Sumário

Se virando no Git

Git

Git é um sistema de controle de versão de arquivos. Através deles podemos desenvolver projetos na qual diversas pessoas podem contribuir simultaneamente no mesmo, permitindo que os mesmos possam existir sem o risco de suas alterações serem sobrescritas. Se não houvesse um sistema de versão, imagine o caos entre duas pessoas abrindo o mesmo arquivo ao mesmo tempo. Uma das aplicações do git é justamente essa, permitir que um arquivo possa ser editado ao mesmo tempo por pessoas diferentes.

GitHub

O GitHub é uma plataforma de hospedagem de código-fonte com controle de versão usando Git. Nele criamos repositórios onde colocamos nossos projetos que vamos desenvolver. No Github o projeto é dividido em branches,elas são separações de código. Normalmente são utilizados para separar alterações ou novas funcionalidades do projeto.

Instalação

Você instalará o Git com este comando (via terminal - Linux):

```sh
sudo apt-get install git
```

Depois de instalar, a primeira coisa que você deve fazer é configurar o Git. Para isso, abra uma janela de terminal e digite os seguintes comandos:

```sh
git config --global user.name "Seu Nome"
git config --global user.email "seu@email.com"
```

A partir daí, o Git irá usar essas informações para registrar quem foi que fez as alterações nos arquivos.

Após isso já poderá realizar os comando do git pelo terminal.

Nota: Esse exemplo configura o mesmo usuário para todos os projetos presentes no computador (isso se dá por conta da flag --global). Podemos também configurar usuários para cada projeto, bastando remover a flag `--global` .

Comando básicos

Criando um repositório:

Criar um repositório no Git é muito simples, apenas siga esta sequência completa dos comandos:

`mkdir meu-projeto` (irá criar o diretório)

`cd meu-projeto` (irá entrar dentro do diretório)

`git init` (irá criar o repositório git)

Após isso, basta começar a trabalhar, criando, removendo e alterando arquivos.

Outros Comandos:

`git clone [link do repositório]` Para clonar o projeto do repositório;

`git checkout -b [nome da branch]` Para criar uma nova branch;

`git checkout [nome da branch]` Para trocar de branch;

`git status` Para você visualizar os arquivos que modificou;

`git add .` Para você adicionar as modificações feitas;

`git commit -m [“mensagem”]` Para você comentar brevemente sobre as modificações feitas;

`git push origin [nome da branch]` Sobe as alterações feitas para a branch remota(para onde quero enviar, por isso usamos *origin*) do **GitHub**

`git pull origin master` Recebe as alterações feitas na branch remota origin master;

Mas se houver conflitos:

`git pull -r origin master` Facilita na hora de resolver conflitos.

Ex: Deram push em uma atualização de código da linha 11, e você localmente modificou esta linha e quer dar push, o git ficará sem saber o que fazer e resultará em conflitos, você terá que escolher entre uma das atualizações,ou em deixar as duas.

Após isso para ter certeza de que não tenha mais nada diferente execute:

```
git rebase --continue
```

Nota: Este comando só funcionará se usar a flag `-r` no `git pull`

Se não houver mais conflito pode dar seu push tranquilo, caso contrario terá que resolver todos os conflitos para dar push.

`git reverse --hard HEAD~1` Exclui commit local

Caso já tenha enviado ao seu repositório será necessário executar este comando também para exclui-lo:

```
git push origin HEAD --force
```

Importante: Não é muito recomendável usar estes últimos dois comandos exceto em casos muito extremos, eles podem causar grandes complicações.

EXERCÍCIOS DE FIXAÇÃO

Baseado no material desta apostila informe os comandos necessários para realizar cada uma das tarefas a seguir:

Imagine que criamos um repositório no GitHub:

- [ ] Clone este repositório para seu computador
- [ ] Crie uma branch
- [ ] Entre na branch criada
- [ ] Envie as modificações para o repositório
- [ ] Acesse o site do GitHub e crie um repositório chamado `exercicioGitHubAcelera`
- [ ] Agora, dentro de algum diretório de seu computador, inicie um repositório Git local
- [ ] Clone este repositório (`exercicioGitHubAcelera`) para seu computador
- [ ] Faça qualquer alteração neste diretório (crie arquivos novos, modifique algum existente e etc) e em seguida adicione estas alterações neste repositório Git local
- [ ] Realize um `commit` destas alterações ao seu repositório Git local, informando uma mensagem explicando o que esta sendo salvo neste `commit`
- [ ] Agora, envie as modificações para o repositório

Introdução ao Java

Neste capítulo, veremos alguns conceitos fundamentais do Java para que possamos começar a utilizar a linguagem.

