Faculdade de Computação - FACOM

Bacharelado em Sistemas de Informação

FACOM32305 - Programação Orientada a Objetos

Prof. Thiago Pirola Ribeiro

Sumário

Armazenamento em memória I

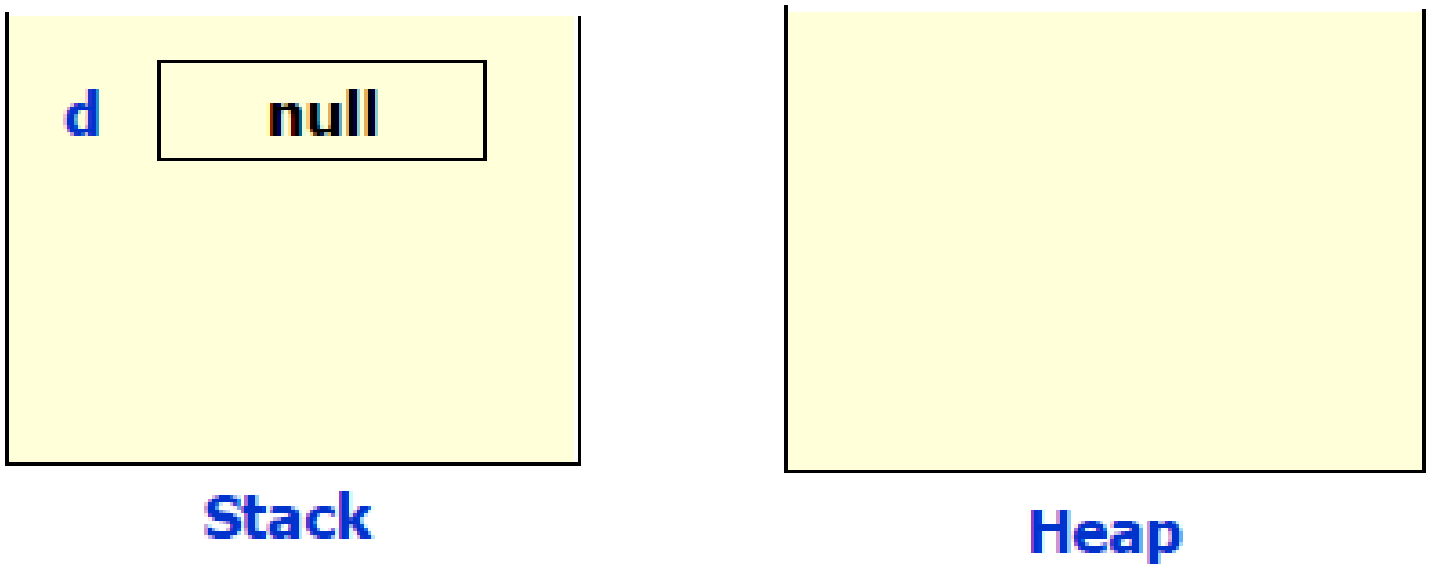
Variáveis de tipos primitivos ou referências são criados em uma área de memória conhecida como Stack (pilha);
Por sua vez, objetos são criados em área de memória conhecida como Heap (monte). Uma instrução `new Zzzzz()`:

1. Aloca memória para a variável de referência ao objeto na pilha e inicia-a com valor null;
2. Executa construtor, aloca memória na *heap* para o objeto e inicia seus campos (atributos);
3. Atribui endereço do objeto na *heap* à variável de referência do objeto na pilha.

Armazenamento em memória

1

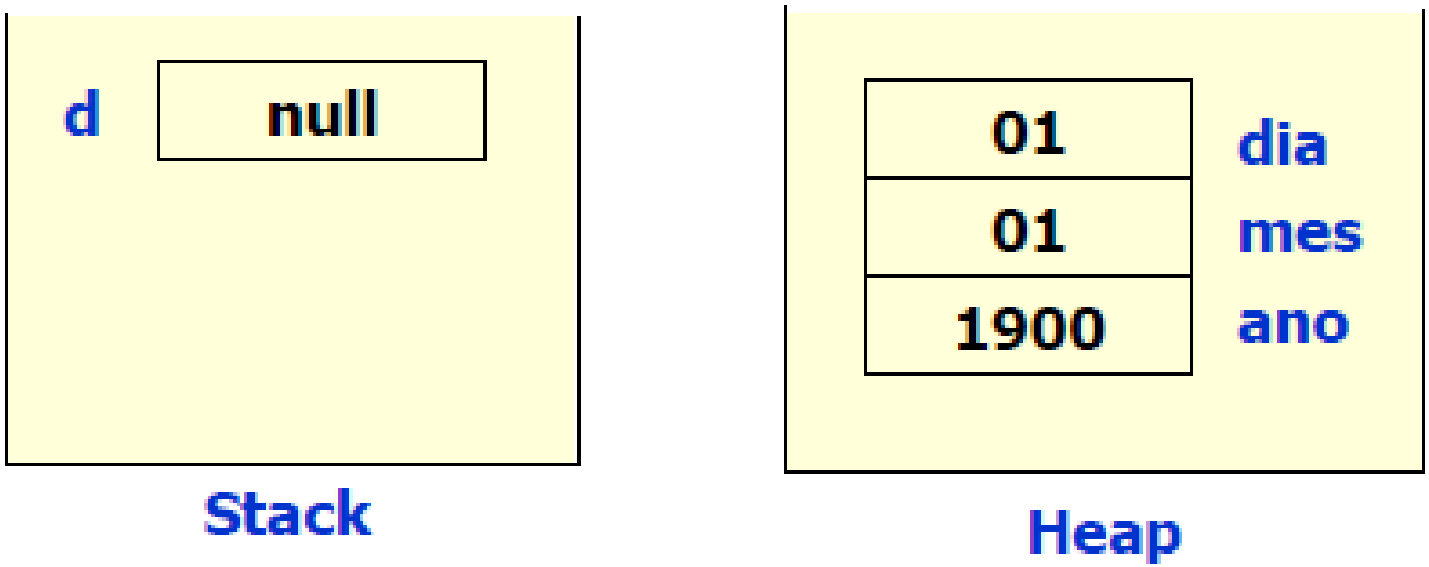
```
MinhaData d = new MinhaData (17, 3, 2009);
```



