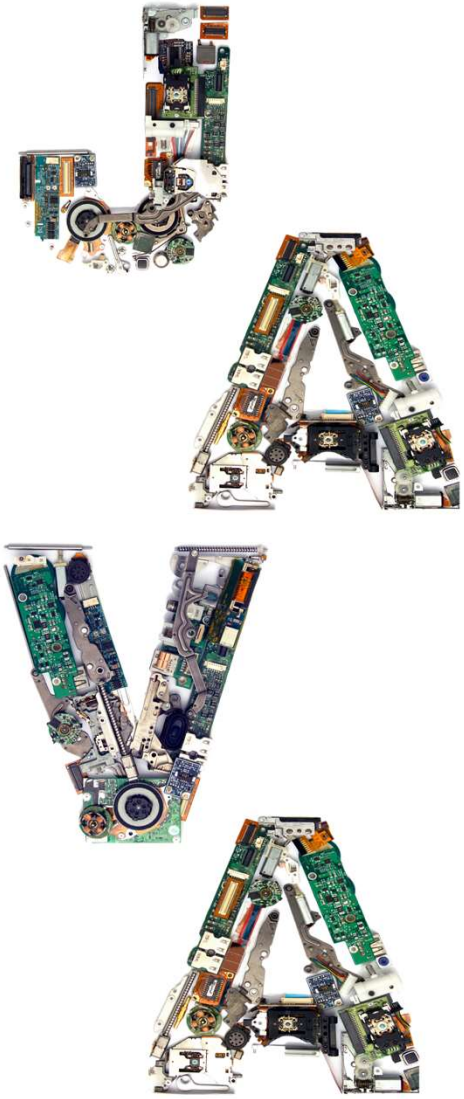
Programação Orientada a Objetos com Java e WEB

Tópicos Adicionais em Implementação de Classes, Métodos e Objetos



Modificador *final*

O modificador *final* pode ser utilizado na declaração de **classes**, **atributos** e **métodos**. **A utilização na declaração de classes e métodos será abordada no estudo de herança**.

O modificador *final* na declaração de um atributos faz com que o atributo seja uma **constante**, ou seja, o atributo tem um valor que não pode ser alterado durante a execução do programa. Sintaxe:

visibilidade final tipo NOME_DO_ATRIBUTO = valor;

será abordada em outro momento

tipos primitivos ou referência

escrito em letra maiúscula seguindo o padrão Java *Beans*

Modificador *final*

Exemplo:

```
public class ContaBancaria {
    String nome;
    double saldo;
    final double LIMITECREDITO = 500;
}
```

```
ContaBancaria conta = new ContaBancaria();
conta.LIMITECREDITO = 1500.00;
```

essa linha de código gera erro porque o valor de uma constante não pode ser alterado durante a execução de um programa

Modificador *final*

O modificador ***static*** pode ser utilizado na declaração de **atributos** e **métodos**. **Pode ser utilizado em classes, mas não é comum devido a aplicação (classes internas)**.

Sintaxe para atributos:

visibilidade ***static*** *tipo nomeDoAtributo;*

Sintaxe para métodos:

```
visibilidade static tipo nomeDoMétodo( parâmetros) {
    corpo do método
}
```

Modificador *final* – atributos

Utilizado quando o atributo está logicamente associado a classe e não os objetos. Por exemplo: o atributo rm de um aluno não pode ser estático porque cada aluno tem um rm único (cada objeto aluno deve ter uma cópia do atributo).

O atributo para armazenar o valor da passagem do metrô pode ser estático porque o valor padrão é o mesmo para todos os objetos.

Uma **variável estática** é instanciada apenas uma vez **por classe e não por objeto**. Você não precisa de um objeto para acessar a variável estática.