- Escrevendo um "olá, mundo" em Java
- Tipos de dados
- Estruturas condicionais (se, senão, senão se)
- Estruturas de repetição (enquanto, para)
- Operadores

Olá, mundo

No Java o seu código sempre será escrito dentro de classes e métodos. Uma classe é um bloco de código que contém atributos (variáveis) e métodos (funções). Os atributos irão guardar dados e os métodos irão executar lógica/comportamento. Para este capítulo, isto é tudo que precisamos saber sobre classes e métodos, veremos mais sobre eles no capítulo de orientação a objetos.

Sabendo disto, vejamos então como escrever um olá, mundo em Java:

```java
public class Ola {
  public static void main(String [] args) {
    System.out.println("Ola, mundo");
  }
}
```

No exemplo acima, criamos a classe `Ola`, que possui o método `main`, o qual irá escrever uma mensagem na tela utilizando o método `System.out.println` do Java.

> Existem algumas palavras chave neste exemplo que podem parecer bastante confusas (`public`, `static`, `void`, `String[]`). Por enquanto, não precisamos nos preocupar com elas, e iremos entender o que cada uma significa em outros capítulos.

Tipos de dados

A linguagem Java oferece diversos tipos de dados com os quais podemos trabalhar. Há basicamente duas categorias em que se encaixam estes tipos de dados: **tipos primitivos** e **tipos de referência**.

Tipos primitivos

Os tipos primitivos correspondem a dados simples escalares (que possuem um tamanho fixo na memória). No java, existem oito tipos primitivos, mas nem todos são comumente utilizados. Os tipos que você utilizará com mais frequência serão:

- **boolean:** Assume os valores booleanos `true` (verdadeiro) ou `false` (falso).
- **int:** serve para armazenar números inteiros entre -2^{31} e $2^{31}-1$
- **double:** armazena números decimais (quebrados, ou com vírgula).
- **char:** O char é um tipo de variável que aceita a inserção de um caractere apenas.

```java
boolean verdade = true;
boolean mentira = false;

int numero = 5;
double numeroQuebrado = 5.00000001;

char umCaractere = 'a';
```

Tipos primitivos menos comuns

Além dos tipos mais comuns, ainda temos alguns outros tipos primitivos para guardar números:

- **float:** armazena números decimais (quebrados, ou com vírgula) com uma precisão menor (menos números depois da vírgula) que o `double` .
- **short:** armazena valores inteiros entre `-32768` e `32767`
- **long:** armazena valores inteiros entre.
- **byte:** armazena valores inteiros entre `-128` e `127`

> Estes tipos são muito similares aos tipos mais comuns. No entanto, eles existem para casos muito específicos, especialmente para quando precisamos economizar memória ou precisamos utilizar valores inteiros muito grandes (declarando-os como `long` em vez de `int`, por exemplo). Não estamos preocupados com estas situações neste momento.

Links da documentação

Alguns destes são tópicos bastante avançados, mas caso queira entender um pouco mais sobre alguns detalhes, aqui estão alguns links da documentação oficial do Java:

- Sobre tipos primitivos
- Explicação sobre a precisão de números decimais
- O padrão Unicode (utilizado pelo Java para representar variáveis do tipo char)

Tipos de referência

Os tipos de referência armazenam objetos. Neste momento, não faz sentido tentarmos entender a fundo o que isto significa. Sugerimos que depois de você dar uma lida no capítulo de orientação a objetos, revisite esta parte da apostila para entender um pouco melhor.

Nas primeiras interações com a linguagem, você irá utilizar quase que constantemente dois tipos de referências:

- Arrays (vetores)
- Strings

Portanto, neste momento, vamos nos concentrar em entender estes tipos primeiro.

Arrays (vetores)

Arrays (ou vetores) são uma estrutura utilizada para quando necessitamos armazenar um conjunto de valores em uma variável, como uma lista.

Por que utilizamos vetores?

Pensemos em um caso de uso. Estamos escrevendo um programa que armazena 10 valores aleatórios inteiros na memória, multiplica cada um por 2 e exibe os resultados na tela. Podemos resolver isto de duas maneiras:

Criando 10 variáveis

```java
int valor0 = 5;
int valor1 = 11;
int valor2 = 8;
int valor3 = 13;
int valor4 = 18;
int valor5 = 20;
int valor6 = 30;
int valor7 = 35;
int valor8 = 2;
int valor9 = 4;

System.out.println(valor0 * 2);
System.out.println(valor1 * 2);
System.out.println(valor2 * 2);
System.out.println(valor3 * 2);
System.out.println(valor4 * 2);
System.out.println(valor5 * 2);
System.out.println(valor6 * 2);
System.out.println(valor7 * 2);
System.out.println(valor8 * 2);
System.out.println(valor9 * 2);
```

Esta alternativa não é um bom caminho, pois nosso código ficaria imenso e confuso. Imagine ter que fazer isto com 100 variáveis?