Armazenamento em memória

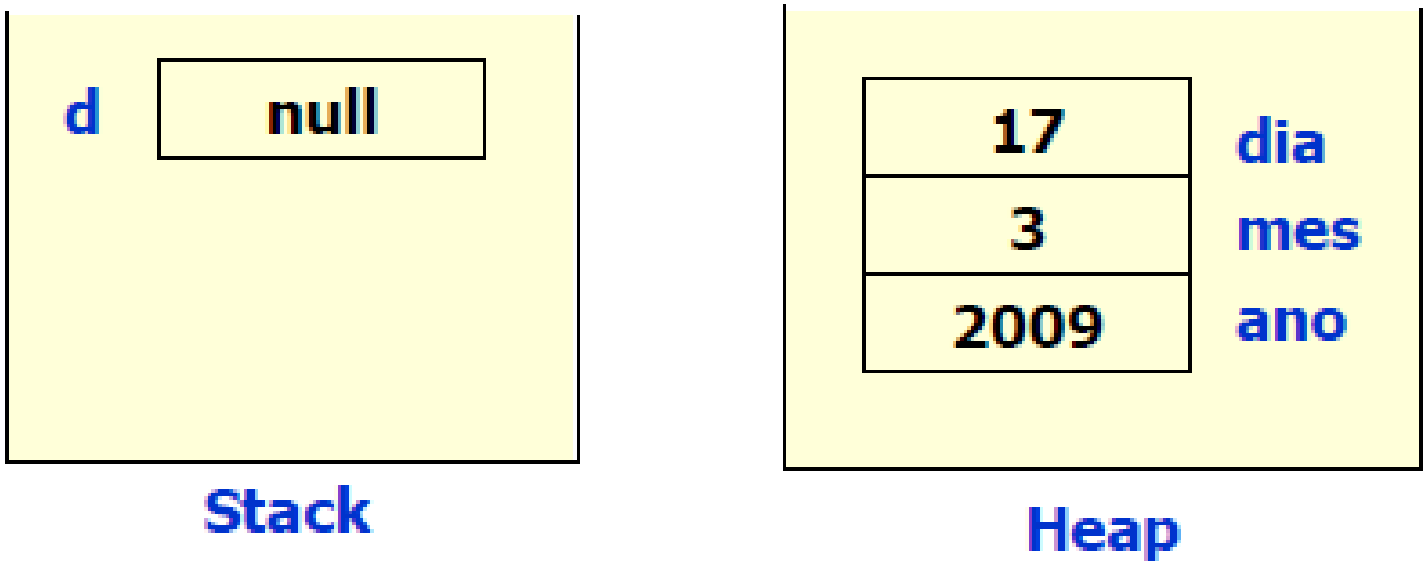
2

```
MinhaData d = new MinhaData (17, 3, 2009);
```



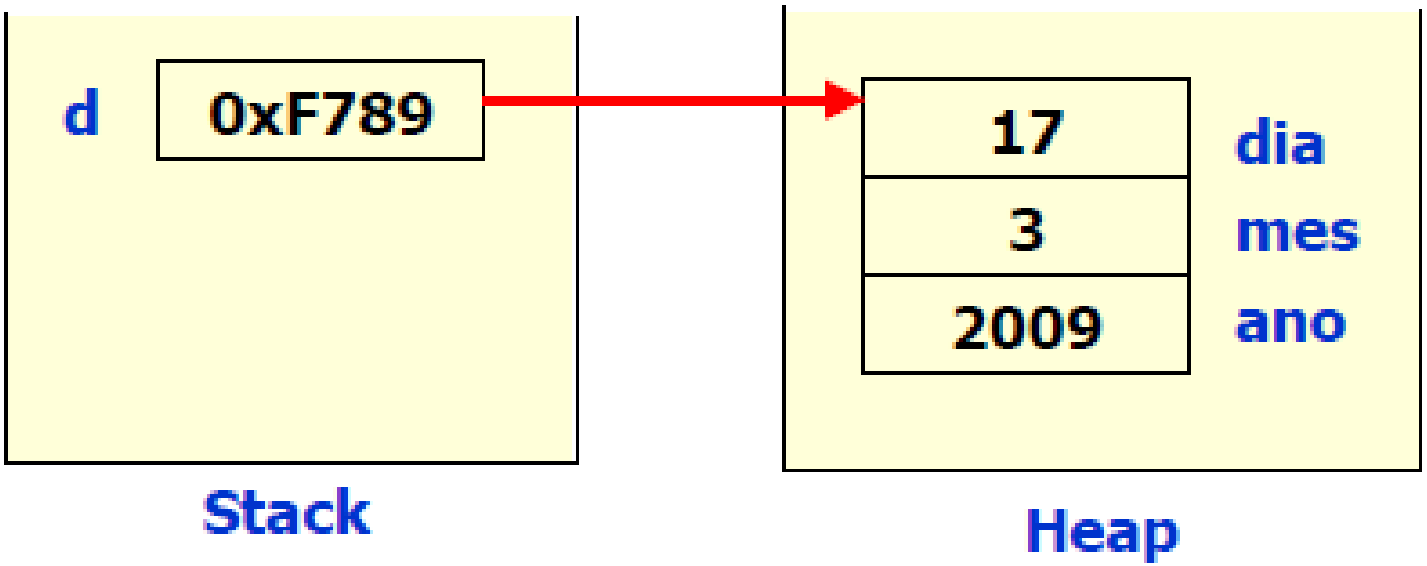
3

```
MinhaData d = new MinhaData (17, 3, 2009);
```



