Revisão para a Primeira Prova de POO e ED

SUMÁRIO

Começando do começo

Para programar em Java precisamos do `JDK` (*Java Development Kit*). O que vem em um `JDK` ?

- Ferramentas de desenvolvimento
    - São variados, e incluem, por exemplo, o compilador `javac` .

- O `JRE` (*Java Runtime Environment*)
    - Tudo o que é necessário para executar um programa em `Java`, como bibliotecas com as classes padrões do `Java`, e uma `JVM` (*Java Virtual Machine*).

Após a instalação do `JDK`, você poderá criar programas em Java a partir de **qualquer editor de texto**. Entretanto, para compilar um programa em `Java` é necessário a criação de um **projeto**, que vai conter na mesma pasta todos os arquivos necessários para transformar o código em um programa executável.

O compilador traduz todo o seu projeto (tudo o que é utilizado) para `bytecodes`, uma espécie de linguagem de máquina, a qual será executada pela `JVM`.

IDEs

As `IDE`s (*Integrated Development Environment*) ajudam os programadores em variadas tarefas, inclusive em montar todo o básico necessário para a criação de um novo projeto. E, por questão de organização, é interessante que você tenha no seu computador uma pasta exclusiva para armazenar os seus projetos. Essa pasta é comumente chamada de `workspace`, ou área de trabalho, em tradução direta. Cada projeto será uma pasta diferente dentro do `workspace`, e terá todo o necessário para a compilação de um novo programa.

Cada `IDE` é única, no sentido de ter um conjunto de ferramentas específicas para ajudar o programador. O `VS Code` é um **editor de código**, ou seja, não é uma `IDE` propriamente dita, porém pode ser transformado em uma, através da **instalação e configuração de extensões**.

Para quem está começando a programar, o uso do `VS Code` não é recomendável, uma vez que são necessárias várias configurações do próprio ambiente de programação. Então, para quem está começando, o recomendável é utilizar uma `IDE` feita especificamente para o `Java`. As mais comuns:

- IntelliJ Idea;
- Eclipse;
- NetBeans.

Caso você não queira instalar uma `IDE`, ou ainda esteja em dúvida de como utilizá-las, o recomendável é procurar alguma solução *online*. Um dos melhores ou mais conhecidos é o replit.com. Basta você abrir uma conta, e depois abrir um projeto em `Java`. A vantagem do `replit` e outras soluções parecidas, é que todo o seu código estará o tempo inteiro na nuvem.

Você com certeza já deve ter percebido, porém vou reforçar: aprenda **Inglês**. Várias das `IDE`s podem ser instaladas com tradução para o Português, e muito material para iniciar na programação já está disponível em Português. Entretanto, é inevitável que uma hora ou outra você irá precisar de materiais que só estarão disponíveis em Inglês.

Sintaxe

A `sintaxe` de uma linguagem de programação consiste, grosso modo, de todas as regras determinadas para a programação nessa linguagem. Nisso estão inclusos:

- Formas de declarar qualquer coisa, desde variáveis a funções;
- Palavras reservadas (por exemplo, **if**, **for**, etc.);
- Como o código deve ser estruturado.

O `Java` possui uma `sintaxe` semelhante à linguagem `C`. Ou seja, todo comando deve ser finalizado com um **ponto-e-vírgula** `;`. Exemplo:

```
int a = 2;
```

Além disso, **todo bloco** deve estar entre **chaves** `{ }`. Exemplo:

```
if (a > 2){
    System.out.println("a é maior do que 2");
        }
```

Existem algumas variações nas regras. Por exemplo, caso um bloco consista de apenas **UM** comando, é possível ignorar as chaves. Exemplo:

```
if (a > 2)
    System.out.println("a é maior do que 2");
```

ou

```
if (a > 2) System.out.println("a é maior do que 2");
```

Uma vez que o `;` é o separador de todos os comandos, e as `{ }` são os separadores dos blocos, é teoricamente possível que você escreva um programa inteiro usando apenas uma linha de código.

