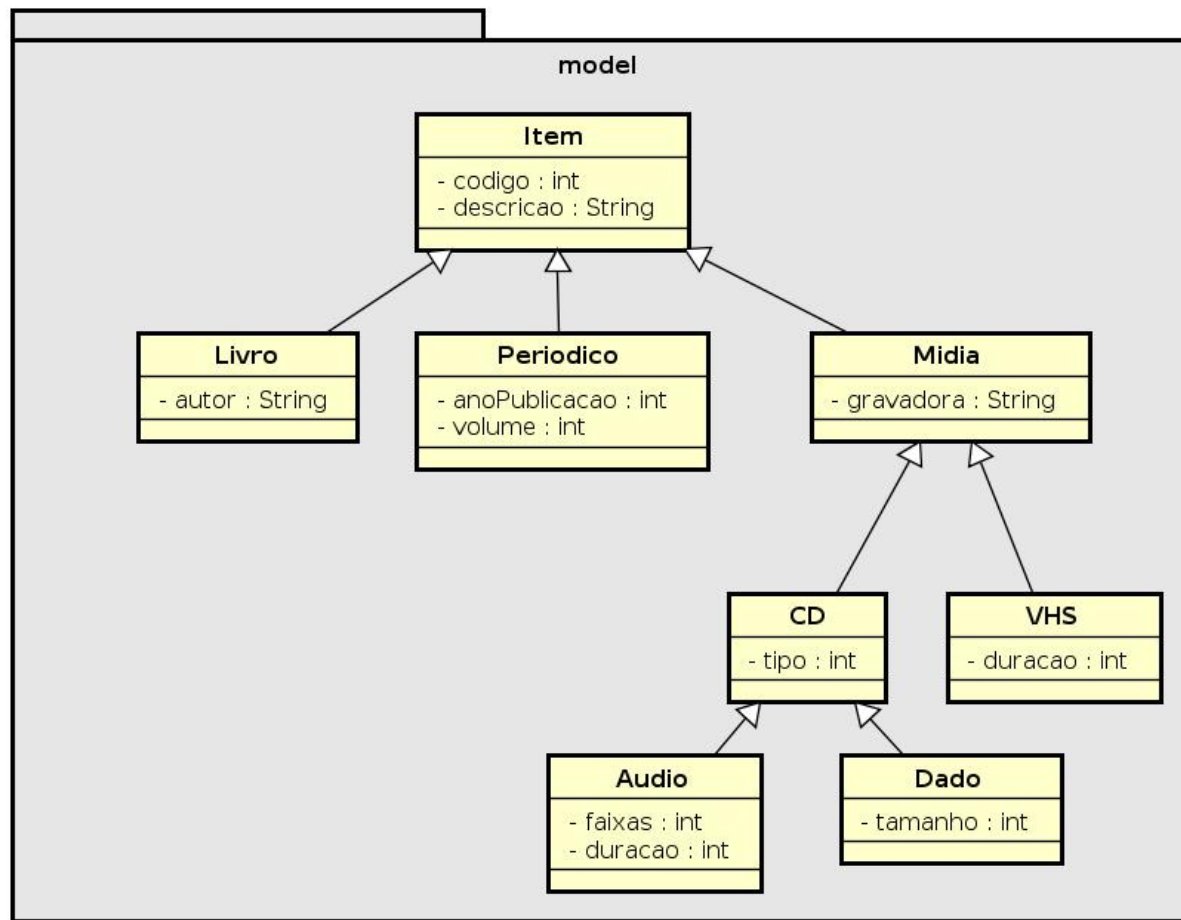


Modelo adequado, não está!



É possível observar a repetição de código, atributos, entre classes. Isso não é adequado. Para resolvermos isso vamos criar uma classe genérica e usar herança.





Organizou-se o modelo de forma que a classe genérica **Item** represente todos os itens da biblioteca.

Perceba que não existe mais a repetição de atributos.

Além disso existem outras vantagens que serão vistas daqui a pouco.