4

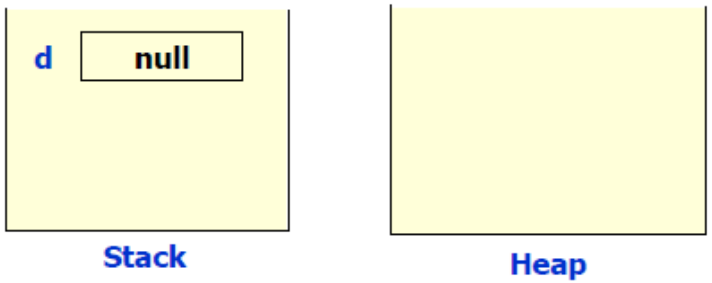
```
MinhaData d = new MinhaData (17, 3, 2009);
```



Armazenamento em memória

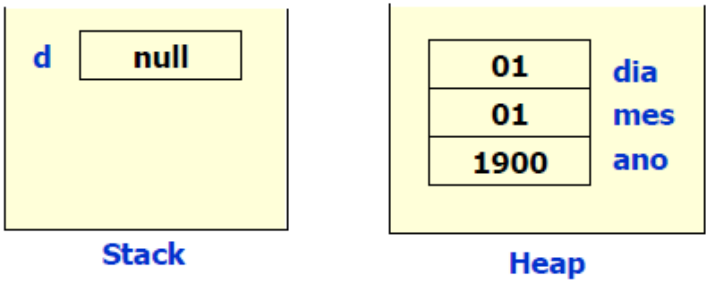
1

```
MinhaData d = new MinhaData (17, 3, 2009);
```



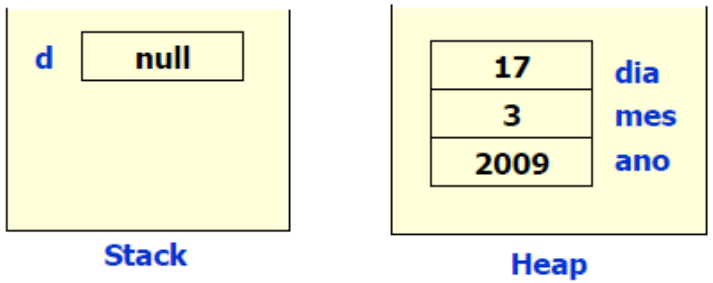
2

```
MinhaData d = new MinhaData (17, 3, 2009);
```



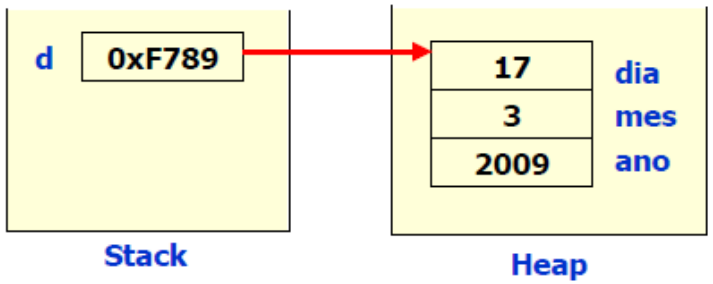
3

```
MinhaData d = new MinhaData (17, 3, 2009);
```



4

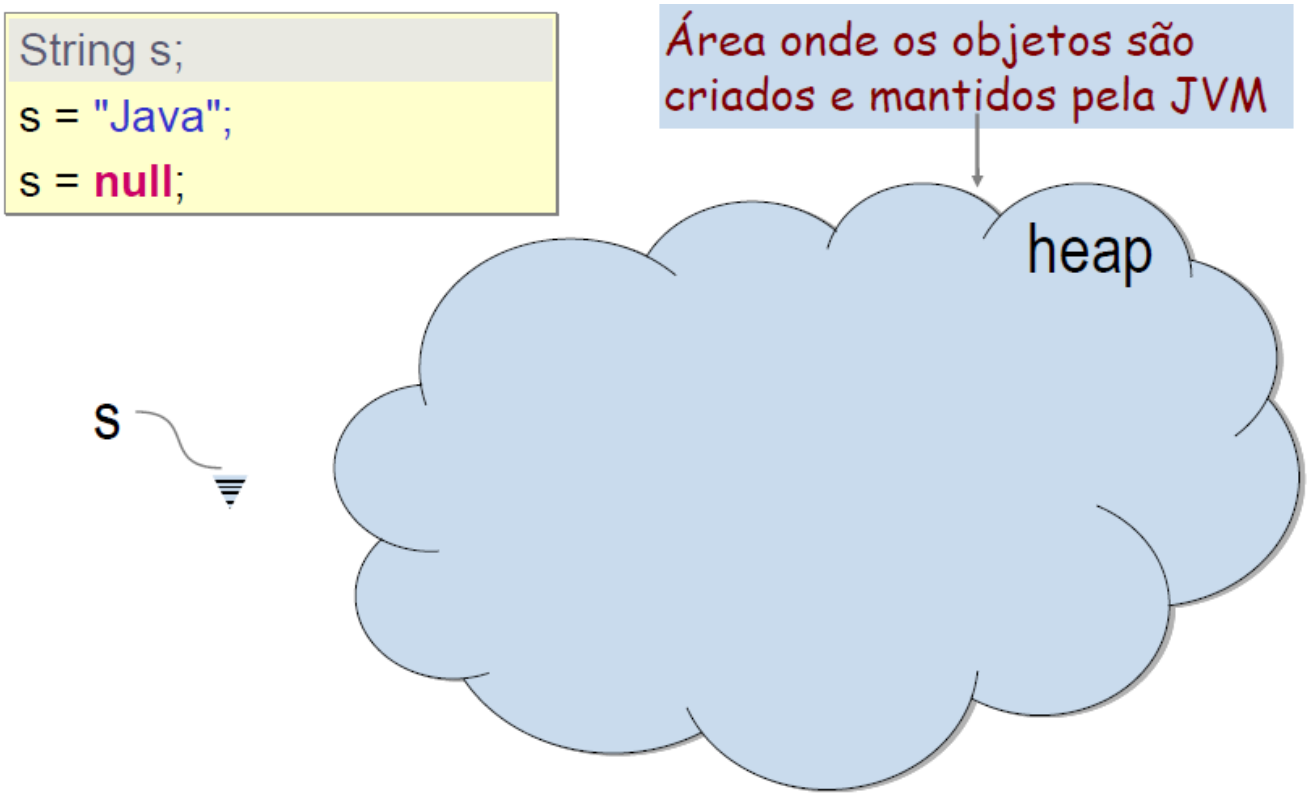
```
MinhaData d = new MinhaData (17, 3, 2009);
```