Vejamos a outra alternativa:

Criando um vetor de 10 posições

Antes de mais nada, para criar um vetor de valores inteiros, fazemos o seguinte:

```java
int[] valores = new int[10];
```

Utilizamos os colchetes `[]` para indicar que a variável será um array de inteiros. Para criar um array, precisamos dizer qual será seu tamanho, ou em outras palavras, quantos elementos ele poderá guardar. Neste caso, estamos dizendo que o array poderá guardar `10` elementos.

Precisamos falar sobre índices

Como mencionando anteriormente, arrays são uma lista de valores e, para guardar ou acessar elementos desta lista, utilizamos um índice, que indica em qual posição da lista está o elemento que queremos acessar.

Ao criar um array de inteiros de tamanho 10, inicialmente, todas as suas 10 posições conterão o valor `0` :

```java
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }
```

Vamos adicionar o valor `15` à segunda posição do array. Os índices começam em `0` , portanto, para acessar a segunda posição, utilizamos o índice `1` :

```java
valores[1] = 15;
```

Ao executar o código acima, o valor do nosso array agora será:

```java
{ 0, 15, 0, 0, 0, 0, 0, 0, 0, 0 }
```

De volta ao problema

Para resolver o nosso problema utilizando arrays, basta adicionar os números em cada posição do array:

```java
valores[0] = 5;
valores[1] = 11;
valores[2] = 8;
valores[3] = 13;
valores[4] = 18;
valores[5] = 20;
valores[6] = 30;
valores[7] = 35;
valores[8] = 2;
valores[9] = 4;
```

Segura o cavaco!

Parando para pensar, até agora não ganhamos muita coisa ao utilizar um array em vez de 10 variáveis, nosso código está ficando muito parecido e até um pouco mais complexo que a primeira alternativa (de criarmos dez variáveis).

Qual é a moral de usar esse negócio então?

Uma das maiores vantagem de utilizar um array em vez de várias variáveis é que eles nos permitem utilizar **estruturas de repetição!**

Vamos resolver nosso problema utilizando um loop `for` :

```java
valores[0] = 5;
valores[1] = 11;
valores[2] = 8;
valores[3] = 13;
valores[4] = 18;
valores[5] = 20;
valores[6] = 30;
valores[7] = 35;
valores[8] = 2;
valores[9] = 4;

for (int i = 0; i < valores.length; i++) {
  System.out.println(valores[i] * 2);
}
```

Sabemos que ainda não chegamos na parte das estruturas de repetição, mas tenha em mente que em algumas situações, é melhor utilizar arrays e eles existem por ótimos motivos, não se preocupe em entender como tudo funciona agora, por enquanto, apenas guarde isso na sua mente:

> **"Arrays existem por bons motivos, fique de olho em onde você poderá utilizá-los."**
>
> - Regina Casé

Bônus: Definindo valores direto na declaração

No caso do nosso problema, os arrays nos dão uma outra vantagem bacana. Podemos criar o array já preenchido com os valores que precisamos, o que simplifica bastante nosso código:

```java
int [] valores = { 5, 11, 8, 13, 18, 20, 30, 35, 2, 4 };

for (int i = 0; i < valores.length; i++) {
  System.out.println(valores[i] * 2);
}
```

Strings

String é uma das classes mais importantes do Java, sendo vastamente utilizada. Strings servem para representar e manipular texto.

Para declarar uma String, basta fazer o seguinte:

```java
String dia = "Sexta";
```

Ao começar no Java, muitas pessoas pensam que String é um tipo primitivo, o que não é verdade, pois ela é uma classe e valores String armazenados em variáveis são do tipo referência. Esta confusão geralmente acontece pois ela é a única classe na linguagem que possui uma **representação literal**, ou seja, é possível criar novas Strings utilizando aspas duplas.

Manipulando Strings

Strings são úteis para resolver incontáveis tipos de problemas, por isso, é interessante revisar como manipulamos valores String utilizando seus métodos. Vejamos alguns métodos úteis:

length

Nome em português:

> length: tamanho

Retorna o tamanho da String. O tamanho é a quantidade de caracteres que a String possui:

```java
String texto = "Oi";

int tamanho = texto.length(); // tamanho sera 2;
```

equals

Nome em português:

> equals: igual ou é igual a

Compara duas strings e retorna verdadeiro (`true`) caso elas sejam iguais:

```java
String texto = "Oi";
String outroTexto = "Oi";

boolean saoIguais = texto.equals(outroTexto); //saoIguais sera true
```

ou falso (`false`) caso elas sejam diferentes:

```java
String texto = "Oi";
String outroTexto = "Opa";

boolean saoIguais = texto.equals(outroTexto); //saoIguais sera false
```

replace

Nome em português:

> replace: substituir

Este método recebe dois argumentos:

- conteudoAntigo
- conteudoNovo

Ele retorna uma nova String substituindo todas as ocorrências do valor `conteudoAntigo` encontradas na String por `conteudoNovo` :

```java
String bomDia = "Bom Dia!";

String boaNoite = bomDia.replace("Dia", "Noite"); // boaNoite sera "Boa Noite!
```

substring

Este método recebe dois argumentos:

- indiceInicial
- indiceFinal

Retorna uma nova String contendo a porção que está entre as posições indiceInicial e indiceFinal:

```java
String texto = "Aceleradora";

String pedaco = texto.substring(0, 3); // pedaco sera "Ace"
```

split

Nome em português:

> split: separar

Este método recebe um argumento:

- token

Retorna um array de Strings, formado pela divisão da String original. Este método irá dividir a String cada vez que encontrar o `token` no conteúdo da String:

```java
String texto = "A,B,C";

String[] pedacos = texto.split(","); // pedacos sera um array contendo {"A", "
```

Dica, caso queira transformar uma String em um array de Strings, utilize este método passando uma String vazia:

```java
String texto = "dica";

String[] pedacos = texto.split(""); //pedacos sera um array contendo {"d", "i"
```

contains

Nome em português:

> contains: contém

Este método recebe um argumento:

- busca

Retorna verdadeiro caso a String contenha o valor especificado na **`busca`** ou falso caso contrário.

```java
String texto = "Aceleradora";

texto.contains("A"); // sera verdadero
texto.contains("B"); // sera falso
texto.contains("Ace"); // sera verdadeiro
texto.contains("radora"); // sera verdadeiro
```

toLowerCase

Nome em português:

> toLowerCase: para minusculas

Retorna uma nova String com todas as letras maiúsculas trocadas por minúsculas:

```java
String texto = "BOM DIA";

String textoMinusculo = texto.toLowerCase(); // textoMinusculo sera "bom dia"


String outroTexto = "Bom Dia";

String outroTextoMinusculo = texto.toLowerCase(); // outroTextoMinusculo sera
```

toUpperCase

Nome em português:

> toUpperCase: para maiusculas

Retorna uma nova String com todas as letras minúsculas trocadas por maiúsculas:

```java
String texto = "bom dia";

String textoMaiusculo = texto.toUpperCase(); // textoMaiusculo sera "BOM DIA"


String outroTexto = "Bom Dia";

String outroTextoMaiusculo = texto.toUpperCase(); // outroTextoMaiusculo sera
```

Documentação Java:

- Lista completa dos métodos da classe string

Operadores

Como o próprio nome diz, os operadores permitem executar operações sobre um ou dois **valores primitivos**.

Alguns links da documentação oficial do Java:

- Introdução a operadores Java
- Resumo sobre operadores

Operador de atribuição

Pode ser que isto passe despercebido, mas ao atribuir um valor à uma variável, estamos utilizando um operador, o operador de atribuição (`=`):

```java
int cinco = 5;
```

Operadores de Igualdade

Os operadores de igualdade são utilizados para fazer a comparação de dois valores, ou seja, utilizamos estes operadores quando precisamos saber se um valor é `igual` , `diferente` , `maior` ou `menor` do que outro:

Nome	Sintaxe	Exemplo	Significado
Igual	`==`	`x == y`	`x` é igual a `y`
Diferente	`!=`	`x != y`	`x` é diferente de `y`
Maior que	`>`	`x > y`	`x` é maior que `y`
Menor que	`<`	`x < y`	`x` é menor que `y`
Maior ou igual	`>=`	`x >= y`	`x` é maior ou igual a `y`
Menor ou igual	`<=`	`x <= y`	`x` é menor ou igual a `y`

O uso de operadores de igualdade resulta em um valor booleano, o que permite utilizar estes operadores de diferentes maneiras:

Podemos utilizá-los diretamente dentro de estruturas condicionais:

```java
if (5 > 2) {
  System.out.println("5 eh maior que 2");
} else {
  System.out.println("5 nao eh maior que 2");
}
```

Ou podemos guardar o resultado em uma variável, o que nos ajuda escrever código de uma maneira um pouco mais legível em algumas situações:

```java
boolean cincoEhMaiorQueDois = 5 > 2;

if (cincoEhMaiorQueDois) {
   System.out.println("5 eh maior que 2");
} else {
  System.out.println("5 nao eh maior que 2");
}
```

Operadores Condicionais

Operadores condicionais são utilizados em valores booleanos. Eles são úteis quando precisamos verificar mais de uma condição ou precisamos inverter o valor de um booleano (trocar de `true` para `false` ou vice-versa):

And (&&):

Em inglês a palavra "and" é equivalente ao "e" do português (como na frase Maria `e` João), logo, este operador verifica duas condições e resulta em verdadeiro somente se as duas forem verdadeiras, caso contrário, resulta em falso:

```java
if (vaiChover == true && ehSexta == true) {
  System.out.println("Hoje irei embora mais cedo, pois eh sexta E esta chovend
} else {
  System.out.println("Hoje ficarei até mais tarde.");
}
```

No código acima, a pessoa só iria para casa somente se fosse sexta e fosse chover.