Outra `regra sintática` do `Java` é a necessidade de cada classe estar separada em seu próprio arquivo. Para garantir isso, o nome de cada arquivo deve ser exatamente o nome da `classe` . Tudo o que você vai fazer em `Java` , estará dependendo de alguma classe.

Convenções

Além das **regras sintáticas**, existem também as **convenções** de escrita de código. Essas convenções servem para que qualquer código em Java seja legível por qualquer programador.

Na escrita de `Java` é adotada a convenção `camelCase` e `UpperCamelCase` . O *camel* desse nome significa literalmente camelo. É como se os nomes seguissem o formato das corcovas dos camelos.

Os nomes das classes devem sempre estar no formato `UpperCamelCase` , ou seja, começam com a primeira leitra em maiúsculo, e o restante das letras da palavra em minúsculo. Caso haja necessidade de se acrescentar outra palavra, cada nova palavra seguirá o padrão de primeira letra em maiúsculo, e o restante em minúsculo. Exemplo: **Pessoa**, **PessoaFisica**, **PessoaJuridica**, **FuncionarioDoMes**.

Os nomes de atributos, métodos e variáveis seguem o padrão `camelCase` , ou seja, a primeira palavra estará completamente em minúsculo. A partir da segunda palavra em diante, a primeira letra estará em maiúsculo, e o restante da palavra em minúsculo. Exemplo: **diaTreino**, **horaAlmoco**, **nomeDoMeio**.

Programação Orientada a Objetos

A `Programação Orientada a Objetos` é um dos `paradigmas` de programação. Aqui nós temos os importantes conceitos de `classe` e `objeto`.

Classe

Uma `classe` é uma **estrutura de dados** abstrata e heterogênea em conjunto com `métodos`. **Abstrata** porque os dados são mais complexos e formados a partir de dados `primitivos` e outros dados `abstratos`. E **heterogênea** porque podem ser formadas a partir de variados tipos de dados. Por fim, os `métodos` são funções próprias da `classe` para manipular seu ambiente, ou ser manipulado.

Vamos ver em `C` como é uma estrutura de dados abstrata e heterogênea, sem ser uma `classe`.

```
#include <stdio.h>
#include <stdlib.h>

struct Pessoa{
    char nome[50];
    int idade;
    float altura;
    float peso;
};

void setAltura(float altura, struct Pessoa *pessoa){
    pessoa->altura = altura;
}

void olaMundo(){
    printf("Olá Mundo!");
}

int main(){
    struct Pessoa pessoa;
    setAltura(1.75, &pessoa);

    olaMundo();
    printf("\nAltura: %.2f", pessoa.altura);

    return 0;
}
```

Pelo código acima vemos que criamos uma estrutura de dados chamada `Pessoa`, porém apenas com os atributos. Fora da estrutura foram criadas duas funções, uma chamada `setAltura` e a outra chamada `olaMundo`. Essas funções independem da estrura `Pessoa`.

Voltando ao conceito de `classe`. A partir do momento em que há a possibilidade de funções serem implementadas dentro das estruturas de dados, temos então as `classes`, e as funções passam a ser chamadas de `métodos`.

Objeto

Um `objeto` é a `instância` de uma classe.

infopédia:

> 2 qualidade do que é feito com dedicação, perseverança, **constância**
>
> 4 conjunto particular de fatores, valores ou condições que determinam e/ou constituem determinado âmbito, categora ou domínio

Dicionário Online de Português:

> Qualidade daquilo que é realizado de modo perseverante; em que há perseverança; **persistência**.

Como `instância` de uma `classe`, um `objeto` é a **persistência** de uma `classe` na memória RAM, ou seja, sua criação e continuação de existência.

