Programaçãolll: Construtores, Herança e Polimorfismo

Profª. Tainá Isabela

Construtores

É um método especial para a criação e inicialização de uma nova instância de uma classe. Um construtor inicializa o **novo objeto** e suas variáveis, cria quaisquer outros objetos de que ele precise e realiza quaisquer outras operações de que o objeto precise para inicializar-se.

É possível definir **diversos** construtores para a mesma classe, tendo os tipos ou a quantidade de parâmetros diferentes para cada um deles.

```java
public Conta (int numero, String nome_titular, double saldo){
    this.numero=numero;
    this.nome_titular=nome_titular;
    this.saldo=saldo;
}

public Conta(int numero, String nome_titular){
    this.numero = numero;
    this.nome_titular=nome_titular;
    saldo=0;
}
```

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

    Conta c1 = new Conta(1, "Jao", 0);
    Conta c2 = new Conta(2, "Hozier");
```

Destrutores

Em Java, o método destrutor de uma classe é o método finalize. Quando não é definido um método destrutor para uma classe, Java utiliza um método destrutor padrão que não faz nada.

Encapsulamento

Encapsulamento é uma técnica utilizada para restringir o acesso a variáveis (atributos), métodos ou até à própria classe. Os detalhes da implementação ficam ocultos ao usuário da classe, ou seja, o usuário passa a utilizar os métodos de uma classe sem se preocupar com detalhes sobre como o método foi implementado internamente.

A ideia do encapsulamento na programação orientada a objetos é que não seja permitido acessarmos **diretamente** as propriedades de um objeto. Nesse caso, precisamos operar sempre por meio dos métodos pertencentes a ele.

- *public* (público): indica que o método ou o atributo são acessíveis por qualquer classe, ou seja, que podem ser usados por qualquer classe, independentemente de estarem no mesmo pacote ou estarem na mesma hierarquia;

- *private* (privado): indica que o método ou o atributo são acessíveis apenas pela própria classe, ou seja, só podem ser utilizados por métodos da própria classe;

- *protected* (protegido): indica que o atributo ou o método são acessíveis pela própria classe, por classes do mesmo pacote ou classes da mesma hierarquia (estudaremos hierarquia de classes quando tratarmos de herança).

```java
package exemplos;
public class Conta
{
    private int numero;
    private String nome_titular;
    private double saldo;

    public void depositar (double valor) {
        this.saldo = this.getSaldo() + valor;

    }


    boolean sacar(double valor) {
        if (this.getSaldo()>=valor){
            this.saldo-=valor;
            return(true);
        }
        else
            return false;
    }

    public double getSaldo() {
        return saldo;
    }

    public int getNumero() {
        return numero;
    }

    public String getNome_titular(){
        return nome_titular;
    }

    public void setNome_titular(String nome_titular){
        this.nome_titular = nome_titular;
    }
    }

```

Herança

Um dos conceitos de orientação a objetos que possibilita a **reutilização de código** é o conceito de herança. Pelo conceito de herança é possível criar uma nova classe a partir de outra classe já existente.

A Herança é um mecanismo que permite que uma classe herde todo o comportamento e os atributos de outra classe.

```java
public class ContaEspecial extends Conta
{
    private double limite;

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite){
        this.limite = limite;
    }
}
```

```
3
4  public class ContaPoupanca extends Conta
5  {
6      public void reajustar (double percentual) {
7          double saldoAtual = this.getSaldo();
8          double reajuste = saldoAtual * percentual;
9          this.depositar(reajuste);
10     }
11 }
```

Quando visualizamos uma hierarquia partindo da classe pai para filhas, dizemos que houve uma especialização da superclasse. Quando visualizamos partindo das classes filhas para as classes ancestrais, dizemos que houve uma generalização das subclasses.

Caso você não tenha definido um construtor em sua superclasse, não será obrigado a definir construtores para as subclasses, pois Java utilizará o construtor padrão para a superclasse e para as subclasses.

Porém, caso haja algum construtor definido na superclasse, obrigatoriamente você precisará criar ao menos um construtor para cada subclasse.