Agora estamos prontos para dominar o código!



```

public class Item {

    private int id;
    private String descricao;

    public Item(int id, String descricao) {
        setId(id);
        setDescricao(descricao);
    }
    @Override
    public String toString(){
        StringBuffer sb = new StringBuffer();
        sb.append("ID: ");
        sb.append(id);
        sb.append("\t");
        sb.append("Descrição: ");
        sb.append(descricao);
        return sb.toString();
    }

```

```

    public int getId() {                return id;        }

    public void setId(int id) {          this.id = id >= 0? id : 0;    }

```

```

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao.toUpperCase();
    }

    public String getDescritivo(){
        StringBuilder sb = new StringBuilder();
        sb.append("Extrato do Objeto\n");
        sb.append("-----\n");
        sb.append("Objeto da classe: ");
        sb.append(getClass().getName());
        sb.append("\n\t");
        sb.append(toString());
        sb.append("\n-----\n");
        return sb.toString();
    }
}

```

```
public class Livro extends Item {
```

```
    private String autor;
```

```
    public Livro(int id, String descricao, String autor) {  
        super(id, descricao);  
        this.autor = autor;  
    }
```

```
    public String getAutor() { return autor; }  
    public void setAutor(String autor) {  
        this.autor = autor.toUpperCase();  
    }
```

```
    @Override  
    public String toString() {  
        StringBuffer sb = new StringBuffer();  
        sb.append(super.toString());  
        sb.append("\n");  
        sb.append("Autor do livro: ");  
        sb.append(autor);  
        return sb.toString();  
    }  
}
```

Observe que Livro é um Item e por isso possui todas as características de Item e as de Livro.



```
public class Periodico extends Item {  
    private int anoPublicacao;  
    private int volume;  
    public Periodico(int id, String descricao, int anoPublicacao, int volume) {  
        super(id, descricao);  
        setAnoPublicacao(anoPublicacao);  
        setVolume(volume);  
    }  
    public int getAnoPublicacao() { return anoPublicacao; }  
    public void setAnoPublicacao(int anoPublicacao) {  
        this.anoPublicacao = anoPublicacao > 1900 ? anoPublicacao:1900;  
    }  
    public int getVolume() { return volume; }  
    public void setVolume(int volume) { this.volume = volume>0?volume:1; }  
    @Override  
    public String toString() {  
        StringBuilder sb = new StringBuilder();  
        sb.append(super.toString());  
        sb.append("\nAno de publicação: ");  
        sb.append(anoPublicacao);  
        sb.append("\tVolume: ");  
        sb.append(volume);  
        return sb.toString();  
    }  
}
```

```
public class Midia extends Item {  
    private String gravadora;  
  
    public Midia(int id, String descricao, String gravadora) {  
        super(id, descricao);  
        this.gravadora = gravadora.toUpperCase();  
    }  
  
    @Override  
    public String toString(){  
        StringBuilder sb = new StringBuilder();  
        sb.append(super.toString());  
        sb.append("\nGravadora: ");  
        sb.append(gravadora);  
        return sb.toString();  
    }  
  
    public String getGravadora() {  
        return gravadora;  
    }  
  
    public void setGravadora(String gravadora) {  
        this.gravadora = gravadora.toUpperCase();  
    }  
}
```

```

public class CD extends Midia{
    public static final int TIPO_CDROM = 1;
    public static final int TIPO_DVD = 2;

    private int tipo;

    public CD(int id, String descricao, String gravadora, int tipo) {
        super(id, descricao, gravadora);
        this.tipo = tipo;
    }

    public int getTipo() { return tipo; }

    public void setTipo(int tipo) {
        if(tipo == TIPO_CDROM ||
           tipo == TIPO_DVD)
            this.tipo = tipo;
        else
            this.tipo = TIPO_CDROM;
    }
}

```

CD é uma Midia que por sua vez é um Item, assim, CD possui todas as características de Item e Midia.