Outra forma de entender a diferença entre `classe` e `objeto`, é que a `classe` é uma **descrição**, enquanto o `objeto` é a **realização** dessa descrição. Por isso é possível que existam vários `objetos` da mesma `classe`.

Tecnicamente, todo `objeto` criado é um espaço reservado na memória, suficiente para armazenar todos os valores declarados como `atributos`, e também `métodos` que podem consultar e/ou editar esses valores.

A criação forma de um `objeto` acontece com o uso da palavra reservada `new`. Exemplo:

```
Pessoa p1 = new Pessoa();
```

No exemplo acima um objeto chamado `p1` é criado como `instância` da `classe` `Pessoa`, utilizando o `método construtor padrão`. Isso faz com que seja reservado um espaço na memória que caiba todos os valores dos `atributos` da `classe` `Pessoa` e também um espaço para os `métodos` da mesma `classe`.

Ou seja, uma `instanciação` de uma `classe` só acontece quando é utilizada explicitamente a palavra reservada `new`.

Estrutura Geral de uma Classe

Em `Java` as `classes` possuem a seguinte estrutura geral:

1. Declaração do pacote;
2. Imports;
3. Declaração da Classe;
4. Atributos;
5. Construtores;
6. Getters e Setters;
7. Métodos.

Declaração do pacote

Cada `classe` está contida em um `pacote`. Os `pacotes` são basicamente a **estrutura hierárquica** das `classes` em pastas. Até o momento em que essa revisão foi criada, esse projeto está organizado da seguinte forma:

```
src
  |__aula01
  |__aula02
  |       |__ifelse
  |       |__repete
  |
  |__aula03
  |       |__exercicio01
  |       |__exercicio04
  |       |__imagens
  |       |__textos
  |
```

```
|__aula04
|       |__codigos
|       |        |__questao01
|       |
|       |__texto
|
|__aula05
|       |__codigos
|       |        |__exemplos
|       |        |         |__encapsulamento1
|       |        |         |__encapsulamento2
|       |        |         |__encapsulamento3
|       |        |
|       |        |__exercicios
|       |                     |__heranca
|       |                     |__polimorfismo
|       |
|       |__texto
|
|__revisao
         |__prova1
```

Dentro do arquivo da classe, o caminho até o arquivo é declarado com a palavra reservada `package` e o nome das pastas separados por ponto. Exemplo: `package aula05.codigos.exemplos.encapsulamento1;`

Imports

Após a declaração do pacote onde a `classe` está, pode haver um conjunto de comandos `import` para importar métodos e atributos de outras `classes` . Após a palavra reservada `import` é declarado o caminho para a `classe` cujos métodos e atributos são desejados. Exemplo: `import java.util.Scanner;`

A maioria das `IDE` s vão lidar com esses `imports` de forma automática, portanto você não precisará se preocupar com isso na maioria dos casos.

Declaração da Classe

Por enquanto nos basta saber que a declaração de uma `classe` segue o seguinte padrão:

> *visibilidade* `classe` Nome [`extends` SuperClasse]{ }

Como vimos na última aula, a `visibilidade` pode ser `default`, `private`, `public` ou `protected`.

A `visibilidade` `default` acontece quando você não define nenhuma das outras, ou seja, deixa o valor vazio. Isto vai fazer com que essa `classe` seja visível SOMENTE para as demais `classes` presentes no mesmo `pacote`.

A `visibilidade` `private` determina que essa `classe` ficará invisível para as demais. No contexto de `classes`, existem poucos casos em que isso seja preciso. Essa `visibilidade` será mais utilizada em `atributos` ou `métodos`.

A `visibilidade` `public` é possivelmente a mais utilizada e permite a visibilidade da classe por todo o projeto.

A `visibilidade` `protected` permite a visualização dessa classe SOMENTE por outras presentes no mesmo `pacote`. Porém pode ser visível fora do `pacote` por suas **subclasses**.

