POO – TP (recurso)

Paradigmas da programação:

- ➔ Imperativa (passos a executar)
- ➔ Declarativa (que resultado obter)

Programação orientada a objetos facilita:

- ➔ Modularidade
- ➔ Reutilização
- ➔ Substituição
- ➔ Information Hiding

Principais características:

- ➔ Encapsulamento (information hiding)
- ➔ Herança
- ➔ Polimorfismo
- ➔ Abstração

Packages em Java correspondem a um sub-diretório. Assim evitamos conflitos de nomes de classes.

Variáveis e tipos primitivos:

Type	Size
boolean	1 bit
byte	8 bits
short	16 bits
char	16 bits
int	32 bits
long	64 bits
float	32 bits
double	64 bits

Operadores:

- atribuição: =
- aritméticos: *, /, +, -, %, ++, --
- relacionais: <, <=, >, >=, ==, !=
- lógicos: !, | |, &&

Nota:

```
int a = 1;                          \, >>, <<
int b = ++a; // a = 2, b = 2 ?
int c = b++; // b = 3, c = 2
```

Reads:

- **nextLine()** – lê uma linha inteira (String)
- **next()** – lê uma palavra (String)
- **nextInt()** – lê um inteiro (int)
- **nextDouble()** – lê um número real (double)

Vetores:

```
int[] vet1;
int vet2[]; // sintaxe alternativa e equivalente, mas não encorajada
```

Para além da sua declaração, precisamos de declarar o seu tamanho:

- inicialização com valores por omissão:
  `int[] v1 = new int[3]; // vetor com 3 elementos: 0, 0 ,0`
- declaração e inicialização com valores específicos
  `int[][] a = { { 1, 2, 3, }, { 4, 5, 6, } } ;` `entos: 1, 2, 3`
  `// ou`
  `int[] v3 = new int[] { 1, 2, 3};`

Tamanho do vetor é dado por: vetor_name.length (sem parênteses)

Operador ternário

```
result = testCondition ? valueIfTrue : valueIfFalse
```

Em Java consideramos cada módulo do programa como ficheiro .java

Funções gerais:

```
Math.abs()
Math.ceil()
Math.floor()
Math.min()
Math.max()
Math.round()
Math.random()
```

```
Math.exp()
Math.log()
Math.log10()
Math.pow()
Math.sqrt()
```

Char e Strings:

```
String s1 = "java"; // creating string by java string literal
char ch[] = { 's', 't', 'r', 'i', 'n', 'g', 's' };
String s2 = new String(ch); // converting char array to string
```

As Strings são imutáveis, por isso, não se alteram mas eliminam-se e substituiem-se.

Imprimir String letra a letra: sout(string.charAt(x))

Métodos String: contains(x), substring(1, 3) //índices , startsWith(x) and EndsWith(x)

Formataçao: String.format("%02f") ou System.out.printf(x)

Expressões regulares (REGEX):

```
String s1 = "123";
System.out.println(s1.matches("\\d{2,4}"));
    // 2-4 dígitos seguidos
s1 = "abcdefg";
System.out.println(s1.matches("\\w{3,}"));
    // pelo menos 3 carateres alfanuméricos
```

```
String frase = "Regular expressions are powerful and "
                + "flexible text-processing tools.";
String[] splitResult = frase.split("\\W");
    // separa com base em carateres não alfanuméricos
System.out.println(splitResult.length + " palavras: "+
                Arrays.toString(splitResult));
splitResult = frase.split("ex");
System.out.println(splitResult.length + " palavras: " +
                Arrays.toString(splitResult));
```

Expressões 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

Todos os objetos são manipulados através de referências, são armazenados na memória heap, manipulados através de uma referência guardada na piha.

Um construtor sem parâmetros é designador por default constructor ou construtor por omissão. No entanto, se já houver um construtor na classe, esta já não cria o construtor por omissão. Valor por omissão do bollean é false.

Podemos usar o mesmo nome para diferentes objetos desde que estes representem diferentes tipos de dados. Não é possível distinguir funções pelo valor de retorno.

Os métodos estáticos não têm associada a referência this. Estes não conseguem invocar métodos não estáticos. É possível ser invocado sem que existam objetos dentro de esta classe ⟶ não estão associados a objetos.

```
class Circulo {
   static private double lista[ ] = new double[100];
   static {  // inicializador estático
      // inicialização de lista[ ]
   }
}
```

```
public interface Interface2 {
    static void stMeth() { //... do something
    }
}
```

Encapsulamento: public, protected, default (omissão), private.

- ➔ Public – pode ser usado por qualquer classe
- ➔ Protected – pode ser utilizado pelas classes filho e classe mãe
- ➔ Omissão – visível dentro do mesmo package
- ➔ Private – visível apenas dentro da própria classe

É preciso notar que qualquer que seja o encapsulamento, os dados devem ser private (exceto nas classes mãe que derivam).