Isso é Hierarquia de classes.

```

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(super.toString());
    if(tipo == TIPO_CDROM)
        sb.append("\nTipo: CDROM");
    else
        sb.append("\nTipo: DVD");
    return sb.toString();
}
}

```

```
public class Audio extends CD {
```

```
    private int faixas;  
    private int duracao;
```

```
    public Audio(int id, String descricao, String gravadora,  
                int tipo, int faixas, int duracao) {  
        super(id, descricao, gravadora, tipo);  
        setFaixas(faixas);  
        setDuracao(duracao);  
    }
```

```
    public int getFaixas() {  
        return faixas; }  
    public void setFaixas(int faixas) {  
        this.faixas = faixas>=1?faixas:1;  
    }
```

```
    public int getDuracao() {  
        return duracao; }  
    public void setDuracao(int duracao) {  
        this.duracao = duracao>=1?duracao:1;  
    }
```

```
@Override  
    public String toString() {  
        StringBuilder sb = new StringBuilder();  
        sb.append(super.toString());  
        sb.append("\n Faixas: ");  
        sb.append(faixas);  
        sb.append("\tDuração: ");  
        sb.append(duracao);  
        sb.append(" minutos");  
        return sb.toString();  
    }  
}
```



```

public class Dado extends CD {
    private int tamanho;
    public Dado(int id, String descricao, String gravadora, int tipo, int tamanho) {
        super(id, descricao, gravadora, tipo);
        setTamanho(tamanho);
    }

```

@Override

```

public String imprimeCapa() {
    StringBuilder sb = new
        StringBuilder();
    sb.append("Título: ");
    sb.append(getDescricao());
    sb.append("\nGravadora: ");
    sb.append(getGravadora());
    sb.append("\nTamanho: ");
    sb.append(tamanho);
    sb.append(" MB");
    return sb.toString();
}

```

```

public int getTamanho() {
    return tamanho;
}

```

```

public void setTamanho(int tamanho) {
    this.tamanho = tamanho>=1?tamanho:1;
}

```

@Override

```

public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(super.toString());
    sb.append("\nTamanho: ");
    sb.append(tamanho);
    sb.append(" MB");
    return sb.toString();
}
}

```

```
public class VHS extends Midia{
    private int duracao;

    public VHS(int id, String descricao, String gravadora, int duracao) {
        super(id, descricao, gravadora);
        setDuracao(duracao);
    }

    public int getDuracao() {    return duracao;    }

    public void setDuracao(int duracao) {
        this.duracao = duracao>=1?duracao:1;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append(super.toString());
        sb.append("\nDuração: ");
        sb.append(duracao);
        sb.append(" minutos.");
        return sb.toString();
    }
}
```

# Problema - Biblioteca

- Observe que nosso sistema possui vários tipos de objetos para gerenciar.

A solução é utilizar  
polimorfismo!



Um array para cada tipo  
resolve?



# Problema - Biblioteca

---

- Com o polimorfismo é possível armazenar um objeto em outro.
  - O polimorfismo de objetos permite que um objeto seja tratado como se fosse outro.
- No exemplo da Biblioteca será criado um **array** de Itens (classe mais genérica) para armazenar todos os objetos que forem filhos de Item.
- Nesse array poderemos armazenar Livros, Periódicos, CD de Audio, CD de Dados, VHS.

```
public class Biblioteca {  
    private static final int MAXIMO = 100;  
    private Item[] itens;  
    private int maximoItens;  
    private int itensCadastrados;  
  
    public Biblioteca(int maximoItens) {  
        setMaximoItens(maximoItens);  
        this.itensCadastrados = 0;  
        itens = new Item[maximoItens];  
    }  
  
    public Biblioteca() {  
        maximoItens = MAXIMO;  
        itensCadastrados = 0;  
        itens = new Item[maximoItens];  
    }  
  
    public int getMaximoItens() {  
        return maximoItens;  
    }  
  
    private void setMaximoItens(int maximoItens) {  
        this.maximoItens = maximoItens >= 1 ? maximoItens : 1;  
    } //...
```

Como todos nossos objetos são filhos de **Item**, podemos ter apenas um array para gerenciar todos os objetos.

Esse é um dos usos do **Polimorfismo** como **Compatibilidade de Tipos**.

```
private boolean novoItem(Item item){  
    boolean deuCerto = false;  
    if(!isFull() && item != null){  
        itens[itensCadastrados] = item;  
        itensCadastrados += 1;  
        deuCerto = true;  
    }  
    return deuCerto;  
}
```

O método que faz o cadastro recebe qualquer subclasse de Item e insere no **array** de Itens.

```
public boolean isFull(){  
    return itensCadastrados == maximoItens;  
}
```

Os métodos de cadastro apenas definem a “interface” para que o usuário insira objetos na biblioteca.

