Introdução ao Java

12 de março de 2021
00:45

JAVA é uma linguagem fortemente tipada, já que todas as variáveis têm um tipo de dados específico e todos os métodos só aceitam como parâmetros tipos de dados bem definidos.

Os objetos são armazenados na memória *heap* e manipulados através de uma referência (variável), guardada na pilha.

Podemos usar o **mesmo nome** em vários métodos, desde que tenham **argumentos distintos** e que concetualmente executem a mesma ação.
Não é possível distinguir funções, pelo valor de retorno.

"Hello" + 2 → "Hello2"
"Hello" * 2 → ERRO: **O Java não permite a multiplicação de *strings***, com recurso ao operador aritmético *. A solução é utilizar o método replace().

```
public class Programa {
	//declaração de dados
	//declaração de métodos
	public static void main(String[] args) {
		//TODO
	}
```

args - argumentos do programa que está a correr.

print(…, end="\n") => System.out.println(…);
print(…, end="") => System.out.print(…);

nextLine() - lê uma linha inteira (string)
next() - lê uma palavra (string)
nextInt - lê um nº inteiro
nextDouble - lê um nº real

int[] vetor = new int[n]; //criação de um vetor de dimensão n
int[] vetor = new int[]{v1, v2, …, vn} ou simplesmente **int[] vetor = {v1, v2, …, vn}**.

Não é possível adicionar elementos a um vetor, apenas redefinir o valor dos seus indexes.

int[][] tabela = new int[lines][columns]

int a = 1;

Linha	a	b
int b = ++a;	2	2
int b = a++;	2	1

int b = a++;
EQUIVALENTE A

int b = a;
a++;

DADOS

Tipo	Dimensão	Representação
boolean	1 bit	0 ou 1
byte	8 bits	Complemento para 2
short	16 bits	Complemento para 2
char	16 bits	Unicode
int	32 bits	Complemento para 2
long	64 bits	Complemento para 2
float	32 bits	IEEE 754 (precisão simples)
double	64 bits	IEEE 754 (precisão dupla)

int(5.5) = 5

float a = 5; OK
float a = 5.5f; OK
float a = 5.5; NO (5.5 is a double)

2/10 = 0
2/10.0 = 2.0/10 = 2.0/10.0 = 0.2

++x e --x têm prioridade sobre as restantes operações aritméticas.

```
int a = 5;
int b = -15;
System.out.println(++a-b/30);
//(5+1) - (-15/30) = 6 - (int(-0.5)) = 6 - 0 = 6
```

Operador ternário (?:)
variable = testCondition ? valueIfTrue : valueIfFalse
Ex: x = (a < b) ? a : b;

do
	//something; (corre, pelo menos, 1 vez);
while (condition);

for (inicialização; condição; atualização) {//corpo;}

for (classeElemento elemento: vetor) {//corpo;}
Exemplo:
int[] vetor = new int[5]
for (int i : vetor) {//do something;}

switch(variável) {

```
        case valor1:
                //do something;
                break;
        case valor2:
                //do something;
                break;
        (…)
        default:
                //do something;
}
```

A instrução '*continue*' permite terminar a execução da iteração em curso, forçando a passagem para a **iteração seguinte**.
A instrução '*break*' permite a **saída** imediata do bloco de código.
A instrução '*return*' é reservada para funções. Não é usada para ciclos, mas frequentemente em ciclos.

Métodos estáticos
y = funcStat(x); //Não é preciso criar uma instância.
y = x.funcNonStat(); // É preciso criar uma instância.

Math m1 = new Math();
m1.pow() NO

pow() OK
No caso de termos importado a biblioteca Math completa, Math.pow().

Módulo com funções matemáticas: Java.lang.Math;
Módulo com funções de manipulação de strings: Java.lang.String;

string.length() OK
string.length NO
Para as strings, length() é um método. Para os arrays, length é um atributo.

As strings são imutáveis. Quando tentamos modificar uma string, estamos, na verdade, a criar uma nova string e a sobrescrever o seu conteúdo, na variável original.
data = "string";

Também é possível utilizar string builders:
String builder sb = new StringBuilder();
sb.**append**(10);
sb.append(" de");
sb.append(" fevereiro");
String data = sb.**toString**();

ou criar strings a partir de uma array:
char lista[] = {'s', 't', 'r', 'i', 'n', 'g'};
String data = new String (lista);

String.format("%s", string) - returns string
String.format("%d", integer) - returns integer
String.format("%f", float/double) - returns float/double

```
System.out.printf("|%8s|", "Hello"); → |   Hello|
System.out.printf("|%-8s|", "Hello"); → |Hello   |
System.out.printf("|%8d|", 1111); → |    1111|
System.out.printf("|%-8d|", 1111); → |1111    |
System.out.printf("|%08d|", 1111); → |00001111|
System.out.printf("|%8.1f|", 19.5); → |    19,5|
System.out.printf("|%-8.1f|", 19.5); → |19,5    |
System.out.printf("|%8.4f|", 19.5); → |19,5000|
```

int x = String1.compareTo(String2)

- String1 > String2 → x = 1
- String1 == String2 → x = 0
- String1 < String2 → x = -1