Modificador *final* – atributos

```
public class ContaBancaria {
    String nome;
    double saldo;
    static double limiteCredito = 500;
}

public static void main(String[] args) {
    ContaBancaria conta = new ContaBancaria();
    ContaBancaria.limiteCredito = 5500.00;
}
```

classe

variável *static*

Quando a classe ContaBancaria é carregada na memória, todas as suas variáveis *static* são inicializadas (já estão prontas para uso, sem que a classe seja instanciada)

variável *static* é acessada usando o nome da classe. Não há a necessidade de instanciar a classe

Modificador *final* – atributos

```
public class Teste {
    static int cont = 0;
    public static void main(String[] args) {
        System.out.println(Teste.cont);
    }
}
```

ou assim

```
public class Teste {
    static int cont = 0;
    public static void main(String[] args) {
        System.out.println(cont);
    }
}
```

Modificador *final* – atributos

```
public class Teste {
    int cont = 0;
    public static void main(String[] args) {
        System.out.println(Teste.cont);
    }
}
```

Esse código gera erro! Como a variável *cont* não é *static* ela não pode ser referenciada pelo nome da classe! A variável *cont* é um atributo e deve ser acessada por um objeto.

```
public class Teste {
    int cont = 0;
    public static void main(String[] args) {
        Teste teste = new Teste();
        System.out.println(teste.cont);
    }
}
```

A classe Teste é instanciada para que o atributo possa ser acessado.

Modificador *final* – métodos

O modificador ***static*** é sempre utilizado quando uma variável ou método terá a sua execução independente de objetos, ou seja, **são métodos e variáveis que não estão logicamente associados aos objetos da aplicação**.

Métodos estáticos (definidos com o modificador ***static***) basicamente tem duas aplicações:

- ❑ Trabalhar com variáveis definidas como estáticas.
- ❑ Definir **classes utilitárias**. **Classes utilitárias** são classes definidas com a finalidade de reunir variáveis e métodos globais, que devam estar disponíveis para todos os objetos da aplicação. Por exemplo, uma classe utilitária que armazena todas as operações de banco de dados de uma aplicação.

Modificador *final* – métodos

Independente de qual instância da classe executa o método, ele sempre se comportará do mesmo modo.

O comportamento do método não depende do estado (valores das variáveis de instância) de um objeto.

Os métodos da classe Math são estáticos: Math.sqrt(), Math.pow(), etc. Por exemplo, o cálculo da raiz quadrada de um número não depende de nenhum objeto (portanto os métodos são definidos como *static*).

Você precisa de um objeto, já que o método nunca será específico da instância? Porque não solicitar à classe para executar o método?

Modificador *final* – métodos

```
public class ContaBancaria {
    String nome;
    double saldo;
    static double limiteCredito = 500;

    public static void aumentarCredito() {
        limiteCredito *= 1.2;
    }
}
```

apesar de funcionar, não é a forma indicada

Como utilizar?

```
ContaBancaria cb = new ContaBancaria();
cb.aumentarCredito();

Ou

ContaBancaria.aumentarCredito();
```

Forma correta

Modificador *final* – métodos

```
public class Foo {
    int size = 42;
    public static void doMore() {
        int x = size;
    }
}
```

Uma variável não-*static* é um atributo e deverá ser acessada por meio de uma variável de referência.

ERRO!!O método *static* não é capaz de acessar uma variável de instância (não-*static*)

```
public class Bar {
    public void go(){ }

        public static void doMore(){
            go();
        }
}
```

ERRO!! O método *static* não é capaz de acessar um método não-*static*. Aqui vale a mesma observação feita para variáveis não estáticas.

Modificador *final* – métodos

```
public class Baz {
    static int count;

    static void woo() { }

    static void doMore() {
        woo();
        int x = count;
    }
}
```

A partir de um contexto (método) *static* pode-se acessar métodos e/ou variáveis estáticas.

O método *static* é capaz de acessar um método ou uma variável *static*.

Modificador *final* – métodos

```
public static void main(String[] args) {
    //corpo do método
}
```

Por que o método *main*() é estático?

O método *main*() é o primeiro método da aplicação que deve ser executado porque ele é responsável por iniciar (executar) a aplicação.