Or (||):

Em inglês a palavra "or" significa "ou", logo, este operador verifica duas condições e resulta em verdadeiro se pelo menos uma das duas for verdadeira, e, somente caso as duas sejam falsas, resulta em falso:

```java
if (vaiChover == true || ehSexta == true) {
  System.out.println("Hoje irei embora mais cedo, pois eh sexta ou esta choven
} else {
  System.out.println("Hoje ficarei até mais tarde.");
}
```

No código acima, a pessoa iria para casa se fosse chover, independentemente do dia da semana. Ou, caso fosse sexta mas não estivesse chovendo, ela também iria para casa.

Not (!):

Em inglês, "not" significa "não" ou negação. Este operador inverte o valor booleano de uma expressão ou variável:

Expressão:

```java
boolean naoEhCincoNemMaiorQueDez = !(numero == 5 || numero > 10)
```

Variável:

```java
boolean verdade = true;
boolean mentira = !verdade;
```

Exemplo de uso:

Temos que escrever um programa que valida o embarque de passageiros em um avião. O programa só deve permitir pessoas maiores de idade `e` que possuam passaporte. Caso a pessoa seja maior de idade mas não possua passaporte o sistema deve notificá-la. Caso a pessoa seja menor de idade, o programa deve notificá-la para estar acompanhada dos pais:

```java
public void verificaEmbarque(int idade, boolean possuiPassaporte) {
  boolean ehMaiorDeIdade = idade >= 18;

  if (ehMaiorDeIdade && possuiPassaporte) {
    System.out.println("Pode embarcar");
  } else if (ehMaiorDeIdade && !possuiPassaporte) {
    System.out.println("Nao pode embarcar. Apresente o passaporte.");
  } else if (!ehMaiorDeIdade) {
    System.out.println("Nao pode embarcar. Venha com seus pais.");
  }
}
```

Operadores Numéricos

Os operadores numéricos servem para executar operações com números. Temos dois tipos de operadores numéricos:

Binários

São os operadores que executam operações entre **dois** números:

Nome	Sintaxe	Exemplo	Resultado
Soma	`+`	`1 + 1`	`2`
Subtração	`-`	`2 - 2`	`0`
Multiplicação	`*`	`2 * 2`	`4`
Divisão	`/`	`4 / 2`	`2`
Módulo	`%`	`4 % 2`	`0`

Exemplos de uso

Podemos utilizá-los para criar uma calculadora em Java:

```java
public class Calculadora {
    public int soma(int a, int b) {
        return a + b;
    }

    public int subtrai(int a, int b) {
        return a - b;
    }

    public int multiplica(int a , int b) {
        return a * b;
    }

    public int divide(int a, int b) {
        return a / b;
    }
}
```

Unários

São operadores que executam operações com apenas **um** número. Estes operadores não funcionam diretamente em números literais, apenas variáveis (veja os exemplos para entender isto melhor):

Nome	Sintaxe
Incrementa	`++`
Decrementa	`--`
Acumula soma	`+=`
Acumula multiplicação	`*=`
Acumula subtração	`-=`
Acumula divisão	`/=`

Exemplo:

```java
5++; // nao funciona

int numero = 4;
numero++; // numero agora tem o valor 5
numero--; // numero agora tem o valor 4
numero += 2; // numero agora tem o valor 6
numero -= 2; // numero agora tem o valor 4
numero *= 2; // numero agora tem o valor 8
numero /= 2; // numero agora tem o valor 4
```

Fim do capítulo

E isso é quase tudo que você deve saber para começar a se aventurar no Java! Este capítulo serve como um pontapé inicial, mas ainda temos muita coisa para ver! Caso você tenha interesse, dê uma lida nos tópicos complementares mais abaixo, que tentam explicar um pouco mais sobre algumas coisas que foram comentadas neste capítulo, mas que podem ser meio confusas neste momento.

Recomendamos revisitar estes tópicos complementares depois da leitura do capítulo de orientação a objetos e da realização de alguns exercícios.

Exercícios de fixação

No repositório da trilha de exercícios, você encontrará alguns desafios de lógica de programação que lhe ajudarão a fixar os conceitos apresentados nesta introdução.

