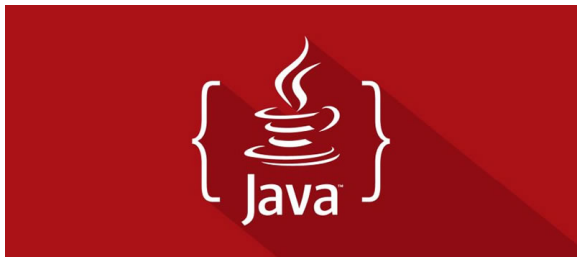
Programação II

Aula 03

Evandro J.R. Silva[1]

[1] Bacharelado em Ciência da Computação
Estácio Teresina

25/08/2022



Sumário

Construtores

- Um **Construtor** é o método responsável por inicializar um objeto com determinados valores.
- Se não é declarado, o Java aplica seu construtor nativo, ou seja, inicializa o objeto com valores *default* (padrão).
- O construtor deve **obrigatoriamente** ter o mesmo nome da classe! E é possível criar mais de um construtor.

Pacotes

- Os pacotes ajudam a organizar o código.
- Classes de mesma finalidade, ou que podem ser agrupadas de acordo com algum critério, ficam no mesmo pacote.

Herança

- Classes podem *herdar* atributos e métodos de outra classe.
- A classe que herda é chamada de **subclasse**. A classe "mãe"é chamada de **superclasse**.
- A subclasse pode ter elementos próprios que não estão presentes em sua superclasse. Ou seja, é como se a superclasse fosse uma descrição mais geral, enquanto as suas herdeiras fossem as decrições mais detalhadas.
- **Curiosidade**: em Java todas as classes são herdeiras da classe Object.
- Para que uma classe seja herdeira de outra, usamos a *palavra reservada* **extends**.

Polimorfismo

- Grosso modo, o polimorfismo é uma técnica de *sobrescrever* os métodos de uma superclasse.
- O uso do polimorfismo pressupõe duas condições: (1) a existência de herança e (2) a redefinição de métodos em todas as classes (as que estão na estrutura da herança).

Polimorfismo

- Outros exemplos
  - Image uma superclasse chamada **Animal**, a qual possui um método chamado **mover()**.
  - A partir da classe Animal criamos com o **extend** as classes **Cachorro**, **Pássaro** e **Peixe**.
  - Cada uma das subclasses vai herdar o método **mover()**, porém vai implementá-lo de forma diferente.

Polimorfismo

```
public class Animal
{
    // atributos ...

    public void mover()
    {
        // mover objeto
    }
}

public class Cachorro extends Animal
{
    // atributos ...

    // override
    public void mover()
    {
        // correr
    }
}
```

```
public class Passaro extends Animal
{
    // atributos ...

    //override
    public void mover()
    {
        // voar
    }
}

public class Peixe extends Animal
{
    // atributos ...

    public void mover()
    {
        // nadar
    }
}
```

Polimorfismo

- E se tiver algum método que eu não queria que seja sobrescrito?
- Basta utilizar o modificador **final**.
  Ex.: **public final void** comer () ...
- Uma classe que tenha o modificador **final** não poderá ter subclasses.
- E um atributo que tenha esse modificador é transformado em uma constante.
  Ex.: **private final int** idadeMorte = 100;

Agrupamento de Objetos

- As vezes precisamos criar manipular várias instâncias de uma classe, ou uma *variável* que possa guardar vários valores ao mesmo tempo.
- Em Java podemos fazer isso através de **Arrays**, **List**, **Set** e **Map**. Vamos ver cada um.

Array

- Um **Array** é considerado (em Java) um objeto indexado, de tamanho imutável (após criado), que pode armazenar tanto tipos primitivos quanto objetos.