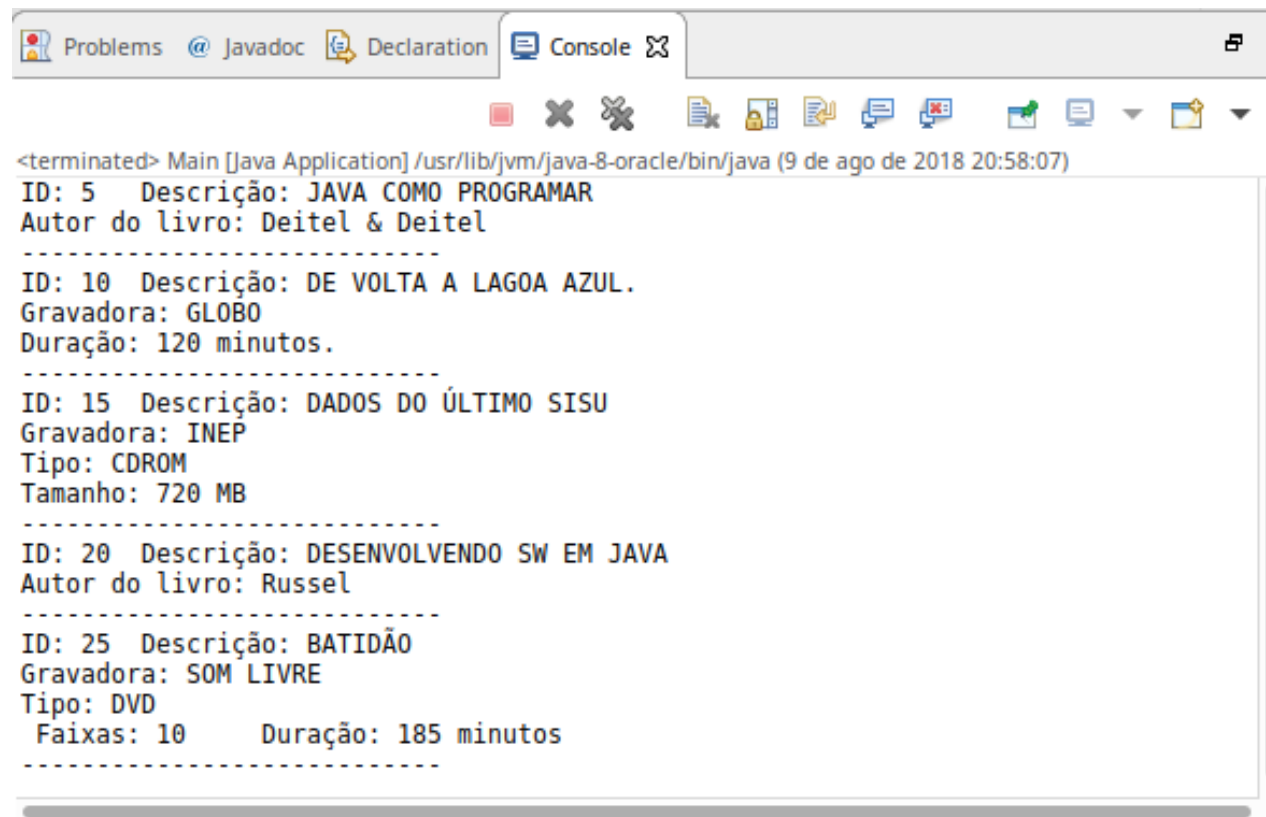
Cada objeto assume a “forma” de um Item.

```
public boolean cadastraLivro(Livro livro){    return novoItem(livro);    }  
  
public boolean cadastraPeriodico(Periodico periodico){  
    return novoItem(periodico);  
}  
  
public boolean cadastraAudio(Audio audio){    return novoItem(audio);    }  
  
public boolean cadastraDado(Dado dado){    return novoItem(dado);    }  
  
public boolean cadastraVHS(VHS vhs){    return novoItem(vhs);    }
```