Acesse o repositório aqui: https://github.com/aceleradora-TW/trilha-de-exercicios

Tópicos complementares

Strings

Representação literal

Quando falamos *representação literal*, estamos nos referindo às *aspas duplas*. No Java, quando queremos criar um objeto de alguma classe, sempre temos que utilizar a palavra `new`. Vamos supor que nosso programa tem as classes `Carro` e `Papagaio`. Para criar objetos destas classes e guardá-los em variáveis, teríamos que utilizar o new:

```java
Carro carro = new Carro();
Papagaio passaro = new Papagaio();
```

Seguindo esta lógica, teríamos que fazer o mesmo com a String, certo? Afinal, ela é uma classe! Teríamos que fazer algo como:

```java
String dia = new String();
```

Ainda que isto seja possível, não é necessário, pois String é uma classe tão comumente utilizada, que o Java nos dá a facilidade de utilizar as aspas duplas em vez de new:

```java
String dia = "Sexta";
```

Imutabilidade

Quando dizemos que as Strings são imutáveis, basicamente significa que o valor de uma variável String não pode ser alterado em algumas situações. Isto gera bastante confusão.

Podemos sobrescrever o valor de uma variável

Qual a diferença entre tipos primitivos e tipos de referência?

Existem grandes diferenças entre estes tipos, no entanto, para nós esta diferença ainda não é clara, pois não exploramos os conceitos de orientação a objetos. Basicamente, tipos primitivos guardam valores, enquanto tipos de referência guardam a referência para um objeto na memória. Esta ideia pode soar bastante estranha por enquanto, pois ainda não sabemos o que é um objeto.

Para saber quando uma variável é primitiva e quando ela é referência, podemos observar o uso da palavra chave `new` (exceto com as Strings). Esta palavra é responsável por criar uma instância de objeto. Em outras palavras, ela colocará os dados do objeto em memória e adicionará na variável uma referência para a posição de memória onde estão estes dados para que possamos manipulá-los (daí o nome tipo de referência).

Não se preocupe se nada disto fizer sentido agora, recapitularemos estas ideias posteriormente com mais detalhes.

Com tudo isto em mente, vejamos uns exemplos:

```java
// Um tipo primitivo:
int numero = 5;

// Um tipo de referencia
Carro carro = new Carro();

/*
Strings sao a unica excessao a regra da palavra new.

Elas tambem sao um tipo por referencia, mas nao precisamos da palavra new, em
podemos utilizar as aspas duplas para declarar uma nova String e o Java vai en
*/
String dia = "Sexta";
```

Uma diferença muito importante entre tipos primitivos e tipos de referência é que tipos de referência, por serem objetos, possuem atributos e métodos. Ou seja, em um tipo por referência, eu posso fazer o seguinte:

```java
String dia = "SEXTA";

// Chamar um metodo da String
String diaMinusculo = dia.toLowerCase();
```

```
5
6   int [] vetor = new int [5];
7
8   // Acessar um atributo do vetor
9   int tamanhoVetor = vetor.length;
```

Já nos tipos, primitivos, nada disto é possível, pois eles não possuem atributos nem métodos, pois variáveis primitivas apenas guardam um valor bruto.

Depois de ler o capítulo de orientação a objetos, recomendamos que vocês revisite esta parte, prometemos que tudo fará um pouco mais de sentido.

Orientação a Objetos

Classes

No Java o seu código sempre será escrito dentro de classes e métodos. Uma classe é um elemento do código Java que utilizamos para representar objetos do mundo real. Na orientação a objetos, sempre tentamos pensar em como abstrair conceitos do mundo real dentro do código.

Vejamos por exemplo uma classe que representa um carro:

```java
public class Carro {

  String marca;
  int quantidadeDePneus;
  //Tipos De Variáveis e atributos


  public Carro() {
    //construtor sem parâmetros
  }

  public void andar() {
    //método
  }
}
```

Na orientação a objetos (ou a até mesmo na programação em geral), sempre teremos dois elementos:

- (Representação de) Dados
- Comportamento

Atributos

Quando se está estudando e utilizando orientação a objetos, muito ouve-se falar dos tais atributos. Estes nada mais são que variáveis que pertencem a uma classe. No caso da nossa classe carro, temos dois atributos:

- Marca
- Quantidade de pneus

Podemos informar a visibilidade da classe, que pode ser `public` , `private` ou `default` . Utilizamos a palavra reservada `class` seguida pelo nome da classe. Logo após, entre chaves, definimos os elementos a ela relacionados: atributos, construtores e métodos.

Construtores

Para que servem?

Métodos construtores servem para construir um objeto da classe. Ao contrário de outros métodos, um construtor não pode ser chamado diretamente. Para isso usamos a palavra `new` para criar o objeto e então atribuí-lo a uma variável de mesmo tipo.

Exemplo de instanciação de classe:

Chama-se instância de uma classe, a criação um objeto (através do método construtor) cujo comportamento e estado são definidos pela classe.

```java
Carro carrinho = new Carro();
```

Extends

Quando uma classe precisa herdar características de outra, fazemos uso de herança. Em Java, é representado pela palavra-chave `extends`. Todos os atributos e métodos não-privados serão herdados pela outra classe. Por isso, é comum dizer que a classe herdada é pai da classe que herdou seus elementos.

