**Tipo String**

A String não é um tipo primitivo, e você consegue chamar as funcionalidades dela através da notação ponto, exemplo:

String a = “Olá pessoal”;

System.out.println(a.charAt(2));

A String é identificada por aspas duplas(“”), e assim como um array, ela tem um tamanho e índice(que também começa por 0), sendo cada letra da String um índice, nesse exemplo anterior, a função “.charAt()” vai retornar a letra que está no índice passado dentro dos parenteses, e nesse caso como foi passado a posição “2”, ele irá imprimir a letra “á”, se você passar a posição “3”, ele irá retornar nada, pois será o espaço entre a letra “á” e a letra “p”.

A String é um objeto imutável, oque significa que você nunca vai conseguir mudar o valor que você atribuiu a uma variável, como a variável “a”, você pode substituir o valor da variável do tipo String, mas não modifica-lo.

Funcionalidades da String:

a.startsWith(“Olá”);

Aqui ele irá retornar true se a String começar com a palavra “Olá”, diferenciando maiúsculas de minusculas, se não ele vai retornar false.<

a.endsWith(“pessoal”);

Aqui ele irá retornar true se a String terminar com a palavra “pessoal ”, diferenciando maiúsculas de minusculas, se não ele vai retornar false.

a.toLowerCase();

Vai fazer com que todas as letras da String fique minusculas.

a.toUpperCase();

Vai fazer com que todas as letras da String fique maiúsculas .

a.length();

Vai retornar um número inteiro do tamanho da String.

a.equals(“Boa tarde”);

Aqui ele irá retornar true se a String for igual a comparação dentro dos parenteses, diferenciando maiúsculas de minusculas, se não ele vai retornar false.

a.equalsIgnoreCase(“Boa tarde”);

Aqui ele vai fazer a mesma coisa do anterior, a diferença é que ele vai ignorar as letras maiúsculas e minusculas e verificar apenas o conteúdo da String.

String.format(“Olá, meu nome é %s e tenho %d anos”, “Gabriel”, 20);

O String.format() serve para você receber os valores das variáveis ou diretamente como no exemplo depois da virgula da frase, sendo:

%s para String

%d para números inteiros

%f para números reais

Para que o código não fique gigante, e mais fácil de entender, e cada valor passado depois da virgula será respectivo a posição de cada um dos %s, %d ou %f.

Caso você for usar um número real, e quer definir a quantidade de casas decimais na String gerada, basta colocar:

String.format(“Olá, meu nome é %s, tenho %d anos e ganho R$%.2f”, “Gabriel”, 20, 10000);

Bastando colocar a quantidade de casas decimais que quer no lugar do “2” entre o ponto e o “f”.

String y = "Bom dia X".replace("X", "Gui");

Aqui ele vai substituir o valor de “X” por “Gui”, serve exatamente para substituir uma letra ou palavra da String por outra.

**Wrappers**

Uma classe empacotadora, em inglês wrapper class, na linguagem de programação Java é uma das oito classes fornecidas no pacote java.lang para fornecer métodos de objeto para os oito tipos primitivos. Todas as classes wrapper em Java são imutáveis.

**Tipos Inteiros:**

**Byte -> byte**

**Short -> short**

**Integer -> int**

**Long -> long**

**Byte b = 100;**

**Short s = 1000;**

**Integer i = 10000;**

**Long l = 100000L;**

**Tipos Reais :**

**Float -> float**

**Double -> double**

**Float f = 10000F;**

**Double d = 100000;**

**Tipos Booleanos:**

**Boolean -> boolean**

**Boolean bo = true;**

**Tipos Caractere:**

**Character -> char**

**Character bo = ‘#’;**

Como o tipo String, os tipos Wrappers são objetos que fornecem funcionalidades através da notação ponto.

**Conversão de Tipos:**

**Convervetendo número para String:**

**Integer num1 = 10000;**

**String s = num.toString();**

Aqui o “s” vai receber o valor de “num1” convertido para String.

