

# *Paradigmas*

*Módulo Básico*





# MÉTODOS

- **Métodos:**
  - Em uma *instituição financeira*, é possível *realizar diversas operações* em uma *conta*, como:
    - *Depósito*
    - *Saque*
    - *Transferência*
    - *Consultas*
    - *Etc.*
  - *Essas operações* podem *modificar* ou *apenas acessar* os *valores* dos *atributos* dos *objetos* que *representam* as *contas*

# Métodos

- **Métodos:**

- Essas **operações** são **realizadas** em **métodos definidos** na **própria classe Conta**
- **Exemplo:**
  - Para **realizar** a **operação de depósito**, podemos **acrescentar o método abaixo** na **classe Conta**

```
void deposita(double valor){  
    // implementação  
}
```

*Código Java: Definindo um método*

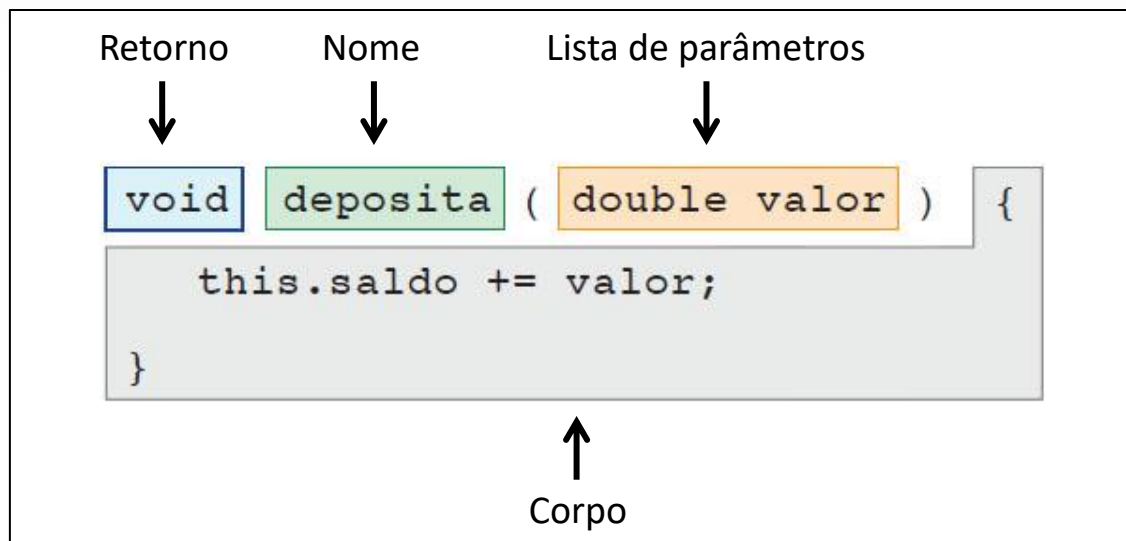
- **Métodos:**
  - **Podemos fragmentar um método em quatro partes:**
    1. **Nome:** é utilizado para **chamar** (invocar) o **método** (em *Java*, é uma *boa prática definir* os *nomes* dos *métodos* com a *primeira letra minúscula*)
    2. **Lista de Parâmetros:** *define* os *valores* que o *método* *deve receber* (*métodos* que *não recebem nenhum valor* possuem uma *lista de parâmetros vazia*)
    3. **Corpo:** *define* o que *acontecerá* quando o *método* for *chamado/invocado*

# Métodos

- **Métodos:**

- **Podemos fragmentar um método em quatro partes:**

- 4. **Retorno:** refere-se a **resposta** que será **devolvida** ao **final** do **processamento do método** (**quando** um **método não devolve nenhuma resposta**, ele deve ser **assinado** com a **palavra reservada void**)



- **Métodos:**
  - *Para realizar um depósito, torna-se necessário chamar o método **deposita()**, por meio da referência do objeto que representa a conta que terá o dinheiro creditado*

```
class TestaConta {  
    public static void main(String[] args) {  
        // Referência de um objeto  
        Conta c = new Conta();  
  
        // Invocando o método deposita()  
        c.deposita(1000);  
    }  
}
```

*Código Java: Invocando o método deposita*

- **Métodos:**

- *Para realizar um depósito, torna-se necessário chamar o método **deposita()**, por meio da referência do objeto que representa a conta que terá o dinheiro creditado*

Normalmente, os métodos **acessam** ou **alteram** os valores armazenados nos atributos dos objetos.

Por exemplo, na execução do método **deposita()**, é necessário alterar o valor do atributo **saldo** do objeto que foi escolhido para realizar a operação.



- **Métodos:**
  - **Dentro de um método**, para **acessar** os **atributos** do **objeto** que está **processando** o **método**, **torna-se necessário utilizar** a **palavra reservada** **this**

```
void deposita(double valor) {  
    this.saldo += valor;  
}
```

*Código Java: Utilizando o this para acessar e/ou modificar um atributo*

- **Métodos:**
  - *Suponha a necessidade de implementar um método para realizar a operação que consulta o saldo disponível das contas*
  - *O saldo disponível é igual a soma do saldo e do limite*
  - *Esse método deve somar os atributos **saldo** e **limite** e devolver o resultado*
  - *Esse método **não** deve receber nenhum valor, pois **todas** as informações necessárias para realizar a operação estão nos atributos dos objetos que representam as contas*

# Métodos

- **Métodos:**
  - **Exemplo:**

```
double consultaSaldoDisponivel() {  
    .....  
    return this.saldo + this.limite;  
}
```

*Código Java: Método com retorno double*

```
class TestaConta {  
    public static void main(String[] args) {  
        // Referência de um objeto  
        Conta c = new Conta();  
  
        // Invocando o método deposita()  
        c.deposita(1000);  
  
        // Atribuindo a resposta de um método em uma variável  
        double saldoDisponivel = c.consultaSaldoDisponivel();  
  
        System.out.println("Saldo disponível: " + saldoDisponivel);  
    }  
}
```

*Código Java: Armazenando a resposta de um método*

# Sobrecarga

- ***Sobrecarga:***

- *Os clientes das instituições financeiras normalmente consultam periodicamente as informações relativas às suas contas*
- *Essas informações são obtidas através de extratos*
- *No sistema computacional das instituições financeiras, os extratos podem ser gerados por métodos da classe **Conta***



