Construtores

Construtores

- São similares à métodos, com exceção de que são invocados exclusivamente durante a criação de objetos
- Geralmente utilizados para “inicializar” um objeto
- A declaração de construtor é semelhante à declaração de métodos, porém não possuem tipo de dado de retorno e seu identificador é igual ao da classe
- Não é preciso criar construtor para a classe. Quando não é implementado um construtor, o compilador automaticamente fornece um construtor padrão.
  - Um “construtor padrão” é um construtor sem argumentos.

Construtores

- Podemos também criar os nossos próprios construtores, passando parâmetros para inicializar os atributos do objeto

```
public class Pessoa{

   double altura;
   double peso;

    public Pessoa() {
    }

   public Pessoa(double altura, double peso) {
      this.altura = altura;
      this.peso= peso;
   }

    public double calcularImc() {
        return peso/ (altura * altura);
    }

}
```

this

- `this` refere-se ao objeto corrente – o objeto na qual o método foi chamado.

- O principal motivo em usar a palavra `this` é quando há algum parâmetro de método que possui o mesmo nome de uma variável de instância.
  - Quando há dois identificadores com o mesmo nome, por padrão Java sempre reconhece que o identificador utilizado no comando é aquele com menor escopo

O operador new

- O operador **new** faz então quatro operações:
  1. Cria o objeto na memória, alocando espaço para armazenar valores para suas variáveis de instância
  2. Inicializa as variáveis de instância
  3. **Executa o construtor que foi utilizado no operador new**
  4. Retorna o endereço de memória criado pelo objeto.

Sobrecarga de métodos

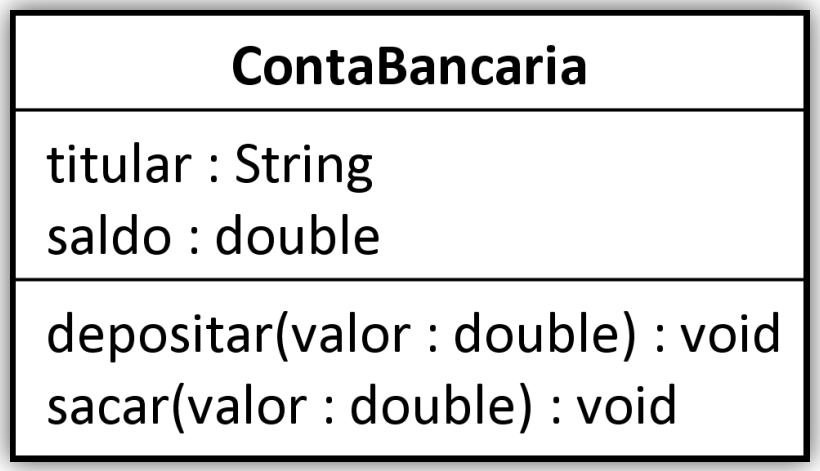
Sobrecarga de métodos

- A linguagem Java suporta a sobrecarga de métodos, isto é, implementação de vários métodos com mesmo nome.
- Os métodos devem ter assinaturas diferentes
  - Métodos podem ter mesmo nome se a lista de parâmetros for diferente.
  - O compilador não considera o tipo de retorno para diferenciar o método. Por isso, dois métodos com a mesma assinatura mas retornos distintos não podem ser implementados na mesma classe
- Deve ser utilizado com moderação pois pode tornar o código menos legível

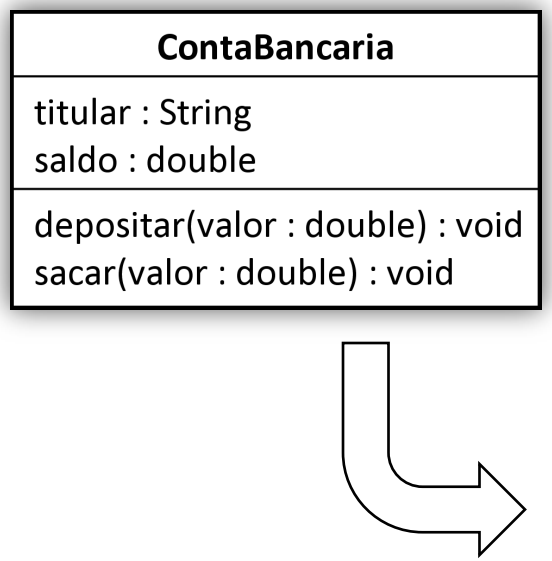
Encapsulamento

Encapsulamento - Motivação

Considerar a classe abaixo, a ser utilizada para representar contas bancárias:



Encapsulamento – Motivação




```
 1 public class ContaBancaria {
 2
 3     String titular;
 4     double saldo;
 5
 6     void depositar(double valor) {
 7         saldo = saldo + valor;
 8     }
 9
10     void sacar(double valor) {
11         saldo = saldo - valor;
12     }
13
14 }
```

Encapsulamento - Motivação

```
 1 public class CaixaEletronico {
 2
 3     public static void main(String[] args) {
 4
 5         ContaBancaria conta1 = new ContaBancaria();
 6         conta1.titular = "Sandro da Silva";
 7         conta1.depositar(500);
 8         conta1.sacar(100);
 9         System.out.println(conta1.saldo);
10
11     }
12
13 }
```

Mostra 400

Encapsulamento - Motivação

```
 1 public class CaixaEletronico {
 2
 3     public static void main(String[] args) {
 4
 5         ContaBancaria conta1 = new ContaBancaria();
 6         conta1.titular = "Sandro da Silva";
 7         conta1.depositar(500);
 8         conta1.sacar(100);
 9         conta1.saldo = 10000;
10
11     }
12
13 }
```

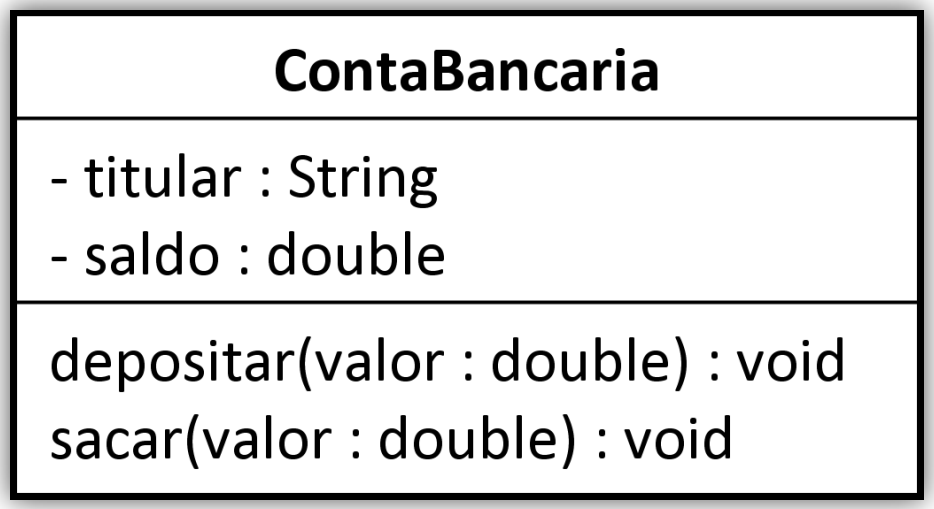
Este comando permite que seja definido um valor de saldo, sem que seja feito um depósito correspondente

Encapsulamento

- O acesso aos atributos deve ser “controlado”, para garantir integridade dos dados
  - Isto é, o estado do objeto precisa ser “controlado”
- Somente o próprio objeto deveria manipular o valor de seus atributos
- Esta técnica se chama **encapsulamento** de dados
- Em Java, para aplicar o encapsulamento é preciso tornar o atributo **privado**.

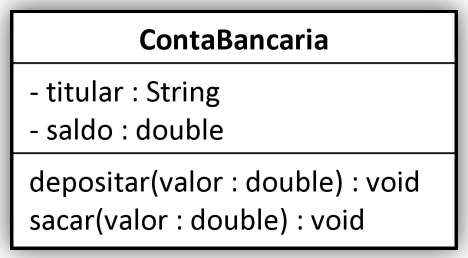
Encapsulamento

Em UML, para expressar que um atributo é encapsulado, será apresentado um sinal de “-” na frente do atributo, como no exemplo:



Encapsulamento

Outros símbolos podem ser utilizados nos membros de uma classe (atributos e operações) para indicar seu grau de visibilidade:



Símbolo UML	Nome	Palavra reservada*	Significado
-	Privado	private	Somente visível pela própria classe
+	Público	public	Visível para qualquer classe
#	Protegido	protected	Estudaremos mais tarde
~	"de pacote"	(ausência de símbolo)	Estudaremos mais tarde

* Conhecido também como "modificador de acesso"

Sintaxe para traduzir um atributo em Java:

```
Modificador de acesso tipo de dado identificador;
```

Exemplo:

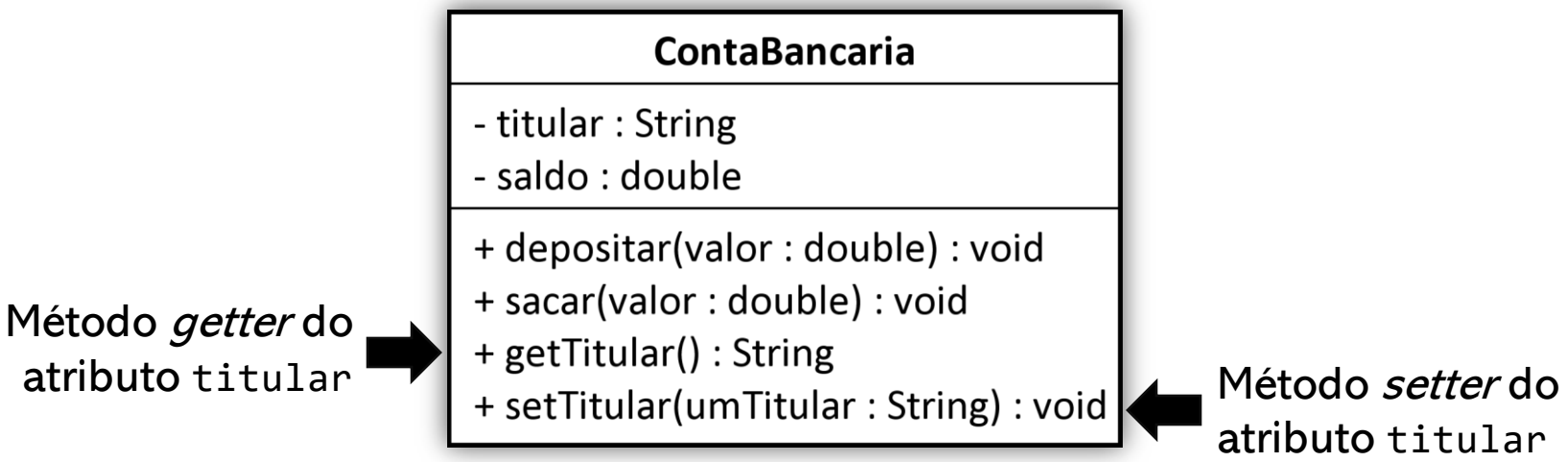
```
private double saldo;
```

Encapsulamento – Métodos de acesso

- Todos os atributos de um objeto deveriam ser encapsulados.
- Os atributos que precisam ser acessados por outras classes poderão ser acessados por meio de **métodos de acesso**.
- Os métodos de acesso são métodos, geralmente públicos. Dividem-se em:
  - **Getters**: métodos usados para recuperar valor de atributos de um objeto da classe. O nome de um *método getter* deve ser escrito com o prefixo *get*, seguido do nome do atributo com a inicial maiúscula.

    Exceção: se o atributo for lógico (booleano), usar o prefixo "is".
  - **Setters**: métodos usados para atribuir valor de atributos da classe. O nome de um método *setter* deve ser escrito com o prefixo *set*, seguido do nome do atributo com a letra inicial maiúscula.

Exemplo



Métodos *getter* :

- nunca têm parâmetro
- sempre são do tipo função
- sempre retornam um dado cujo tipo é igual ao da variável em que é *getter*

Métodos *setter* :

- sempre têm um parâmetro
- sempre são do tipo procedimento
- o parâmetro deve ser declarado com um tipo de dado igual ao da variável que é *setter*

Exemplo de *getter* e *setter*

```
public class ContaBancaria {

   private String titular;
   private double saldo;

   void depositar(double valor) {
      saldo = saldo + valor;
   }

   void sacar(double valor) {
      saldo = saldo - valor;
   }

   public void setTitular(String umTitular) {
      titular = umTitular;
   }

   public String getTitular() {
      return titular;
   }

}
```

Exemplo de *getter* e *setter*

```
public class ContaBancaria {

   private String titular;
   private double saldo;

   void depositar(double valor) {
      saldo = saldo + valor;
   }

   void sacar(double valor) {
      saldo = saldo - valor;
   }

   public void setTitular(String titular) {
      this.titular = titular;
   }

   public String getTitular() {
      return titular;
   }

}
```

Encapsulamento

- Sempre dar preferência por encapsular todos os atributos de uma classe
- Somente é admissível utilizar `public` para constantes
- Se for necessário expor o valor de atributo para outros objetos/classes, implementar um método *getter* para o atributo
- Se for necessário permitir que outros objetos/classes definam o valor de um atributo, é necessário implementar um método *setter* para o atributo

Encapsulamento de métodos

- Ao utilizar POO, é possível ocultar a complexidade do trabalho interno executado pelo objeto:
  - Criar uma forma simplificada e compreensível de utilizar o objeto – favorece a reutilização
  - Por exemplo: o motorista não precisa compreender como o mecanismo interno de combustão funciona para ligar o carro.