**String s = Integer.toString(10000);**

Aqui será a mesma coisa do outro exemplo, porém usando diretamente o Integer.

**Convervetendo String para número:**

**String valor1 = “35”;**

**String valor2 = “3.87”;**

**int num1 = Integer.parseInt(valor1);**

Aqui ele vai converter o valor da String para um número inteiro.

**double num2 = Double.parseInt(valor1);**

Aqui ele vai converter o valor da String para um número real .

Mas essa conversão de String para número só funcionar se o valor dentro das aspas da String for um número, pois se tiver algum texto ele não vai funcionar.

**Classes e métodos**

**Classes**

Uma **classe é um** elemento do código **Java** que utilizamos para representar objetos do mundo real. Dentro dela é comum declararmos atributos e métodos, que representam, respectivamente, as características e comportamentos desse objeto.

**Métodos**

métodos é um comportamento, é aquele que é chamado com os “( )”, como uma função, podendo ter ou não parâmetros. Para criar um método você deve definir o tipo dele (int, double, String...) que será retornado, ou caso não retorne nada deve ser declara com o void, e se for criar parâmetros para o método, deve definir o tipo do parâmetro também.

Exemplo:

int soma(int a, int b) {

...

return a + b;

}

Ou sem passar parâmetros:

int soma() {

...

return ...;

}

E dentro do método usa o return para retorna algo dentro do método.

**Construtores**

O construtor usa a palavra reservada “new” seguido pelo nome do construtor, ex:

Produto p1 = new Produto();

Um construtor padrão é aquele que não recebe nenhum parâmetro e é definido de forma implícita pelo Java, mas da mesma forma que o método pode receber parâmetros, o construtor também pode receber parâmetros, ao se definir um novo construtor de forma explicita recebendo parâmetros, o construtor padrão que é criado pelo Java de forma implícita deixa de existir, e terá que criar um novo construtor de forma explicita que não recebe parâmetros para usá-lo sem os parâmetros. Ao se criar uma nova classe construtora, não se passa o tipo que será retornado como por exemplo(int, double, String... ou void para não retornar nada), pois o java já sabe que o retorno do construtor é uma instancia do tipo do nome da classe.

Exemplo:

class Produto {

// Construtor implícito

}

Aqui foi criado um construtor de forma implicita, ou seja, foi criado pelo próprio Java e não possui parâmetros, para usar esse construtor é so usar a palavra reservada new seguido do nome da classe construtora:

Produto p1 = new Produto();

class Produto {

Produto (int a) {

// Construtor explicito

}

}

Produto p1 = new Produto(3);

Aqui foi criado uma nova variável do tipo Produto através da classe construtora Produto, e passado o valor do parâmetro como o número inteiro 3, pois esse construtor foi declarado de forma explícita com parâmetros.

class Produto {

Produto (int a) {

// Construtor explicito com os parâmetros

}

Produto () {

// Construtor explicito sem os parâmetros

}

}

Produto p1 = new Produto(3);

Produto p2 = new Produto();

Aqui foi criado uma nova variável do tipo Produto através da classe construtora Produto, e passado o valor do parâmetro como o número inteiro 3, pois esse construtor foi declarado de forma explícita com parâmetros, mas também podemos usar o construtor sem os parâmetros, pois foi criado de forma explicita um construtor sem os parâmetros.

O retorno dos construtores do exemplo da classe Produto, será do tipo Produto. O construtor é um método especial.