- **Sobrecarga:**
  - O método **imprimeExtrato()** recebe a *quantidade* de *dias* que deve ser **considerada** para **gerar** o **extrato da conta**
  - Se esse **método** receber o **valor 30** então ele **deve gerar** um **extrato** com as **movimentações** dos **últimos 30 dias**
  - **Normalmente**, **extratos** dos **últimos 15 dias** **atendem** as **necessidades** dos **clientes**
  - Dessa forma, seria **oportuno adicionarmos** um **método** na **classe Conta** para **gerar extratos** com essa **quantidade fixa** de **dias**

# Sobrecarga

- ***Sobrecarga:***
  - ***Exemplo:***
    - ***Implementando sobrecarga de métodos***

```
class Conta {  
    public double saldo;  
    public double limite;  
  
    public void imprimeExtrato(int dias){  
        // extrato  
    }  
  
    public void imprimeExtrato(){  
        // extrato dos últimos 15 dias  
    }  
}
```

*Código Java: Conta.java*

# Sobrecarga

- ***Sobrecarga:***
  - *Os dois métodos possuem o mesmo nome e lista de parâmetros distintos*
  - Quando ***dois*** ou ***mais métodos*** são ***definidos*** na ***mesma classe*** com o ***mesmo nome***, ***ocorre*** uma ***sobrecarga*** de ***métodos***
  - ***Uma sobrecarga de métodos*** apenas é ***válida*** se as ***listas de parâmetros*** dos ***métodos*** são ***distintas entre si***



# Sobrecarga

- **Sobrecarga:**

- **Exemplo:**

- **No caso dos *dois métodos* que *geram extratos*, para *evitar repetição de código*, podemos fazer um *método chamar o outro***

```
class Conta {  
    public double saldo;  
    public double limite;  
  
    public void imprimeExtrato(int dias){  
        // extrato  
    }  
  
    public void imprimeExtrato(){  
        this.imprimeExtrato(15);  
    }  
}
```

*Código Java: Conta.java*





# CONSTRUTORES

# Construtores

- **Construtores:**
  - *No domínio de uma instituição financeira:*
    - *Todo cartão de crédito deve possuir um número*
    - *Toda agência deve possuir um número*
    - *Toda conta deve estar associada a uma agência*



# Construtores

- **Exemplo:**
  - *Posteriormente a criação de um objeto para representar um cartão de crédito, poderíamos definir um valor para o atributo **numero***
  - *De maneira similar, podemos definir um número para um objeto da classe **Agencia** e uma agência para um objeto da classe **Conta***

# Construtores

- **Exemplo:**

- ***Definindo um número para um cartão de crédito***

```
CartaoDeCredito cdc = new CartaoDeCredito();  
cdc.numero = 12345;
```

- ***Definindo um número para uma agência***

```
Agencia a = new Agencia();  
a.numero = 0865;
```

# Construtores

- **Exemplo:**

- ***Definindo uma agência para uma conta***

```
Conta c = new Conta();  
c.agencia = a;
```



- **Observação:**

- ***Definir os valores dos atributos obrigatórios de um objeto logo após a criação dele resolveria as restrições do sistema bancário***
    - ***Nada garante que todos os desenvolvedores sempre lembrem de inicializar esses valores***
    - ***Para evitar tal risco, podemos utilizar construtores***

# Construtores

- **Construtor:**

- *Permite que um determinado trecho de código seja executado toda vez que um objeto é criado, ou seja, toda vez que o operador **new** é chamado*
- *Análogo aos métodos, os construtores podem receber parâmetros, todavia, **não** devolvem nenhum tipo de resposta*
- *Em Java, um construtor deve ter o mesmo nome da classe na qual ele foi definido*



# Construtores

- **Exemplo:**
  - **Construtor aplicado a classe CartaoDeCredito**

```
public class CartaoDeCredito {  
    public int numero;  
    public String dataDeValidade;  
  
    public CartaoDeCredito(int numero) {  
        this.numero = numero;  
    }  
}
```

Código Java: CartaoDeCredito.java

- **Construtor aplicado a classe Agencia**

```
public class Agencia {  
    public int numero;  
  
    public Agencia(int numero) {  
        this.numero = numero;  
    }  
}
```

Código Java: Agencia.java

# Construtores

- **Exemplo:**
  - **Construtor aplicado a classe Conta**

```
public class Conta {  
    public Agencia agencia;  
  
    public Conta(Agencia agencia){  
        this.agencia = agencia;  
    }  
}
```

*Código Java: Conta.java*



# Construtores

- **Exemplo:**

- *Na criação do objeto com o comando **new**, os argumentos passados devem ser compatíveis com a lista de parâmetros de algum construtor definido na classe que está sendo instanciada*
- *Caso contrário, um **erro** de compilação ocorrerá para avisar o desenvolvedor dos valores obrigatórios que devem ser passados para criar um objeto*

# Construtores

- **Exemplo:**
  - **Utilização correta dos construtores**

```
public class TestandoConstrutor {  
    // Passando corretamente os parâmetros para os construtores  
    CartaoDeCredito cdc = new CartaoDeCredito(1111);  
  
    Agencia a = new Agencia(865);  
  
    Conta c = new Conta(a);  
}
```

*Código Java: TestandoConstrutor.java*



# Construtores

- **Exemplo:**
  - **Utilização inadequada dos construtores**

```
public class TestandoConstrutor {  
    // Erro de compilação  
    CartaoDeCredito cdc = new CartaoDeCredito();  
  
    // Erro de compilação  
    Agencia a = new Agencia();  
  
    // Erro de compilação  
    Conta c = new Conta();  
}
```

Código Java: TestandoConstrutor.java



# Construtores

- **Construtor Padrão:**
  - *Toda vez que um objeto é criado, um construtor da classe correspondente deve ser chamado*
  - *Mesmo quando nenhum construtor for definido explicitamente, há um construtor padrão que será inserido pelo próprio compilador*
  - *O construtor padrão **não** recebe parâmetros e será inserido sempre que o desenvolvedor **não** definir pelo menos um construtor explicitamente*

- ***Construtor Padrão:***

```
class Conta{  
  
}
```

*Código Java: Conta.java*

```
// Chamando o construtor padrão  
Conta c = new Conta();
```

*Código Java: Utilizando o construtor padrão*

# Construtores

- **Construtor Padrão:**

```
class Conta{
```

O construtor padrão só será inserido pelo compilador se nenhum construtor for definido no código fonte.