Array

- Um **Array** é considerado (em Java) um objeto indexado, de tamanho imutável (após criado), que pode armazenar tanto tipos primitivos quanto objetos.
- Exemplo de como criar um Array:

```
int[] a1 = new int[10];
                    OU
int a1[] = new int[10];
```

Array

- Um **Array** é considerado (em Java) um objeto indexado, de tamanho imutável (após criado), que pode armazenar tanto tipos primitivos quanto objetos.
- Exemplo de como criar um Array:

```
int[] a1 = new int[10];
                    OU
int a1[] = new int[10];
```

- No exemplo criamos um array com 10 posições. Os índices vão de 0 a 9, e os valores são os *default*.

Collections

- E se quisermos criar um conjunto cujo tamanho seja dinâmico?
- E se quisermos manipular o conjunto sem precisar ficar percorrendo o array índice por índice?

Collections

- E se quisermos criar um conjunto cujo tamanho seja dinâmico?
- E se quisermos manipular o conjunto sem precisar ficar percorrendo o array índice por índice?
- A solução dos nossos problemas no Java é a Framework Collections. A partir dela temos **listas**, **conjuntos** e **mapeamentos** (estão lembrando de *estruturas de dados*?).

List, Map e Set

- Na maioria das vezes criamos um **ArrayList**.

```
List<CLASSE> variavel = new ArrayList<>();
                OU
ArrayList<CLASSE> variavel = new ArrayList<>();
```

- Map:

```
Map<TIPO CHAVE, TIPO VALOR> variavel = new HashMap<>();
Para adicionar itens: variavel.put(chave, valor);
```

- SET:

```
Set<TIPO SET> variavel = CONJUNTO;
```

Encapsulamento

- É um princípio de design de código, geralmente ligado a programação orientada a objetos, e nos orienta a esconder as funcionalidades e funcionamento do nosso código dentro de pequenas unidades (normalmente métodos e funções).
- Isso possibilita que modificações no sistema possam ser feitas de maneira mais cirúrgicas, sem que uma funcionalidade esteja espalhada por diversas partes do sistema.

Encapsulamento

- Quando devo usar?
  - Basicamente sempre, pois a forma como classes e objetos conversam uns com os outros, deve sempre estar isolada da forma como executam o que propuseram a fazer.

Encapsulamento

- **Default**
  - É quando o encapsulamento não é definido.
  - Um atributo default pode ser acessado por todas as classes que estiverem no **mesmo pacote** da classe que possui aquele atributo.

Encapsulamento

- **Public**
  - TODAS as classes têm acesso.

Encapsulamento

- **Private**
  - Somente a própria classe tem acesso!
  - Só a classe mesmo, ou seja, nem o objeto dessa classe tem acesso.

Encapsulamento

- **Protected**
  - Praticamente a mesma coisa do default.
  - DIFERENÇA: pode ser acessado de uma **subclasse** que está em **outro pacote**.

Classes Abstratas e Interfaces

- Já vimos e treinamos o conceito de herança!
- A classe `Pessoa` já faz parte das nossas famílias!
- Lembrando: a classe `Pessoa` era a superclasse.
- Era extendida por pelo menos `PessoaFísica` e `PessoaJurídica`. Dava para utilizar os atributos e métodos herdados, e ainda fazer polimorfismo!
- Ainda assim a gente podia instanciar a classe `Pessoa`.
- E se ...

Classes Abstratas e Interfaces

- Já vimos e treinamos o conceito de herança!
- A classe `Pessoa` já faz parte das nossas famílias!
- Lembrando: a classe `Pessoa` era a superclasse.
- Era extendida por pelo menos `PessoaFísica` e `PessoaJurídica`. Dava para utilizar os atributos e métodos herdados, e ainda fazer polimorfismo!
- Ainda assim a gente podia instanciar a classe `Pessoa`.
- E se ...
  - ... fosse possível criar uma superclasse apenas para servir de superclasse? Ou seja, não precisa (e nem pode) ser instanciada?