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
concat()	Appends a string to the end of another string	String
contains()	Checks whether a string contains a sequence of characters	boolean
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
format()	Returns a formatted string using the specified locale, format string, and arguments	String
compareTo()	Compares two strings lexicographically.	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int
isEmpty()	Checks whether a string is empty or not	boolean

lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
matches()	Searches a string for a match against a regular expression, and returns the matches	boolean
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Extracts the characters from a string, beginning at a specified start position, and through the specified number of character	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

Alguns exemplos de padrões regex

- . qualquer caracter
- \d dígito de 0 a 9
- \D não dígito [^0-9]
- \s "espaço": [\t\n\x0B\f\r]
- \S não "espaço": [^\s]
- \w carater alfanumérico: [a-zA-Z_0-9]
- \W carater não alfanumérico: [^\w]
- [abc] qualquer dos carateres a, b ou c
- [^abc] qualquer carater exceto a, b e c
- [a-z] qualquer carater entre a-z, inclusive
- X? um ou nenhum X
- X* nenhum ou vários X
- X+ um ou vários X

(?!) - ignora maiúsculas e minúsculas.

String x = "ola\n";
System.out.println(x); OK

System.out.println("ola\n"); NO
System.out.println("ola\\n") OK
Num argumento de uma função, as expressões regulares devem ser precedidas por "\".

Quantificadores em expressões regulares:

X{n}	X aparece n vezes.
X{n,}	X aparece, pelo menos, n vezes.
X{n,m}	X aparece, pelo menos, n vezes, mas não mais que m vezes.
X?	X não aparece ou aparece 1 vez.
X*	X não aparece ou aparece várias vezes.
X+	X aparece 1 vez ou aparece várias vezes.

Metacaracteres de fronteira:

^	Inicia
$	Finaliza
|	Ou (condição)

Alguns exemplos:

```
//Pesquisa se uma palavra existe no texto
boolean palavra = "Hello World Java".matches(".*Java.*");

//Pesquisa pela palavra "Benfica" ou "Porto"
boolean equipa = "Benfica".matches("Benfica|Porto");

//Java + qualquer char 0 ou várias vezes = começar por "Java"
System.out.println("Java Hello World".matches("Java.*"));

//Qualquer char 0 ou várias vezes + Java = terminar em "Java"
System.out.println("Hello World Java".matches(".*Java"));

//Qualquer char 0 ou várias vezes + Java + Qualquer char 0 ou várias veze
s = conter "Java"
System.out.println("Hello Java World".matches(".*Java.*"));


System.out.println("!!!!Hello0000".matches("\\W{4}Hello\\d{4}")); //true
```

Pode ser útil indicar a string vazia, representada por ".{0}":

```
System.out.println("!!!!Hello0000".matches("\\W{4}.{0}Hello.{0}\\d{4}"));
//true
```

Pode ser útil indicar um intervalo para um char ou um int.

```
System.out.println("!!!!Hello0000".matches("\\W{4}.{0}Hello.{0}[0-
9]{4}")); //true

System.out.println("aa3a?".matches(".*(\\d|\\W).*")); //true
System.out.println("aa3a".matches(".*(\\d|\\W).*")); //true
System.out.println("aaa?".matches(".*(\\d|\\W).*")); //true
System.out.println("aaa".matches(".*(\\d|\\W).*")); //false
```

Classes e Objetos

9 de maio de 2021
11:16

Encapsulamento
public - pode ser usado em qualquer classe.
(default) "omissão" - visível dentro do *mesmo package*.
protected - visível dentro do *mesmo package e classes derivadas*.
private - apenas visível dentro da classe.

```
class nomeClasse {
	//atributos
	ex.: int num; String name; …
	// Seletores (gets)
	public int getNum() {return num;}
	public String getName() {return name;}
	// Modificadores (sets)
	public void setNum(int num) {this.num = num;}
	public void setName(String name) {this.name = name;}
	//construtor
	public nomeClasse(int num, String name) {
		this.num = num;
		this.name = name;
	}
	//toString
	@Override // boa prática
	public String toString() {return name + " " + num;}
	//outros métodos
}
```

@Override - marcador usado para reescrever um método.

Tendo a seguinte classe,

```
public class Person {
	private String name;
	private int age;
	public Person(String name, int age) {
		this.name = name;
		this.age = age;
	}
}
Person p1 = new Person();
System.out.println(p1.name); ERRO -> "name is not visible"
```

É preciso criar a função public String getNome(){}, para aceder ao nome.

Se quisermos incrementar um atributo, a cada instanciação da classe (criação de um novo objeto):

- ATRIBUTOS: private **static** int counter = 0;
- CONSTRUTOR: counter ++;

Instanciação:

- nomeClass c1 = new nomeClass();
- nomeClass c1 = new nomeClass(num, name);

Garbage collector é o mecanismo que apaga atributos da memória, quando estes já não são visíveis.
Por omissão dos níveis de controlo de acesso (public, protected e private), são as chavetas { } que definem o alcance (*scope*) das referências a objetos.

```
{
      int num = 0;
}
int ++; // não vai funcionar
```

```
Tipo          Valor por omissão
-----------------------
boolean     false
char           '\u0000' (null)
byte           (byte)0
short          (short)0
int            0
long           0L
float          0.0f
double         0.0d
```

int x; // Como é um tipo de dados primitivo, tem valor predefinido.
RandomClass c1; // Como não é um tipo de dados primitivo, não tem valor predefinido.