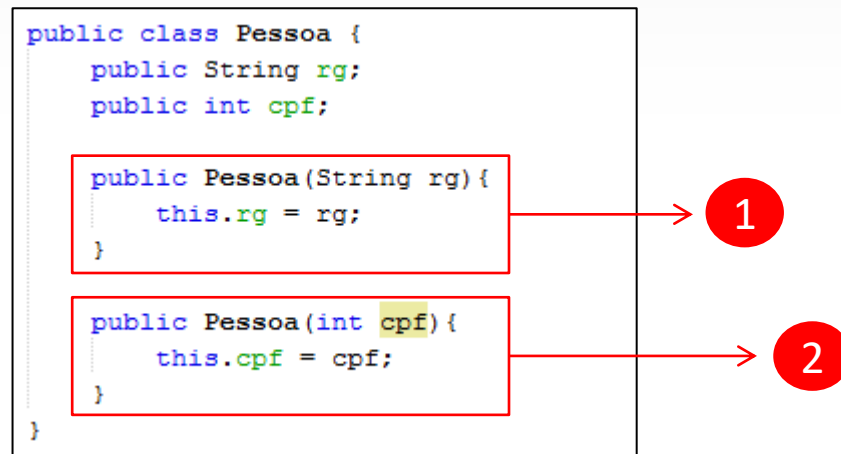
Dessa forma, se você adicionar um construtor com parâmetros então não poderá utilizar o comando new sem passar argumentos, pois um erro de compilação ocorrerá.

*Código Java: Utilizando o construtor padrão*

# Construtores

- ***Sobrecarga de Construtores:***
  - *O conceito de sobrecarga de métodos pode ser aplicado para construtores*
  - **Exemplo:**
    - ***Definindo diversos construtores para a classe **Pessoa*****

```
public class Pessoa {  
    public String rg;  
    public int cpf;  
  
    public Pessoa(String rg) {  
        this.rg = rg;  
    }  
  
    public Pessoa(int cpf) {  
        this.cpf = cpf;  
    }  
}
```



Código Java: Pessoa.java

# Construtores

- ***Sobrecarga de Construtores:***

- **Exemplo:**

- ***Quando dois construtores são definidos, há duas opções no momento de utilizar o comando `new`***

```
public class TestaPessoa {  
  
    // Chamando o primeiro construtor  
    Pessoa p1 = new Pessoa("123456Y");  
  
    // Chamando o segundo construtor  
    Pessoa p2 = new Pessoa(123456789);  
  
}
```

*Código Java: Utilizando dois construtores diferentes*



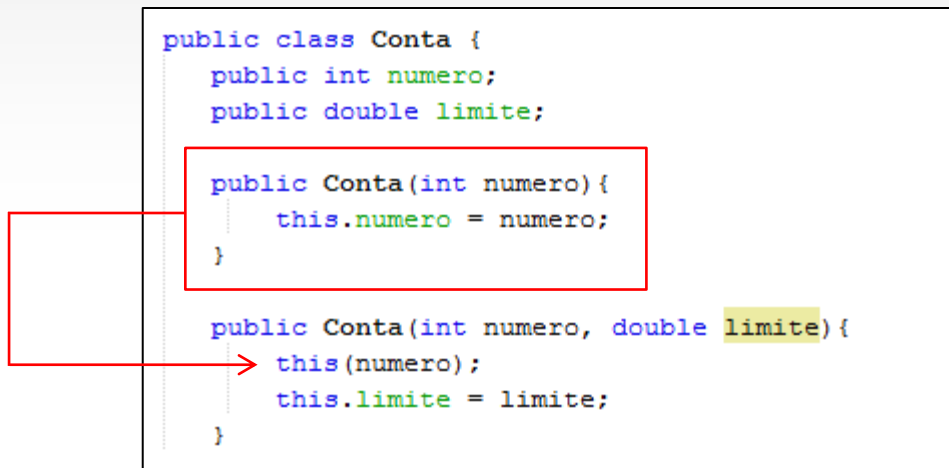
# Construtores

- **Construtores chamando Construtores:**

- Exemplo:

- É permitido encadear construtores (*análogo* o que já é *feito* com *os métodos*)

```
public class Conta {  
    public int numero;  
    public double limite;  
  
    public Conta(int numero){  
        this.numero = numero;  
    }  
  
    public Conta(int numero, double limite){  
        this(numero);  
        this.limite = limite;  
    }  
}
```



*Código Java: Conta.java*



# REFERÊNCIAS

# Referências

- ***Referências como Parâmetro***
  - *Similar a passagem de valores primitivos como parâmetro para um método ou construtor*
  - *É possível passar valores **não** primitivos (**referências**)*



- ***Referências como Parâmetro***

- ***Exemplo (1):***

- ***Considere*** um ***método*** na ***classe*** **Conta** que ***implemente*** a ***lógica*** de ***transferência*** de ***valores entre contas***
    - Esse ***método*** ***deve receber como argumento***, além do ***valor*** a ***ser transferido***, a ***referência*** da ***conta*** que ***receberá*** o ***dinheiro***

```
void transfere( Conta destino, double valor ) {  
    this.saldo -= valor;  
    destino.saldo += valor;  
}
```

*Código Java: Método transfere()*

- *Referências como Parâmetro*

- Exemplo (2):

- Quando invocarmos o método *transfere()*, devemos possuir duas referências de contas: *uma para chamar o método* e *outra para passar como parâmetro*

```
Conta origem = new Conta();
```

```
origem.saldo = 1000;
```

```
Conta destino = new Conta();
```

```
origem.transfere (destino, 500);
```

*Código Java: Invocando o método transfere()*

# Referências

- **Referências como Parâmetro**

- **Exemplo (2):**

- Quando invocarmos o método **transfere()**, devemos possuir duas referências de contas: **uma para chamar o método** e **outra para passar como parâmetro**

Quando a variável **destino** é passada como parâmetro, somente a **referência** armazenada nessa variável é enviada para o método **transfere()** e não o objeto em si.

Em outras palavras, somente o “**endereço**” para a conta que receberá o valor da transferência é enviado para o método **transfere()**.

