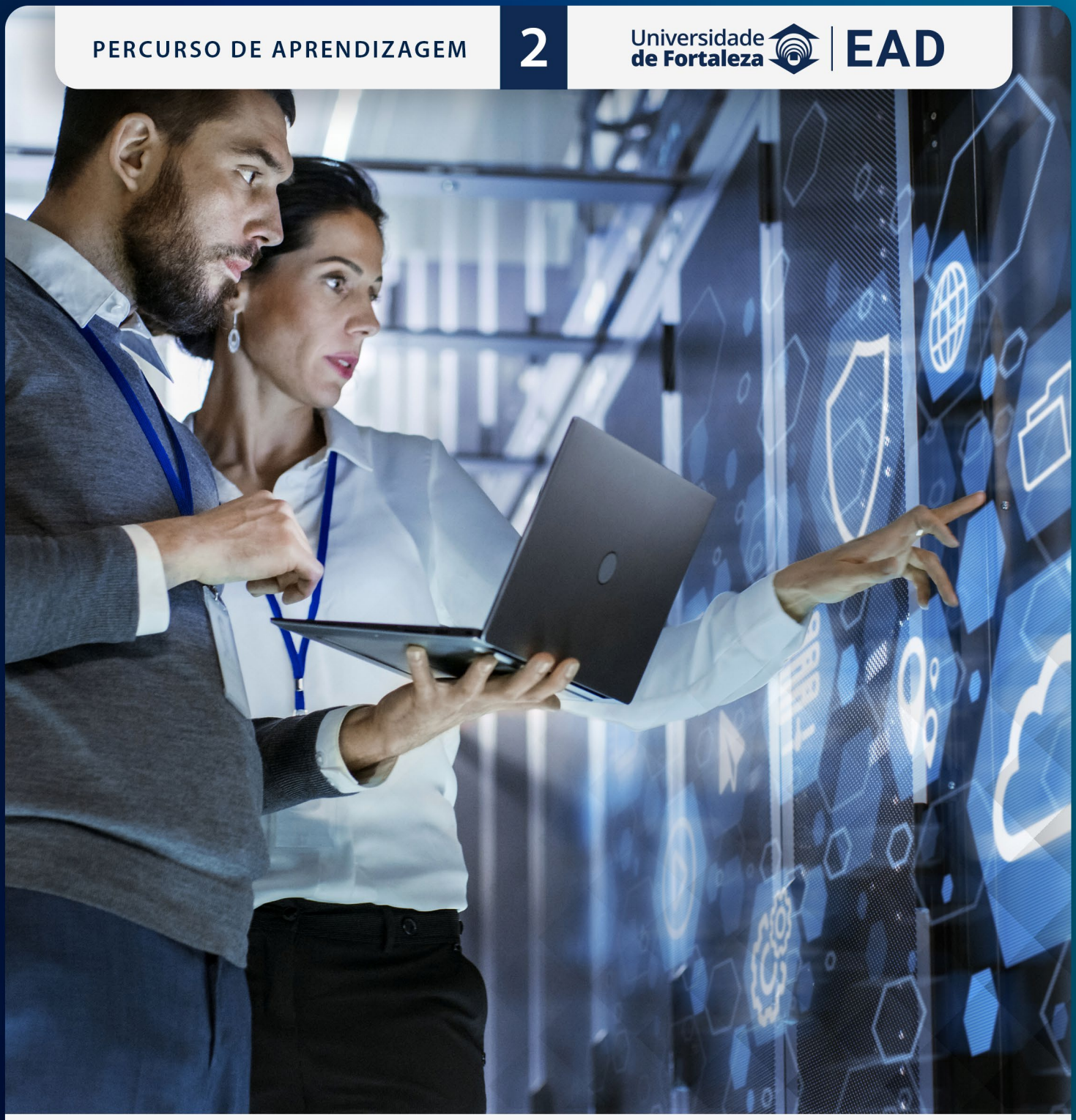
PERCURSO DE APRENDIZAGEM 2

Universidade de Fortaleza | EAD



PROGRAMAÇÃO ORIENTADA A OBJETOS

IMPLEMENTAÇÃO DE ALGORITMOS UTILIZANDO PARADIGMA ORIENTADO A OBJETOS

Universidade de Fortaleza | EAD

Sumário

Sumário clicável

Nesse percurso de aprendizagem, vamos aprofundar os pilares apresentados no percurso anterior. Iniciando com o encapsulamento e a definição de classes e objetos, atributos e métodos, instanciação, modificadores de acesso e métodos construtores e destrutores. Ademais, falaremos sobre a importância dos métodos acessores e modificadores (os famosos *getters* e *setters*). A seguir, definiremos e exemplificaremos, por meio de códigos Java e C++, herança simples e múltipla e, por fim, entraremos no último, mas não menos importante pilar, o polimorfismo, definindo sobrescrita de sobrecarga de métodos, classes abstratas e interfaces.

Olá

1. Classes e objetos: atributos, métodos, instanciação, modificadores de acesso, métodos construtores e métodos destrutores

Como visto no percurso de aprendizagem anterior, o paradigma orientado a objetos (POO) tem diversas vantagens em relação ao paradigma procedural ou estruturado. Nele encapsulamos comportamentos e características nas entidades mapeadas, além da reutilização de código, confiabilidade, facilidade de manutenção e extensão. No paradigma estruturado ainda temos o problema dos tipos de dados e procedimentos ficarem separados e isso "exige" que o desenvolvedor tenha o controle de quais dados manipulam quais procedimentos, tendo assim a manutenção em cascata e bem mais complexa.

Embora a Orientação a Objetos tenha vantagens em relação ao paradigma estruturado e não estruturado, existe uma desvantagem inicial: por ser uma forma diferente de "pensar", na qual modelamos o problema por meio de objetos e as mensagens trocadas entre eles, faz com que essa representação seja mais difícil e complexa. Isso ocorre devido à grande quantidade de conceitos que devem ser "aprendidos" para utilizarmos POO de forma correta. Contudo, após sua assimilação e domínio poderemos trabalhar de forma efetiva e consistente.

No início desta árdua — mas empolgante — jornada, apresentamos uma introdução aos principais conceitos de POO e nos exemplos evidenciamos as vantagens da utilização de sua utilização. A seguir, será apresentado o que são classes, atributos, métodos e objetos. O objetivo deste circuito de estudo é prover o contato inicial do aluno com esse paradigma e subconceitos inerentes a eles. Para um estudo com maior profundidade, existem vários livros que tratam exclusivamente desses conceitos e, como indicação de leitura, temos Manzano, 2014 e Horstmann, 2009.

1.1 Classe

Apesar do termo “Objeto” estar presente nas denominações: Paradigma Orientado a Objeto (POO), Programação Orientada a Objeto, Modelo Orientado a Objetos, tudo inicia com uma classe. Antes mesmo de ser possível manipular objetos, é preciso criar uma classe, pois esta é a unidade inicial e mínima de código na OO. É a partir de classes que futuramente será possível criar objetos.

Conforme foi visto no percurso de aprendizagem anterior, Classes são modelos, esqueletos de um entidade que será mapeada em um sistema OO. No mundo real, nós, seres humanos, já agrupamos essas entidades, por exemplo, os animais, professor, carro. Esse agrupamento representa a CLASSE de animais, conjunto de animais que têm características e comportamentos semelhantes. Para tanto, podemos exemplificar **a classe** carro com os comportamentos: percorrer distâncias, gastar combustível acelerar, frear e características (cor, marca, modelo); a classe professor com comportamento: receber salário, receber décimo terceiro salário, tirar férias, tirar licença e características (gênero, data de nascimento, carga horária, data de admissão).

Dessa forma, podemos considerar as classes com um tipo abstrato de dados (TADs) que possuem ações e características, os quais denominaremos daqui para frente de métodos (ações) e atributos (características), respectivamente.

Uma dúvida que pode surgir, como podemos definir as classes de um *software*. Exemplo um sistema que gerencia uma universidade, como o Unifor Online. Ele precisaria representar as classes disciplinas, cursos, alunos, dentre outros. O nome dessas entidades representadas pelas classes seria muito importante, por exemplo, usar a classe pessoa para representar alunos, professores, coordenadores, diretores nesse sistema acadêmico não seria uma boa escolha, porque poderia confundir e dificultar a manutenção do *software*, como foi visto no circuito de aprendizagem anterior uma parte essencial do desenvolvimento de software. A seguir, exemplos de classes em Java e C++.

Utilizaremos essas duas linguagens, uma vez que além de serem linguagens bastante utilizadas para desenvolvimento de *software* (TIOBE, 2021) são tipicamente definidas como linguagens orientadas a objetos, apesar de nas versões atuais, também, ser possível lançar mão do paradigma funcional.

1.1.1 Classe em C++

Uma classe em C++ é um "melhoramento" do *struct* (registro) em C (EDELWEISS,2014), pois, além de encapsular características (variáveis), iremos acrescentar as funções (métodos). A sintaxe em C++ seria:

(fonte: http://www.inf.furb.br/~marcel/k19-k11-orientacao-a-objetos-em-java.pdf):
class nome_da_classe {
//especificador de acesso:
// declaração das variáveis (atributos da classe)
//declaração das funções (métodos).