Caso a `classe` que esteja sendo implementada seja `subclasse` de outra, então é necessário explicitar essa relação com o uso da palavra reservada `extends`, seguido do nome da `superclasse`.

Atributos

Os `atributos` são normalmente declarados nas primeiras linhas no interior de uma `classe`. Não é obrigatório que toda `classe` possua `atributos`, e também não é obrigatório que sejam sempre declarados no início.

A declaração dos atributos segue o seguinte formato:

> `visibilidade` `tipo` nomeAtributo [= valor];

A `visibilidade` segue o mesmo princípio mostrado no formato da declaração das `classes`. O `tipo` pode ser desde um dentre os primitivos (ex.: `int`, `double`, etc.) como também alguma classe (ex.: `String`, `Pessoa`). Caso o programador deseja, ou seja necessário, o atributo já pode ser iniciado com algum valor.

Construtores

Os `construtores` são `métodos` especiais **que devem ter o mesmo nome da** `classe` . Eles servem para inicializar os `atributos` quando um `objeto` é `instanciado` .

Não é obrigatório implementar um `construtor` . Nesse caso, será atrelado à `classe` o `construtor padrão` , o qual inicalizará os `atributos` com seus valores padrões. Por exemplo, `Strings` são inicializadas como `null` (nulo), e valores numéricos são inicializados com o valor 0.

Não existe um limite para quantos `construtores` uma `classe` pode ter. O programador pode implementar quantos desejar, desde que cada `construtor` tenha uma `assinatura` diferente. Quanto mais `construtores` uma `classe` tiver, maiores serão as possibilidades de se `instanciar` essa `classe` .

Getters e Setters

Os `métodos getters & setters` fazem sentido quando aplicamos o conceito de `visibilidade` nos `atributos` . Esses são `métodos` "padronizados" para acessar ou editar valores dos atributos de forma segura (no contexto de programação).

Métodos

Os demais `métodos` são todos aqueles que o programador deseja acrescentar à sua `classe` , além dos construtores e *getters* e *setters*.

Um `método` é uma função atrelada a uma `classe` . Seu formato é como segue:

> `visibilidade retorno nomeMetodo` ([parâmetros]) {}

A `visibilidade` é o mesmo mostrado no formato das `classes` .

O `retorno` é o tipo de dado que o método irá retornar a "quem" o tiver chamado. Portanto, se estiver escrito `int` , o método deverá terminar com uma linha contendo a palavra reservada `return` seguido de uma valor inteiro, ou uma variável que contenha um valor inteiro. O mesmo raciocínio pode ser aplicado para qualquer outro tipo.

Caso o `método` não deva retornar qualquer valor, deve ser utilizada a palavra reservada `void` .

Os `parâmetros` são variáveis de entrada (*input*) com seus respectivos tipos, cujos valores serão utilizados em alguma computação dentro do `método`. Não existe um limite para a quantidade de parâmetros.

Os demais conceitos

Para finalizar a revisão, veremos os demais conceitos vistos em sala de aula.

Sobrecarga

A `sobrecarga` acontece quando temos vários `métodos` com o **mesmo nome**, porém com `assinaturas` diferentes.

A `assinatura` de um `método` consiste em:

- Nome;
- Quantidade de parâmetros;
- Tipo dos parâmetros;
- Ordem dos parâmetros.

Herança

A `herança` pode ser vista como uma técnica de programação orientada a objetos em que podemos utilizar da generalização de `atributos` e `métodos` para evitar replicação de código.

Seu funcionamento é basicamente da seguinte forma. Uma classe mais geral é implementada, e após ela, subclasses são implementadas, de forma a *herdar* seus `atributos` e `métodos`, precisando implementar apenas o que for mais específico.

Polimorfismo

O `polimorfismo` pode ser visto como uma junção entre `herança` e `sobrecarga`. Mais especificamente, o `polimorfismo` acontece quando `métodos` herdados são implementados de forma diferente nas `subclasses`.