Para que ele possa ser carregado na memória antes dos outros métodos, deve ser definido como *static*. Tudo que é definido como *static* é carregado na memória antes do resto da aplicação (sem ter objetos).

Sobrecarga de métodos

Cada método possui uma **assinatura** (nome, tipo de retorno, o número e tipos de seus parâmetros).

Dois métodos podem ter o mesmo nome se eles tiverem diferentes números e/ou tipos de parâmetros. **Isso é chamado de sobrecarga de método.**

Sobrecarga é geralmente usada quando um método “trabalha” com tipos ou número de parâmetros diferentes.

Condição: dois ou mais métodos na mesma classe com o mesmo nome, mas com uma lista de parâmetros diferentes

Sobrecarga de métodos

```
public void meuMetodo(int p1) {...}

public void meuMetodo(String p1) {...}

public void meuMetodo(int p1, String p2) {...}

public void meuMetodo(String p1, int p2) {...}

public void meuMetodo(int p1, String p2, double p3) {...}

public int meuMetodo(int p1) {...}
```

CUIDADO com esse método!!!
Ele está sobrecarregado???

NÃO, porque a diferenciação deve estar nos parâmetros, independente do tipo de retorno do método.

Sobrecarga de métodos

Regras para a sobrecarga de métodos:

- **Métodos sobrecarregados DEVEM mudar a lista de parâmetros.**
- **Métodos sobrecarregados PODEM mudar o tipo de retorno.**
- **Métodos sobrecarregados PODEM mudar o modificador de acesso.**

Sobrecarga de métodos

```
public class Teste {
    int x;
    double y;

    public void potencia(int n) {
        x = Math.pow(x, n);
    }

    public void potencia(double n) {
        y = Math.pow(y, n);
    }
}
```

Sobrecarga → dois ou mais métodos com o mesmo nome variando a lista de parâmetros, independente do tipo de retorno.

Sobrecarga de métodos

O método *main()* pode ser sobrecarregado?

```
public class Teste {
	public static void main(String[] args) {
		main(1);
	}

	static void main(int i) {
		System.out.println("Sobrecarga");
	}
}
```

Não!! Porque a assinatura do método *main*() não pode ser alterada!

Método construtor

Desempenha papel essencial no processo de instanciação de uma classe.

Inicializador da classe que solicita à JVM a alocação de espaço em memória.

Os construtores são utilizados para inicializar os atributos com valores padrão ou com valores informados.

Entenda-se por inicializar os atributos como inicializar o estado de variáveis de instância.

São métodos especiais que são invocados juntamente com o operador new.

Método construtor

Construtores não possuem valor de retorno (nem mesmo *void*) e possuem o mesmo nome da classe

Toda vez que um objeto é instanciado, um construtor é chamado. Construtores são chamados apenas em tempo de execução.

Toda classe possui, por default, um constructor padrão: público e sem parâmetros

Podem ser definidos vários construtores para uma classe (SOBRECARGA DE CONSTRUTOR)

Método construtor

```
public Pessoa() { ...}
```
construtor padrão (público e sem parâmetros)

```
public Pessoa(long rg) {...}

public Pessoa(long rg, String nome) {...}

public Curso(int codigo) {...}

public Curso(int codigo, String nomeCurso) {...}
```

CONSTRUTORES SOBRECARREGADOS → dois ou mais métodos com o mesmo nome variando (obrigatoriamente) a lista de parâmetros (ou em quantidade e/ou tipo do parâmetro)

Método construtor

```
public class ContaBancaria {
    String nome;
    double saldo;
    static double limiteCredito = 500;

    public ContaBancaria(String n, double s) {
        nome = n;
        saldo = s;
    }

    public static void aumentarCredito() {
        limiteCredito *= 1.2;
    }
}
```

Método construtor

Regras para codificação do construtor:

- O nome do construtor deve coincidir com o nome da classe.
- Os construtores não tem ter um tipo de retorno (nem mesmo *void*).
- Os construtores podem usar qualquer modificador de acesso (visibilidade).
- É válido ter um método qualquer com o mesmo nome da classe, mas inútil.
- Um construtor é chamado quando um objeto é instanciado, ou, um construtor poderá ser chamado a partir de outro construtor.