};

em que, **nome_da_classe** identifica a classe, especificador de acesso tem o domínio: *public, private e protected* os quais serão abordados em detalhes no próximo circuito de aprendizagem. A declaração dos métodos nos quais são declaradas as funções encapsuladas na classe definida com seus respectivos especificador de acesso, tipo de retorno, nome e lista de parâmetros, e, por fim, os atributos das classe nas quais são definidas as variáveis com especificador de acesso, tipo e nome.

Exemplo:

```
class Conta {
 public:
  int agencia;
  long int numero;
 private:
  float saldo;
 public:
  void debitar(float valor);
  void creditar(float valor);
  void transferir(float valor, Conta destino);
 private:
  boolean autenticacaoUsuario(Usuario usuario);
  void calcularReajusteConta(int porcentagem);
}
```

No exemplo da ***class*** Conta em C++, temos a identificação dos atributos: "agencia", "numero" e "saldo", sendo os dois primeiros públicos e o último privado e os métodos debitar, creditar e transferir públicos e o método privado autenticacaoUsuario(Usuario usuario), essas diferenças entre os critérios de acessos serão detalhados mais à frente.

1.1.2 Classe em Java

Como foi visto, é necessário criar classe em Java até para escrever um "Alo Mundo" na tela, ou seja, o menor programa possível em Java necessita de uma classe. A sintaxe

para criação de uma classe em Java é muito semelhante ao C++ ou C#, mas tem suas particularidades como veremos a seguir:

<visibilidade_de_acesso> class nome_da_classe {

<visibilidade_de_acesso> atributos (variáveis);
<visibilidade_de_acesso> métodos (funções);

}
Em que, nome da classe, atributos e métodos são definidos de forma semelhante ao C++. Uma diferença na *visibilidade_de_acesso* é no caso do Java termos o identificador *protected*.

A seguir, vamos para o exemplo em **Java:**

```
public class Conta {

// definição dos atributos

public int agencia;
public int numero;
public float saldo;

//definição dos métodos
private void calcularJurosInvestimentos(int porcentagem);
public void debitar(float valor);
public void creditar(float valor);
public void transferir(float valor, Conta destino);
private boolean autenticacaoUsuario(Usuario usuario);


};
```

Nesse exemplo, a ideia de como fizemos no C++ é apenas mostrar a estrutura de uma classe em Java, como podemos perceber, não nos preocupamos ainda em implementar cada método. Comparando com o código anterior, as diferenças não são apenas na forma de indicar o especificador de acesso e nos tipos de dados. No C++, existe a figura dos modificadores de tipos: long, short e unsigned, já no Java existem apenas os tipos primitivos.

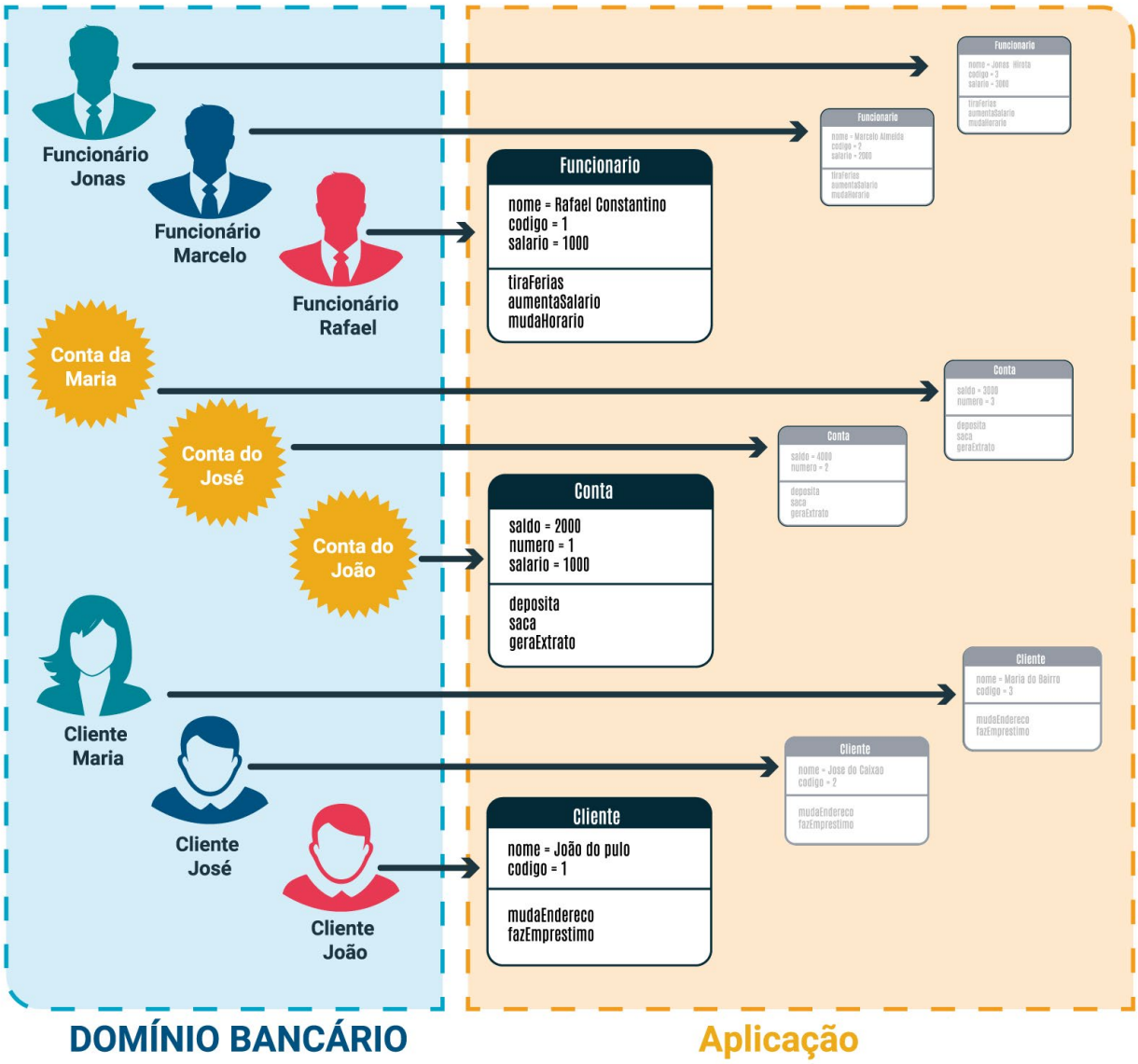
Em C++, C# e Java, por convenção, os nomes seguem o padrão “Camelcase”, ou seja, palavras compostas ou frases são escritas com a primeira letra maiúscula, com exceção da primeira palavra (Exemplo: iPhone, eBay, calcularJurosInvestimentos). Existem outras formas de escrita como PascalCase na qual todas as palavras, inclusive a primeira, são escritas com letras maiúsculas, exemplo: CalculoDesconto(), ValidaEmail() e ImprimeCadastro( )

(fonte: https://celsokitamura.com.br/pascal-case-e-camel-case/): --

1.2 Objeto

Antes de definirmos o objeto é necessário entender um pouco melhor a ideia de domínio da aplicação e o que será representado por meio das classes e objetos. Para tanto, veja a *Figura 1* que exemplifica um domínio de uma aplicação bancária, na qual temos funcionários, clientes e contas e cada um deles podem ser representados por objetos, que por sua vez são definidos (modelado) por meio das classes. Então, teríamos, pelo menos, a definição de três classes: funcionário, cliente e conta e a partir dessas classes a criação de vários objetos. Por exemplo, definida a classe funcionário, poderíamos ter os funcionários Jonas, Marcelo e o Rafael, já, por meio da classe Cliente, poderíamos criar os objetos Maria, José e João, conforme a figura 1 e suas respectivas contas.

Figura 1 – Domínio Bancário.



Fonte: Elaborada pelo autor (2021).

No exemplo da *Figura1*, podemos ver que, a partir de uma classe, podemos gerar várias instâncias dela, que por sua vez seriam os objetos, ou seja, a concretização de uma classe em memória seriam os objetos. Por meio de uma classe, podemos criar vários objetos. Prezados, é hora de ir para código, bora lá:

Para não ficarmos “bitolados” só em uma linguagem, vamos mostrar um exemplo em C++ e outro em Java.