Coleta de lixo I

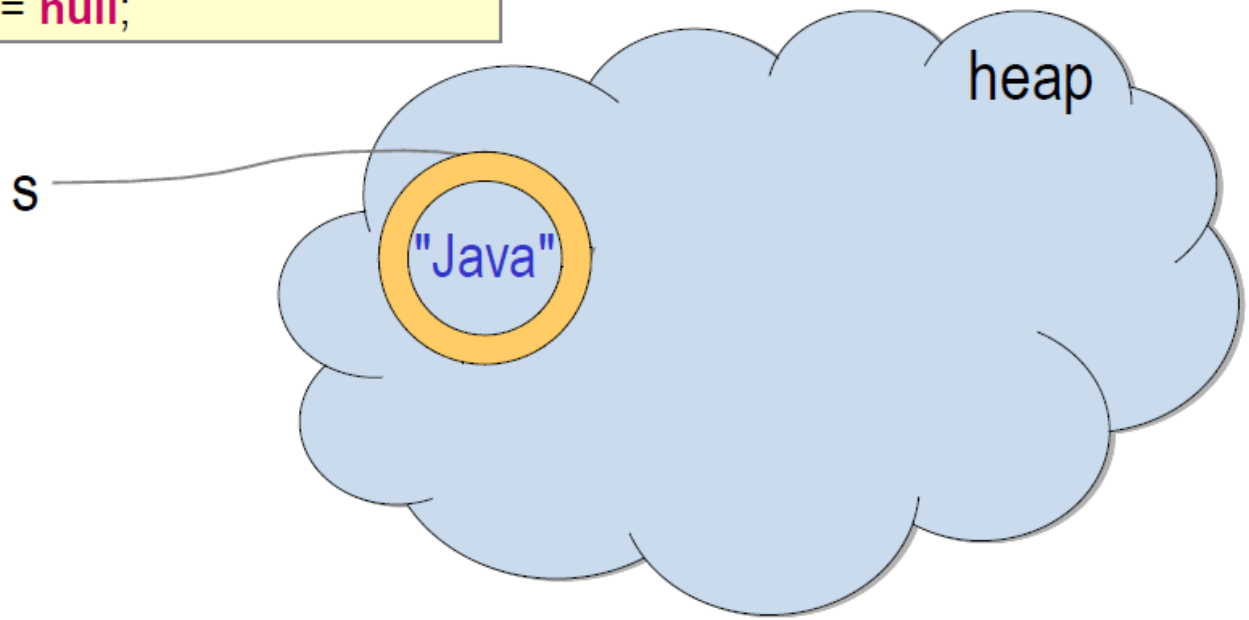
- Periodicamente, a JVM realiza uma coleta de lixo:
  - Retorna partes de memória não usadas;
  - Objetos não referenciados;
  - Não existem então primitivas para alocação e desalocação de memória (como `calloc`, `free`, etc).