Código Java: Invocando o método **transfere()**



# ENCAPSULAMENTO

# Encapsulamento

- ***Atributos Privados:***

- *Em nosso sistema bancário hipotético, cada objeto da classe **Funcionario** possui um atributo para armazenar o salário do funcionário que ele representa*

- ***Exemplo:***

```
class Funcionario {  
    double salario;  
}
```



# Encapsulamento

- **Atributos Privados:**

- O atributo **salario** pode ser *acessado ou modificado* por *código escrito* em *qualquer classe* que *esteja* no *mesmo diretório* que a *classe* **Funcionario** (controle descentralizado)
- *Podemos obter* um *controle centralizado* tornando o atributo **salario** **privado** e *definindo métodos* para *implementar todas as lógicas* que *utilizam ou modificam* o *valor desse atributo*

# Encapsulamento

- ***Atributos Privados:***

- ***Exemplo:***

```
class Funcionario {  
    private double salario;  
  
    void aumentaSalario(double aumento){  
        // lógica para aumentar o salário  
    }  
}
```

- Um **atributo privado** apenas pode ser **acessado** ou **alterado** por **código escrito dentro da classe na qual ele foi definido**

# Encapsulamento

- ***Atributos Privados:***

- **Observação:**

- ***Definir todos os atributos como **privado** e utilizar métodos para implementar as lógicas de **acesso** e **alteração** é quase uma regra da orientação a objetos***
    - ***O objetivo é sempre ter um controle centralizado dos dados dos objetos, o que promoverá falicitadores no que se refere a manutenção do sistema e, eventuais detecção de erros***

# Encapsulamento

- ***Métodos Privados:***

- O *propósito* de *alguns métodos* pode ser o de *auxiliar outros métodos* da *mesma classe*
- *Em alguns casos, não é correto chamar* esses *métodos auxiliares* de *fora* da *sua classe diretamente*



# Encapsulamento

- ***Métodos Privados:***

- ***Exemplo (Parte 1):***

- O **método** `descontaTarifa()` é um **método auxiliar** dos **métodos** `deposita()` e `saca()`
    - ***Ele não deve ser invocado diretamente***, pois a **tarifa apenas deve ser descontada** quando **ocorrer** um **depósito** ou **um saque**

# Encapsulamento

- ***Métodos Privados:***

- ***Exemplo (Parte 2):***

```
class Conta {  
    private double saldo;  
  
    void deposita(double valor){  
        this.saldo += valor;  
        this.descontaTarifa();  
    }  
    void saca(double valor){  
        this.saldo -= valor;  
        this.descontaTarifa();  
    }  
    void descontaTarifa(){  
        this.saldo -= 0.1;  
    }  
}
```

# Encapsulamento

- ***Métodos Privados:***

- Para ***garantir*** que ***métodos auxiliares*** ***não*** sejam chamados por ***código escrito fora da classe*** na qual ***eles foram definidos***, podemos torná-los privados, adicionando o modificador ***private***

```
private void descontaTarifa(){  
    this.saldo -= 0.1;  
}
```

- ***Qualquer chamada*** ao ***método*** ***descontaTarifa()*** realizada ***fora*** da classe ***Conta*** gera ***um erro*** de ***compilação***

# Encapsulamento

- **Métodos Públicos:**

- *Os métodos que devem ser chamados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade **public***

```
class Conta {  
    private double saldo;  
  
    public void deposita(double valor){  
        this.saldo += valor;  
        this.descontaTarifa();  
    }  
    public void saca(double valor){  
        this.saldo -= valor;  
        this.descontaTarifa();  
    }  
    ...  
}
```



# Encapsulamento

- ***Implementação e Interface de Uso: - (Parte 1)***
  - ***Dentro de um sistema orientado a objetos, cada objeto realiza um conjunto de tarefas de acordo com as suas responsabilidade***
  - ***Por exemplo, os objetos da classe **Conta** realizam as operações de **saque**, **depósito**, **transferência** e **geração de extrato*****
  - ***Para descobrir o que um objeto pode fazer, basta olhar para as assinaturas dos métodos públicos definidos na classe desse objeto***

# Encapsulamento

- *Implementação e Interface de Uso: - (Parte 2)*
  - A assinatura do método é composta pelo seu **nome** e seus **parâmetros**
  - As assinaturas dos **métodos públicos** de um **objeto** formam a sua **interface de uso**

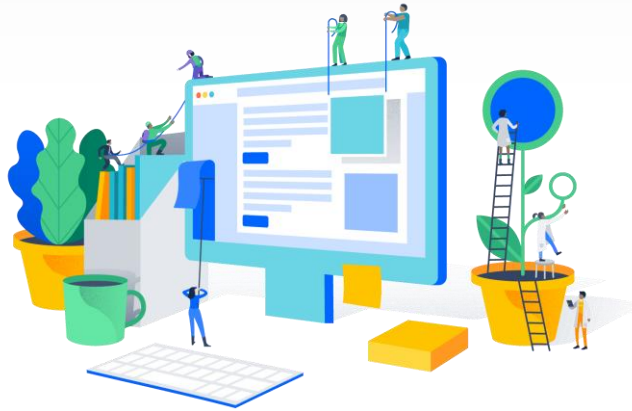
# Encapsulamento

- ***Por quê encapsular? (Parte 1)***
  - ***Encapsular significa esconder a implementação dos objetos***
  - ***O encapsulamento favorece principalmente dois aspectos do sistema: a **manutenção** e o **desenvolvimento*****
  - **Manutenção:**
    - ***Uma vez aplicado o encapsulamento, quando o funcionamento de um objeto deve ser alterado, em geral, basta modificar a classe do mesmo***



# Encapsulamento

- ***Por quê encapsular? (Parte 2)***
  - **Desenvolvimento:**
    - ***Uma vez aplicado o encapsulamento, conseguimos determinar precisamente as responsabilidades de cada classe da aplicação***



# Encapsulamento

- ***Celular - Escondendo a Complexidade (Parte 1)***
  - ***Atualmente, as pessoas estão acostumadas com os celulares***
  - Os ***botões***, a ***tela*** e os ***menus*** de ***um celular formam a interface de uso*** do mesmo
  - O ***usuário interage*** com ***esses aparelhos através*** dos ***botões***, da ***tela*** e dos ***menus***