1.2.1 Instanciando objetos em C++

Em C++, a sintaxe para que o objeto seja criado, ou seja, instanciado, é a seguinte:

nome_da_classe nome_do_objeto;

Exemplo:

Conta c1, c2;

Nesse exemplo, estamos criando dois objetos c1 e c2, por meio da classe Conta, na qual cada um deles terá uma agência, saldo e contas diferentes. O acesso por meio do atributo ocorre dessa forma:

nome_do_objeto.atributo;

Exemplo:

cout << c1.agencia << “ “ << c1.numero;
cout << c2.saldo;

No exemplo acima, estamos imprimindo as informações da agência e do número da conta c1 e o saldo da conta c2, ou seja, as propriedades estão “gravadas” encapsuladas em cada objeto. Temos, também, a sintaxe para executarmos os métodos:

nome_do_objeto.nome_do_metodo;

Exemplo:
c1.creditar(100);
c2.debitar(50);

Nesse exemplo, estamos creditando **R$ 100,00** reais a conta c1 e debitando **R$ 50,00** reais da conta c2. Veja como é interessante, no código acima a mensagem.

1.2.1 Instanciando objetos em Java

No Java a sintaxe é um pouco diferente do C++, uma vez que, para o objeto ser criado e instanciado, é necessário usarmos o comando **new,** sendo assim, a sintaxe para criar e instanciar um objeto:

```
//criando um objeto
nome_da_classe nome_do_objeto;
//instanciando um objeto
nome_do_objeto = new nome_da_classe();
```

ou

```
nome_da_classe nome_do_objeto = new nome_da_classe();
```

Como podemos ver no exemplo acima para criarmos em memória um objeto no java é necessário criar a referência e instanciar:

```
Conta c = null;

c = new Conta();
```

No momento que fazemos “Conta c=null;” estamos criando uma referência para um objeto da classe Conta que aponta para *null*, mas neste momento ainda não temos nenhum atributo criado em memória.

Figura 2 - Representação da referência null para o objeto c da classe Conta.

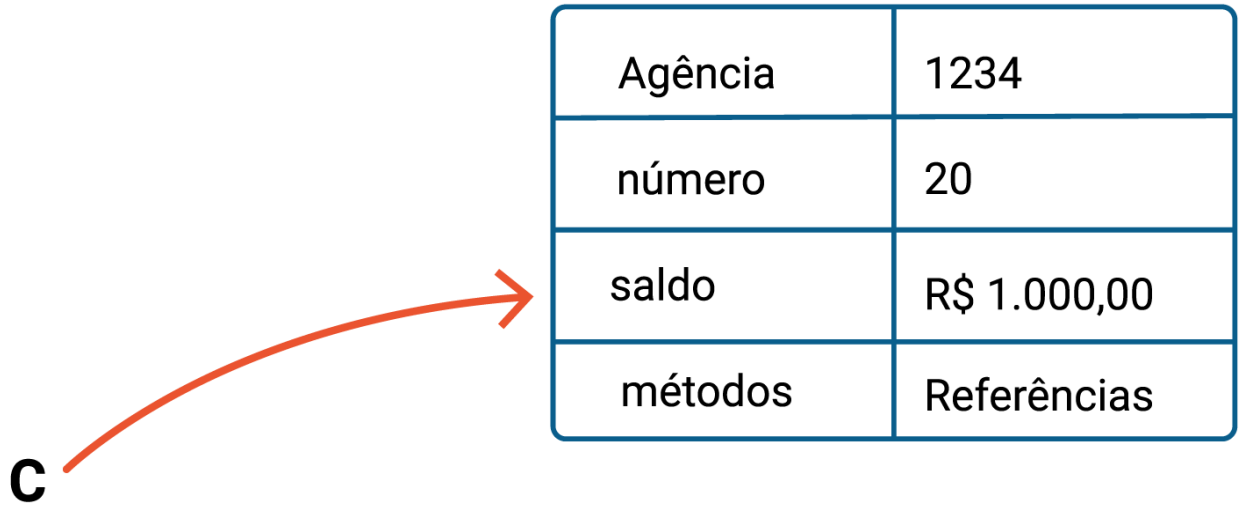
Fonte: Elaborada pelo autor (2021).

Como mostrado no exemplo, para criar o objeto em memória é necessário usar o comando **new** (new Conta()). Para exemplificar, se fizermos:

```
c = new Conta();
c.agencia = 1234;
c.numero=20;
c.saldo=1000;
```

No código acima, temos a criação de um objeto da **classe** Conta e a atribuição de valores para esse objeto a representação gráfica pode ser vista na figura 3.

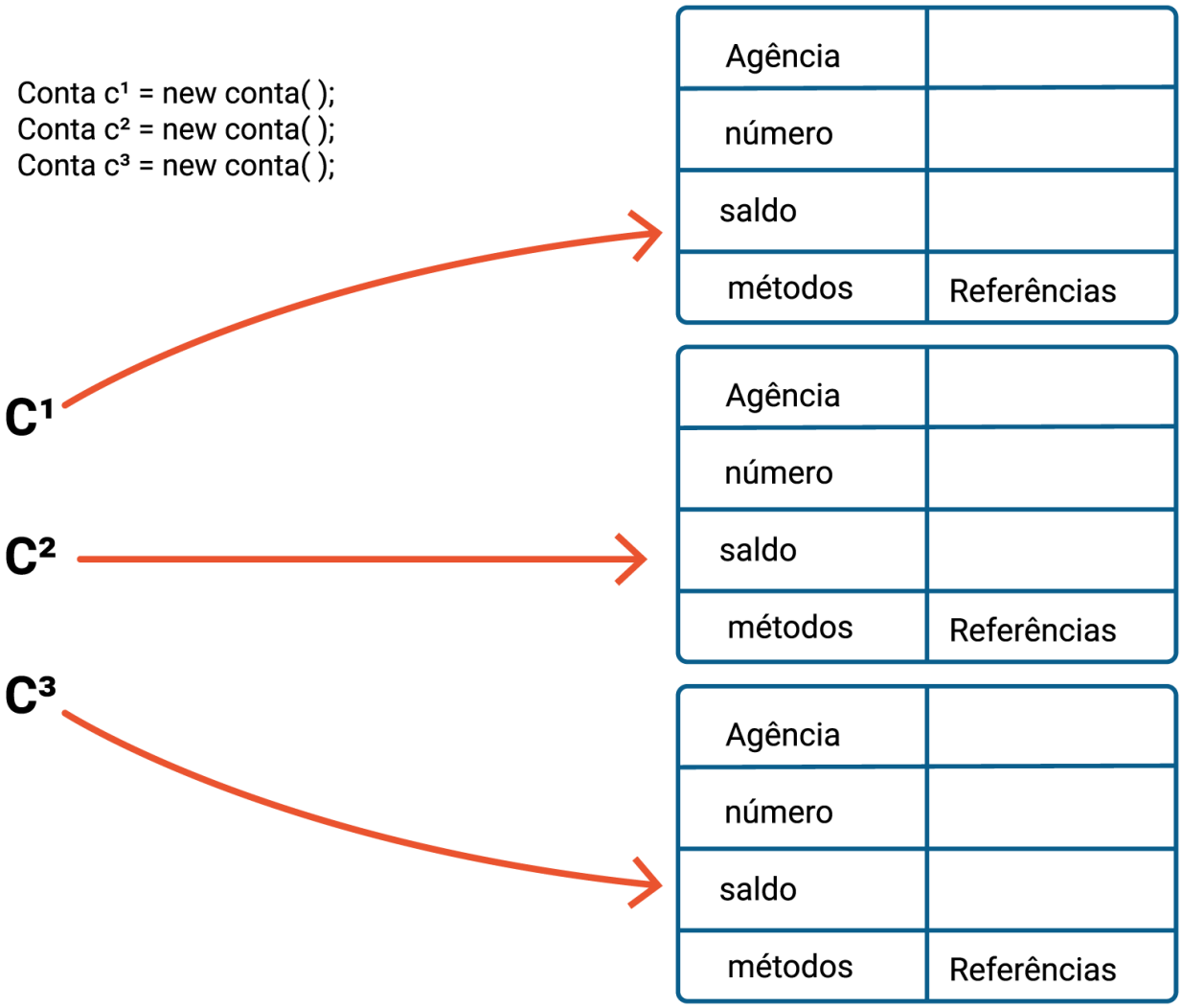
Figura 3 - Representação da referência c apontando para o objeto com os valores iniciados.



Fonte: Elaborada pelo autor (2021).

Para deixar mais claro como o **new** funciona criando instâncias dos objetos. Caso tenha várias chamadas de **new**, criaremos vários objetos em memória. Segue exemplo na *Figura 4*, na qual são criados três objetos do tipo conta: c1, c2 e c3. Podemos ver que todos os objetos estão em espaço de memória diferentes e alterar um objeto não necessariamente altera o outro.

Figura 4 - Exemplo de objetos do tipo Conta sendo instanciados.



Fonte: Elaborada pelo autor (2021).

Importante!

Uma analogia ao comando **new**, instanciando um objeto Java, é como estivéssemos contratando uma cozinheira para fazer uma receita de bolo (a receita seria a classe). Ou com uma planta de uma casa em mãos (planta seria a classe) pagássemos uma construtora para executar a planta.

Outra analogia importante para referência em Java é relativa à referência. É como em um carro de controle remoto o carro fosse o objeto e o "controle remoto" a referência, através dele você pode controlar o carro e acessar as suas ações (funcionalidades), inclusive desligar o carro...