# O Main

```
public class Main {  
    public static void main(String[] args) {  
        Biblioteca library = new Biblioteca(5);  
        library.cadastraLivro(new Livro(5, "Java como programar", "Deitel & Deitel"));  
        library.cadastraVHS(new VHS(10, "De volta a lagoa azul.", "globo", 120));  
        library.cadastraDado(new Dado(15, "Dados do último SISU", "INEP",  
            Dado.TIPO_CDRM, 720));  
        library.cadastraLivro(new Livro(20, "Desenvolvendo SW em Java", "Russel"));  
        library.cadastraAudio(new Audio(25, "Batidão", "Som Livre",  
            Audio.TIPO_DVD, 10, 185));  
        int i=0;  
        Item objeto;  
        do{  
            objeto = library.getItemAt(i);  
            if(objeto != null){  
                System.out.println(objeto.toString());  
                System.out.println("-----");  
            }  
            i++;  
        }while (objeto != null);  
    }  
}
```

# Execução



The screenshot shows an IDE console window with the following tabs: Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of a Java application. The output is a list of book records, each separated by a dashed line. The records are as follows:

```
<terminated> Main [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (9 de ago de 2018 20:58:07)
ID: 5   Descrição: JAVA COMO PROGRAMAR
Autor do livro: Deitel & Deitel
-----
ID: 10  Descrição: DE VOLTA A LAGOA AZUL.
Gravadora: GLOBO
Duração: 120 minutos.
-----
ID: 15  Descrição: DADOS DO ÚLTIMO SISU
Gravadora: INEP
Tipo: CDR0M
Tamanho: 720 MB
-----
ID: 20  Descrição: DESENVOLVENDO SW EM JAVA
Autor do livro: Russel
-----
ID: 25  Descrição: BATIDÃO
Gravadora: SOM LIVRE
Tipo: DVD
Faixas: 10      Duração: 185 minutos
-----
```



# O Cliente ....

---

- Cliente quer saber quantos livros estão cadastrados na biblioteca.
  - Nem todos os itens do array são da classe Livro.
  - O que fazer?
    - Contar!!!

# instanceof

O operado **instanceof** compara se um objeto é uma instancia daquela classe ou não.

## Classe Biblioteca

```
public int quantidadeLivros(){
    int i, conta;
    for(i=0, conta=0; i<itensCadastrados; i++){
        if(itens[i] instanceof Livro){
            conta++;
        }
    }
    return conta;
}
```

```
<terminated> Main [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (9 de ago de 2018 21:06:43)
ID: 5   Descrição: JAVA COMO PROGRAMAR
Autor do livro: Deitel & Deitel
-----
ID: 10  Descrição: DE VOLTA A LAGOA AZUL.
Gravadora: GLOBO
Duração: 120 minutos.
-----
ID: 15  Descrição: DADOS DO ÚLTIMO SISU
Gravadora: INEP
Tipo: CDROM
Tamanho: 720 MB
-----
ID: 20  Descrição: DESENVOLVENDO SW EM JAVA
Autor do livro: Russel
-----
ID: 25  Descrição: BATIDÃO
Gravadora: SOM LIVRE
Tipo: DVD
Faixas: 10      Duração: 185 minutos
-----
Total de livros: 2
```

# Polimorfismo

---

- Também é a troca de mensagens entre métodos de uma subclasse com sua superclasse e assim sucessivamente.

```

public class Item {

    private int id;
    private String descricao;

    //...
    public String getDescritivo(){
        StringBuilder sb = new StringBuilder();
        sb.append("Extrato do Objeto\n");
        sb.append("-----\n");
        sb.append("Objeto da classe: ");
        sb.append(getClass().getName());
        sb.append("\n\t");

        sb.append(toString());

        sb.append("\n-----\n");
        return sb.toString();
    }
}