Nota: Em Java não existe herança múltipla. Assim, uma classe pode herdar apenas de outra.

Exemplo:

```java
public class Produto {

  public double valorCompra;
  public double valorVenda;

  public class Computador extends Produto {
    private String processador;

  }
}
```

A palavra-chave `extends` foi utilizada na declaração da classe Computador. Assim, além do atributo processador, devido à herança, a classe Computador também terá os atributos valorCompra e valorVenda, sem que seja necessário declará-los novamente, sem repetir código.

Implements

Quando uma classe precisa implementar os métodos de uma interface, utiliza-se a palavra reservada `implements`:

Exemplo:

Considerando a interface `IProduto`:

```java
public interface IProduto {

  double calculaFrete();

}
```

Podemos ter a classe `Televisao` implementando-a:

```java
public class Televisao implements IProduto {
  private double peso;
  private double altura;

  @Override
    public double calculaFrete() {
      //código para cálculo do frete
    }
}
```

A anotação `@Override` explicita os métodos que foram codificados/sobrescritos.

Nota: Podemos implementar várias interfaces. Para isso, basta separá-las por vírgula.

Também é possível utilizar `extends` conjuntamente com `implements` . Trata-se de um recurso útil quando deseja-se tornar uma classe mais específica e implementar novos comportamentos definidos em interfaces.

Exemplo:

```java
public class ClasseFilha extends ClassePai implements NomeInterface {
  // Atributos, construtores e métodos da ClasseFilha
  //Métodos implementados da interface
}
```

Regras para nomeação de classes:

- Manter o nome simples e descritivo;
- Usar palavras inteiras, isto é, sem siglas e abreviações;
- A primeira letra de cada palavra devem ser maiúsculas. (camel casing)

Constantes

Uma constante é declarada quando precisamos lidar com dados que não devem ser alterados durante a execução do programa. Para isso, utilizamos conjuntamente as palavra reservadas `final` e `static` .

Exemplo:

```java
public static final float PI = 3.14;
public static final String MEU_NOME = "Cassia";
```

- A palavra `final` indica que a variável não pode ter seu valor modificado.
- A palavra `static` indica que todos os objetos de uma classe compartilharão o mesmo valor.

Nota: Por convenção, usamos letras maiúsculas e underscores (_)para declarar constantes e assim distingui-las das variáveis.

Enums

Em Java, uma enum é um tipo especial de classe no qual declaramos um conjunto de valores constantes pré-definidos. usamos a palavra chave `enum` que antecede seu nome.

Exemplo:

```java
public enum Turno {
  MANHA, TARDE, NOITE;
}
```

Por serem os campos de uma enum constantes, seus nomes são escritos em letras maiúsculas.

Para atribuir um desses valores a uma variável podemos fazer como no código abaixo:

Exemplo:

```java
Turno turno = Turno.MANHA;
```

Por que usar enums?

Enums são extremamente úteis quando precisamos representar um conjunto restrito de valores de uma maneira mais segura em vez de usar apenas Strings. Eles nos garantem que o compilador irá aceitar somente o conjunto de possibilidades que nós definimos, o que deixa o código menos propenso a erros. Vejamos um exemplo à respeito disso.

Precisamos fazer um programa para gerenciar as turmas de uma escola. Para isso, criamos primeiro uma classe para representar as turmas:

```java
public class Turma {
  private String nome;
  private String turno;

  public String getTurno() {
    return turno;
  }
}
```

Uma das funcionalidades do programa é informar o horário de início das turmas de acordo com seus turnos. Para isso, implementamos o seguinte:

```java
public class GestaoDeTurmas {

  public void mostraHorarioDaTurma(Turma turma) {
    if (turma.getTurno().equals("manha")) {
      System.out.println("As aulas comecam as 7h30min");
    } else if (turma.getTurno().equals("tarde")) {
      System.out.println("As aulas comecam as 13h30min");
    } else {
      System.out.println("As aulas comecam as 18h30min");
    }

  }
}
```

Não temos como garantir a integridade dos valores que receberemos, ou seja, pode ser que as nossas usuárias escrevam `manha`, `manhã`, `Manhã`, `De tardezinha` para representar o turno, o que causará comportamentos estranhos.