2. Encapsulamento: métodos acessores e métodos modificadores

Implementando, de forma completa, a classe Conta podemos notar várias características importantes. A seguir vamos implementá-la com o intuito de exemplificar e explicar os conceitos de construtor e destrutor:

```
#include <stdio.h>
class Conta {

public:
// atributos valor e tipo
float saldo;
long int numero;
int agencia;
// método construtor
Conta() {
printf("\nExecutando o método construtor padrão");
}
Conta (long int numero n, int a) {
printf("\nExecutando o método construtor");
agencia = a;
numero = n;
saldo = 0;
}

// método destrutor
~Conta() {
printf("\nExecutando o método destrutor");
}
// .....
};

int main() {
float agencia, numero;
printf("\n\ndigite a agencia: ");
scanf("%f",&agencia);
printf("\ndigite o numero da conta: ");
```

```
scanf("%f",&numero);
Conta c(agencia,numero); // Execução do construtor;
getchar();
return 0;
}
```

Na implementação em C++, temos a implementação do construtor padrão com o mesmo nome da classe e sem parâmetros e do construtor com parâmetros "**Conta (long int numero n, int a)** {" o qual foi executado na "Conta c (agencia,numero);" criando uma conta com a "agencia" e "numero" informado pelo usuário e saldo igual a ZERO (veja a definição do construtor). Temos, também, a definição do destrutor com a implementação de métodos com prefixo **~**. O construtor tem o papel de inicialização dos atributos da classe enquanto o destrutor tem a função de liberar o espaço. O construtor é chamado apenas uma vez na criação do objeto, já o destrutor é executado quando o objeto é destruído. No caso do exemplo acima em C++, é executado na linha "Conta c (agencia,numero); ***// Execução do construtor;"*** e destrutor é executado quando o método main termina (ou quando usamos o método delete). A seguir, vamos para o exemplo em **Java:**

```
1. public class Conta {
// definição dos atributos
2. public int agencia;
   3. public int numero;
   4. public float saldo;
//definição dos métodos construtor
5. public Conta() {
6. this.saldo = 0;
7. this.numero = 0;
8. this.agencia = 0;
9. }
//definição dos métodos construtor
10. public Conta(int numero, int agencia) {
11. this.saldo = 0;
12. this.numero = numero;
13. this.agencia = agencia;
14. }
//definição dos métodos
15. private void calcularJurosInvestimentos(int porcentagem) {
16. this.saldo = this.saldo + this.saldo*porcentagem;
17. }
18. public void debitar(float valor) {
19. this.saldo = this.saldo - valor;
20. }
21. public void creditar(float valor) {
22. this.saldo = this.saldo + valor;
23. }
24. public void transferir(float valor, Conta destino) {
25. destino.creditar(valor);
26. debitar(valor);
27. }
28. private boolean autenticacaoUsuario(Usuario usuario) {
29. //autenticar usuário...
30. }
31. };
```

Nesse exemplo do Java temos a definição de dois construtores: o padrão, sem parâmetros (linha 5), e o da linha 10 com passagem de dois parâmetros: o número da conta e o número da agência. Conforme já mencionado, o construtor tem o papel de inicialização das propriedades da classe.

```
1. public class CriaConta {
2. public static void main(String[] args) {
        3. Conta primeiraConta = new Conta ();
        4. primeiraConta.saldo = 200;
        5. System.out.println(primeiraConta.saldo);
}
}
```

Na classe *CriarConta* que é uma classe executável em Java, já que tem o método **main**, temos a instanciação da conta chamando o construtor padrão e, em seguida, a alteração e impressão do saldo desta conta. O número da conta e da agência seria ZERO já que o construtor padrão faz essa inicialização. Caso a chamada da linha 3. fosse "Conta primeiraConta = **new** Conta(10,123);", a conta seria a 10 e o saldo 123...

Vamos, agora, falar um pouco sobre os métodos acessores e modificadores. Para tanto, vamos criar a classe Funcionario, abaixo:

```
1. public class Funcionario {
2. private String nome;
3. private String cpf;
4. private double salario;
5. public String getNome() {
6. return nome;
7. }

8. public void setNome(String nome) { //método modificador
9. this.nome = nome;
10. }

11. public String getCpf() { //método acessores
12. return cpf;
13. }
14. public void setCpf(String cpf) { //método modificador
15. this.cpf = cpf;
16. }
17. public double getSalario() { //método acessores
18. return salario;
19. }
20. public void setSalario(double salario) { //método modificador
21. this.salario = salario;
22. }
23. }
```

Os atributos “nome”, “cpf” e “salario” são privados para viabilizar o encapsulamento (linhas 2, 3 e 4). Temos os métodos acessores (gets) linhas 5, 11 e 17 os quais são públicos e têm a ideia de retornar os valores das propriedade dos objetos da classe Funcionario. Por fim, temos os métodos modificadores (sets), linhas 8, 14 e 20, os quais têm a função de alterar o valor dos atributos da classe. Como podemos ver no exemplo, ele tem o retorno void e recebem como parâmetro uma variável do mesmo tipo da qual o atributo quer ser alterado. Esses métodos são chamados de *getters* e *setters*.

Importante!

É importante notar que só devemos criar os métodos gets e sets que realmente precisam ser utilizados. Na classe Conta, vista no circuito de aprendizagem anterior, não teria sentido, por exemplo, criar o setSaldo(float valor) e o getSaldo(), uma vez que, para alterar o saldo, é necessário creditar ou debitar. Apesar de alguns programadores já terem o hábito de gerar todos os *getters* e *setters* dos atributos das classes, essa não é uma boa estratégia uma vez que estamos deixando todos as características da classe “expostas”. Essas classes são conhecidas como classes “fantoche”.

Suponha agora que precisássemos de um método para fazer uma bonificação de 15% de salário para um funcionário. Vamos ao código:

```
public class Funcionario {
private String nome;
private String cpf;
private double salario;
public double getBonificacao() {
return this.salario * 0.15;
}
// ...

}
```

Nesse caso, teríamos um novo método *getter* (*getBonificacao*) mesmo sem o atributo bonificacao. Não existe problema em fazer isso. Para testar a classe Funcionario, poderíamos ter:

```
1.      public class TesteFuncionario {
2. public static void main(String[] args) {
3. Funcionario nico = new Funcionario();
4. nico.setNome(“Nico Steppat”);
5. nico.setCpf(“223355646-9”);
```

```
nico.setSalario(2600.00);
System.out.println(nico.getNome());
System.out.println(nico.getBonificacao());

}
}
```

Nessa implementação estaríamos criando o objeto nico da classe Funcionario na linha 3 e executando o construtor padrão.

> Toda a classe tem construtor, mesmo que não seja criado explicitamente, a máquina virtual java cria um construtor padrão inicializando valores padrões para os atributos, exemplo: tipo boolean é false, int é zero, float é 0.0 e os objetos são *null*).

Como podemos perceber, na classe Funcionario, temos atributos identificados como modificadores de acesso **private** e os métodos acessores e modificadores, como **public.** Além desses dois, temos, também, no Java, os modificadores **default** e o **protected.** Sendo que o **public** é aquele no qual os atributos ou métodos poderão ser chamados em outra classe ou pacote, como próprio nome identifica, são "públicos". Os privados (**private**) são aqueles que, ao contrário do **public**, só podem ser acessados dentro da definição da própria classe, enquanto que o default é dentro do mesmo pacote Java e o protected pode ser acessado nas classes filhas da mesma hierarquia, esse último ficará mais claro no próximo circuito de aprendizagem que será apresentado herança. Abaixo a *Tabela 1*, na qual apresentamos os critérios de acesso da linguagem Java e C++.

Tabela 1 - Modificadores de Acesso nas linguagens C + + e Java.

Modificadores de Acesso nas linguagens C + + e Java			
Critério de Acesso	C++	Java	Observação
Público	Public	Public	Atributos e métodos podem ser acessados em outra classe ou pacote.
Privado	Private	Private	Atributos e métodos só podem ser acessados dentro da mesma classe.
Padrão	–	default	Atributos e métodos só podem ser acessados dentro do mesmo pacote Java. Não existe um equivalente em C++
Protegido	Protected	Protected	Atributos e métodos podem ser acessados em outra classe ou pacote hierarquia de classe. É utilizado juntamente com a herança.

Fonte: Elaborada pelo autor (2021).

Como podemos notar uma diferença entre C++ e Java é o modificador de acesso **default**, no C++ ele não existe, já no java, caso não seja colocado nenhum modificador explicitamente, ele é considerado como padrão. Ou seja, se na linha 2 da classe Funcionario a definição fosse “String nome;” o atributo nome seria do tipo padrão e poderia ser acessado dentro do mesmo pacote (*package*) Java (*exemplo de um pacote Java: java.util, java.io*).