Relações IS-A ou HAS-A – ambas com extends:

- ➔ IS-A: Subclasses de uma classe são (IS-A) main, fazem parte da main
- ➔ HAS-A: Main tem (HAS-A) subclasses.

Nota: as classes derivadas têm acesso aos métodos da main e métodos declarados como public ou protected na classe Main, também têm de ser ou public ou protected na classe derivada.

Atributos final significam que não podem ser mudados ou são métodos não herdáveis.

Equals:

```
Circulo p1 = new Circulo(0, 0, 1);
Circulo p2 = new Circulo(0, 0, 1);
System.out.println(p1 == p2);      // false
System.out.println(p1.equals(p2));  // false
```

Downcast é double para int e upcast é automático e é de int para double

Referência polimórfica – dynamic binding: Obj1 = new Obj2(); é corretor desde que Obj2 seja um Obj1

+ Generalização = melhor (ou seja, reutilizar, simplificar e usar heranças)

Uma classe é abstrata de tiver pelo menos um método abstrato. Esta não é instanciável (não podemos criar objetos desta). PERMITEM HERANÇA SIMPLES.

Uma interface é uma classe que só contém assinaturas. Todos os seus métodos são implicitamente abstratos e as variáveis são implicitamente estáticas e constantes. Uma interface pode ser vazia. Podemos criar uma referência para outra interface. PERMITEM HERANÇA MÚLTIPLA

Interfaces:

- ➔ Default methods: reescritos nas classes que implementam com o @override
- ➔ Static methods: não podem ser reescritos nas classes que implementam a interface (NÃO PODEMOS USAR @OVERRIDE). Têm de ser feitos na própria interface.

Enumerate – limitação: não podemos usar a partir do scanner. A instrução switch funciona com enum. Enum é uma classe, não um tipo primitivo. Não são inteiros. Podemos desenvolver métodos como toString dentro de Enums assim como implementar interfaces.

- valueOf(String val) : converte a String (elemento do conjunto) para um valor
- ordinal(): posição (int) do valor na lista de elementos
- values(): devolve a lista de elementos

```
Season s1 = Season.valueOf("WINTER");
System.out.println(s1);
System.out.println(s1.ordinal());
for (Season s2: Season.values())
    System.out.println(s2);
```

```
WINTER
3
SPRING
SUMMER
FALL
WINTER
```

```
Color myColor = Color.BLACK;

switch(myColor){
    case WHITE: …;
    case BLACK: …;
    …
    case BLUE: …;
    default: …;
}
```

Java Collections são estruturas de classes, interfaces e algoritmos que representam vários tipos de estruturas armazenadas de dados. Não suportam tipos primitivos int, float, double: temos que usar classes adaptadoras (Integer, Double, Float)…

Interfaces:

- ➔ Conjuntos – sets: sem noção de posição e ordem. Sem repetição.
  - HashSet: usa uma hash function para inserir elementos. A inserção de um novo elemento não será efetuada até que a função equals() do elemento retorne false, isto é, o espaço já não estiver ocupado. Desempenho de O(n). Atenção com a ordem de inserção quando se introduzem elementos repetidos:

```
// vector para simular a entrada de dados no Set
String[] str = {"Rui", "Manuel", "Rui", "Jose",
                "Pires", "Eduardo", "Santos"};

Set<String> group = new HashSet<>();
  for (String i: str ) {
    if (!group.add(i))
      System.out.println("Nome duplicado: " + i);
```

```
Nome duplicado: Rui
6 nomes distintos

Manuel
Rui
Jose
Eduardo
Santos
Pires
```

- TreeSet: permite a ordenação dos seus elementos por ordem alfabética ou numérica (árvore balanceada). Complexidade de O(log n). Quando se inserem objetos, é ordenado de acordo com o último atributo desse objeto:

```
public class TestTreeSet {

    public static void main(String[] args) {
        Collection<Quadrado> c = new TreeSet<>();
        c.add(new Quadrado(3, 4, 5.6));
        c.add(new Quadrado(1, 5, 4));
        c.add(new Quadrado(0, 0, 6));
        c.add(new Quadrado(4, 6, 7.4));
        System.out.println(c);

        for (Quadrado q: c)
            System.out.println(q);
    }
}
```

```
[Quadrado de Centro (1.0,5.0) e de lado 4.0, Quadrado de Centro (3.0,4.0) e de lado 5.6, Quadrado de Centro (0.0,0.0) e de lado 6.0, Quadrado de Centro (4.0,6.0) e de lado 7.4]
Quadrado de Centro (1.0,5.0) e de lado 4.0
Quadrado de Centro (3.0,4.0) e de lado 5.6
Quadrado de Centro (0.0,0.0) e de lado 6.0
Quadrado de Centro (4.0,6.0) e de lado 7.4
```

Ordem

➔ Listas – lists: sequências sem noção de ordem. Com repetições. Permitem ainda acesso posicional, pesquisa, listIterator e range-view.