Restrições do construtor:

1. Nome da Classe = Nome do Construtor
2. Um construtor NÂO TEM RETORNO

**Membros de Classe vs Instância**

Ao de criar uma classe, e chamar ela com a palavra reservada new, você estará criando uma nova instancia(Objetos), reservando a quantidade de memoria daquela instancia, de acordo com oque tem na classe, por exemplo, se criarmos a classe Data:

public class Data {

int dia;

int mes;

int ano;

}

Ao se criar uma nova instância do tipo Data, ele irá criar uma cópia do valor do atributo para cada nova instância criada com a palavra new, e assim vai acontecer para cada nova instancia do tipo Data que for criado.

public class Data {

static int dia;

int mes;

int ano;

}

Data p1 = new Data();

Ao se usar a palavra reservada static antes do tipo do atributo dentro da classe, você estará dizendo que o atributo pertencerá a classe, automaticamente o valor que está associado a instância já não vai mais estar associado a ela, e sim vai estar associado ao própria classe, e ele gerará apenas uma única copia para todos as novas variáveis que você criar, no caso objetos. Ou seja, se você dizer p1.dia = 30, todas os outros objetos do tipo data terá o dia igual a 30, e se você declarar outo objeto com o nome p2, e dizer que p2.dia = 10, tanto p1.dia quanto o p2.dia será igual a 10.

**Atribuição: Valor vs Referência**

Atribuição por Valor para tipos primitivos

Atribuição por Referência para objetos

Atribuição por Valor

Ao se atribuir o valor de uma variável para outra variável, exemplo:

int a = 8;

int b = a;

Você estará criando uma copiar de a em b, só que cada um com seu espaço reservado na memória de forma individual, se eu alterar o valor de a na linha de baixo para 10, o valor de b continuara valendo 8, pois ela já recebeu o primeiro valor de a.

**Atribuição por Referência**

Já se você criar um objeto, e atribuir o valor desse objeto para outro objeto, por exemplo:

Data d1 = new Data();

Data d2 = d1;

Eles vão compartilhar o mesmo espaço de memória, fazendo assim com oque você altere em d1, também seja alterado em d2 e vise e versa. Exemplo:

d1.dia = 10;

d2.dia também será igual a 10.

se você fazer o d2 receber um novo objeto Data, ele vai apontar para outra área de memoria, fazendo assim com que oque você altere em d2 não seja alterado em d1 e vice e versa.

Exemplo:

d2 = new Data();

d2.dia = 23;

O valor de d2.dia será igual a 23 e o valor de d1.dia continuará igual a 10.

**This e This()**

A palavra-chave "this" é uma referência ao objeto atual (ele mesmo). isso pode ser usado dentro de algum método ou construtor. ... Esta é uma palavra-chave em **Java**. Que pode ser usado dentro do método ou construtor da classe.

public class Data {

int dia;

int mes;

int ano;

Data(int dia, int mesInicial, int anoInicial) {

dia = dia;

mes = mesInicial;

ano = anoInicial;

}

}

Se você colocar o nome do parâmetro por exemplo dia, que é o mesmo nome da variável de instancia “int dia” dentro da classe Data, e você atribuir o valor de dia para o parâmetro dia, você não estará atribuindo o valor passado como parâmetro para a variável de instância dia, e sim para o próprio parâmetro, e para você acessar a variável de instância da classe e usar o nome do parâmetro com o mesmo nome da variável de instância, você precisa usar a palavra reservada this, fazendo assim com que o valor passado no parâmetro seja atribuído a variável de instancia dia, pois o this estará referenciando o própria objeto que estará sendo criado naquele momento.

public class Data {

int dia;

int mes;

int ano;

Data(int dia, int mesInicial, int anoInicial) {

this.dia = dia;

mes = mesInicial;

ano = anoInicial;

}

}

O this aponta para o objeto atual que estará executando o código. O this não pode ser usado dentro de um método estático pois o this é uma das instâncias da classe, e um método static ele existe associado a classe.

This():

Você também pode usar o this para chamar a partir de um construtor, você chamar outro construtor. Exemplo:

public class Data {

int dia;

int mes;

int ano;

Data() {

dia = 1;

mes = 1;

ano = 1970;

}

}

Em vez de usar dia, mes e ano um em cada linha passando os valores padrões, você pode usar o this para passar os valores diretamente nele, chamando o this como uma função(método). Exemplo:

public class Data {

int dia;

int mes;

int ano;

Data() {

this(1, 1, 1970);

}

}

Você só pode usar o this como método dentro de outro construtor, que serve para a partir de um construtor você chamar outro construtor.