3. Heranças simples e múltiplas

Anteriormente, no paradigma estruturado, identificamos problemas em trabalhar tudo em única classe ou arquivo, dificultando o reuso. Nesse circuito de aprendizagem, daremos início ao aprendizado com o conceito de herança.

Vamos supor um exemplo de um sistema de Recursos Humanos diferentes tipos de funcionários, cada um com suas especificidades, ou seja, juntarmos tudo **“um só lugar”** tornará nosso programa muito difícil de manter.

Nosso objetivo então será separar as classes. Teremos uma para Funcionario e outra para Supervisor.

A partir da classe Funcionario, criaremos a classe Supervisor. Selecionaremos e utilizaremos o atalho **"Ctrl + C" e "Ctrl + V" - famoso *Copy & Paste* - *que às vezes alguns alunos fazem nos trabalhos***, para que seja criada uma cópia sua. A esta cópia daremos o nome de Supervisor. Como você já deve estar pensando, isso não é uma boa estratégia nem para os trabalhos (principalmente se o professor descobrir) quanto na implementação de sistemas. Código duplicado dificilmente é uma boa escolha, porque estamos indo de encontro com um dos princípios da POO: reuso de código.

Essa classe terá todos os atributos que a classe Funcionario possui, exceto pelos atributos de quantidade de funcionários subordinados e a senha, que serão criadas especificamente para ele. Além disso, agora a bonificação funcionará de forma diferente para o supervisor, uma vez que nesse cargo os empregados terão direito a um salário extra (conhecido como o 14o. décimo quarto salário, em referência ao 13o.). Por fim, se vamos armazenar uma senha, teremos também um método que a autentica:

```
public class Supervisor {

private String nome;
private String cpf;
private double salario;
private int senha;
private int quantidadeFuncionariosSubordinados;
public boolean autentica(int senha) {
if(this.senha == senha) {
return true;
} else {
return false;
}
}
public double getBonificacao() {
return this.salario;
}

public String getNome() {
return nome;
}

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

```
public String getCpf() { //método acessores
return cpf;
}
public void setCpf(String cpf) { //método modificador
this.cpf = cpf;
}
public double getSalario() { //método acessores
return salario;
}
public void setSalario(double salario) { //método modificador
this.salario = salario;
}
}
```

Assim, nosso código melhorou, mas ainda pode ficar menos verboso e com menos repetições. Como podemos observar, diversos dos atributos foram repetidos em ambas as classes, bem como muitos dos *getters* e *setters*.

O ideal seria termos um mecanismo que nos permitisse indicar que o Supervisor já possui estes elementos presentes em Funcionario. E Temos :) ! Herança. Ou seja, teríamos que indicar ao Java que o Gerente **herda tudo** de Funcionario.

Na sintaxe da linguagem, a herança é expressada pela palavra "**extends**". Nossa classe Supervisor ficaria então da seguinte forma:

```
    public class Supervisor extends Funcionario {
private int senha;
public boolean autentica(int senha) {
if(this.senha == senha) {
return true;
} else {
return false;
}
}

    public double getBonificacao() { /
return this.salario; //erro de acesso
}
}
```

Ou seja, o Supervisor tem, e sabe fazer, tudo que o Funcionário faz. Ainda assim, nosso método getBonificacao() apresenta um erro, pois o modificado de acesso dele é **privado** e não pode ser acessado em uma outra classe.

Relembrando a *Tabela 1*, podemos verificar que o modificador de acesso mais adequado, aqui, seria *protected*, uma vez que o mesmo dá acesso às classes filhas. Dessa forma, a linha 4 da classe Funcionario poderia ser "**protected** double salario;".

Suponha agora que teríamos, também, além da classe Supervisor, a classe Professor e Contador e que um Professor ainda pudesse ser coordenador. Ou seja, o diagrama de classe fosse de acordo com a *Figura 5*:

```
1. public class Professor extends Funcionario{
2. protected float horasAula = 0;
3. public Professor() { }
4. public void lecionar(float horas){
     5. this.horasAula += horas;
     6. }
     7.}
```

Note que na linha 1 da classe Professor já usando a palavra reservada **extends** que indica que a *class* professor herda todos os métodos e atributos da classe Funcionario. Aqui temos uma dúvida muito comum entre os alunos: os atributos **privados** são herdados? Sim, eles são herdados apesar de não termos acesso a eles de forma direta, já que estamos em outra classe, mas podemos acessá-los de forma indireta.

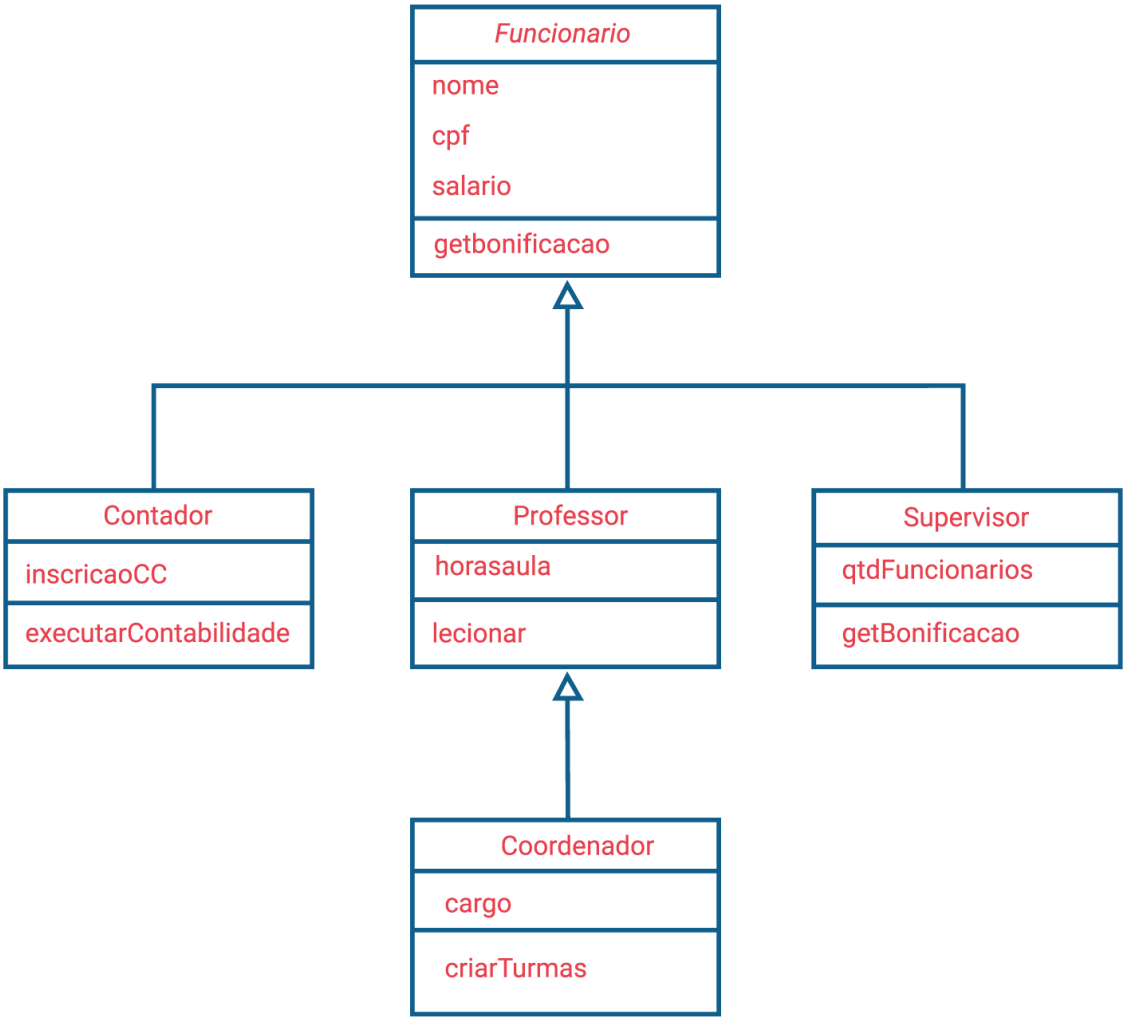
Outro ponto importante é que, como podemos ver, na classe Professor, o atributo horasAula e o método lecionar foram acrescentados, ou seja, além das propriedades "nome", "cpf" e "salario" que foram herdadas da classe pai podemos adicionar outros atributos ou métodos específicos para as classes derivadas.

Como ocorreu na classe Professor a classe Contador, também, adiciona o atributo inscricaoCC e método executarContabilidade.

```
     public class Contador extends Funcionario{
private boolean inscricaoCC;
public Contador(boolean inscricaoCC) {
super();
this.inscricaoCC = inscricaoCC;
}
public void executarContabilidade(float horas){ }
}
```

Por sua vez, na classe Coordenador, incluímos o atributo cargo e o método *criarTurmas()*. Essa possibilidade de acrescentar métodos e propriedades nas classes filhas é muito importante, uma vez que podemos incluir especializações nas classes mais específicas, o que ocorre na maioria das vezes na prática.

Figura 5 - Diagrama de classe representando a herança de um Funcionário.



Fonte: Elaborada pelo autor (2021).

Podemos usar como exemplo a *Figura 5*, um coordenador tem atribuições e características que não pertencem ao professor, mas como todo coordenador é um professor ele "herdaria" todas as características do professor.