Vejamos como os enums podem ajudar a garantir um comportamento mais previsível:

Primeiro, criamos um enum para representar os turnos que o programa suporta:

```java
public enum Turno {
  MANHA, TARDE, NOITE;
}
```

Depois, mudamos a nossa classe `Turma` para que ela utilize o enum:

```java
public class Turma {
  private String nome;
  private Turno turno;

  public Turno getTurno() {
    return turno;
  }
}
```

Agora, a `GestaoDeTurmas` pode ser um pouco mais precisa, utilizando somente os tipos que o programa aceita e informando o usuário caso a informação recebida seja inválida:

```java
public class GestaoDeTurmas {

  public void mostraHorarioDaTurma(Turma turma) {
    if (turma.getTurno() == Turno.MANHA) {
      System.out.println("As aulas comecam as 7h30min");
    } else if (turma.getTurno() == Turno.TARDE) {
      System.out.println("As aulas comecam as 13h30min");
```

```
8       } else {
9         System.out.println("As aulas comecam as 18h30min");
10      }
11    }
12  }
```

Os 4 pilares da Programação Orientada a Objetos

Abstração

É utilizada para a definição de entidades do mundo real. Sendo onde são criadas as classes. Essas entidades são consideradas tudo que é real, tendo como consideração as suas características e ações.

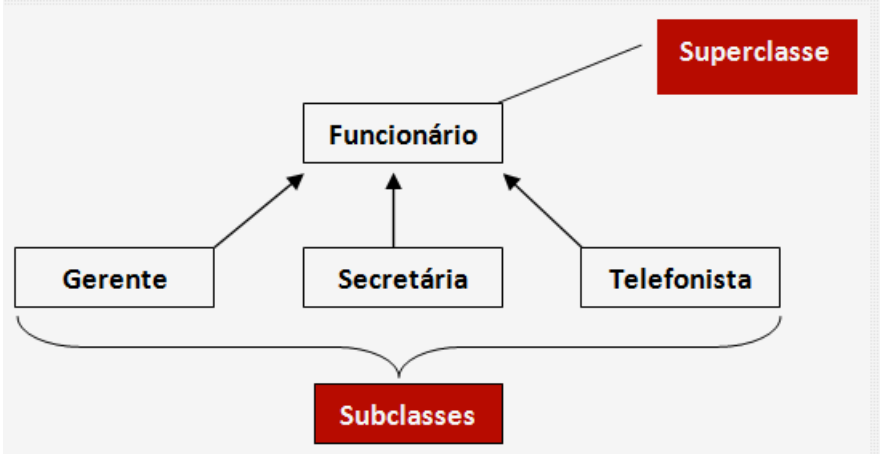
Entidade	Características	Ações
Carro, Moto	tamanho, cor, peso, altura	acelerar, parar, ligar, desligar
Elevador	tamanho, peso máximo	subir, descer, escolher andar
Conta Banco	saldo, limite, número	depositar, sacar, ver extrato

Encapsulamento

É a técnica utilizada para esconder uma ideia, ou seja, não expor detalhes internos para o usuário, tornando partes do sistema mais independentes possível. Por exemplo, quando um controle remoto estraga apenas é trocado ou consertado o controle e não a televisão inteira. Nesse exemplo do controle remoto, acontece a forma clássica de encapsulamento, pois quando o usuário muda de canal não se sabe que programação acontece entre a televisão e o controle para efetuar tal ação.

Herança

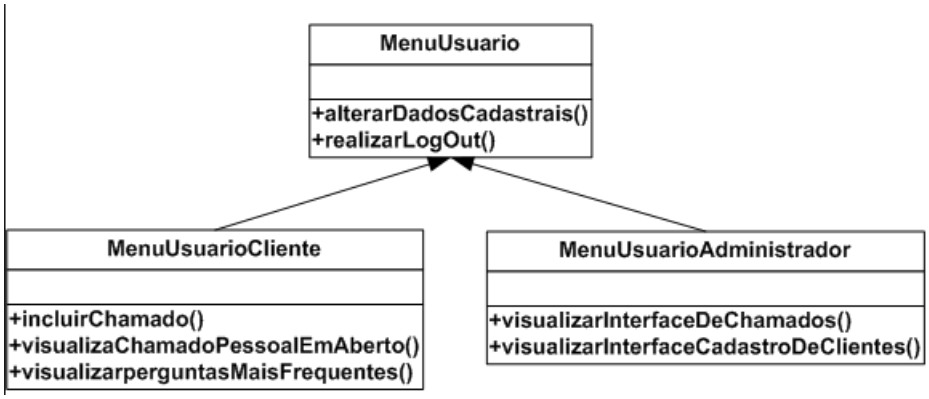
Na Programação Orientada a Objetos o significado de herança tem o mesmo significado para o mundo real. Assim como um filho pode herdar alguma característica do pai, na Orientação a Objetos é permitido que uma classe possa herdar atributos e métodos da outra, tendo apenas uma restrição para a herança. Os modificadores de acessos das classes, métodos e atributos só podem estar com visibilidade `public` e `protected` para que sejam herdados.



Polimorfismo

O polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.

Exemplo:



EXERCÍCIOS DE FIXAÇÃO

Clone o projeto do GitHub link: https://github.com/aceleradora-TW/laboratorio-oo-java

Faça os exercícios e deixe os testes passar

Divirta-se!