Não inicializar uma variável é diferente de esta ter valor "null".
Contudo, num vetor de um tipo **não primitivo**, **o valor predefinido de cada index é "null"**.

```
String[] texto = new String[10];
System.out.println(texto[0]);
//devolve "null"
Animal[] vetor = new Animal[10];
System.out.println(vetor[0]);
//devolve "null"
```

final int i;
i = 1; //a partir do momento em que é atribuído um valor à variável, este não pode ser alterado, ou seja, é constante.

static attribute	Atributo comum a todas as instâncias da classe.
static method	Operação que não depende da instância da classe.
static class	**Não pode ser instanciada**, tal como a abstract class.
final attribute	Não permite reatribuição de valor.
final method	Não permite "override".
final class	Não permite "extends".

Herança e Polimorfismo

9 de maio de 2021
11:16

Herança (is-A)
class Pessoa {…}
class Aluno extends Pessoa {…}
Aluno herda todas os atributos e métodos de Pessoa e tem outros atributos e métodos particulares.
Em Java, todas as classes derivam da super class java.lang.Object.

Uma classe base pode ter múltiplas classes derivadas, mas **uma classe derivada não pode ter múltiplas classes base**.

Não podemos reduzir a visibilidade dos métodos herdados numa classe derivada.
Public > Protected > Default > Private

Podemos aumentar a visibilidade de métodos herdados numa classe derivada, excetuando métodos private, uma vez que estes nem são herdados.

Classe Base	Classe Derivada	Permitido?
public	protected/default/private	X
protected	default/private	X
default	private	X
private	default/protected/public	Nem sequer é herdado.
default	protected/public	✔
protected	public	✔

Nos construtores das classes derivadas,
super()
this.attribute = attribute
OK

this.attribute = attribute
super()
NO

super(), caso seja enunciado, é sempre a primeira instrução.

Composição (has-A)
class Pilha {}
class Relógio {Pilha p; …}

Métodos comuns a todos objetos:

- toString()

- equals()
- hashCode()

Upcasting:
`Pessoa p1 = new Aluno("Rafael Gonçalves", 15186693);` Específico → Geral (OK)

Downcasting:
`Aluno a1 = new Pessoa("Rafael Gonçalves", 15186693);` Geral → Específico (NO)
`Aluno a1 = (Aluno) new Pessoa("Asdrúbal", 15186693);` (OK)

double x = 5;
geral para específico

int x = (int) 5.0;
específico para geral

Método instanceof testa o tipo do objeto. É válido, tanto para a classe e superclasses, como para uma interface.

Polimorfismo
Uma classe abstrata (basta conter pelo menos um método abstrato) **não é instanciável**. Contudo, **pode criar-se uma referência** para uma classe abstrata.
Numa hierarquia, a classe **só deixa de ser abstrata, quando implementar todos os métodos abstratos**.

```
class abstract Forma {
        (…)
        public abstract double area();
        public abstract double perimetro();
        (…)
}
class Circulo extends Forma {
        (…)
        public double area() {return Math.PI*r*r;}
        public double perimetro() {return 2*Math.PI*r;}
        (…)
}
```
Como Circulo implementa os métodos abstratos de Forma, já é possível instanciar: Circulo c1 = new Circulo(…);

A interface é uma classe, que atua como um protocolo que é seguido pelas classes que a implementam.
Só contém assinaturas de funções, isto é, uma lista das funções que devem ser definidas nas classes que a implementam.
Todos os seus métodos são implicitamente abstratos, pelo que **não pode ser instanciada**, e todas as suas variáveis são implicitamente estáticas e constantes **(static final)**. **Não tem construtores e permite herança múltipla**.
Uma classe (não abstrata) que implemente uma interface deve implementar todos os seus métodos.
Uma classe pode implementar uma ou mais interfaces.

Semelhanças com a classe abstrata:

Não se pode criar uma instância da interface.
Pode criar-se uma referência para uma interface.

Uma interface pode ser vazia.

```
inteface Desenho() {
        (…)
        public cor(Color c);
        static corDeFundo(Color cf);
        default void desenha(DrawWindow dw);
        (…)
}
class Circulo extends Forma implements Desenho {
        (…)
        public cor(Color c);
        @Override
        public void desenha(Draw Window dw) {…};
        (…)
}
```

Tanto os métodos default como static oferecem uma implementação, por omissão. Contudo, enquanto que os primeiros podem ser reescritos, os segundos não.

Enquanto que classes abstratas descrevem entidades e permitem heranças simples, isto é, só podem herdar atributos de uma única classe, interfaces descrevem comportamentos funcionais e permitem herança múltipla, isto é, podem herdar atributos de várias classes.

Enums

14 de junho de 2021
15:01

Têm como finalidade restringir os valores possíveis de uma variável. **Não são primitivas**, só têm **construtores privados**, podem implementar **interfaces**, suportam **comparação** e os seus valores são objetos.
Por convenção, os valores enumerados são escritos em maiúsculas.
Os valores enumerados são **constantes (public static final)**, criadas aquando da compilação, mas são **instâncias** da enum.
Contribuem para a "*compile-time type safety*", isto é, o compilador tem a garantia de que certos erros não vão existir no código.
Geralmente, são declaradas em ficheiros próprios, mas também podem surgir dentro de uma classe já existente.
A instrução switch funciona com os valores enumerados.
Limitação importante: **Não podemos ler um enum através de um Scanner**.