```java
public ContaEspecial(int numero, String nome_titular, double limite){
    super(numero,nome_titular);
    this.limite = limite;
}
```

```java
public ContaPoupanca(int numero, String nome_titular){
    super(numero,nome_titular);
}
```

Utilização de atributos protected

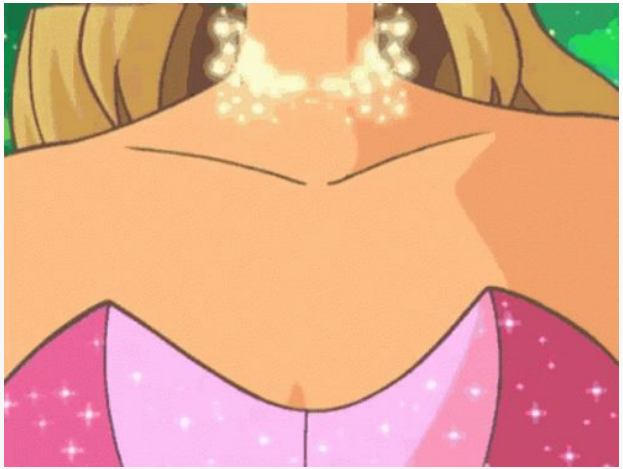
Apesar de potencialmente facilitar a implementação de métodos nas sub classes, a utilização de atributos protegidos é perigosa, pois o atributo **ficará acessível** a todas as classes que estejam no mesmo pacote e não somente às subclasses. Assim, pense bastante sobre as vantagens e desvantagens antes de se decidir por definir um atributo como protected.

Polimorfismo

O polimorfismo permite escrever programas que processam objetos que **compartilham** a mesma superclasse (direta ou indiretamente) como se todos fossem objetos da superclasse; isso pode **simplificar** a programação.

O polimorfismo pode ser obtido pela utilização dos conceitos de herança, sobrecarga de métodos e sobrescrita de método (também conhecida como redefinição ou reescrita de método).

Polimorfismo



Em outras palavras, podemos ver o polimorfismo como a possibilidade de um mesmo método ser executado de **forma diferente** de acordo com a classe do objeto que aciona o método e com os parâmetros passados para o método

Com o polimorfismo podemos projetar e implementar sistemas que são **facilmente extensíveis**, novas classes podem ser adicionadas com pouca ou nenhuma alteração a partes gerais do programa, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente.

Sobrescrita

A técnica de sobrescrita permite **reescrever** um método em uma subclasse de forma que tenha comportamento **diferente** do método de mesma assinatura existente na sua superclasse.

@Override

```java
public class Conta {

    public void imprimirTipoConta(){
        System.out.println("Conta Comum");
    }
}
public class ContaEspecial extends Conta {

    @Override
    public void imprimirTipoConta(){
        System.out.println("Conta Especial");
    }
}
public class ContaPoupanca extends Conta {

    @Override
    public void imprimirTipoConta() {
        System.out.println("Conta Poupança");
    }
}
```

```java
package banco;
import java.util.Scanner;

public class UsaContaPolimorfa {
    public static void main(String[] args) {
        Conta c = null;
        Scanner scan = new Scanner(System.in);
        int opcao;
        System.out.println("Qual tipo de conta deseja criar para José?");
        System.out.println("1 - Conta");
        System.out.println("2 - Conta especial");
        System.out.println("3 - Conta poupança");
        opcao = scan.nextInt();
        switch (opcao) {
            case 1:
                c = new Conta(1, "José");
                break;
            case 2:
                c = new ContaEspecial(1, "José", 100.00);
                break;
            case 3:
                c = new ContaPoupanca(1, "José");
                break;
        }
        c.imprimirTipoConta();
    }
}
```

```java
@Override
public boolean sacar(double valor){
    if (valor <= this.limite + this.saldo) {
        this.saldo -= valor;
        return true;
    } else {
        return false;
    }
}
```

Figura 4.8: Método *sacar* sobrescrito na classe *ContaEspecial*

Sobrecarga

Métodos de **mesmo nome** podem ser declarados na mesma classe, contanto que tenham diferentes conjuntos de parâmetros (determinado pelo número, tipos e ordem dos parâmetros). Isso é chamado **sobrecarga de método**.

Para que os métodos de mesmo nome possam ser distinguidos, eles devem possuir assinaturas diferentes. A assinatura (signature) de um método é composta pelo nome do método e por uma lista que indica os tipos de todos os seus argumentos.

```java
public void imprimirTipoConta() {
    System.out.println("Conta Comum");
}
public void imprimirTipoConta(String s) {
    System.out.println("Conta Comum - String recebida:" + s);
}
```

Classe Object

Todas as classes no Java herdam direta ou indiretamente da classe Object; portanto, seus 11 métodos são herdados por todas as outras classes.

toString(): esse método indica como transformar um objeto de uma classe em uma String. Ele é utilizado, automaticamente, sempre que é necessário transformar um objeto de uma classe em uma String.

```
@Override

public String to String() {

        return ("Conta: " + this.numero);

}
```

Classe Object

***getClass*:** retorna a classe de um objeto. Muito utilizado quando se trabalha na criação de ferramentas geradoras de código ou frameworks.

***equals()*:** esse método possibilita comparar os valores de dois objetos. Se considerarmos esses objetos iguais, devemos retornar true; caso sejam diferentes, devemos retornar false.