```

```

<terminated> Main [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (9 de
Extrato do Objeto
-----
Objeto da classe: model.Livro
ID: 5 Descrição: JAVA COMO PROGRAMAR
Autor do livro: Deitel & Deitel
-----

Extrato do Objeto
-----
Objeto da classe: model.VHS
ID: 10 Descrição: DE VOLTA A LAGOA AZUL.
Gravadora: GLOBO
Duração: 120 minutos.
-----

Extrato do Objeto
-----
Objeto da classe: model.Dado
ID: 15 Descrição: DADOS DO ÚLTIMO SISU
Gravadora: INEP
Tipo: CDROM
Tamanho: 720 MB
-----

Extrato do Objeto
-----
Objeto da classe: model.Livro
ID: 20 Descrição: DESENVOLVENDO SW EM JAVA
Autor do livro: Russel
-----

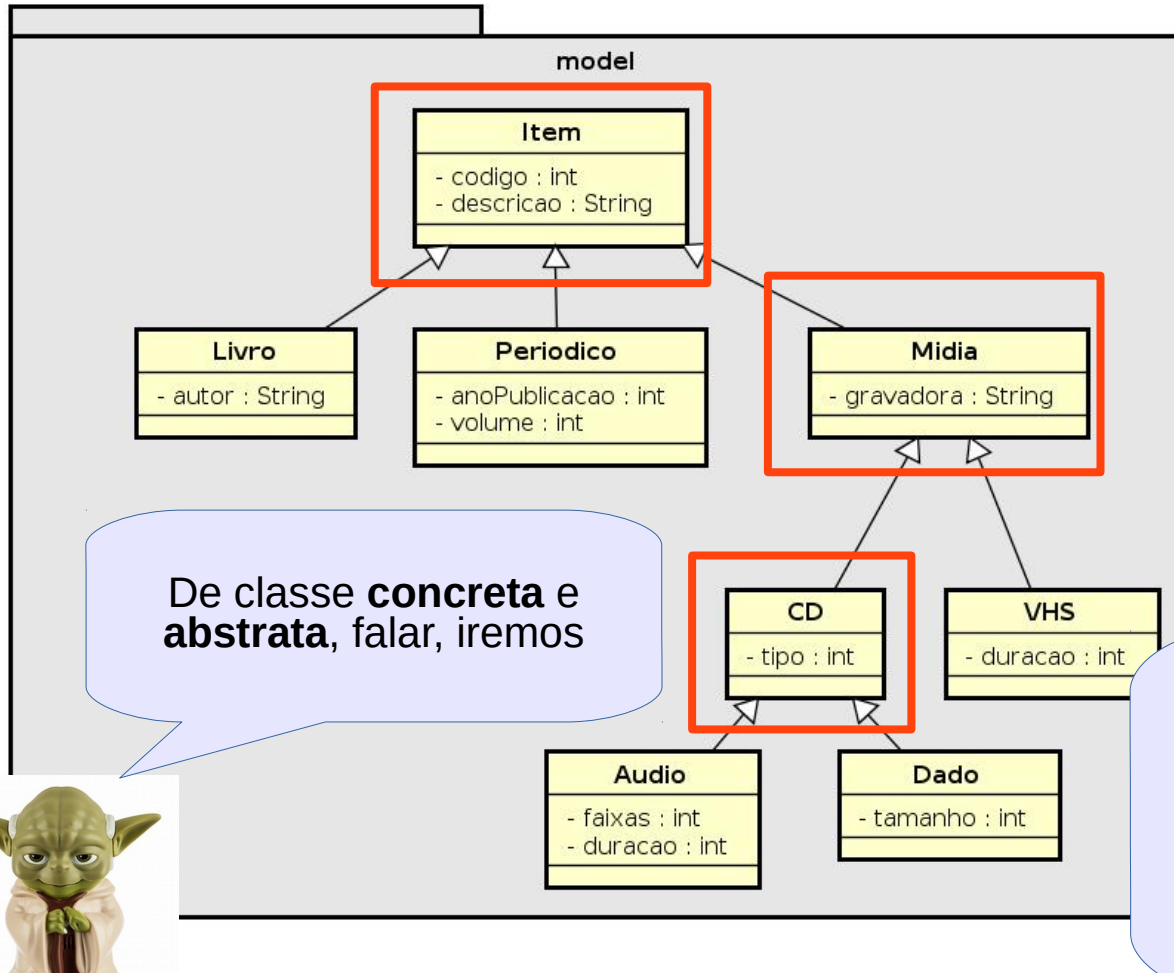
Extrato do Objeto
-----
Objeto da classe: model.Audio
ID: 25 Descrição: BATIDÃO
Gravadora: SOM LIVRE
Tipo: DVD
Faixas: 10 Duração: 185 minutos
-----

Total de livros: 2

```

Qual método toString() é chamado?