Métodos úteis:

- **Enum.valueOf**(String val): passa de "ELEMENTO" (string, em maiúsculas) para ELEMENTO (valor enumerado);
- **Enum.values**(): lista de elementos.
- ELEMENTO.**ordinal**(): posição (int) na lista de elementos;
- ELEMENTO.**name**(): devolve a string correspondente.

Método que deve ser implementado:

```
@Override
public String toString() {return name().charAt(0) + name().substring(1).toLowerCase();}
```

Exceções

24 de maio de 2021
09:35

Vantagens das exceções:
Separação clara entre o código regular e o código de tratamento de erros, não sendo necessárias complexas estruturas condicionais.
Propagação dos erros em chamadas sucessivas de métodos.
Agrupamento de erros por tipos, uma vez que esses estão hierarquizados.

Não devem ser usadas para controlo de fluxo (solução: estruturas condicionais), nem para um simples teste (asserções).

Alguns exemplos de exceções
- IllegalArgumentException
- IllegalStateException
- NullPointerException
- IndexOutOfBoundsException

Controlo de exceções com try-catch:

```
try {
        /*o que se pretende fazer*/
}
catch (Exception1 e) {
        /*fazer algo com a exceção1*/
}
catch (Exception2 e) {
        /*fazer algo com a exceção2*/
}
catch (Exception e ) [
        /*fazer algo com qualquer outra exceção*/
}
…
finally {
        /*o que se pretende fazer, independentemente de haver exceção ou não*/
}
```

A ordem dos catch baseia-se na hierarquia das exceções.

Controlo de exceções com try-with-resources - assegura que os recursos são fechados e não há bloco finally:

```
try (código que pode gerar Exception1, Exception2, …) {
catch (Exception1 e) {
        /*fazer algo com a exceção1*/
}
catch (Exception2 e) {
        /*fazer algo com a exceção2*/
}
```

Delegação do erro:

```
void method1() {
```

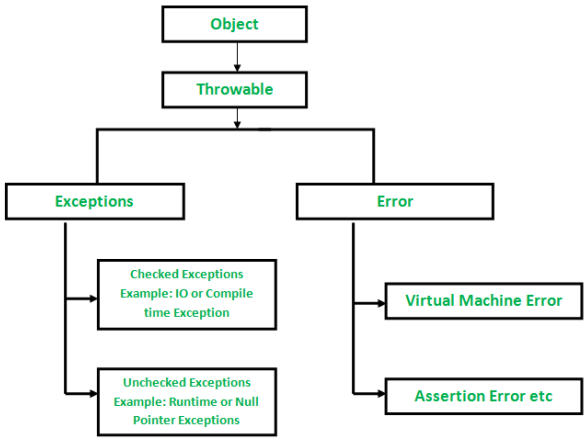
```
        try {
                method2();
        }
        catch (SomeException e) {
                /*processar o erro*/
        }
}
void method2() throws SomeException {
        method3();
}
void method3() throws SomeException {
        /*o que gera someException*/
}
```

SomeException => Exception/SpecificException

Posso delegar mais do que uma exceção da seguinte forma:
void someMethod() throws Exception1, Exception2 {}

Tipos de exceções

- **checked**: são detetadas durante a compilação, pelo que os métodos que as geram devem resolvê-las (try-catch/try-with-resources) ou delegá-las (throw).
- **unchecked**: não são detetadas durante a compilação, mas sim durante a execução. Podem ser evitadas, com a utilização de *asserções*.



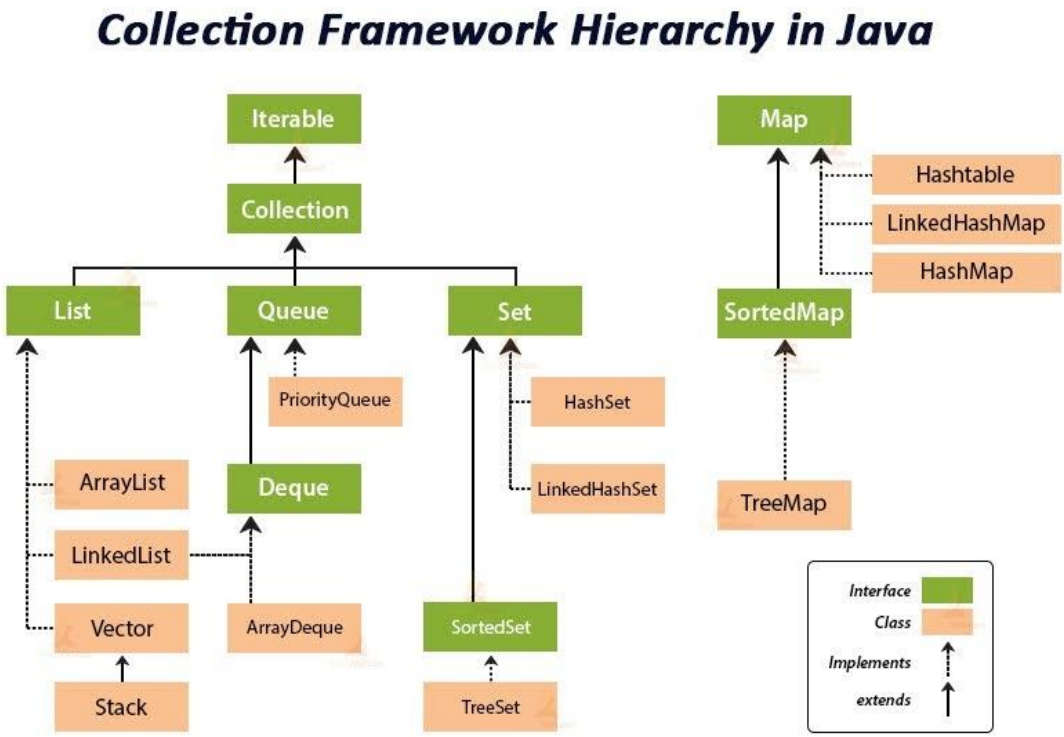
Nova exceção

```
class MyException extends Exception {
        //interface base
        //Construtores:
        public MyException() {}
        public MyException(String msg) {
                super(msg);
        }
        //podemos acrescentar construtores e dados
}
```