Coleta de lixo II

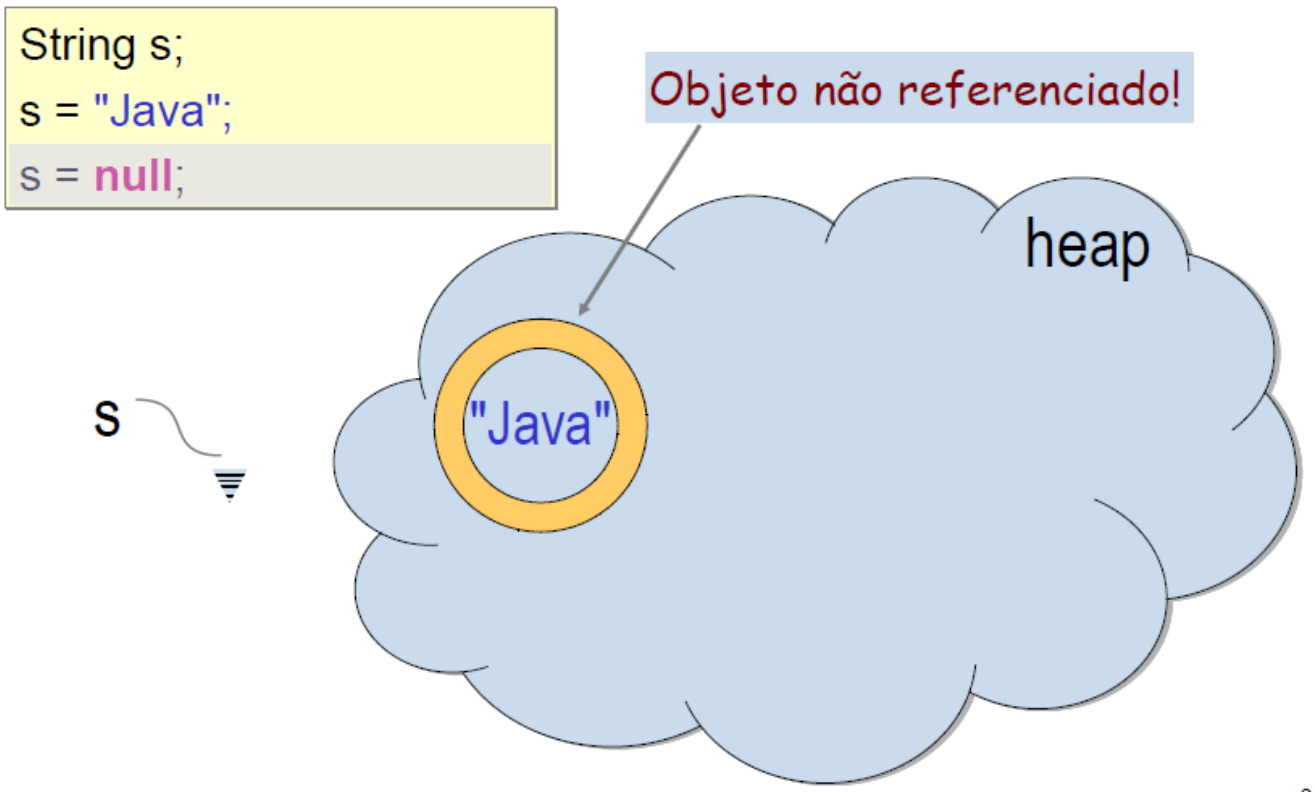


Coleta de lixo III

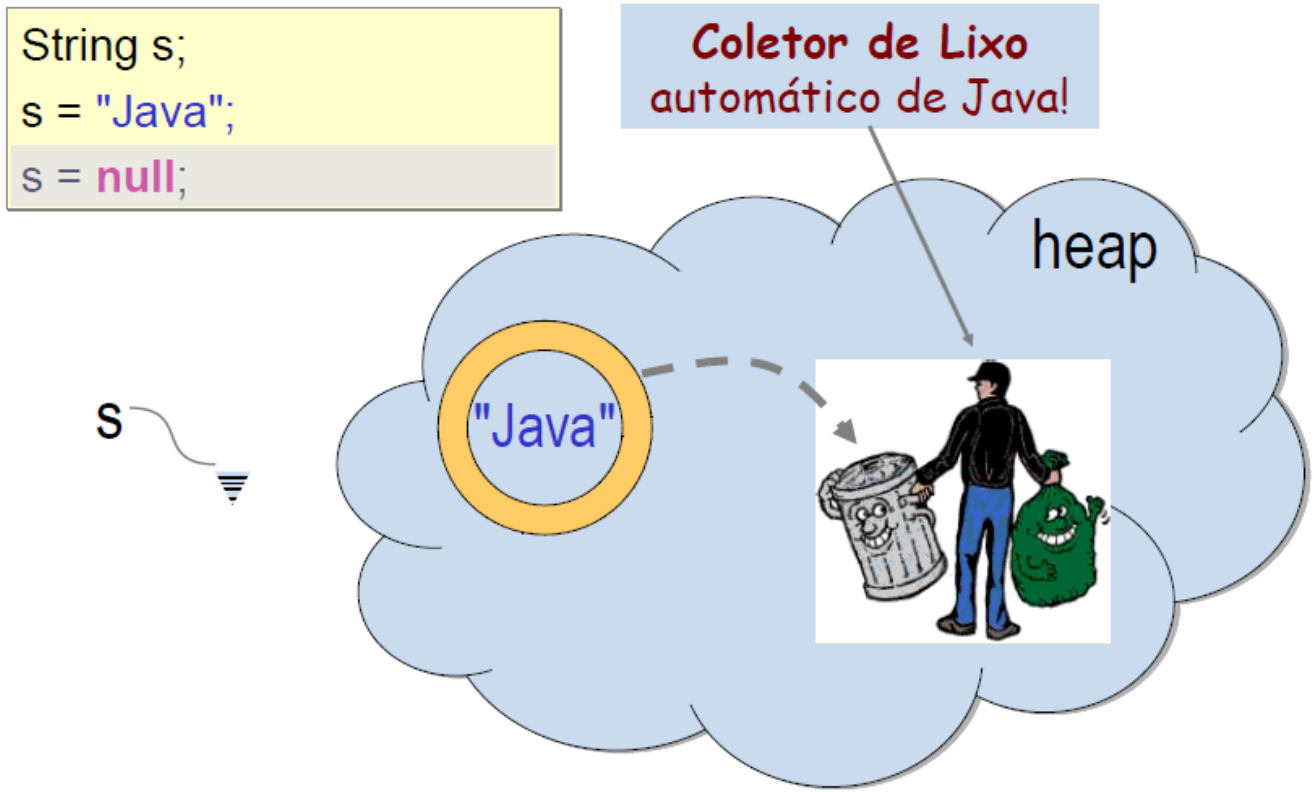
```
String s;
s = "Java";
s = null;
```



Coleta de lixo IV



Coleta de lixo V



Encapsulamento e modificadores de acesso

Encapsulamento e modificadores de acesso

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos.

Encapsulamento e modificadores de acesso

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos.

- O encapsulamento é implementado através dos modificadores de acesso;

Encapsulamento e modificadores de acesso

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos.

- O encapsulamento é implementado através dos modificadores de acesso;
  - São palavras reservadas que permitem definir o encapsulamento de classes, atributos e métodos;

Encapsulamento e modificadores de acesso

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos.

- O encapsulamento é implementado através dos modificadores de acesso;
  - São palavras reservadas que permitem definir o encapsulamento de classes, atributos e métodos;
- **Modificadores de acesso**: `public`, `private`, `protected`.

Encapsulamento e modificadores de acesso

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos.

- O encapsulamento é implementado através dos modificadores de acesso;
  - São palavras reservadas que permitem definir o encapsulamento de classes, atributos e métodos;
- **Modificadores de acesso**: `public`, `private`, `protected`.
- Quando se omite, o acesso é do tipo *package-only*.

Encapsulamento e modificadores de acesso

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos.

- O encapsulamento é implementado através dos modificadores de acesso;
  - São palavras reservadas que permitem definir o encapsulamento de classes, atributos e métodos;
- **Modificadores de acesso**: `public`, `private`, `protected`.
- Quando se omite, o acesso é do tipo *package-only*.
- Classes: apenas `public` – ou omitido (*package-only*) – é permitido (exceção para classes internas).

Encapsulamento e modificadores de acesso

- *Package-only*

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.
- `public`

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.
- `public`
  - Permite acesso irrestrito a partir de qualquer classe (mesmo que estejam em outros arquivos);

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.
- `public`
  - Permite acesso irrestrito a partir de qualquer classe (mesmo que estejam em outros arquivos);
  - Único que pode ser usado em classes (externas).