# Encapsulamento

- ***Celular - Escondendo a Complexidade (Parte 2)***
  - Os ***dispositivos internos*** de ***um celular*** e os ***processos*** que ***transformam*** o ***som capturado*** pelo ***microfone*** em ***ondas*** que ***podem ser transmitidas*** para ***uma antena*** da ***operadora*** de ***telefonia móvel*** constituem a ***implementação*** do celular
  - ***Do ponto de vista*** do ***usuário*** de ***um celular***, para ***fazer uma ligação***, ***basta digitar*** o ***número*** do ***telefone desejado*** e ***clicar*** no ***botão*** que ***efetua*** a ***ligação*** (***diversos processos complexos*** são ***realizados*** pelo ***aparelho*** para que as ***pessoas*** possam ***conversar através dele***)

# Encapsulamento

- ***Carro – Evitando Efeitos Colaterais (Parte 1)***
  - A ***interface*** de ***um carro*** é ***composta*** pelos ***dispositivos*** que ***permitem*** que o ***motorista conduza*** o ***veículo*** (***volante, pedais, alavanca*** do ***câmbio***, etc.)
  - A ***implementação*** do ***carro*** é ***composta*** pelos ***dispositivos internos*** (***motor, caixa de câmbio, radiador, sistema de injeção eletrônica***, etc.) e pelos ***processos realizados internamente*** por ***esses dispositivos***



# Encapsulamento

- ***Carro – Evitando Efeitos Colaterais (Parte 2)***
  - ***Aplicando o conceito de encapsulamento, as implementações dos objetos ficam “escondidas”***
  - ***Dessa maneira, podemos modificá-las sem afetar a maneira de utilizar esses objetos***
  - ***Por outro lado, se alterarmos a interface de uso que está exposta, afetaremos a maneira de usar os objetos***



# Encapsulamento

- ***Acessando ou Modificando Atributos (Parte 1)***
  - ***Aplicando a ideia de encapsulamento, os atributos deveriam ser todos **privados** (dessa forma, os atributos não podem ser acessados ou modificados por código escrito fora da classe na qual eles foram definidos)***
  - ***As informações armazenadas nos atributos precisam ser consultadas de qualquer lugar do sistema***

# Encapsulamento

- ***Acessando ou Modificando Atributos (Parte 2)***
  - ***Disponibilizando métodos para consultar os valores dos atributos***

```
class Cliente {  
    private String nome;  
  
    public String consultaNome() {  
        return this.nome;  
    }  
}
```

# Encapsulamento

- ***Acessando ou Modificando Atributos (Parte 3)***
  - ***Modificando o valor de um atributo a partir de qualquer lugar do sistema, por meio de um método***

```
class Cliente {  
    private String nome;  
  
    public void alteraNome(String nome){  
        this.nome = nome;  
    }  
}
```

# Encapsulamento

- ***Acessando ou Modificando Atributos (Parte 4)***
  - *Normalmente, é necessário **consultar** e **alterar** o valor de um atributo a partir de qualquer lugar do sistema*
  - *Nessa situação, podemos definir os dois métodos discutidos anteriormente*
  - ***Mas, o que é melhor?** Criar os dois métodos (um de leitura e outro de escrita) ou deixar o atributo público?*
  - *Quando queremos consultar a quantidade de combustível de um automóvel, olhamos o painel ou abrimos o tanque de combustível?*

# Encapsulamento

- ***Acessando ou Modificando Atributos (Parte 5)***
  - ***Acessar*** ou ***modificar*** as ***propriedades*** de ***um objeto*** ***manipulando diretamente os seus atributos*** é uma ***abordagem*** que ***normalmente gera problemas***
  - É ***mais seguro*** para a ***integridade*** dos ***objetos*** e, conseqüentemente, para a ***integridade*** da ***aplicação***, que ***esse acesso*** ou ***essa modificação*** seja ***realizados*** por meio de ***métodos*** do ***objeto***
  - ***Através dos métodos***, é ***possível controlar*** como as ***alterações*** e as ***consultas*** são ***realizadas***

# Encapsulamento

- **Getters e Setters (Parte 1)**
  - Em Java, há **uma convenção de nomenclatura** para os **métodos** que têm como **finalidade** **acessar** ou **alterar** as **propriedades** de **um objeto**
  - Segundo essa convenção, os **nomes** dos **métodos** que **permitem** a **consulta** das **propriedades** de **um objeto** **devem possuir** o **prefixo** **get**
  - Analogamente, os **nomes** dos **métodos** que **permitem** a **alteração** das **propriedades** de **um objeto** **devem possuir** o **prefixo** **set**

# Encapsulamento

- ***Getters e Setters (Parte 2)***

- ***É fortemente indicado seguir essa convenção***, pois os ***desenvolvedores Java*** já ***estão acostumados*** com essas ***regras de nomenclatura*** e o ***funcionamento*** de ***muitas bibliotecas do Java*** ***depende fortemente*** desse ***padrão***

```
class Cliente {  
    private String nome;  
  
    public String getNome(){  
        return this.nome;  
    }  
  
    public void setNome(String nome ){  
        this.nome = nome;  
    }  
}
```



# HERANÇA



- ***Reutilização de Código: (1)***
  - Uma ***instituição financeira*** oferece ***diversos serviços*** que ***podem ser contratados individualmente*** pelos ***clientes***
  - ***Quando um serviço é contratado, o sistema bancário deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data da contratação***



- ***Reutilização de Código: (2)***
  - ***Com o objetivo de ser produtivo, a modelagem dos serviços da instituição financeira deve minimizar a repetição de código***
  - ***A ideia é reaproveitar o máximo do código já criado***
  - ***Essa ideia está diretamente relacionada ao conceito **Don't Repeat Yourself*****
    - ***Minimizar ao máximo a utilização do “copiar e colar”***
    - ***Aumentar a produtividade e reduzir o custo com manutenção***

- ***Uma classe para todos os serviços:***
  - ***Definindo apenas **uma classe** para **modelar todos** os **tipos de serviços** que **nossa instituição financeira** oferece:***

```
class Servico {  
    private Cliente contratante;  
    private Funcionario responsavel;  
    private String dataDeContratacao;  
  
    // métodos  
}
```

- ***Empréstimo: (1)***
  - *Quando um cliente contrata esse serviço, são definidos o **valor** e a **taxa de juros mensal** do empréstimo*
  - *Necessidade de adicionar dois atributos na classe **Servico**: um para o valor e outro para a taxa de juros do serviço de empréstimo*