Pode ser útil fazer no main throws IOException.

Coleções

14 de junho de 2021
20:30



Coleções: conjunto de classes, interfaces e algoritmos que representam várias estruturas de dados.
Não suportam tipos primitivos, pelo que temos de usar as suas classes adaptadoras.
int -> Integer
float -> Float
double -> Double

Esta framework de Java apresenta várias interfaces, separa a especificação (classes) da implementação (interfaces). Deste modo, pode-se substituir uma implementação por outra mais eficiente, sem grandes impactos na estrutura.

ITERABLE: **forEach**(lambda/methodReference), **iterator**().

- **Coleção(Collection<TypeObject>)**: **add**(TypeObject), addAll(Collection<TypeObject>), **remove**(Object), removeAll(Collection<Object>), **contains**(Object), containsAll(Collection<Object>), **isEmpty**(), **size**(), stream(), retainsAll(Collection<Object>) [faz uma interseção de coleções], **toArray**().
    - **Conjunto(Set)**: sem ordem e repetição. A implementação de equals(), na classe dos elementos do conjunto, é imperativa, uma vez que um elemento não será inserido, se esse método devolver true.
        - HASHSET - Usa uma tabela de dispersão (Hash Map) para armazenar os elementos. // Tem um desempenho *constante*.
        - **Conjunto ordenado (SortedSet**): first(), last().

        - TREESET - Permite a ordenação dos elementos pela sua "ordem natural". Para isso, a classe dos objetos inseridos deve implementar a interface Comparable ou o próprio TreeSet deve incluir, na chamada do seu construtor, um objeto do tipo Comparator. // Tem um desempenho *logarítmico* para add, remove e contains.
    - **Lista(List)**: com ordem e repetição - **get**(int index), **indexOf**(Object), **lastIndexOf**(Object), listIterator() [este iterador implementa hasPrevious(), previous(), previousIndex(), hasNext(), next(), nextIndex(), add(TypeObject), remove(TypeObject)], listIterator(int index) [iterador a partir de um certo index], **subList**(int fromIndex, int toIndex), subList(int fromIndex, int toIndex).**clear**() [para remover uma sublista] - Podem ser percorridas com um for(Type element : list) ou com um listIterator. - É possível obter uma lista, a partir de um array, recorrendo a **Arrays.asList**(elementos).
        - ARRAYLIST - usa um array dinâmico e é indicada para leitura e escrita de dados.
        - LINKEDLIST - usa uma lista duplicada, cujas cópias estão ligadas, o que permite que um elemento seja rapidamente removido, sem deslocamento de bits. É indicada para manipulação de dados.
        - VECTOR - também array dinâmico.
            - STACK
    - **Fila(Queue)**: fila do tipo First in First Out (efeito dominó*) - **offer**(TypeObject) insere o elemento especificado, **remove**() [*retorna exceção*, se a fila está vazia] e **poll**() [não retorna exceção] removem e retornam o elemento do topo da fila, **element**() [*retorna exceção*, se a fila está vazia] e **peek**() [*não retorna exceção*] retornam o elemento do topo da fila.
        - LINKEDLIST também é uma fila.

forEach() executa o que é passado no argumento, para cada elemento do iterável, até todos serem processados ou até a ação gerar uma exceção.
iterator() devolve um iterador, em função do tipo de dados do iterável. O iterador, por sua vez, é uma interface, desenhada para alterar rapidamente a coleção que percorre, pelo que implementa **hasNext()**, **next()** e **remove()**.

```
Iterator<Type> it = randomCollection.iterator();

while(it.hasNext()) {
  System.out.println(it.next());
}
```

MAP: **put**(key, value), **get**(key), **getOrDefault**(key, valor caso a key não seja encontrada), remove(key), **containsKey**(key), **containsValue**(value), **size**(), **isEmpty**(), putAll(someMap), clear() - estrutura associativa, onde os objetos são representados por pares chave-valor. Também é denominado por dicionário. // As vistas dos mapas, que os permitem percorrer, são coleções. Há 3 vistas disponíveis: **entrySet**() [não há pares chave-valor repetidos], **keySet**() [não há chaves repetidas], **values**() [pode ter valores repetidos].