**Variáveis locais**

Quando você declara uma variável fora do método, dentro da classe, ela pode ser uma variável de instância e pode ser uma variável de classe, ou membro(ou atributo) estático ou um atributo de instância(objeto), quando você defini uma variável dentro de um método nos chamamos essa variável(ou constante) de variável(ou constante) local. E uma variável local só estará visível dentro do método em que foi declarada a partir da linha em que foi definida, ou seja, você não pode acessar essa variável fora do método em que ela foi definida. Se você precisar usar uma variável local em outros métodos, basta declarar ela dentro da classe e fora do método, e caso você declare ela na linha de baixa em que o método foi definido, você ainda poderá usar essa variável em um método que está na linha acima da declaração dessa variável, exemplo:

public class Data {

int dia;

int mes;

int ano;

String obterDataFormatada() {

String formato = “%d/“%d/“%d”;

return String.format(formato, this.dia, mes, ano);

}

}

Nesse exemplo a variável “formato” só estará acessivel dentro do método obterDataFormatada, se você precisa usar essa variável dentro de outro método, basta declarar a variável dentro da classe, e fora de qualquer método, exemplo:

public class Data {

int dia;

int mes;

int ano;

String obterDataFormatada() {

return String.format(formato, this.dia, mes, ano);

}

void imprimirDataFormatada() {

formato = “...”;

System.out.println(this.obterDataFormatada());

}

String formato = “%d/“%d/“%d”;

Nesse caso, a variável foi “formato” foi declarada depois de todos os métodos, mas você poderá usar ela nos métodos que estão acima da declaração dessa variável ou depois dela também, podendo alterar o valor dela dentro de cada método, ou usar com o valor que ela foi definida. Para ficar organizado, o padrão é colocar no inicio da classe todas as variáveis de classe ou variáveis de instância ou constantes, e depois os métodos.

**Valores padrão**

Quando você for criar uma variável de instância ou variável de classe do tipo byte, short, int ou long, o valor padrão será 0.

O valor padrão do float e double é 0.0.

O valor padrão do booelan é false.

O valor padrão do char é ‘\u000’ que é o primeiro elemento da tabela unicode.

Objetos por padrão terão o valor nulo. Exemplo:

String s = null;

Uma variável nula significa que ela não aponta para nenhum local de memória.

E isso só acontece quando a variável pertence a classe ou é uma variável de instância, se você declarar uma variável dentro de um método ou um construtor ela não será inicializada por padrão, se você não atribuir um valor para ela você não poderá usar essa variável.

**Arrays e Collections**

**Array:**

Um array é uma variável unidimensional, que pode guardar mais de uma informação ao mesmo tempo, mas ele tem um tamanho fixo, ou seja, você tem que definir o tamanho do array no momento da sua declaração.

Caracteristicas do Array:

1. Estático(Tem um tamanho fixo).
2. Homógeno(Ele só aceita um tipo, ou seja, se você dizer que o array é do tipo inteiro, você só pode adicionar valores inteiros nele).
3. O array é uma estrutura indexada(Você acessa uma posição do array através de um valor numérico, começando por 0).
4. Array é um objeto.

Declaração de um array:

double[ ] a = new double [6];

Aqui você declarou um array do tipo real com tamanho igual a 6. E você so poderá adicionar valores do tipo double, não poderá adicionar valores do tipo boolean ou String.

Adicionando um valor no Array:

a[2] = 10.0;

Aqui você estará adicionando o valor 10 no índice 2 do array.

a[0] = 9.2;

Aqui você estará adicionando o valor 9.2 no índice 0 do array.