```java
@Override
public boolean equals(Object o) {
    if (o == null) {
        return false;
    } else if (o.getClass() != this.getClass()) {
        return false;

    } else if (((Conta) o).getNumero() != this.getNumero()) {
        return false;

    } else {
        return true;
    }
}
```

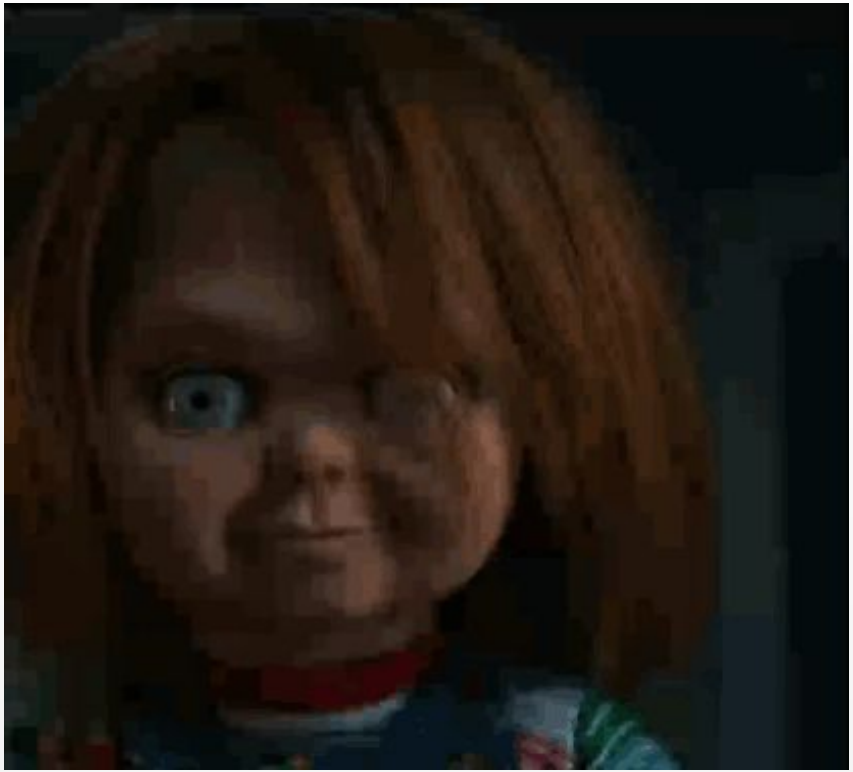
No método equals apresentado são feitas as seguintes verificações:

- *if (o == null)*: estamos prevendo que se pode tentar comparar um objeto Conta com um valor nulo (variável não instanciada). Como o objeto Conta que acionou o método equals está instanciado, ele não pode ser igual a null.

- *if (o.getClass() != this.getClass())*: estamos verificando se o objeto passado como parâmetro é da mesma classe que o objeto que está invocando o método, ou seja, se estamos comparando duas instâncias da classe Conta. Caso os objetos sejam de classes diferentes, consideramos que eles são diferentes.

- *if (((Conta) o).getNumero() != this.getNumero())*: caso os dois objetos sejam do mesmo tipo (Conta), então comparamos os valores do atributo numero dessas contas. Se os números são diferentes, consideramos que são contas diferentes; caso contrário, as consideramos iguais.

Momento Questionário



Atividades

1. Crie uma classe Livro com os atributos titulo e autor. Implemente dois construtores: Um sem parâmetros que define valores padrão. Um com parâmetros para inicializar os atributos. No método main, instancie dois objetos utilizando os dois construtores diferentes e exiba os dados.

2. Crie uma classe Pessoa com os atributos nome e idade. Depois, crie a subclasse Aluno que herda de Pessoa e adiciona o atributo matricula. Crie um construtor para Aluno que receba todos os dados e utilize super() para inicializar nome e idade. Instancie um objeto de Aluno e mostre suas informações.

3. Modifique a classe Aluno do exercício anterior para: Tornar os atributos privados. Criar métodos get e set para todos os atributos. No método main, teste os métodos de acesso e modificação

Atividades

4. Crie uma classe Animal com um método emitirSom(). Crie duas subclasses: Cachorro e Gato. Sobrescreva o método emitirSom() para imprimir sons diferentes. No main, crie um array de Animal contendo um Cachorro e um Gato, e percorra chamando emitirSom() para cada um.

5. Crie uma classe Calculadora com métodos somar() sobrecarregados para: dois inteiros, dois doubles, três inteiros. No main, teste cada versão do método e exiba os resultados.

6. Crie uma classe ContaBancaria com o atributo numero. Sobrescreva os métodos toString() e equals(): Teste no main: Crie duas contas com números iguais e compare com equals(). Imprima um objeto diretamente para verificar o funcionamento do toString().