- HASHMAP - Utiliza uma tabela de dispersão (hash table) e *não existe ordenação* nos pares.
- LINKEDHASHMAP - É semelhante ao hashmap, mas *preserva a ordem de inserção*.
- **Mapa ordenado (SortedMap)**: first(), last().
  - TreeMap - apresenta uma estrutura em árvore e os pares são ordenados, com base na chave. Para isso, a classe das chaves deve implementar a interface Comparable ou o próprio TreeMap deve incluir, na chamada do seu construtor, um objeto do tipo Comparator. //O desempenho para a inserção e remoção é *logarítmico*.

Entry é uma interface que lista os pares chave-valor de um mapa. Suporta os métodos: **getKey()**, **getValue()**, **setValue**(SomeValue), **comparingByKey**(nothing or comparator), **comparingByValue**(nothing or comparator).

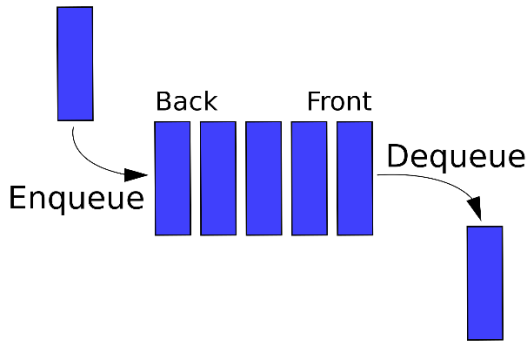
Como incrementar um valor inteiro de um mapa, dentro de um ciclo?
mapa.put(key, mapa.getOrDefault(key, 0) + 1)

Como adicionar um elemento a uma lista de um mapa, dentro de um ciclo?
mapa.put(key, mapa.getOrDefault(key, new ArrayList<>()).add(elemento)))

Como adicionar mais do que um elemento a uma lista de um mapa, dentro de um ciclo?
mapa.put(key, mapa.getOrDefault(key, new ArrayList<>()).addAll(Array.asList(elementos))))

Nota: Uma classe parametrizada, isto é, com <T>, não pode ser instanciada com tipos primitivos.

*FIFO (First In First Out):



Entra um elemento, sai o primeiro.

Ficheiros

15 de junho de 2021
12:33

JAVA IO

- File: canRead(), canWrite(), **exists()**, **getName()**, **isFile()**, **isDirectory()** , **listFiles()** , toPath() [remete para o Java NIO], toURI(), listRoots(), **mkdir()** [cria a pasta com o caminho indicado no construtor de File], **getAbsolutePath()** [home/../file.txt], getCanonicalPath() [home/desktop/Documents/file.txt].
- **Scanner**
  1. Sem tratamento de exceções [input fechado na main]

```
public class ReadFile() {
    public static void main(String[] args) throws FileNotFoundException  {
        Scanner input = new Scanner(new File(LOCALIZAÇÃO));
        while (input.hasNextLine()) {
            System.out.println(input.nextLine();
        }
        input.close()
    }
}
```

  2. Try-catch (sem finally) [input fechado no try]

```
public class ReadFile() {
    public static void main(String[] args) {
        try {
            Scanner input = new Scanner(new File(LOCALIZAÇÃO));
            while (input.hasNextLine()) {
                System.out.println(input.nextLine();
            }
            input.close();
        }
        catch (FileNotFoundException e) {
            System.out.println("Ficheiro não encontrado.");
        }
    }
}
```

  3. Try-catch (com finally) [input fechado no finally]

```
public class ReadFile() {
    public static void main(String[] args) {
        Scanner input = null;
        try {
            input = new Scanner(new File(LOCALIZAÇÃO));
            while (input.hasNextLine()) {
                System.out.println(input.nextLine();
            }
        }
        catch (FileNotFoundException e) {
            System.out.println("Ficheiro não encontrado.");
        }
        finally {
            if (input != null) input.close();
        }
    }
```

```
}
```

4. Try-with-resources [input fechado automaticamente]

```
public class ReadFile() {
    public static void main(String[] args) {
        try (Scanner input = new Scanner(new File(LOCALIZAÇÃO))) {
            while (input.hasNextLine()) {
                System.out.println(input.nextLine();
            }
        }
        catch (FileNotFoundException e) {
            System.out.println("Ficheiro não encontrado.");
        }
    }
}
```

- **PrintWriter**
  1. **Sem append**

```
public class WriteFile() {
    public static void main(String[] args) throws IOException {
        PrintWriter output = new PrintWriter(new File(LOCALIZAÇÃO));
        output.println("Hello World");
        output.printf("Idade: %d\n", 19);
        output.close(); //Não fechar impede a escrita!
    }
}
```

  2. **Com append**

```
public class WriteFile() {
    public static void main(String[] args) throws IOException {
        FileWriter filewriter = new FileWriter(LOCALIZAÇÃO, true);
        PrintWriter output = new PrintWriter(filewriter);
        output.append("Hello World");
        output.close(); //Não fechar impede a escrita!
    }
}
```