Para imprimir um Array inteiro, basta usar a classe chamada Arrays, converte o Array para String, exemplo:

Arrays.toString(a);

Ele irá converter o Array para String somente nessa linha, e você pode usar ele diretamente para imprimir no console, ou pode atribuir a uma variável do tipo String:

String vetor = Arrays.toString(a);

ForEach:

for (double nota : notas) {

soma += nota;

}

Um ForEach em Java se faz quase da mesma forma que um laço for normal, só que você apenas passa o tipo da variável que vai iterar pelo Array, no caso vai iterar pelo valor, depois o nome do “parâmetro”, depois dois ponto e por ultimo o Array que irá ser iterado no ForEach.

**Matriz**

Uma matriz, é um Array multidimensional, ou seja, que tem mais de uma dimensão dentro do Array, ou seja, uma **matriz** é um vetor de vetores, todos de mesmo tamanho. Em **Java**, é possível criar um vetor de vetores onde cada elemento tem um tamanho diferente. Nesse caso, cada elemento deve ser criado de forma independente. Todo vetor em **Java** tem o atributo length que define o seu número de elementos.

Para declara uma matriz em java é a seguinte sintaxe:

double[ ][ ] notas = new double [6][2];

Para imprimir uma Matriz inteira, basta usar a classe chamada Arrays, converte o Array para String, exemplo:

System.out.println(Arrays.deepToString(notas));

Ou atribuir esse valor diretamente a uma String:

String matrizDeNotas = Arrays.deepToString(notas);

**Equals e Hashcode**

Em Java se você cria um objeto com o nome de p1, e esse objeto tem o atributo nome, e você atribuir a esse atributo o valor “Caneta”, e depois você criar outro objeto chamado p2, e atribuir o valor “Caneta” para o atributo nome de p2, e tentar comparar o valor dos dois valores usando dois iguais (==), o resultado da comparação será falso, pois no Java não estar á comparando os valores, e sim para o local de memória de cada objeto, pois as comparações com “==” comparam o local de memória dos objetos, e não o seus valores, exemplo:

Produto p1 = new Produto();

p1.nome = “Caneta”;

Produto p2 = new Produto();

p2.nome = “Caneta”;

p1 == p2

O resultado da comparação retornara falso, pois seria a mesma coisa que você colocar pra comparar o local de memória de cada objeto, exemplo:

0x123 == 0x234

O loca de memória do Java é representado por um número hexadecimal.

Todo objeto tem o método “equals” por padrão, e ele funciona da mesma forma que o “==”:

p1.equals(p2);

Retornará falso.

No Java temos a função EQUALS que vai retorna um boolean (true ou false), e também temos a função HASHCODE que vai retornar um inteiro.

O método EQUALS é lento, e por isso se usa o HASHCODE com ele, por exemplo, se você está quer procurar dentro de um conjunto de objetos algum usuário com o nome “Luca” dentro de um conjunto com mais de 100 mil usuários, primeiro você usa o HASHCODE (pois o HASHCODE é muito mais rápido do que o EQUALS) para fazer uma comparação da quantidade de letras do nome do usuário, para que o método HASHCODE separe aquilo que tem probabilidade de ser oque está sendo buscado dos que não tem nenhuma chance de ser oque está sendo buscado, e depois disso você usa o EQUALS num subconjunto dos usuários comparar apenas os que tem chance de ser o usuário buscado, para retornar aquele usuário. OEQUALS e HASHCODE sempre andam junto para comparar igualdades entre objetos em Java.

**Collections:**

Diferente do Array, as collections tem tamanhos variados, e você pode tem uma collection com tipos misturados(inteiro , real , String), mas a boa prática é sempre usar uma collection de um único tipo.

**SET:**

* Não ordenado (Por padrão).
* Não indexado.
* Não aceita repetição(não aceita colocar dois elementos dentro do SET que são iguais.

**LIST:**

* Indexada.
* Aceita repetição.