- ArrayList: array dinâmico
- LinkedList: listas ligadas

Estes dois formatos são iguais na inserção, no entanto, quando se elimina um certo index da linkedList, então tudo o que está para trás, também é eliminado:

```
public static void main(String args[])
{

    // Creating an object of the
    // class linked list
    LinkedList<String> object
        = new LinkedList<String>();

    // Adding the elements to the object created
    // using add() and addLast() method

    // Custom input elements
    object.add("A");
    object.add("B");
    object.addLast("C");

    // Print the current LinkedList
    System.out.println(object);

    // Removing elements from the List object
    // using remove() and removeFirst() method
    object.remove("B");
    object.removeFirst();

    System.out.println("Linked list after "
                       + "deletion: " + object);
}
}
```

Output:

```
[A, B, C]
Linked list after deletion: [C]
```

➔ Filas – queues: fisrt in first out.

- A queue tem essencialemente 3 operações: remove() – que remove o primeiro elemento da fila, add() – que adiciona um elemento no fundo da fila e peek() que vê o primeiro elemento no início da fila. Para printar uma queue x basta só fazer sout(x).

➔ Mapas – maps: estruturas representadas por chave-valor. Não descende de collections. Tem put(key, object), remove(key), e get(key).

- HashMap: usa uma hash table e não existem ordenação nos pares.
- LinkedHashMap: semelhante ao HashMap, mas preserva a ordem de inserção
- TreeMap: os pares são ordenados com base na chave.

Collections					
	Implementações				
Interfaces	**Resizable array**	**Linked list**	**Hash table**	**Hash table + Linked list**	**Balanced Tree (sorted)**
List	ArrayList	LinkedList			
Queue	ArrayDeque	LinkedList			
Set			HashSet	LinkedHashSet	TreeSet
Map			HashMap	LinkedHashMap	TreeMap

Controlode execeções: um try...catch permite um "try", infintos "cath" e zero ou um "finally". A ordem dos catch é importante: primeiro os mais específicos e depois os mais abrangentes. Podem haver checked (try catch ou throw) ou unchecked (sistema).

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Java.io:

- ➔ Byte
- ➔ Character
- ➔ File
- ➔ Scanner
- ➔ FileReader
- ➔ FileWriter
- ➔ RandomAcessFile

Java.nio:

- ➔ Path
- ➔ Paths
- ➔ Files
- ➔ SeekableByteChannel

Leitura de ficheiros: FileNotFoundExeption

PrintWriter and RandomAcessFile ⟶ binário, acho

Paths: Path p1 = Paths.get("/tmp/foo");

LAMBDA

Uma expressão lambda descreve uma expressão anónima:

- (argument) -> (body)
  (int a, int b) -> { return a + b; }
- () -> { body }
  () -> System.out.println("Hello World");

Exemplos:

lambda expression	equivalent method
`() -> { System.gc(); }`	`void nn() { System.gc(); }`
`(int x) -> { return x+1; }`	`int nn(int x) return x+1; }`
`(int x, int y) -> { return x+y; }`	`int nn(int x, int y) { return x+y; }`
`(String... args) ->{return args.length;}`	`int nn(String... args) { return args.length; }`
`(String[] args) -> { if (args != null) return args.length; else return 0; }`	`int nn(String[] args) { if (args != null) return args.length; else return 0; }`

```
NumericTest isEven = (n) -> (n % 2) == 0;
if (isEven.test(10)) System.out.println("10 is even");
```

```
public class Lamdba1 {
    public static void main(String[] args) {
        MyNum n1 = (x) -> x+1;
            // qualquer expressão que transforme double em double
        System.out.println(n1.getNum(10));
        n1 = (x) -> x*x;
        System.out.println(n1.getNum(10));
    }
}
```

```
11.0
100.0
```

Referências a métodos:

- Podemos substituir:
  - str -> System.out.println(str)
  - (s1, s2) -> {return s1.compareToIgnoreCase(s2); }
- por:
  - System.out::println
  - String::compareToIgnoreCase

TreeSet tem usa ordem alfabética ou numérica, mas com referências a métodos, podemos alterar a ordem:

TreeSet aceita um java.util.Comparator<T>

```
public class Test {
    public static void main(String args[]) {
        TreeSet<String> ts =
            new TreeSet<>(Comparator.comparing(String::length));

        ts.add("viagem");
        ts.add("calendário");
        ts.add("prova");
        ts.add("zircónio");
        ts.add("ilha do sal");
        ts.add("avião");
        for (String element : ts)
            System.out.println(element + " ");
        }
}
```

```
prova
viagem
zircónio
calendário
ilha do sal
```

Nota: Java Collections != Java Collection