- **FileReader**: new FileReader(LOCALIZAÇÃO), **read()** [lê apenas um caracter], read(char[]) [lê caracteres e guarda-os no array indicado] e **read**(char[], start, length) [lê um nº de caracteres igual a length e guarda no array, a partir da posição start], close(), getEncoding() - os métodos de leitura devolvem -1, quando esta chega ao fim. // Para definir a codificação de leitura, podemos indicá-la no construtor new FileReader(LOCALIZAÇÃO, Charset.forName("UTF8")).
- **FileWriter**: new FileWriter(LOCALIZAÇÃO), new FileWriter(LOCALIZAÇÃO, **boolean append?**), **write()** [escreve apenas um caracter], read(char[]) [escreve os caracteres do array indicado] e write(String) [escreve a String], close(), getEncoding() - Para definir a codificação de leitura, podemos indicá-la no construtor new FileWriter(LOCALIZAÇÃO, Charset.forName("UTF8")).
- **RandomAccessFile**: new RandomAccessFile(LOCALIZAÇÃO, mode), **seek(int)**, **read(byte[])**, **write(byte[])**, writeByte(byte), readInt(), writeInt(int), readInt(), writeBoolean(booelan), readBoolean(), writeUTF(String), readUTF(String), length() - modos: "r" [leitura], "w" [escrita] e "rw" [leitura e escrita].

  Copiar bytes 10-19 para 0-9:

```
RandomAccessFile input = new RandomAccessFile(LOCALIZAÇÃO, "rw");
```

```
byte[] buf = new byte[10];
input.seek(10);
input.read(buf);
input.seek(0);
input.write(buf);
input.close();
```

JAVA NIO

- **Paths** - métodos estáticos, que permitem converter Strings ou um URI num Path: get(Strings), get(LOCALIZAÇÃO).
- Path - classe que é criada com os métodos Paths.get(): toString(), **getFileName()**, **getName(index)** [devolve o item, pasta ou ficheiro, da hierarquia na posição indicada], getNameCount(), subpath(fromIndex, toIndex), **getParent()**, **getRoot()**.
  - Path p = Paths.get(LOCALIZAÇÃO);
  - Path p = **FileSystems.getDefault()**.getPath(LOCALIZAÇÃO)
  - Path p = Paths.get(args[0]);
  - Path p = Path.get(URI.create("file:///Users/Rafa/Teste.java");
- **Files**: **readAllLines(Path, Charset.forName("UTF8"))** [devolve uma lista com as linhas], readAllBytes(Path), copy(src, dst, StandardCopyOption.COPY_ATTRIBUTES, StandardCopyOption.REPLACE_EXISTING), move(src, dst, StandardCopyOption.ATOMIC_MOVE), **delete(Path)** [gera NoSuchFileException, DirectoryNotEmptyException, IOException], **deleteIfExists**(path) [não gera exceções].
- **FileChannel** - usado para aceder a metadata (data de criação/modificação, tamanho, etc.).
- **SeekableByteChannel** - semelhante ao RandomAccessFile do IO.

. indica a pasta **atual**.
./ indica a pasta **raiz**.
../ indica a pasta **acima** da pasta raiz, na hierarquia.

Para um ficheiro "home/…/file.txt", para imprimir apenas "file.txt":
System.out.println(f.getAbsolutePath()); ✗
System.out.println(f.getName()); ✓

AbsolutePath -> Varia com o sistema operativo.
CanonicalPath [getCanonicalPath() gera uma IOException] é único -> home/desktop/file.txt

Listar um diretório

1. Com Java IO:

```
File pasta = new File("src/");
File[] ficheiros = pasta.listFiles();
for (File f : ficheiros) {
        System.out.println(f.getAbsolutePath());
}
```

2. Com Java NIO;

```
Path pasta = Paths.get("/src");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
        for (Path f : stream) {
                System.out.println(f.getAbsolutePath());
```

```
        }
}
```

Algumas exceções frequentes no acesso a ficheiros:

- NoSuchFileException
- DirectoryNotEmptyException
- IOException (geral)

O código seguinte gera um erro de compilação, uma vez que a FileNotFoundException não é tratada nem delegada:

```
Scanner input = new Scanner(new File("readfiles.txt"));
while(input.nextLine().startsWith("#")) {
    System.out.println(input.nextLine());
}
input.close();
```

Lambda

16 de junho de 2021
21:37

EXPRESSÃO LAMBDA

Sintaxe: (arguments) -> {body}

- Pode ter 0, 1 ou + argumentos.
- O tipo de argumentos pode ser explicitamente declarado ou inferido.
- O corpo pode ter **1 ou + instruções**.
- Não pode ser usada isoladamente.
- Nas interfaces funcionais, as expressões lambda resultantes são implementações de métodos abstratos, que serão convertidas em métodos privados da classe, pelo compilador.

INTERFACE FUNCIONAL

- Contém **apenas um método abstrato**.
  Exemplo:
  ```
  @FunctionalInterface
  interface MyNum {
      double getNum(double n);
  }
  ```

Agora podemos criar uma expressão que transforme um double noutro double. Para isso, basta enunciar:
MyNum metodo1 = (x) -> x + 1 ou **MyNum metodo1 = x -> x + 1**,
uma vez que, neste caso, a expressão só tem um único argumento.

Nota: Para encurtar o código, T designa a classe de um objeto genérico.

Podemos definir interfaces funcionais genéricas:

```
@FunctionalInterface
interface MyFunc<T> {
    T someFunc(T obj);
}
```

…

Concretizando para T = String,

```
static String stringOp(MyFunc<String> stringFunc, String s) {
    return stringFunc.someFunc(s);
}
```

Por fim, podemos criar uma expressão lambda, que maiusculiza a String.
String supper = stringOp(s::toUpperCase, inputString);

Algumas interfaces funcionais genéricas (java.util.function) já definidas em Java:

- **Predicate<T>**: **boolean** teste(**T** obj) - para lambda booleano.
- **Consumer<T>**: **void** tarefa(**T** obj) - para lambda sem retorno.
- **Function<T, R>**: **R** tarefa(**T** obj) [R designa a classe do retorno da função] - para lambda genérico.
- **Supplier<T>**: **T** get() - para lambda sem argumentos.
- **Comparator<T>**: int compare(T obj1, T obj2) - para lambda comparador.