**EMPRÉSTIMO**

- ***Empréstimo: (2)***

```
class Servico {  
    // Geral  
    private Cliente contratante;  
    private Funcionario responsavel;  
    private String dataDeContratacao;  
  
    // Empréstimo  
    private double valor;  
    private double taxa;  
  
    // métodos  
}
```

- ***Seguro de Veículos: (1)***
  - ***Para esse serviço*** devem ser ***definidas*** as ***seguintes informações:***
    - ***Veículo segurado***
    - ***Valor do seguro***
    - ***Franquia***
  - ***Necessidade*** de ***adicionar três atributos*** na ***classe*** ***Servico***



- ***Seguro de Veículos: (2)***

```
class Servico {  
    // Geral  
    private Cliente contratante;  
    private Funcionario responsavel;  
    private String dataDeContratacao;  
  
    // Empréstimo  
    private double valor;  
    private double taxa;  
  
    // Seguro de Veículo  
    private Veiculo veiculo;  
    private double valorDoSeguroDeVeiculo;  
    private double franquia;  
    // métodos  
}
```

- **Observação: (1)**

- **Apesar de seguir a ideia do DRY, modelar todos os serviços com apenas uma classe pode dificultar o desenvolvimento**
- **Imaginando que dois ou mais desenvolvedores são responsáveis pela implementação dos serviços, eles provavelmente modificariam a mesma classe concorrentemente**
- **Além disso, os desenvolvedores, principalmente os recém chegados no projeto da instituição financeira, ficariam confusos com o código extenso da classe **Servico****



- **Observação: (2)**

- **Outro inconveniente** é que **um objeto** da **classe *Serviço*** possui **atributos** para todos os serviços que a instituição oferece (**ele deveria possuir apenas os atributos relacionados a um serviço**)
- **Do ponto de vista de performance**, essa abordagem **causaria um consumo desnecessário de memória**



- ***Uma classe para cada serviço: (1)***
  - ***Para melhor modelar os serviços, evitando uma quantidade significativa de atributos e métodos desnecessários, criaremos uma classe para cada serviço***

```
class SeguroDeVeiculo {  
    // Geral  
    private Cliente contratante;  
    private Funcionario responsavel;  
    private String dataDeContratacao;  
  
    // Seguro de Veículo  
    private Veiculo veiculo;  
    private double valorDoSeguroDeVeiculo;  
    private double franquias;  
    // métodos  
}
```

- ***Uma classe para cada serviço: (2)***
  - ***Para melhor modelar os serviços, evitando uma quantidade significativa de atributos e métodos desnecessários, criaremos uma classe para cada serviço***

```
class Emprestimo {  
    // GERAL  
    private Cliente contratante;  
    private Funcionario responsavel;  
    private String dataDeContratacao;
```

```
    // EMPRÉSTIMO
```

```
    private double valor;  
    private double taxa;
```

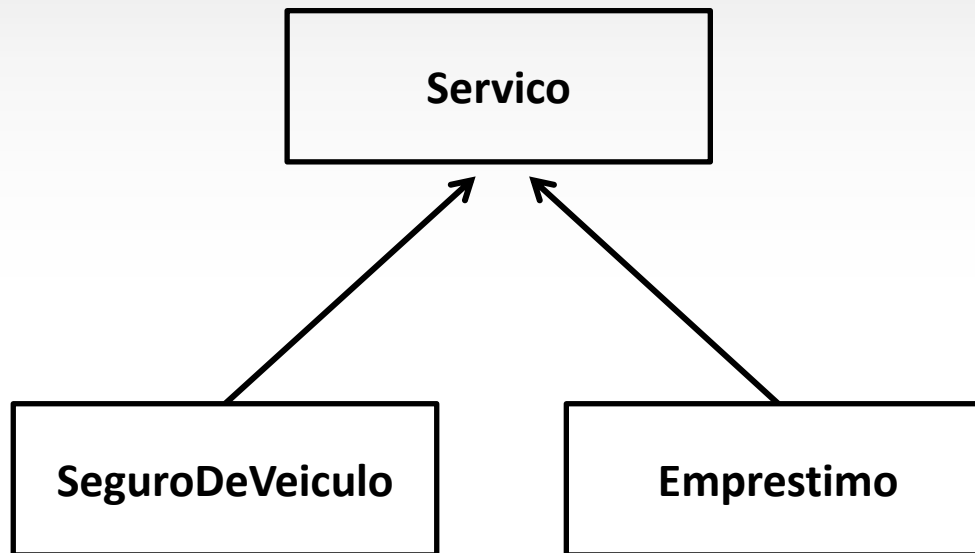
```
    // métodos
```

```
}
```

- **Observação:**
  - *Criar uma classe para cada serviço torna o sistema mais flexível, pois qualquer alteração em um determinado serviço **não** causará efeitos colaterais nos outros*
  - *Por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do DRY*
  - *Qualquer alteração que deve ser realizada em todos os serviços precisa ser implementada em cada uma das classes*

- ***Uma classe genérica e várias específicas:***
  - Na ***modelagem dos serviços*** de nossa ***instituição financeira***, ***podemos aplicar*** um ***conceito*** de ***orientação a objetos*** denominado ***Herança***
  - ***Aplicando herança***, ***teríamos a classe Servico*** com os ***atributos e métodos*** que ***todos os serviços devem ter*** e ***uma classe*** para ***cada serviço*** com os ***atributos e métodos específicos*** do ***determinado serviço***
  - As ***classes específicas*** seriam “***ligadas***” de ***alguma forma*** à ***classe Servico*** para ***reaproveitar*** o ***código*** nela definido

- ***Uma classe genérica e várias específicas:***
  - ***Relacionamento entre as classes é representado pelo diagrama da UML***



- ***Uma classe genérica e várias específicas:***
  - Os **objetos** das **classes específicas** *Emprestimo* e *SeguroDeVeiculo* possuiriam tanto os **atributos** e **métodos** *definidos* nessas **classes** quanto os **definidos** na **classe** *Servico*

```
Emprestimo e = new Emprestimo();
```

```
// Chamando um método da classe Servico  
e.setDataDeContratacao ("10/10/2010");
```

```
// Chamando um método da classe Emprestimo  
e.setValor (10000);
```