Classes Abstratas e Interfaces

- Já vimos e treinamos o conceito de herança!
- A classe `Pessoa` já faz parte das nossas famílias!
- Lembrando: a classe `Pessoa` era a superclasse.
- Era extendida por pelo menos `PessoaFísica` e `PessoaJurídica`. Dava para utilizar os atributos e métodos herdados, e ainda fazer polimorfismo!
- Ainda assim a gente podia instanciar a classe `Pessoa`.
- E se ...
  - ... fosse possível criar uma superclasse apenas para servir de superclasse? Ou seja, não precisa (e nem pode) ser instanciada?
  - Como fazer?

Classes Abstratas e Interfaces

- Já vimos e treinamos o conceito de herança!
- A classe `Pessoa` já faz parte das nossas famílias!
- Lembrando: a classe `Pessoa` era a superclasse.
- Era extendida por pelo menos `PessoaFísica` e `PessoaJurídica`. Dava para utilizar os atributos e métodos herdados, e ainda fazer polimorfismo!
- Ainda assim a gente podia instanciar a classe `Pessoa`.
- E se ...

- Seus problemas acabaram!
- Conheçam a **CLASSE ABSTRATA**!

Classes Abstratas

- Uma **classe abstrata**, como já diz o nome, é desenvolvida para representar entidades e conceitos abstratos.
- É declarada com a palavra reservada **abstract**:
  - **abstract class** NOME ...

Classes Abstratas

- Uma **classe abstrata**, como já diz o nome, é desenvolvida para representar entidades e conceitos abstratos.
- É declarada com a palavra reservada **abstract**:
  - **abstract class** NOME ...
- Mas não somente as classes podem ser abstratas, como também os métodos!

Classes Abstratas

- Uma **classe abstrata**, como já diz o nome, é desenvolvida para representar entidades e conceitos abstratos.
- É declarada com a palavra reservada **abstract**:
  - **abstract class** NOME ...
- Mas não somente as classes podem ser abstratas, como também os métodos!
  - **public abstract void** nomeMétodo();
  - Nesse caso o método abstrato é declarado apenas para ser sobrescrito, e as subclasses poderão implementar/polimorfar *livremente*.

Classes Abstratas

- Uma **classe abstrata**, como já diz o nome, é desenvolvida para representar entidades e conceitos abstratos.
- É declarada com a palavra reservada **abstract**:
  - **abstract class** NOME ...
- Mas não somente as classes podem ser abstratas, como também os métodos!
  - **public abstract void** nomeMétodo();
  - Nesse caso o método abstrato é declarado apenas para ser sobrescrito, e as subclasses poderão implementar/polimorfar *livremente*.
  - Ou seja, é uma maneira de declarar, na superclasse, métodos cujos códigos serão especificados apenas nas subclasses!

Classes Abstratas

- Uma **classe abstrata**, como já diz o nome, é desenvolvida para representar entidades e conceitos abstratos.
- É declarada com a palavra reservada **abstract**:
  - **abstract class** NOME ...
- Mas não somente as classes podem ser abstratas, como também os métodos!
  - **public abstract void** nomeMétodo();
  - Nesse caso o método abstrato é declarado apenas para ser sobrescrito, e as subclasses poderão implementar/polimorfar *livremente*.
  - Ou seja, é uma maneira de declarar, na superclasse, métodos cujos códigos serão especificados apenas nas subclasses!

Classes Abstratas

- E se eu quiser fazer uma classe 100% abstrata?

Classes Abstratas

- E se eu quiser fazer uma classe 100% abstrata?
- A solução neste caso se chama **INTERFACE**!

Interface

- Como já vimos, é uma classe 100% abstrata!
- Todos os métodos são abstratos, e se houve algum atributo deve ser uma constante!
- É declarada com a palavra reservada **interface**.
  - **interface** Nome;
- Ajuda também a contornar a não existência de múltipla herança no Java, pois é possível "herdar"os métodos de várias interfaces!
  - **public class** Nome **implements** Interface1, Interface2, ...
- Vamos ver um exemplo!

FIM

Só conseguimos aprender a programar quando treinamos!

Até a próxima!