Para utilizar as lambdas, usamos nameLambda.apply(argumentos).

MÉTODO REFERÊNCIA

Gera uma expressão lambda, a partir de métodos existentes.
Estrutura: lib.subLib1.subLib2.(…).subLibn::staticMethod [lib é abreviatura de Library]
Exemplos:

- System.out::println
- String::compareToIgnoreCase()
- Math::abs

UTILIZAÇÃO DE EXPRESSÕES LAMBDA EM COLEÇÕES

- Percorrer coleção: someCollection.forEach(System.out::println);
- Definir ordem de um TreeSet:
    - TreeSet<String> ts = new TreeSet<>(**Comparator.comparing**(String::length));
    - TreeSet<String> ts = new TreeSet<>((s1, s2) -> s1.compareTo(s2));

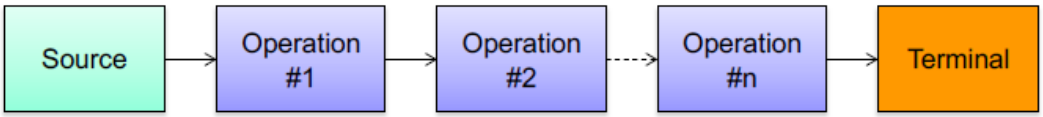
java.util.Collections (≠ interface java.util.Collection) e java.util.Arrays são duas classes abstratas, que apresentam **métodos estáticos úteis**, tais como: sort, binarySearch, copy, shuffle, reverse, max, min, etc.
Métodos estáticos mais usados:

- *Collections*.**sort**(someCollection);
- *Collections*.**reverse**(someCollection);
- someList.**sort**(lambda);
- *Arrays*.**sort**(someArray);
- *Arrays*.**fill**(someArray, valor para preencher o array)

Comparable<T> exige a implementação de int compareTo(obj) e é definida na própria classe T.
Comparator<T> exige a implementação de int compare(T, obj1, T obj2) e é definida numa classe externa (ex.: Tcompare).

Stream

16 de junho de 2021
22:49



A interface permite modificar coleções e arrays.

- FONTE
  - No caso das coleções, usam-se os métodos **stream()** e parallelStream().
  - No caso dos arrays, usa-se **Arrays.stream(someArray)**.
- OPERAÇÕES INTERMÉDIAS
  - **filter()** - exclui todos os elementos que não cumpram uma dada condição.
  - **map()** - aplica a função dada aos elementos.
  - flattMap() = map + flattening (achatamento/regularização). Só o flattMap aceita como argumento um lambda, do tipo x -> x.stream(), pois isso gerará streams aninhadas.
  - **peek()** - executa uma ação em cada elemento.
  - **distinct()** - exclui elementos duplicados (exige equals(), no caso de os elementos da coleção serem objetos de uma classe não primitivas).
  - **sorted()** - ordena os elementos, segundo um Comparator.
  - **limit()** - limita o número de elementos da stream.
  - substream(fromIndex, toIndex) - fatia a stream.
- OPERAÇÕES TERMINAIS
  - *Reducers*:
    - **reduce()** - combina uma sequência de inputs, de modo a originar um único resultado.
      - someStream.reduce(Inicial, (Intermédio, Elemento) -> operação entre o resultado intermédio e o próximo elemento do fluxo).
      - Exemplo 1: int resultado = lista.stream().reduce(0, (subtotal, item) -> subtotal + item);
      - Exemplo 2: int soma = lista.stream().reduce(0, Integer::sum);
    - **count()** - retorna o nº de elementos da stream.
    - **sum()**
    - **max()**
    - **min()**
    - **average()**
    - **findAny()**
    - **findFirst()**
  - *Collectors*:
    - collect(Collectors.toList())
    - collect(Collectors.toSet())
    - collect(Collectors.toMap())
  - **forEach()** - é a operação terminal correspondente ao peek().
  - **iterator()** - retorna um iterador da stream.

Diferença entre peek e map: Enquanto que o peek tem como argumento uma Consumer (sem retorno), o map tem como argumento uma Function (com retorno).

Diferença entre peek e forEach: ao contrário do peek, o forEach é uma operação terminal, não podendo preceder outra operação.

Exemplo importante:

```
List<String> linhas = Files.readAllLines(Paths.get("numeros.txt"), Charse
t.forName("UTF8"));
System.out.println(linhas.stream().mapToInt(s -
> Integer.parseInt(s)).average().getAsDouble());
System.out.println(linhas.stream().mapToInt(s -
> Integer.parseInt(s)).max().getAsInt());
System.out.println(linhas.stream().mapToInt(s -
> Integer.parseInt(s)).min().getAsInt());
```

A expressão `(s -> Integer.parseInt(s))` pode ser substituída por `Integer::parseInt`.
A partir de um int[] x, bastava x.stream().average(), x.stream().max() e x.stream().min().