```
public class Coordenador extends Professor{

     private String cargo;

     public Coordenador(String cargo){
this.cargo = cargo;
}

public void criarTurmas(int turmas){
System.out.println("Criou "+turmas+" turmas");

     }
}
```

Na classe “Main” podemos verificar a criação de três objetos: um da classe Professor (linha 3), outro na linha 5 Contador e um na linha 7 Funcionario. Na linha 4, vemos que um professor pode executar o método lecionar, já o Contador e o Funcionario não poderiam, o mesmo acontecendo com a ação executarContabilidade na linha 6. Reforçando, temos as chamadas dos construtores toda vez que um objeto é instanciado (linhas 3, 5 e 7) e como foi visto o Java não implementa os métodos destrutores de forma explícita.

```
1. class Main {
2.public static void main(String[] args) {
3. Professor p1 = new Professor(“Fernando”, 3420194, false, “22/04/1989”);
4.p1.lecionar(8);
5.Contador c1 = new Contador(true);
6.c1.executarContabilidade(8);
7.Funcionario f1 = new Funcionario();
8.}
9.}
```

Como vimos no exemplo da *Figura 5*, a Herança tenta organizar em classes, chamadas superclasses ou classes pai, todas as propriedades e ações comuns a vários tipos de objetos dentro de um contexto de sistema. Caso alguns objetos possuam especificidades, estas deverão ser descritas em classes chamadas derivadas ou subclasses. Assim, dizemos que subclasses estendem superclasses ou classes pai. Herança implementa uma relação do tipo “é um”. Por exemplo, analisando a *Figura 5*, podemos dizer que um Professor é um tipo de Funcionario e que Contador também é um tipo de Funcionario. Ou seja, tudo que está definido na classe Funcionario serve tanto para Professor quanto para Contador. Por isso, dizemos que Funcionario é a superclasse (ou classe mãe) e que Contador e Professor e, por sua vez, Coordenador é filha de Professor e neta de Funcionario. Essa classe herda as duas classes ancestrais, ou seja, uma classe herda da linhagem de ancestrais da sua classe.

Tudo o que estiver definido em uma superclasse (atributos e métodos) é, automaticamente, repassado para as classes filhas, sem que haja necessidade de repetição de código. Essa passagem acontece como uma herança da vida real, em que os filhos sempre herdam algo dos pais (por exemplo: a herança genética, como cor dos olhos e cor dos cabelos, ou herança financeira, como bens móveis e imóveis). Herança permite definir classes em função de outras.

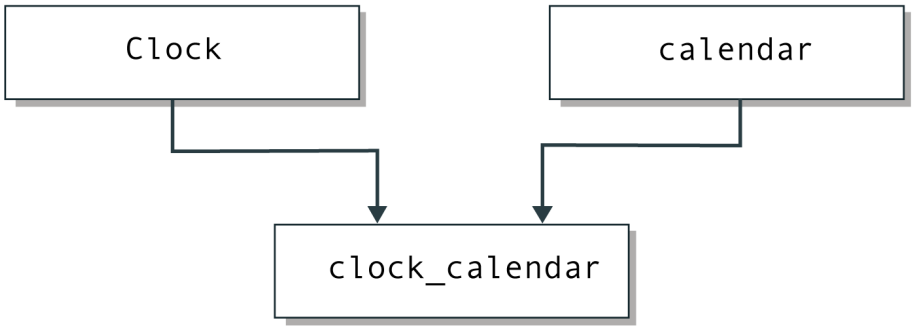
Como podemos ver nos exemplos em Java, apenas uma classe foi herdada de outra. Isso porque, Java implementa herança simples, ou seja, não é possível herdar de mais de

uma classe ao mesmo tempo (pelo menos classes concretas, como veremos no próximo circuito de aprendizagem). Por um lado, essa restrição é benéfica, já que simplifica o entendimento do código. Pense, por exemplo, em uma situação que duas classes tivessem o método *getBonificacao()* e uma terceira classe herdasse as características e ações dessas duas primeiras classes, qual método seria realmente herdado ?

Contudo algumas linguagens de programação implementam o que chamamos de herança múltipla, ou seja, a oportunidade de uma classe herdar de uma ou mais classes. Como podemos ver no *Figura 6*.

A classe *clock_calendar* sendo criada a partir da classe clock e *calendar*. No caso do C++, isso é possível uma vez que a linguagem implementada essa característica de herança múltipla. E, com certeza, você deve estar pensando e no Java como poderíamos implementar essa situação da *Figura 6*? hmm, isso será respondido no próximo ciclo de aprendizagem através de interfaces...

Figura 6 - Exemplo de Herança Múltipla.



Fonte: Elaborada pelo autor (2021).

Usando o exemplo da *Figura 6*, teríamos a implementação da classe clock_calendar da segunda forma, supondo a existência da classe clock e calendar (para mais detalhes veja

```
class clock_calendar : public clock, public calendar {
     public:
     clock_calendar(int mt, int d, int y, int h, int mn,
int s,int pm);
void advance();
     };
```

4. Polimorfismo: sobrecarga e sobrescrita de métodos, classes abstratas e interfaces

Vimos até aqui uma parte importante do conceito de herança, que tratou da reutilização de código. Como preconiza a POO, não queremos repetir código, em vez disso, estendemos a classe, utilizando o extends - Professor estende a classe Funcionario, por exemplo. Isso significa dizer também que o Professor herdou todos os atributos e características da classe Funcionario. Seguiremos, agora, para exemplificação de um novo pilar, não menos importante, Polimorfismo. Criaremos uma nova classe, chamada TesteReferencias e nela, criaremos um novo gerente chamado g1:

```
public class TesteReferencias {
public static void main(String[] args) {
Professor p1 = new Professor();
p1.setNome("Paulo");
String nome = p1.getNome();
System.out.println(nome);
}
}
```

Assim, podemos chamar qualquer um dos métodos definidos, primeiro, teremos o setNome() e, em seguida, recuperamos o nome com o g1.getNome. Por fim, imprimimos o nome. Com o objeto p1 da classe Professor podemos acessar qualquer método ou atributo da classe Professor que ou da classe Funcionario que tem sua propriedades e métodos herdados.

Porém, no código da classe *TesteReferencias* temos um problema em relação ao que a POO indica como boa prática de programação. Não deixar o código "engessado", ou seja, se por algum motivo precisássemos mudar o objeto p1 para ser um contador, por exemplo, no caso de um professor mudar de profissão e querer ser contador, não seria possível. Dessa forma, um código mais "orientado a objetos" seria:

```
public class TesteReferencias {

public static void main(String[] args) {
Funcionario p1 = new Professor();
p1.setNome("Paulo");
String nome = p1.getNome();
System.out.println(nome);
}
}
```

Pois a referência para professores seria de um tipo mais genérico “Funcionario” e não do tipo “Professor”, ou seja, um professor que resolvesse mudar de profissão e ser Contador não teria problema, pois todo Funcionário pode ser Contador ou Professor. O código abaixo funcionaria sem problemas:

```
public class TesteReferencias {

public static void main(String[] args) {
Funcionario p1 = new Professor();
p1.setNome(“Paulo”);
String nome = p1.getNome();
//FIXME: Funcionaria, porque todo contador é Funcionaário
Contador c1 = (Contador) p1;
System.out.println(nome);
}
}
```

Outro ponto importante é que o contrário não é verdadeiro, ou seja, uma referência para Professor não pode apontar para um objeto concreto Funcionario, uma vez que nem todo funcionário é professor, ele pode ser contador ou supervisor. (SCHILDT, 2015)

```
//...

public static void main(String[] args) {

Professor p1 = new Funcionario();

//...
```

Outro ponto importante, vamos supor o código abaixo, eu sei já tá cansado de código :(... mas é necessário, porque é mais fácil explicar por meio de exemplos:

```
     public class TesteReferencias {
public static void main(String[] args) {
1. Funcionario s1 = new Supervisor();
2. s1.setNome(“Paulo”);
3. String nome = s1.getNome();
4. s1.autentica(1111); //erro
5. System.out.println(nome);
}
}
```

Nesse código, teríamos um erro do compilador na linha 4. Porque para compilador o que importa é que **s1** é da classe Funcionario, apesar de ter sido instanciado a classe Supervisor. E na classe Funcionario não temos o método autentica. Sendo assim, seria necessário fazer uma mudança de tipo como fizemos em um exemplo anterior (***(Supervisor)****s1.autentica(1111)*);

Agora você deve estar pensando, então, qual a vantagem de se instanciar um Supervisor tendo uma referência para Funcionario. Boa pergunta!!!

Pense na seguinte situação, precisamos somar o salário, juntamente com a bonificação de todos os funcionários de uma empresa. Pois é necessário ver o custo fixo da empresa.