Lembrando que estamos implementando a classe Item que tem o método toString().



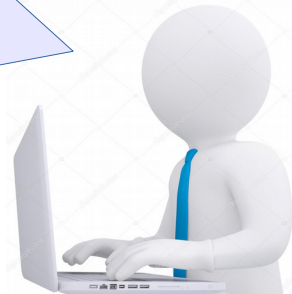
De classe **concreta** e **abstrata**, falar, iremos

Observe as classes

Nosso sistema  
instanciou alguns  
objeto dessas  
classes?

Tem sentido instanciar  
objetos dessas classes?

É possível ter uma Mídia  
em nosso sistema? Ou  
apenas VHS, CD de  
Dados ou Áudio?



# Classe Concreta x Classe Abstrata

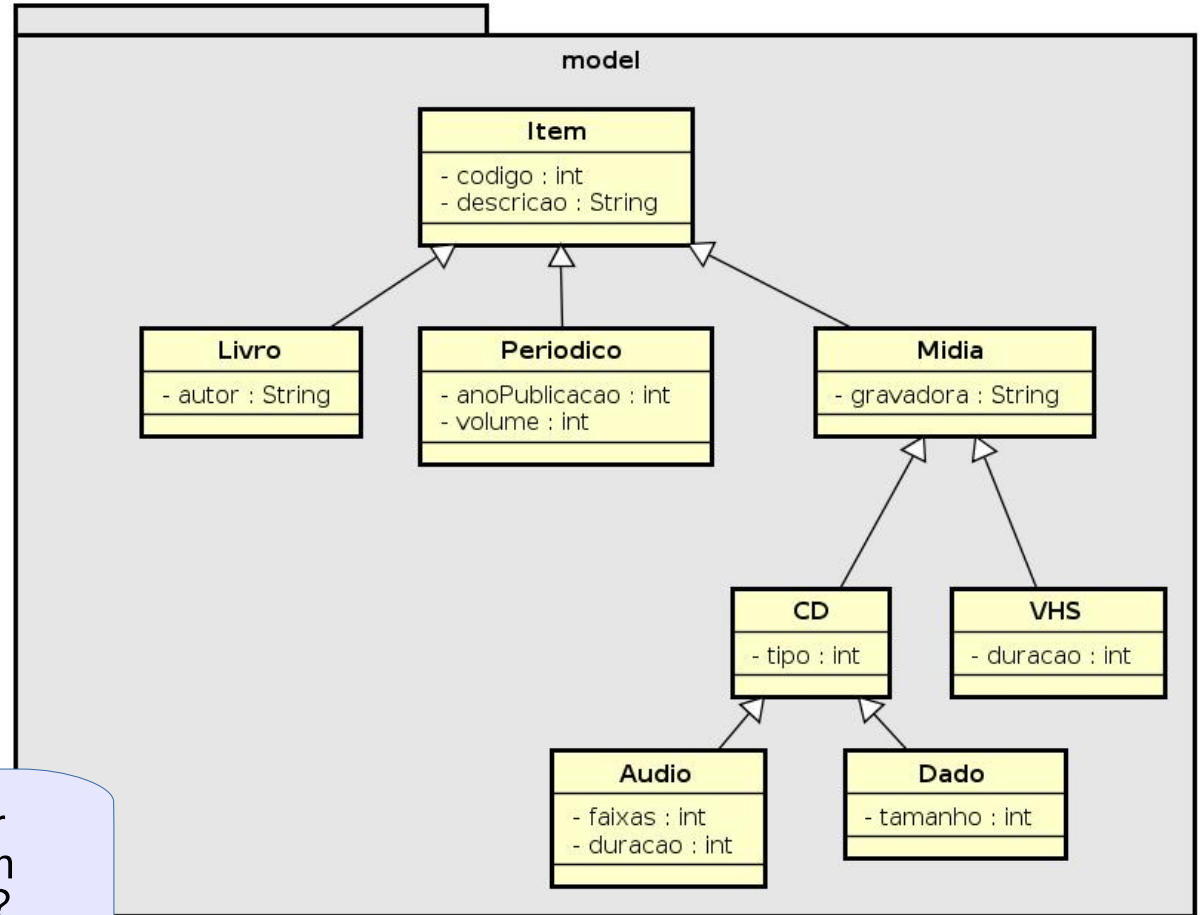
- Uma classe concreta permite a instância de um objeto daquela classe.
    - Na biblioteca temos instância de algumas classes concretas:
- ```
library.cadastraLivro(new Livro(5, "Java como programar", "Deitel & Deitel"));  
library.cadastraVHS(new VHS(10, "De volta a lagoa azul.", "globo", 120));  
library.cadastraDado(new Dado(15, "Dados do último SISU", "INEP", Dado.TIPO_CDROM, 720));  
library.cadastraLivro(new Livro(20, "Desenvolvendo SW em Java", "Russel"));  
library.cadastraAudio(new Audio(25, "Batidão", "Som Livre", Audio.TIPO_DVD, 10, 185));
```
- Classe abstrata não permite que objetos sejam instanciados a partir dela, apenas de suas subclasses. Assim, uma Classe Abstrata é um modelo a ser seguido pelas subclasses.
    - Uma classe abstrata não precisa ser implementada totalmente, mas pode “forçar” que suas subclasses implementem comportamentos específicos.