**MAP:**

* Chave/Valor.
* Chave não aceita repetição
* Valor aceita repetição

**QUEUE:**

* Implementa uma fila.
* First in/ First out (FIFO).

**STACK:**

* Implementa uma pilha(stack).
* Last in/First out(LIFO)

**SET:**

* Pode ser Heterogêneo
* Pode ser Homogêneo
* Não aceita obj. duplicados
* Pode ser ordenado
* Não é indexado

O Set é um dos tipos da Collections, ela é como um conjunto que não aceita repetição, e pode ser heterogênea(que aceita tipos diferentes dentro do conjunto), exemplo:

HashSet conjunto = new HashSet();

Aqui nos estamos criando uma Collection do tipo Set, sem tipo definido, podendo adicionar diferentes tipos, para adicionar um novo valor dentro do Set é:

conjunto.add(1.2);

conjunto.add(true);

conjunto.add(“Teste”);

conjunto.add(1);

conjunto.add(‘x’);

Se usa o “.add” para adicionar um novo valor dentro da Collection Set, e como as Collections não aceitam os tipos primitivos como int, double, char, boolean... Ele vai converter para os tipos objeto como Integer, Double, Character, Boolean....

Para ver o tamanho do Set, diferente do Array que é o “.length”, as Collections normalmente é “.size()”:

conjunto.size();

Para remover um valor da Collection Set se usa o “remove”:

conjunto.remove(“Teste”);

O “.remove” retorna um boolean, se tiver aquele valor que foi passado para remover, ele retorna true e remove o valor, se não ele retorna false e não faz nada. Para verificar se tem algum valor dentro do Set, se usa o “.contains” que também ira retornar true ou false:

conjunto.contains(‘x’);

Para você fazer uma união entre dois Set, você usa o “.addAll”, que vai pegar todos os valores de outro conjunto criado, e adicionado dentro do primeiro conjunto:

HashSet nums = new HashSet();

conjunto.add(1);

conjunto.add(2);

conjunto.add(3);

conjunto.addAll(nums);

O Set “conjunto” vai receber todos os valores do Set “nums” e adicionar dentro dentro dele.

E para você fazer uma intersecção você usa o “.retainAll”, fazendo assim com que retorne true ou false, se ele retornar true é porque os dois conjuntos tem algum elemento em comum, e o conjunto que fazer a intersecção vai ficar apenas com os valores em comum da intersecção:

conjunto.retainAlll(nums);

O Set “conjunto” vai retornar um Set com apenas o valor 1, que é oque os dois tem em comum.

Para limpar uma Collection Set se usa o “.clear”:

conjunto.clear();

Ele vai retornar um “Array” vazio.

Para você criar um Set com o tipo definido, para não misturar os tipos de valores, você usa “<>”, e dentro coloca o tipo do Set:

Set<String> listaAprovados = new HashSet<>();

listaAprovados.add(“Ana”);

listaAprovados.add(“Carlos ”);

listaAprovados.add(“Luca ”);

Nesse exemplo o listaAprovados vai aceitar somente valores do tipo String, E você pode declarar o tipo tanto com HashSet ou apenas Set. Por padrão ele não é ordenado, ou seja, não vai sair os valores na ordem em que foram adicionados, para você fazer com que ele ordene na ordem em que os valores foram adicionados, basta usar o TreeSet no lugar do HashSet:

SortedSet<String> listaAprovados = new TreeSet<>();

E por padrão se usa o SortedSet na declaração ou o próprio TreeSet também. E diferente do Array, as Collections imprimem como se fossem um Array, mas inteiro, sem mostrar o código do local da memória, e sim o próprio “Array”.