Método construtor

Se você não codificar um construtor o Java codifica o construtor padrão (default).

O construtor default (padrão) somente é criado quando nenhum outro construtor for definido para a classe;

Se um construtor for codificado na classe, o Java não codificará o construtor padrão.

Método construtor – valor padrão

Tipo	Conteúdo	Valor Padrão	Tamanho em Bits	Mínimo	Máximo
boolean	Valor lógico	False	8	-	-
char	Caractere unicode	\u0000	16	\u0000	\uFFFF
byte	Inteiro com sinal	0	8	2^{-7}	$2^{-7}-1$
short	Inteiro com sinal	0	16	2^{-15}	$2^{-15}-1$
int	Inteiro com sinal	0	32	2^{-31}	$2^{-31}-1$
long	Inteiro com sinal	0	64	2^{-63}	$2^{-63}-1$
float	Número real	0.0	32	IEEE 754	IEEE 754
double	Número real	0.0	64	IEEE 754	IEEE 754

Método construtor – sobrecarga

Como ter certeza se o método *Pessoa()* é realmente o construtor da classe?

Lembrando das regras do construtor: método com o mesmo nome da classe e sem tipo de retorno (nem mesmo *void*)

```
public class Pessoa {
    String nome;
    int idade;

    public Pessoa(String nome) {
        this.nome = nome;
    }

    public Pessoa (String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
}
```

referência *this* → utilizada neste caso para diferenciar o parâmetro do atributo (as variáveis têm o mesmo nome)

Referência *this*

É uma referência para a instância corrente, ou seja, para o objeto ativo na memória.

Resumidamente, ***this*** sempre será a própria classe ou o objeto já instanciado. Esse conceito de autorreferência é importante para criar métodos construtores sobrecarregados e métodos que acessam outros métodos.

A referência ***this*** pode ser usada em três situações:

- Para diferenciar um atributo (variável da classe) de uma variável local ou de um parâmetro.
- Para referenciar o objeto corrente (objeto ativo na memória) ou passar o objeto corrente como argumento para um método.
- Para chamar outro construtor.

Referência *this*

A referência *this* também pode ser usada para chamar outro construtor dentro da mesma classe.

```
public class Aluno {
    int rm;
    String nome;

    public Aluno(int rm) {
        this(rm, null);
    }

    public Aluno(int rm, String nome) {
        this.rm = rm;
        this.nome = nome;
    }
}
```

referência *this* chamando o construtor *Aluno* sobrecarregado.

referência *this* utilizada para diferenciar variável local da variável de instância.

Referência *this*

```
public class Aluno {
     int rm;
     String nome;

     public Aluno(int rm, String nome) {
          this.rm = rm;
          this.nome = nome;
     }

     public void meuTeste() {
          imprimir(this);
     }

     private void imprimir(Aluno aluno) {
          System.out.println("RM -> "+aluno.rm);
          System.out.println("Nome -> "+aluno.nome);
     }
}
```