Observe que nunca iremos instanciar um Item, mas os filhos de Item. Isso nos indica que Item pode ser uma **Classe Abstrata**, assim como Mídia e CD.

Não faz sentido para nossa Biblioteca termos um CD que não seja um CD de Audio ou um CD de Dados.

Como transformar nossas classes em classes abstratas?

pkg



```
public abstract class Item { ... }  
public abstract class Midia extends Item { ... }  
public abstract class CD extends Midia { ... }
```

Basta inserir a palavra “**abstract**” antes da classe para tornar nossa classe abstrata.



Legal professor, ficou “bunitinho”.

Mas o que mudou?

Vá no método main() e instancie um objeto da classe Item, usando o construtor de Item.

```
Item obj = new Item(1, “Bla”);
```

É possível instanciar uma classe abstrata desde que ela seja implementada por você, em tempo de execução. Veremos isso no futuro.

Por hora, temos que classes abstratas são modelos para classes concretas. Mas não para por aqui. Temos que falar de métodos abstratos.

Uma classe abstrata pode possuir métodos (assinaturas) abstratos e fazer com que os filhos o implementem. Vejamos.





```
public abstract class CD extends Midia{
    public static final int TIPO_CDRom = 1;
    public static final int TIPO_DVD = 2;
```

```
private int tipo;
```

```
public abstract String imprimeCapa();
```

```
public CD(int id, String descricao, String gravadora, int tipo) {
    super(id, descricao, gravadora);
    this.tipo = tipo;
}
```

```
public class Audio extends CD {
    private int faixas;
    private int duracao;
    @Override
    public String imprimeCapa() {
        StringBuilder sb = new StringBuilder();
        sb.append("Título: ");
        sb.append(getDescricao());
        sb.append("\nGravadora: ");
        sb.append(getGravadora());
        sb.append("\nFaixas: ");
        sb.append(faixas);
        return sb.toString();
    }
}
```

O método abstrato `imprimeCapa()` não possui implementação em `CD`, contudo, as subclasses de `CD` são **obrigadas** a implementar esse método.

```
public class Dado extends CD {
    private int tamanho;
```

```
@Override
public String imprimeCapa() {
    StringBuilder sb = new StringBuilder();
    sb.append("Título: ");
    sb.append(getDescricao());
    sb.append("\nGravadora: ");
    sb.append(getGravadora());
    sb.append("\nTamanho: ");
    sb.append(tamanho);
    sb.append(" MB");
    return sb.toString();
}
```