- ***Uma classe genérica e várias específicas:***
  - ***As classes específicas são vinculadas a classe genérica utilizando o comando **extends** (não é necessário redefinir o conteúdo já declarado na classe genérica)***

```
class Servico {  
    private Cliente contratante;  
    private Funcionario responsavel;  
    private String dataDeContratacao;  
}
```



- *Uma classe genérica e várias específicas:*
  - As classes específicas são *vinculadas* a classe genérica utilizando o **comando** **extends** (não é necessário redefinir o conteúdo já declarado na classe genérica)

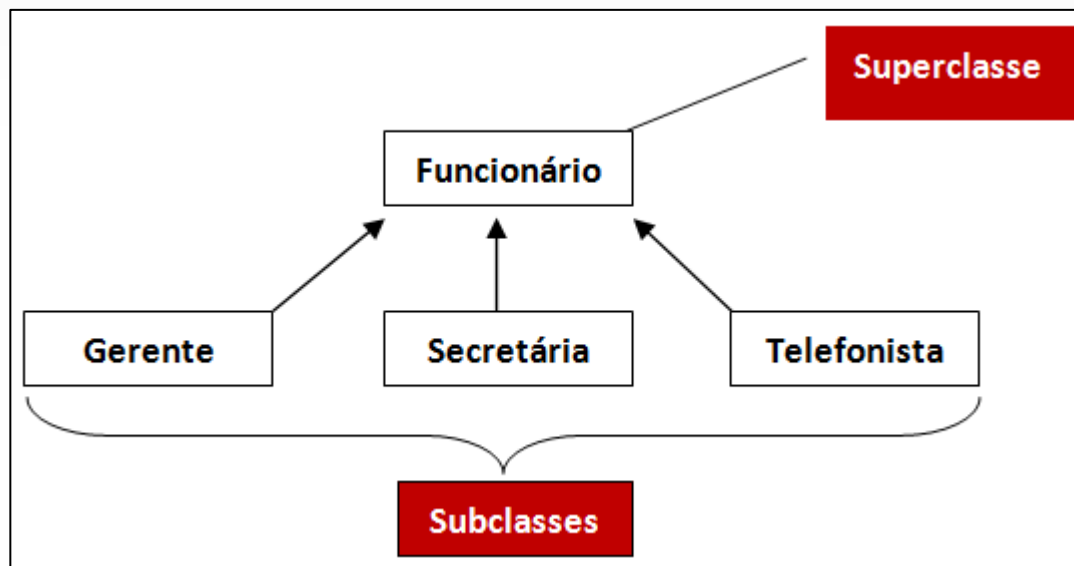
```
class Emprestimo extends Servico {  
    private double valor;  
    private double taxa;  
}
```

- *Uma classe genérica e várias específicas:*
  - As classes específicas são *vinculadas* a classe genérica utilizando o comando **extends** (não é necessário redefinir o conteúdo já declarado na classe genérica)

```
class SeguroDeVeiculo extends Servico {  
    private Veiculo veiculo;  
    private double valorDoSeguroDeVeiculo;  
    private double franquia;  
}
```

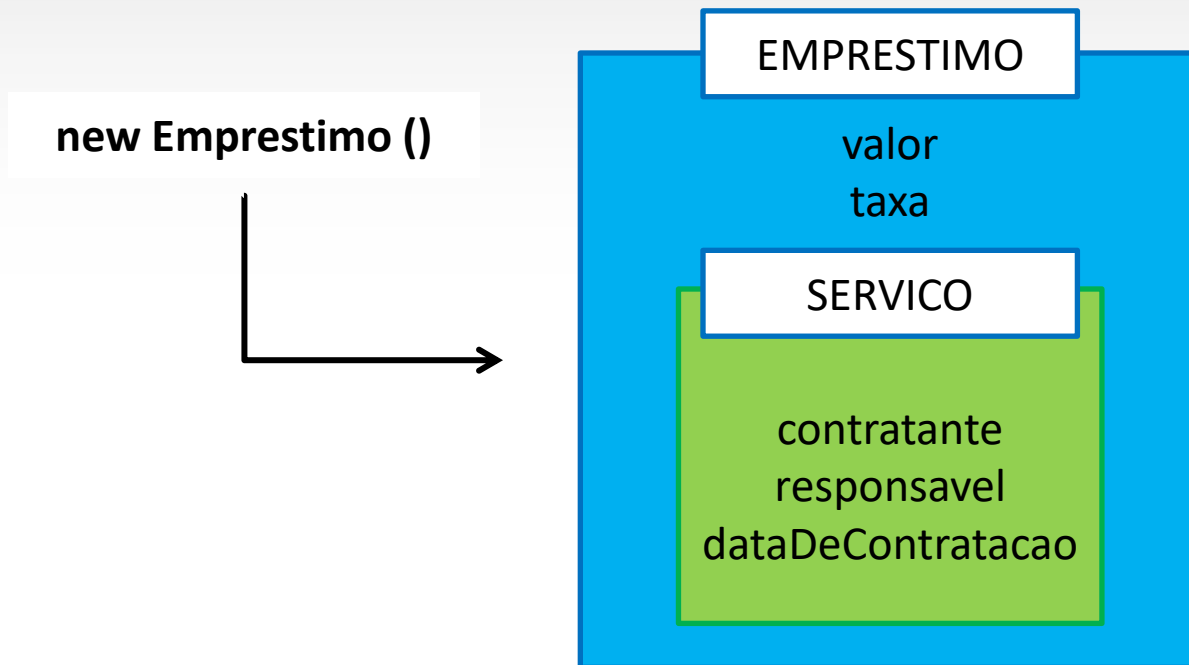
# Herança

- *Uma classe genérica e várias específicas:*
  - A *classe genérica* é denominada **super classe**, **classe base** ou **classe mãe**
  - As *classes específicas* são denominadas **sub classes**, **classes derivadas** ou **classes filhas**



# Herança

- ***Uma classe genérica e várias específicas:***
  - Quando o operador **new** é aplicado em uma **sub classe**, o objeto construído possuirá os **atributos** e **métodos** definidos na **sub classe** e na **super classe**



- **Preço Fixo:**
  - *Imagine* que *todo serviço bancário* possui *uma taxa administrativa* que deve *ser paga* pelo *cliente* que *contratar o serviço*
  - *Podemos considerar inicialmente* que o *valor dessa taxa* seja *igual* para *todos os serviços do banco*
  - *Dessa forma*, podemos *implementar* um *método* na *classe Servico* para *calcular o valor da taxa*
  - *Esse método* será *reaproveitado* por *todas as classes* que *herdam* da *classe Servico*