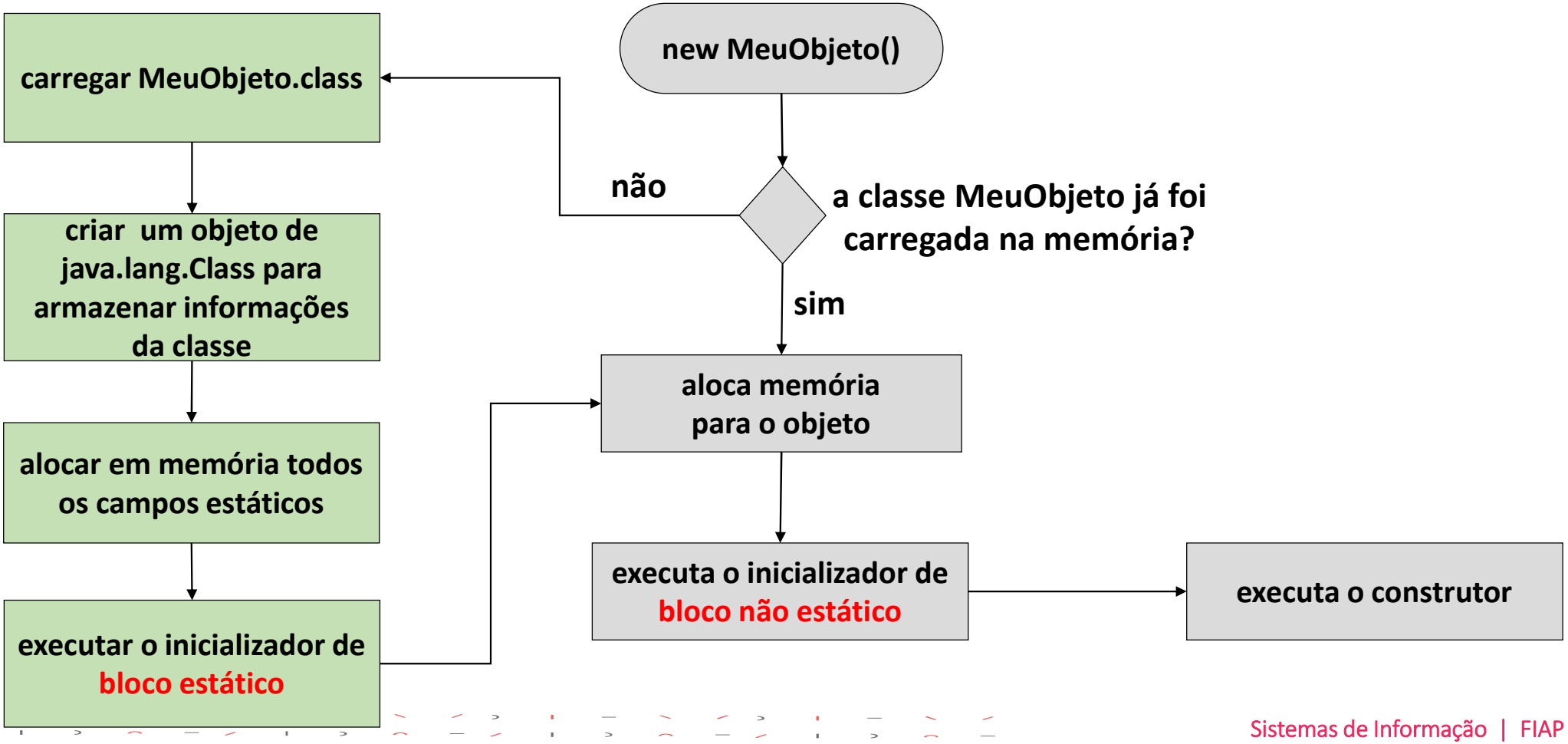
construtor inicializa os atributos do objeto que está sendo instanciado

referência *this* diferencia a variável local da variável de instância

como o método meuTeste() é chamado? → objetoAluno.meuTeste()

this referencia o objeto que fez a chamada do método

Ciclo de vida de um objeto



Ciclo de vida de um objeto

Em um programa Java, objetos são criados e inicializados (com o construtor), são manipulados e podem ser destruídos em determinado momento.

Diferente de outras linguagens de programação como C e C++, a linguagem de programação Java faz o gerenciamento automático da memória, chamado de **coleta de lixo**.

A **finalidade da coleta de lixo** é excluir objetos que não são mais utilizados, ou seja, que não possam mais ser alcançados (referenciados).