```
     public class ControleSalario {
private double soma;
public void registra(Supervisor s) {
double boni = s.getBonificacao();
this.soma = this.soma + boni + s.getSalario();
}
public void registra(Contador s) {
double boni = s.getBonificacao();
this.soma = this.soma + boni + s.getSalario();
}
public void registra(Professor s) {
double boni = s.getBonificacao();
this.soma = this.soma + boni + s.getSalario();
}

public double getSoma() {
return soma;
}
}
```

Como podemos ver no código acima, é necessário “registrar” todos os Supervisores, Contadores e Professores. E isso não é tão interessante, pois imagine agora que estamos contratando outro tipo de Funcionário um Administrador de Empresa, hmmm. E aí teríamos que lembrar de incluir outro método registra (Administrador a);

> Percebemos que, nessa implementação, temos vários métodos registrados com o mesmo nome, isso já seria polimorfismo? Sim, mas um polimorfismo estático, criar vários métodos com o mesmo nome.

Para ficar mais claro vamos para o exemplo:

```
     public class TesteReferencias {
public static void main(String[] args) {
Supervisor s1 = new Supervisor();
s1.setNome(“Paulo”);
s1.setSalario(5000.0);
Funcionario f = new Funcionario();
```

```
f.setSalario(2000.0);
Administrador a1 = new Administrador();
ev.setSalario(2500.0);
ControleSalario controle = new ControleSalario();
controle.registra(s1);
controle.registra(f);
controle.registra(a1);
System.out.println(controle.getSoma());
}
}
```

Pense que agora precisaremos retirar a bonificação do cálculo do salário, teríamos que alterar em todos os métodos :(. Isso está ocorrendo, porque não estamos utilizando o polimorfismo de forma correta.

O ideal, aqui, seria criar a classe ControleSalario da seguinte forma:

```
     public class ControleSalario {
private double soma;
public void registra(Funcionario f) {
double boni = f.getBonificacao();
this.soma = this.soma + boni;
}
public double getSoma() {
return soma;
}

     }
```

Em que o método *registra(Funcionario f)* pode receber qualquer objeto da hierarquia de funcionário, ou seja, Contador, Professor, Supervisor, Coordenador e, inclusive, classes que não existiam inicialmente, como a do Administrador ou que poderão ser acrescentadas no sistema. Resumindo só com uma implementação do método registra conseguimos registrar a bonificação dos “sistema”.

Outro ponto interessante é a sobrescrita de método, no qual sempre podemos redefinir um método em uma classe filha. Vamos para exemplo:

```
1.public class Administrador extends Funcionario {
2.private int senha;
3.public void setSenha(int senha) {
4.if(this.senha == senha) return true;
5.else return false;
6.}
7.public double getBonificacao() {
8.System.out.println(“Chamando o método bonificacao:”);
```

```
9.return super.getBonificacao() + super.getSalario();
10.}
11.}
```

Na linha 7, estamos redefinindo o método *getBonificacao()* no qual, para o caso do administrador, seria a mesma bonificação do Funcionário mais o 14o. salário. Suponha que o coordenador ganhasse a mesma do professor mais a do Funcionário, como seria o código? Tem ideia.

Vamos inicialmente para explicação da *classe* Administrador. Na linha 9, temos a execução do método **getBonificacao ()** da classe pai (através da palavra reservada **super** que indica superclasse a classe pai) mais o salário. Através do comando **super.getBonificacao()** foi possível acessar o método da classe pai (superclasse).

Aproveitando a oportunidade, além da palavra reservada **super**, temos, também, o método **super()** que se refere ao construtor da classe pai. No caso de acessar, por exemplo, o construtor da classe avô, bastaria colocar **super.super()**.

4.1 Encadeamento de construtores

Voltando para *Figura 5*, vemos que, para instanciar um objeto da classe coordenador, é necessário executar o construtor da classe pai (Professor) e avô (Funcionario), pois, como a classe coordenador herda as características das duas outras classes, o construtor tem o papel de inicializar os atributos. Sendo assim, temos a ideia do encadeamento de construtores, no qual, nesse ponto, podemos ter acesso aos atributos privados de uma classe. Por isso, falamos no circuito de aprendizagem anterior que mesmo os atributos privados são herdados, mas seu acesso é feito de forma indireta, por isso, quando o construtor da classe pai é executado, o atributo privado pode ser atribuído.

Importante!

Vale ressaltar que, mesmo que uma classe não tenha herança de forma direta no Java, ela herda os métodos e atributos da classe Object (classe base do Java). Ou seja, direta ou indiretamente todos os objectos advêm de **Object.**

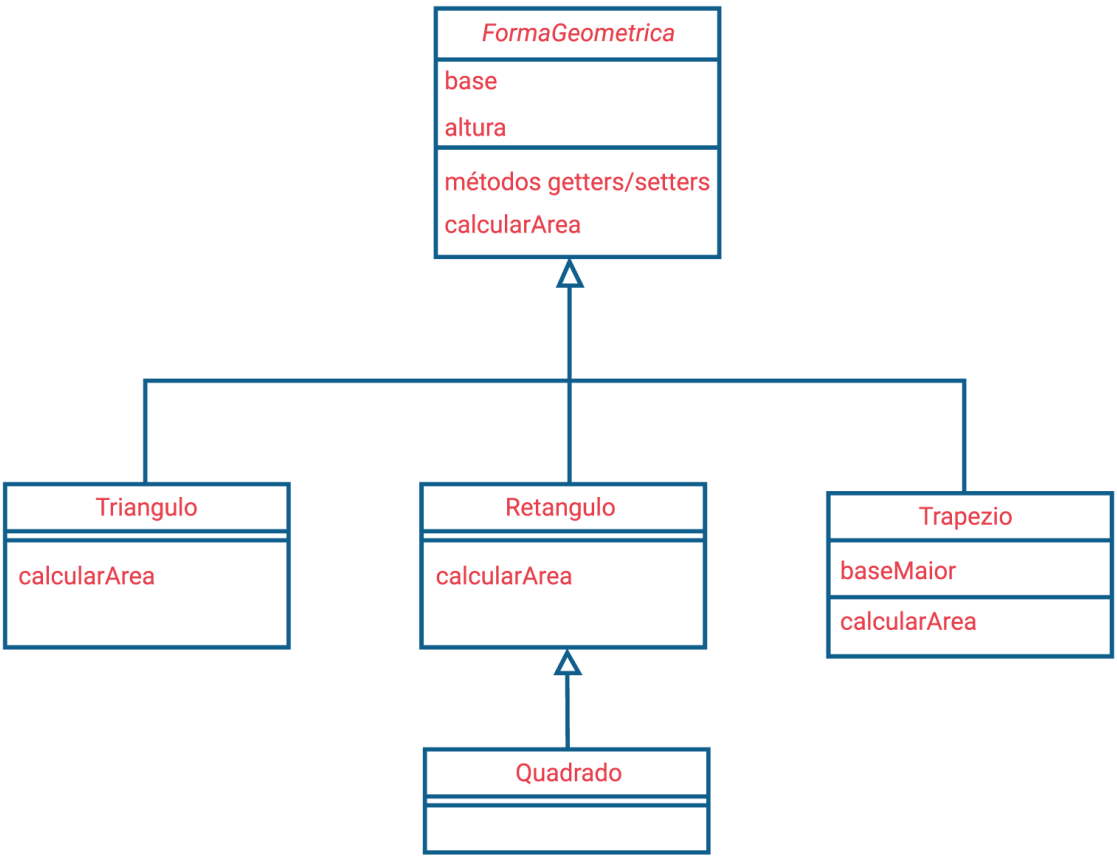
4.2 Classes Abstratas

Existem situações em que a implementação de alguns métodos nas classes mais genéricas não tem sentido, ou seja, naquela abstração e para o domínio do sistema esse método não tem como ser implementado, salvo que seja de forma *fake* (falsa, sem sentido

ou implementação vazia). Com certeza, meu caro leitor, você deve estar se perguntando, então, porque implementar esse método nessa classe e não só nas classes específicas, nas quais realmente ele seria necessário.

O motivo, aqui, de se implementar na classe pai seria dar visibilidade às referências e não ser necessário fazer a mudança de tipo como no circuito anterior. Outro ponto importante, é que podemos fazer com que essas classes que não têm todos os métodos implementados (concretos) sejam classes abstratas que não possam ser instanciadas. Tá confuso, não? São muitas muita informações! Então, vamos para um exemplo concreto para não ficar muito abstrato. Não resisti à piada!

Figura 6 - Exemplo do diagrama de classes: *Forma Geometrica*.



Fonte: Elaborada pelo autor (2021).

Como podemos ver na *Figura 6* a classe FormaGeometrica seria a superclasse com os métodos modificadores (*setters*) e acessores (*getters*) e o calcularArea(), porém, nesse nível de abstração, não teríamos como implementar o calcularArea, uma vez que, sem saber qual forma geométrica está sendo criada, não sabemos qual fórmula deverá ser executada (até que seria bom ter uma fórmula para calcular a forma geométrica de uma figura qualquer, mas sabemos que não é possível!). Então, nesse nível de abstração, a FormaGeometrica não implementaria esse método e, assim, uma ideia seria só implementar nas classes derivadas, porém "dificultaria" o

polimorfismo. Pois, o comando *FormaGeometrica f = new Trinagulo()*; *f.calcularArea()*, não funcionaria, uma vez que o compilador verificaria que a classe FormaGeometrica não tem o método em questão (*calcularArea()*). Outro ponto mais importante seria o raciocínio de "indicar" ou melhor "obrigar" que as classes concretas de FormaGeométrica (exemplo: Quadrado, Triangulo, Retangulo e Trapezio) tivessem o método *calcularArea()*, inclusive outras Formas Geométricas que poderiam ser adicionadas ao sistema depois. A ideia aqui seria, em vez de termos um manual ou uma "cartilha" que indicaria: "Caro programador, toda vez que criar uma classe derivada de FormaGeometrica implemente o método calcularArea(), uma vez que o sistema pressupõe que esse método existe etc... É muito mais fácil indicarmos que o método calcularArea é abstrato e portanto a classe é abstrata.

> Se existir pelo menos um método abstrato em uma classe, essa classe é abstrata e, por conseguinte, não pode ser instanciada. Inclusive uma classe que não tenha nenhum método abstrato pode ser abstrata e, dessa forma, ela não poderá ser instanciada diretamente.

Para uma classe ser concreta ela tem que implementar todos os métodos abstratos advindos das classes ascendentes.

Vamos para o código:

```
public abstract class FormaGeometrica {

     private float base, altura;
public abstract calcularArea();
// Implementação dos métodos gets e sets
public float getBase() {
return base;
}
//...
}
```

Como podemos ver na implementação da *FormaGeometrica* temos a palavra reservada *abstract* para indicar que a classe e o método *calcularArea()* são abstratos.

4.2 Interface

Em algumas situações a classe está em um nível de abstração muito elevado e, desse modo, não é possível implementar nenhum método concreto. Para esses casos podemos ter uma classe abstrata com todos os métodos abstratos ou implementar uma interface

(não confundir com interface gráfica).

Interface é uma forma de contrato que uma classe assina, ou seja, ela se "obriga" a implementar os métodos indicados pela interface. Vamos para exemplo (*#partiucodigo*):

```
public interface Veiculo {
public String getNome();
public String getId();
}

public interface Motor {
public String getModelo();
public String getFabricante();
}

public class Carro implements Veiculo,Motor {

private int id;
private String nome;
private String fabricante;
private String modelo;

public String getId() {
return id;
}

public String getNome() {
return nome;
}

public String getFabricante() {
return fabricante;
}

public String getModelo() {
return modelo;
}

}
```

Como podemos ver na implementação da classe Carro temos a assinatura de duas interfaces: Motor e Veiculo. Isso é possível, porque no java não existe a herança múltipla de classes, contudo, existe de interface. Na verdade, é uma maneira de simular herança múltipla. Outro ponto importante são as palavras chaves utilizadas quando trabalhamos com interfaces.

Para criar usá-se: "interface" e para "assinar" o contrato:" implements".

Voltar ao sumário

Resumo:

Iniciamos nosso estudo desse percurso aprendendo a diferença entre classe e objetos. Aqui, vale aquela pergunta, quem “nasceu” primeiro? Quem define quem? Aprendemos que as classes definem como os objetos se comportam e o que armazenam. Classes são como fossem uma planta de uma casa uma receita de um bolo. Os quais quando são executados (concretizados) criam os objetos que são reais. Estudamos, também, quais são os modificadores de acessos tanto da linguagem C++ como Java e a definição de construtores e destrutores que tem o papel de inicializar e destruir os objetos em sua “criação” e “remoção”.

No segundo circuito de aprendizagem aprofundamos um dos principais pilares da Orientação a Objetos o encapsulamento detalhando os modificadores: private, public, protected e default, além de definir os métodos acessores (getters) e modificadores (setters). Reforçando que não é uma boa prática de programação orientada a objetos simplesmente criar todos os atributos privados e gets e sets correspondentes. A ideia é criar apenas os métodos acessores e modificadores que realmente são necessários para o domínio do sistema em questão.

No circuito seguinte definimos e exemplificamos através de códigos e diagramas UML a herança simples e múltiplas. Vimos que em linguagens de programação com o Java onde a herança é implementada com árvores não temos a possibilidade de herança múltipla, apenas de forma indireta com a utilização de interfaces.

Por fim, definimos e exemplificamos outro pilar da POO, evidenciando a sobrecarga e sobrescrita de métodos na utilização de herança e polimorfismo.

Referências

EDELWEISS, Nina ; LIVI, Maria Aparecida Castro. Algoritmos e programação com exemplos em Pascal e C. Porto Alegre: Bookman, 2014. Disponível em: https://integrada. minhabiblioteca.com.br/books/9788582601907.(DIGITAL) (Cód.:612)

FOWLER, Martin. UML essencial: um breve guia para linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2011. Disponível em: https://integrada. minhabiblioteca.com.br/books/9788560031382. (DIGITAL) (Cód.:6903)

HORSTMANN, Cay. Conceitos de computação com Java. 5. ed. Porto Alegre: Bookman, 2009. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788577804078. (DIGITAL) (Cód.:2813)

MANZANO, José Augusto N. G. . Programação de computadores com Java. São Paulo: Érica, 2014. Disponível em: https://integrada.minhabiblioteca.com.br/books/ 9788536531137. (DIGITAL) (Cód.:30259)

SEBESTA, Robert W. Conceitos de linguagens de programação. 4. ed. Porto Alegre: Bookman, 2000.

SCHILDT, Herbert. Java para iniciantes: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015. Disponível em: https://integrada. minhabiblioteca.com.br/books/ 9788582603376. (DIGITAL) (Cód.:6269)

SOUZA, Marco A. Furlan; GOMES, Marcelo Marques; SOARES, Márcio Vieira; CONCILIO, Ricardo. **Algoritmos e lógica de programação: um texto introdutório para a engenharia.** 3ed. São Paulo: Cengage Learning, 2019

USINAGEM: o que é e qual a sua importância. **www.ccvindustrial.com.br**, 2019. Disponível em: <https://ccvindustrial.com.br/usinagem-o-que-e-e-qual-a-sua-importancia>. Acesso em: 22 de dez. De 2020.

WAZLAWICK, Raul Sidnei. Análise e design orientados a objetos para sistema de informação : modelagem com UML, OCL e IFML. 3. ed. Rio de Janeiro: Elsevier, 2015. Disponível em: https://integrada.minhabiblioteca.com.br/#/books/ 9788595153653/ cfi/6/2!/4/2@0.00:0.00. (DIGITAL) (Cód.:30192)

WINDER, Russel ; GRAHAM, Roberts (autor). Desenvolvendo software em Java. 3. ed. Rio de Janeiro: LTC, 2009. Disponível em: https://integrada.minhabiblioteca.com. br/ books/978-85-216-1994-9. (DIGITAL) (Cód.:3517)

AUTOR
MAIKOL MAGALHÃES RODRIGUES

Possui graduação em Ciência da Computação pela Universidade Estadual do Ceará (1998) e mestrado em Ciência da Computação pela Universidade Estadual de Campinas (2001). Atualmente é analista de sistemas - Serviço Federal de Processamento de Dados, professor assistente da Faculdade Farias Brito e professor assistênte da Universidade de Fortaleza. Tem experiência na área de Ciência da Computação, com ênfase em Modelos Analíticos e de Simulação, atuando principalmente nos seguintes temas: programação linear inteira, programação matemática, otimização combinatória, softwares de otimização e modelos de programação linear.

RESPONSABILIDADE TÉCNICA

UNIVERSIDADE DE FORTALEZA ENSINANDO E APRENDENDO
VRE Vice-Reitoria de Ensino de Graduação e Pós-Graduação
Universidade de Fortaleza | EAD

COORDENAÇÃO DA EDUCAÇÃO A DISTÂNCIA

Coordenação Geral de EAD
Douglas Royer

Coordenação de Ensino e Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Ensino e Aprendizagem
Carla Dolores Menezes de Oliveira

Supervisão de Planejamento Educacional
Ana Flávia Beviláqua Melo

Supervisão de Recursos EAD
Andrea Chagas Alves de Almeida

Supervisão de Operações e Atendimento
Mírian Cristina de Lima

Analista Educacional
Lara Meneses Saldanha Nepomuceno

Projeto Instrucional
Igor Gomes Rebouças

Revisão Gramatical
José Ferreira Silva Bastos

Identidade Visual / Arte
Francisco Cristiano Lopes de Sousa

Editoração / Diagramação
Régis da Silva Pereira

Produção de Áudio e Vídeo
Pedro Henrique de Moura Mendes

Programação / Implementação
Francisco Weslley Lima