**List:**

* Pode ser Heterogêneo
* Pode ser Homogêneo
* Aceita obj. duplicados
* É ordenado
* É indexado

A List é um das Collections que é indexada, aceita repetição, e é ordenada da mesma forma que um Array, organizando pela ordem que foi adicionado o valor, e como o Set o List pode ser de apenas um tipo, ou aceitar vários tipos diferentes:

List<Usuario> lista = new ArrayList<>();

Aqui nos declaramos uma lista do tipo Usuario, que é um objeto, e para adicionar novos Objetos do tipo Usuario dentro da lista é:

lista.add(new Usuario(“Ana”));

Ou declarar o usuário e passar direto o objeto:

Usuario u1 = new Usuario(“Pedro ”);

lista.add(u1);

Para acessar uma lista pelo índice usa o “.get”:

System.out.println(lista.get(1));

O resultado no console vai ser “Pedro”. Da mesa forma que o Set tem o “.remove” e o “.contains” a lista também tem essas propriedades.

Em relação ao “.remove” da lista é que você também pode remover algum valor pelo índice:

lista.remove(1);

Ele vai remover o item que está no indice “1”.

**Queue(Fila):**

A Collections Queue(Fila) usa o método First in/ First Out(primeiro que entra, é o primeiro que sai), e para declara uma fila é:

Queue<String> fila = new LinkedList<>();

Aqui estaremos criando uma Fila do tipo String, e para adicionar um valor na fila temos 2 métodos:

fila.add(“Ana”);

fila.offer(“Bia”);

Os dois métodos vão adicionar os valores na fila, a diferença é que se você tiver um tamanho definido da fila de 100, e usar o “.add” para adicionar e chegar a 100 ou passar a quantidade de valores que pode adicionar, vai dar um erro, já o “.offer” vai retornar false, e só vai adicionar quando “liberar” lugar na fila.

Da mesma forma temos 2 métodos para selecionar um elemento da fila:

System.out.println(fila .peek());

Ele vai retornar a “Ana”.

E se você usar de novo ele não vai remover a “Ana”, vai retornar ela de novo.

E tambem tem o “.element”:

System.out.println(fila .element());

Que também vai retornar a “Ana”, a diferença dos dois é quando a fila estiver vazia o “.peek” vai retornar null, e o “.element” vai dar um erro.

O “.poll” ele retorna o primeiro elemento da fila, e o remove:

System.out.println(fila .poll());

Vai retornar a “Ana” e já remove-la. Se a lista estiver vazia ele vai retornar null, e se você usar o “.remove” e a lista estiver vazia, ela vai lançar uma exceção e dar um erro.

**Stack (Pilha):**

A Collections Stack (Pilha ) usa o método Last in/ First Out(o último que entra, é o primeiro que sai), e para declara uma pilha é:

Deque<String> livros = new ArrayDeque<>();

Aqui nos declaramos uma pilha do tipo String, e os métodos “.add”, “.poll”, “.remove”, “.element”, “.peek” das outras Collections também se aplicam na Collection Pilha, além também do “.size” para ver o tamanho da pilha, “.clear” para limpar a pilha, “.contains” para verificar se tem algo dentro daquela Collection, “.isEmpty” para verificar se aquela Collection está vazio, além de outros métodos. A diferença é que a pilha também tem o método “.push” para adicionar no começo(ou no caso da pilha, no topo dela) da pilha, e o “.add” vai adicionar no final(no caso da pilha, em baixo).

livros .add(“O Pequeno Príncipe”);

livros.push(“O Hobbit”);

O livro “O Hobbit” vai ser adicionado no começo da pilha(na teoria, no topo dela), e se você adicionar outro valor com o “.push”, ela via ficar antes do “O Hobbit”, e se você usar o “.add” vai ficar depois do “O Pequeno Príncipe”. E também tem o “.pop” que vai funcionar que vai funcionar da mesma forma que o “.remove”, vai remover o primeiro item da pilha, e se a pilha estiver vazia vai retornar um erro.