- **Preço Fixo:**

```
class Servico {  
    // Atributos
```

```
    public double calculaTaxa () {  
        return 10;  
    }  
}
```

– *Invocando o método calculaTaxa()*

```
Emprestimo e = new Emprestimo();
```

```
SeguroDeVeiculo sdv = new SeguroDeVeiculo();
```

```
System.out.println ("Emprestimo : " + e.calculaTaxa());
```

```
System.out.println ("SeguroDeVeiculo : " + sdv.calculaTaxa());
```

- **Reescrita de Método:**

- *Imagine* que o **valor da taxa administrativa** do **serviço de empréstimo** seja **distinto** dos **demais serviços**, pois ele é **calculado a partir** do **valor emprestado** ao **cliente**
- *Como está* **lógica** é **específica** para o **serviço de empréstimo**, torna-se **necessário adicionar** um **método** para **implementar** esse **cálculo** na **classe** **Emprestimo**

```
class Emprestimo extends Servico {  
    // Atributos  
  
    public double calculaTaxaDeEmprestimo() {  
        return this.valor * 0.1;  
    }  
}
```

- **Reescrita de Método:**

- Para os **objetos** da **classe** **Emprestimo**, devemos **invocar** o **método** `calculaTaxaDeEmprestimo()`
- **Para todos** os **demaís serviços**, devemos **invocar** o **método** `calculaTaxa()`
- Ainda sim, **nada impediria** que o **método** `calculaTaxa()` **fosse invocado** em **um objeto** da **classe** **Emprestimo**, pois **ela herda** esse **método** da **classe** **Servico**
- Dessa forma, **existe** o **risco** de **alguém** **erroneamente** **invocar** o **método incorreto**



- **Reescrita de Método:**

- A *forma mais segura* é “*substituir*” a *implementação* do método *calculaTaxa()* herdado da classe *Servico* na classe *Emprestimo*
- Para isso, *basta escrever* o método *calculaTaxa()* também na classe *Emprestimo* com a *mesma assinatura* que ele *possui* na classe *Servico*

```
class Emprestimo extends Servico {  
    // Atributos
```

```
    public double calculaTaxa() {  
        return this.valor * 0.1;  
    }  
}
```

- ***Reescrita de Método:***

- ***Observação:***

- Os ***métodos*** das ***classes específicas*** têm ***prioridade sobre os métodos*** das ***classes genéricas***
    - Isso significa que, ***se o método invocado existe na classe filha ele será invocado, caso contrário o método será procurado na classe mãe***
    - ***Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de *reescrita de método****

- ***Fixo + Específico:***

- *Imagine* que o **preço** de **um serviço** é a **soma** de **um valor fixo** mais **um valor** que **depende** do **tipo do serviço**

- **Exemplo:**

- O **preço** do **serviço** de **empréstimo** é **R\$ 5,00** mais **uma porcentagem** do **valor emprestado** ao **cliente**
    - O **preço** do **serviço** de **seguro** de **veículo** é **R\$ 5,00** mais **uma porcentagem** do **valor** do **veículo segurado**



- **Fixo + Específico:**
  - *Em cada classe específica, podemos reescrever o método `calculaTaxa()`*

```
class Emprestimo extends Servico {  
    // Atributos
```

```
    public double calculaTaxa () {  
        return 5 + this.valor * 0.1;  
    }  
}
```

- **Fixo + Específico:**
  - *Em cada classe específica, podemos reescrever o método `calculaTaxa()`*

```
class SeguraDeVeiculo extends Servico {  
    // Atributos
```

```
    public double calculaTaxa () {  
        return 5 + this.veiculo.getTaxa () * 0.05;  
    }  
}
```

- ***Fixo + Específico:***

- Se o **valor fixo** dos **serviços** for **atualizado**, ***todas as classes específicas devem ser modificadas***
- ***Outra alternativa*** seria ***criar*** um ***método*** na ***classe Serviço*** para ***calcular*** o **valor fixo** de ***todos*** os **serviços** e ***chamá-lo*** dos ***métodos reescritos*** nas ***classes específicas***

```
class Servico {  
    public double calculaTaxa () {  
        return 5;  
    }  
}
```

- **Fixo + Específico:**

- Se o **valor fixo** dos **serviços** for **atualizado**, **todas as classes específicas** devem ser modificadas
- **Outra alternativa** seria **criar** um **método** na **classe Serviço** para **calcular** o **valor fixo** de **todos** os **serviços** e **chamá-lo** dos **métodos reescritos** nas **classes específicas**

```
class Emprestimo extends Servicio {  
    // Atributos  
  
    public double calculaTaxa () {  
        return super.calculaTaxa() + this.valor * 0.1;  
    }  
}
```

- ***Construtores e Herança:***
  - ***Quando temos uma hierarquia de classes, as chamadas dos construtores são mais complexas do que o normal***
  - ***Pelo menos um construtor de cada classe de uma mesma sequência hierárquica deve ser chamado ao instanciar um objeto***





- **Construtores e Herança:**

- **Exemplo:**

- **Quando** um **objeto** da **classe** **Emprestimo** é **criado**, pelo **menos** um **construtor** da **própria classe** **Emprestimo** e um da **classe** **Servico** devem ser **executados**
    - Os **construtores** das **classes** mais **genéricas** são **chamados** antes dos **construtores** das **classes específicas**

```
class Servico {  
    // Atributos  
  
    public Servico () {  
        System.out.println ("Servico");  
    }  
}
```

- ***Construtores e Herança:***

- **Exemplo:**

```
class Emprestimo extends Servico {  
    // Atributos  
  
    public Emprestimo () {  
        System.out.println ("Emprestimo");  
    }  
}
```

- *Por padrão, todo **construtor** chama o construtor sem argumentos da classe mãe se **não** existir nenhuma chamada de **construtor explícita***

# Referências

SANTOS, Rafael; Introdução à programação orientada a objetos usando Java / Campus; Rio de Janeiro, 2003.

ALVES, William Pereira; Java 2 - Programação Multiplataforma. – Ed. Érica, 2006.

HORSTMANN, Cay; Conceitos de Computação com o Essencial de Java. Ed. Bookman – Porto Alegre, 2005.