A JVM decide exatamente quando executar o coletor de lixo, mas você (programador) pode ***sugerir*** que seja feita essa operação.

Ciclo de vida de um objeto

Para que os objetos possam ser coletados como lixo, eles devem ser considerados *elegíveis*. Um objeto é *elegível* quando não é mais referenciado.

```
public class Aluno {
    int rm;
    String nome;

    public Aluno(int rm, String nome) {
        this.rm = rm;
        this.nome = nome;
    }
}
```

```
public class Principal {
    public static void main(String[] args) {
        Aluno aluno = new Aluno(10, "Selmini");
        System.out.println(aluno.nome);
        aluno = null;
    }
}
```

ao atribuir *null* para a variável de referência, o objeto se torna elegível para a coleta de lixo.

Ciclo de vida de um objeto

```
public class Principal {
   public static void main(String[] args) {
      Aluno aluno1 = new Aluno(10, "Selmini");
      Aluno aluno2 = new Aluno(20, "Flávio");
      aluno1 = aluno2;
   }
}
```

redireciona *aluno*1 para *aluno*2

o objeto referenciado por *aluno*1 está elegível para a coleta de lixo, uma vez que não é mais referenciado por nenhuma variável.

Ciclo de vida de um objeto

```
public class Principal {
    public static void main(String[] args) {
        Aluno aluno = criarObjeto();
        System.out.println(aluno.nome);
    }

    public static Aluno criarObjeto() {
        Aluno aluno = new Aluno(55, "Carlos");
        return aluno;
    }
}
```

a variável *aluno* referencia o objeto instanciado no método criarObjeto(), portanto, o objeto instanciado não está elegível para a coleta de lixo.

assim que o método criarObjeto() finalizar a execução, a variável *aluno* será destruída porque é uma variável local ao método

Ciclo de vida de um objeto

```
public class Ilha {
    Ilha i;
    public static void main(String[] args) {
        Ilha i2 = new Ilha();
        Ilha i3 = new Ilha();
        Ilha i4 = new Ilha();

        i2.i = i3;
        i3.i = i4;
        i4.i = i2;

        i2 = null;
        i3 = null;
        i4 = null;

        //quando chegar aqui quem está elegível?
    }
}
```

Os objetos referenciados por $i2$, $i3$ e $i4$ são elegíveis para a coleta de lixo porque não tem mais nenhum vínculo com o meio externo.

Ciclo de vida de um objeto

```
public class Lixo {
   public static void main(String[] args) {

      Runtime rt = Runtime.getRuntime();

      System.out.println("Memória total da JVM -> "+rt.totalMemory());
      System.out.println("Antes da coleta -> "+rt.freeMemory());
      Aluno aluno = null;
      for(int i = 0; i < 10000; i++) {
         aluno = new Aluno(i, "xyz");
         aluno = null;
      }
      System.out.println("Memória total da JVM -> "+rt.totalMemory());

      rt.gc();

      System.out.println("Depois da coleta -> "+rt.freeMemory());
   }
}
```

retorna o objeto para acesso direto a JVM

sugere a execução do coletor de lixo

Bibliografia

- DEITEL, H. M., DEITEL, P. J. **JAVA como programar**. 10ª edição. São Paulo: Prentice-Hall, 2010.
- SCHILDT, H. **Java para Iniciantes – Crie, Compile e Execute Programas Java Rapidamente**. 6ª Edição, Editora Bookman, Porto Alegre, RS, 2015.

Bibliografia

- KNUDSEN, J., NIEMEYER, P. **Aprendendo Java**. Rio de Janeiro: Editora Elsevier Campus, 2000.
- FLANAGAN, D. **Java – o guia essencial**. Porto Alegre: Editora Bookman, 2006.

Bibliografia

- ARNOLD, K., GOSLING, J., HOLMES, D., **Java programming language**. 4th Edition, Editora Addison-Wesley, 2005.

- JANDL JUNIOR, P. **Introdução ao Java**. São Paulo: Editora Berkeley, 2002.