**Map:**

O Map é um objeto que mapeia valores para chaves, ou seja, através da chave consegue ser acessado o valor configurado, sendo que a chave não pode ser repetida ao contrário do valor, mas se caso tiver uma chave repetida é sobrescrito pela última chamada. Para declarar um Map é:

Map <Integer, String> usuarios = new HashMap<>();

A Collection Map não tem o “.add”, para você adicionar algo nele se usa o “.put”:

usuarios.put(1, “Roberto”);

Sendo o primeiro a chave, e o segundo o valor.

usuarios.put(20, “Ricardo ”);

usuarios.put(3, “Rafaela ”);

usuarios.put(4, “Rebeca ”);

Para ver todas as chaves do Map, se usa o “.keySet”:

System.out.println(usuarios .keySetl());

Ele vai imprimir como um Array com todas as chaves do Map. Para pegar apenas os valores se usa o “.values”:

System.out.println(usuarios .values());

Ele vai imprimir todos os valores do Map como “Roberto”, “Rafaela” e etc. E caso você queira pegar os dois valores se usa o “.entrySet”:

System.out.println(usuarios .entrySet());

Ele vai imprimir tanto a chave quanto o valor do Map.

Para remover um valor dentro do Map, você pode passar apenas a chave, que ele vai remover aquele item que tem a chave que foi passada, ou a chave e o valor, que ele só vai remover se o valor for referente aquela chave:

System.out.println(usuarios .remove(1));

Aqui ele vai remover o item cuja chave é igual a 1.

System.out.println(usuarios .remove(4, “Gui ”));

E aqui ele só vai remover se o valor da chave 4 for igual a “Gui”.

No caso do Map, não temos o “.contains” e sim o “.containsKey” e o “.containsValue” para verificar se contem um valor ou chave:

System.out.println(usuarios.containsKey(4));

Aqui estamos verificando se o Map usuarios tem a chave 4.

System.out.println(usuarios.containsValue(“Rebeca ”));

Aqui estamos verificando se o Map usuarios tem o valor “Rebeca ”.

E também podemos pegar um valor pela chave usando o “.get”:

System.out.println(usuarios.get (4));

Aqui retornara o valor “Rebeca”, que está na chave 4.

E temos 3 formas de percorrer um Map usando o forEach, pela chave, pelo valor ou pela chave e pelo valor:

for (int chave : usuarios .keySetl()) {

System.out.println(chave);

}

Aqui será retornado cada uma das chaves do Map usuarios.

for (String valor : usuarios.values()) {

System.out.println(chave);

}

Aqui será retornado cada uma dos valores do Map usuarios.

for (Map.Entry<Integer, String> registro : usuarios .entrySet()) {

System.out.println(chave.getKey);

System.out.println(chave.getValues);

}

Aqui será retornado cada uma das chaves e valores do Map usuarios.

**Orientação a Objetos**

rever a parte de relacionamento

**Pilares:**

**Encapsulamento:**

Conceitua-se **encapsulamento** como sendo o processo utilizado para proteger os campos e operações de uma classe (atributos e métodos), permitindo que apenas os membros públicos - em **Java** métodos Get / Set - sejam acessados pelos usuários de determinada classe.

Encapsulamento é o ato de privar(deixar oculto) algo que só pode ser acessado pelo próprio objeto, mas não pode ser acessado por outro objeto que faz referencia ao objeto que tem uma classe privada.

**Modificadores de Acesso:**

**Public:**

Ao criar um atributo publico, significa que qualquer classe do sistema pode acessar o atributo ou método publico de outra classe.

**Protected :**

Um atributo protegido é visível no próprio pacote, na própria classe e é transmitido por herança para um outro pacote.

**Package(Default):**

Um atributo de nível pacote(padrão), só pode ser acessado por outra classe do mesmo pacote, e não por outras classes de pacotes diferentes.

**Private:**

Ao criar um atributo privado dentro da classe, ele só pode ser acessado dentro da classe, ou seja, não pode ser acessado por outras classes.

**Herança:**