Encapsulamento e modificadores de acesso

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.
- `public`
  - Permite acesso irrestrito a partir de qualquer classe (mesmo que estejam em outros arquivos);
  - Único que pode ser usado em classes (externas).
- `private`

Encapsulamento e modificadores de acesso

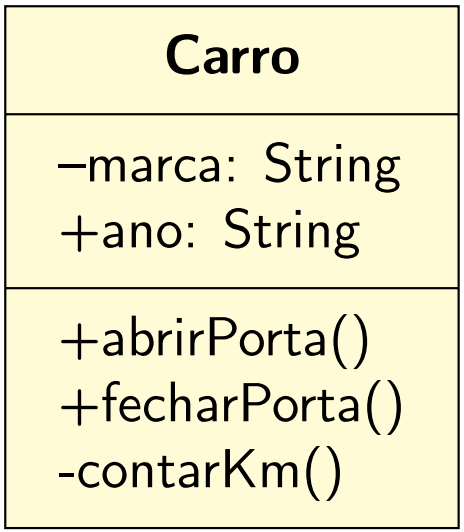
- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.
- `public`
  - Permite acesso irrestrito a partir de qualquer classe (mesmo que estejam em outros arquivos);
  - Único que pode ser usado em classes (externas).
- `private`
  - Permite acesso apenas por objetos da própria classe. O elemento é visível apenas dentro da classe onde está definido.

Modificadores de Acesso

Especificador	Classe	Subclasse	Pacote	Todos
`public`	X	X	X	X
`private`	X			
`protected`	X	X	X	
`package`	X		X	

Encapsulamento e diagrama de classes I

- +: `public` – visível em qualquer classe;
- –: `private` – visível somente dentro da classe.
- #: `protected` – visibilidade associada à herança




```
1  public class Carro
2  {
3      private String marca;
4      public String ano;
5      public void abrirPorta()
6      {
7          //corpo do método
8      }
9      public void fecharPorta()
10     {
11         //corpo do método
12     }
13     private void contarKm()
14     { /*corpo do método*/ }
15 }
```

Encapsulamento e diagrama de classes II

```
1  public class Carro
2  {
3      private String marca;
4      public String ano;
5      public void abrirPorta()
6      {
7          //corpo do método
8      }
9      public void fecharPorta()
10     {
11         //corpo do método
12     }
13     private void contarKm()
14     { /*corpo do método*/ }
15 }
```

```
1  public class UsaCarro
2  {
3      public static void main(
        String args[])
4      {
5          Carro car1 = new Carro()
        ;
6          car1.ano = "2000";
7          car1.marca = "Fiat";
8
9          car1.fecharPorta();
10         car1.contarKm();
11     }
12 }
```

Erros de semântica: linhas 7 e 10.

Encapsulamento e métodos

- Um método `public` pode ser invocado (chamado) dentro da própria classe, ou a partir de qualquer outra classe;

- Um método `private` é acessível apenas dentro da classe a que pertence.

Encapsulamento e atributos

- Atributos públicos podem ser acessados e modificados a partir de qualquer classe;

Encapsulamento e atributos

- Atributos públicos podem ser acessados e modificados a partir de qualquer classe;
- A menos que haja razões plausíveis, os atributos de uma classe devem ser definidos como `private`;

Encapsulamento e atributos

- Atributos públicos podem ser acessados e modificados a partir de qualquer classe;
- A menos que haja razões plausíveis, os atributos de uma classe devem ser definidos como `private`;
- Tentar acessar um componente privado de fora da classe resulta em erro de compilação.

Encapsulamento e atributos

Mas então como acessar atributos, ao menos para consulta (leitura)?

Encapsulamento e atributos

Mas então como acessar atributos, ao menos para consulta (leitura)?

- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar *dois métodos*;

Encapsulamento e atributos

Mas então como acessar atributos, ao menos para consulta (leitura)?

- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar *dois métodos*;
- Os dois métodos são definidos na própria classe onde o atributo se encontra;

Encapsulamento e atributos

Mas então como acessar atributos, ao menos para consulta (leitura)?

- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar *dois métodos*;
- Os dois métodos são definidos na própria classe onde o atributo se encontra;
- Um dos métodos retorna o valor da variável

Encapsulamento e atributos

Mas então como acessar atributos, ao menos para consulta (leitura)?

- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar *dois métodos*;
- Os dois métodos são definidos na própria classe onde o atributo se encontra;
- Um dos métodos retorna o valor da variável
- Outro método muda o seu valor;

Encapsulamento e atributos

Mas então como acessar atributos, ao menos para consulta (leitura)?

- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar *dois métodos*;
- Os dois métodos são definidos na própria classe onde o atributo se encontra;
- Um dos métodos retorna o valor da variável
- Outro método muda o seu valor;
- Padronizou-se nomear esses métodos colocando a palavra *get* ou *set* antes do nome do atributo.

Métodos *getters* e *setters*

- Com atributos sendo `private`, é frequente usar métodos acessores/modificadores (*getters*/*setters*) para manipular atributos;

Métodos *getters* e *setters*

- Com atributos sendo `private`, é frequente usar métodos acessores/modificadores (*getters*/*setters*) para manipular atributos;
- Porém, devemos ter cuidado para não quebrar o encapsulamento:

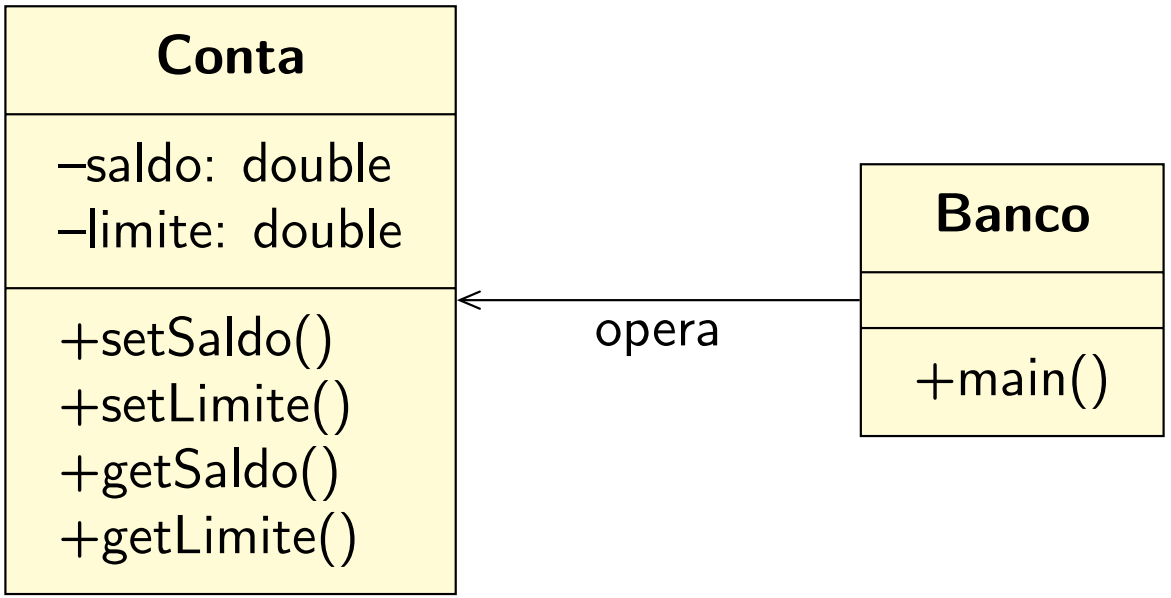
Métodos *getters* e *setters*

- Com atributos sendo `private`, é frequente usar métodos acessores/modificadores (*getters*/*setters*) para manipular atributos;
- Porém, devemos ter cuidado para não quebrar o encapsulamento:

Se uma classe chama `objeto.getAtrib()`, manipula o valor do atributo e depois chama `objeto.setAtrib()`, o atributo é essencialmente público. De certa forma, estamos quebrando o encapsulamento!

Exemplo – simulação de um banco (v0)

Construa um programa para simulação de um banco que possui apenas uma `conta`, com os atributos privados `saldo` e `limite`. Utilize métodos *getters* e *setters* para manipular os valores dos atributos e visualizá-los. Uma entidade `banco` é responsável pela criação da conta e sua operação.



Classe Conta.java (versão 0)

```
1  public class Conta {
2      private double limite;
3      private double saldo;
4      public double getSaldo() {
5          return saldo;
6      }
7      public void setSaldo(double x) {
8          saldo = x;
9      }
10     public double getLimite() {
11         return limite;
12     }
13     public void setLimite(double y) {
14         limite = y;
15     }
16 }
```

Questão

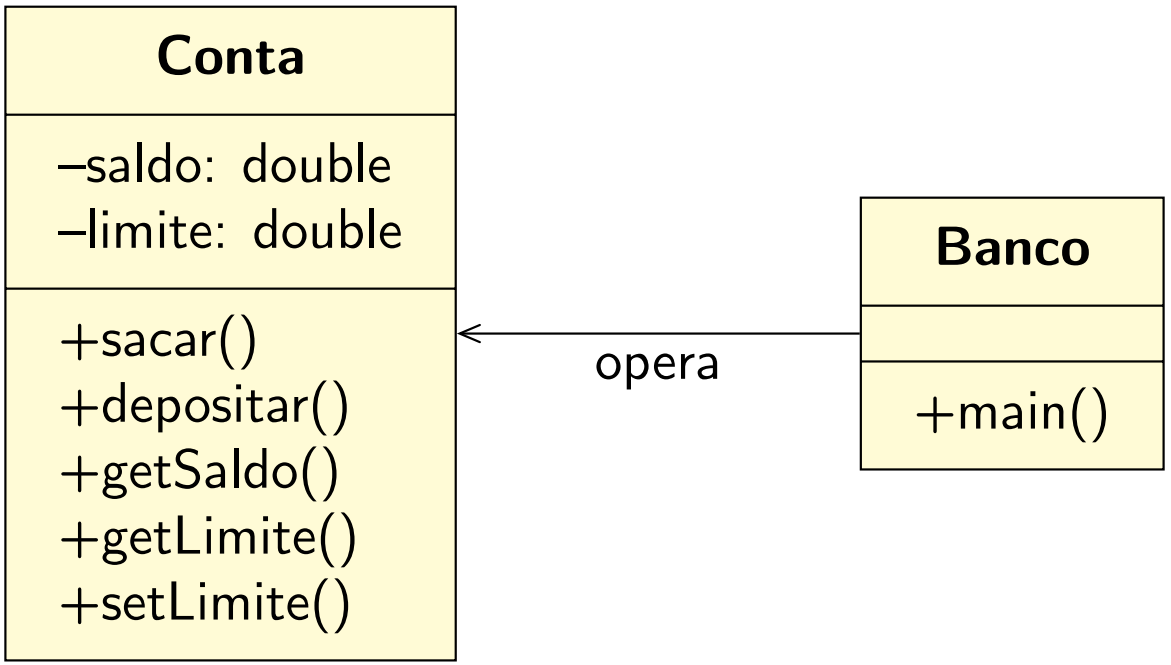
Esta classe permite alterar seus atributos como se fossem públicos!

Classe Banco.java (versão 0)

```
1  public class Banco
2  {
3      public static void main ( String args[] )
4      {
5          Conta c1 = new Conta();
6          c1.setSaldo(  1000 );
7          c1.setLimite( 1000 );
8          double saldoAtual = c1.getSaldo();
9          System.out.println( "Saldo atual é " + saldoAtual );
10         double limiteConta = c1.getLimite();
11         System.out.println( "Limite é " + limiteConta );
12     }
13 }
```

Exemplo – simulação de um banco (v1)

Construa um programa para simulação de um banco que possui apenas uma `conta`, com os atributos privados `saldo` e `limite`. Uma entidade `banco` é responsável pela criação da conta e sua operação. Ela pode sacar e depositar dinheiro da conta, visualizar seu saldo atual, assim como verificar e atualizar o limite.



Classe Conta.java (versão 1)

```
1  public class Conta {
2      private double saldo;
3      private double limite;
4      public void depositar(
         double x )
5      {
6          saldo = saldo + x;
7      }
8      public void sacar( double x
         )
9      {
10         saldo = saldo - x;
11     }
12     public double getSaldo()
13     {
14         return saldo;
15     }
16     public void setLimite(
         double x )
17     {
18         limite = x;
19     }
20
21     public double getLimite()
22     {
23         return limite;
24     }
25 }
```

Classe Banco.java (versão 1) I

```
1  public class Banco
2  {
3      public static void main ( String args[] )
4      {
5          Conta c1 = new Conta();
6          c1.setLimite( 300 );
7          c1.depositar( 500 );
8          c1.sacar( 200 );
9          System.out.println( "O saldo é " + c1.getSaldo() );
10     }
11 }
```

Classe Banco.java (versão 1) II

E se sacarmos mais que o disponível?

Classe Conta.java (versão 1.1)

Reescrevemos o método `sacar()`:

```
8     public void sacar(double x) {
9         if ( saldo + limite >= x )
10            saldo = saldo - x;
11    }
```

Métodos permitem controlar os valores / atributos, evitando que qualquer objeto altere seu conteúdo sem observar regras.

Construtores

Métodos construtores são utilizados para realizar *toda a inicialização necessária* a uma nova instância da classe;

Construtores

Métodos construtores são utilizados para realizar *toda a inicialização necessária* a uma nova instância da classe;

- Diferente de outros métodos, um método construtor não pode ser chamado diretamente:
  Um construtor é invocado pelo operador `new` quando um novo objeto é criado;

Construtores

Métodos construtores são utilizados para realizar *toda a inicialização necessária* a uma nova instância da classe;

- Diferente de outros métodos, um método construtor não pode ser chamado diretamente:
  Um construtor é invocado pelo operador `new` quando um novo objeto é criado;
- Determina como um objeto é inicializado quando ele é criado;

Construtores

Métodos construtores são utilizados para realizar *toda a inicialização necessária* a uma nova instância da classe;

- Diferente de outros métodos, um método construtor não pode ser chamado diretamente:
  Um construtor é invocado pelo operador `new` quando um novo objeto é criado;
- Determina como um objeto é inicializado quando ele é criado;
- Vantagens: não precisa criar métodos *get*/*set* para cada um dos atributos privados da classe (reforçando o encapsulamento), tampouco enviar mensagens de atribuição de valor após a criação do objeto.

Construtores

A declaração de um método construtor é semelhante a qualquer outro método, com as seguintes particularidades:

- O nome do construtor deve ser o mesmo da classe;

Construtores

A declaração de um método construtor é semelhante a qualquer outro método, com as seguintes particularidades:

- O nome do construtor deve ser o mesmo da classe;
- Um construtor não possui um tipo de retorno – sempre `void`, mas isso não é indicado diretamente no programa.

Construtores

A declaração de um método construtor é semelhante a qualquer outro método, com as seguintes particularidades:

- O nome do construtor deve ser o mesmo da classe;
- Um construtor não possui um tipo de retorno – sempre `void`, mas isso não é indicado diretamente no programa.

Observações importantes:

Construtores

A declaração de um método construtor é semelhante a qualquer outro método, com as seguintes particularidades:

- O nome do construtor deve ser o mesmo da classe;
- Um construtor não possui um tipo de retorno – sempre `void`, mas isso não é indicado diretamente no programa.

Observações importantes:

- Por padrão, o Java já cria um construtor sem parâmetros para todas as classes.

Construtores

A declaração de um método construtor é semelhante a qualquer outro método, com as seguintes particularidades:

- O nome do construtor deve ser o mesmo da classe;
- Um construtor não possui um tipo de retorno – sempre `void`, mas isso não é indicado diretamente no programa.

Observações importantes:

- Por padrão, o Java já cria um construtor sem parâmetros para todas as classes.
- Poderá criar mais de um construtor para uma mesma classe. Por exemplo, pode-se criar um construtor sem parâmetros, com dois parâmetros e outro com três parâmetros.

Classe Veiculo.java I

```
1  class Veiculo
2  {
3      private String marca;
4      private String placa;
5      private int kilometragem;
6      public Veiculo( String m, String p, int k )
7      {
8          marca = m;
9          placa = p;
10         kilometragem = k;
11     }
12     public String getPlaca()
13     {
14         return placa;
15     }
16     public String getMarca()
17     {
18         return marca;
19     }
```

Classe Veiculo.java II

```
20      public int getKilometragem()
21      {
22          return kilometragem;
23      }
24      public void setKilometragem(int k)
25      {
26          kilometragem = k;
27      }
28  }
```

Classe AcessaCarro.java

```
1  class AcessaCarro
2  {
3      public static void main(String args[])
4      {
5          Veiculo meuCarro = new Veiculo("Escort","XYZ-3456",60000);
6          String marca;
7          int kilometragem;
8          marca = meuCarro.getMarca();
9          System.out.println( marca );
10         kilometragem = meuCarro.getKilometragem();
11         System.out.println( kilometragem );
12         meuCarro.setKilometragem( 100000 );
13         System.out.println( kilometragem );
14     }
15 }
```

This I

This

This é usado para fazer auto-referência ao próprio contexto em que se encontra.

This sempre será a própria classe ou o objeto já instanciado.

This II

```
1  class Veiculo
2  {
3      private String marca;
4      private String placa;
5      private int kilometragem;
6      public Veiculo( String marca, String placa, int kilometragem )
7      {
8          this.marca = marca;
9          this.placa = placa;
10         this.kilometragem = kilometragem;
11     }
12     public String getPlaca()
13     {
14         return placa;
15     }
16     public String getMarca()
17     {
```

This III

```
18          return marca;
19      }
20      public int getKilometragem()
21      {
22          return kilometragem;
23      }
24      public void setKilometragem(int kilometragem)
25      {
26          this.kilometragem = kilometragem;
27      }
28  }
```

Referências

- HORSTMANN, Cay S.; CORNELL, Gary. *Core Java 2: Vol.1 – Fundamentos*, Alta Books, SUN Mircosystems Press, 7a. Edição, 2005.

- DEITEL, H. M.; DEITEL, P. J. *JAVA – Como Programar*, Pearson Prentice-Hall, 6a. Edição, 2005.

- https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html