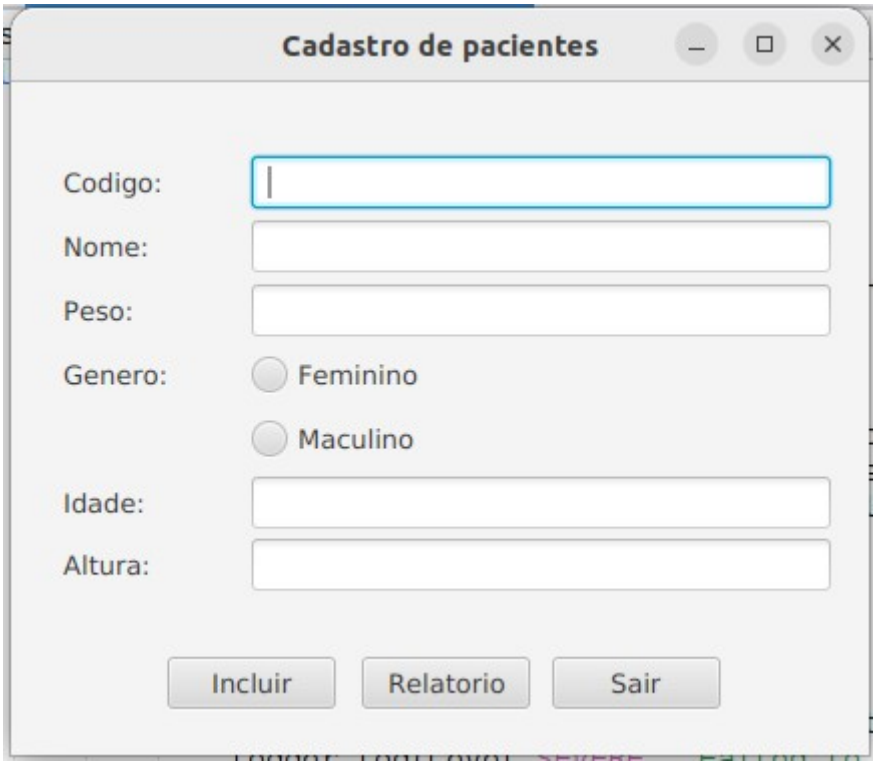
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
Engenharia de Computação

PROGRAMAÇÃO DE COMPUTADORES II

Atividade 10 – Cadastro de pacientes

Crie um formulário para entrada de dados de pacientes de uma clínica, contendo:

nome, o código de identificação, o gênero, o peso, a idade e a altura.



Exiba um relatório contendo:

(a) a quantidade de pacientes.

(b) a média de idade dos homens.

(c) a quantidade de mulheres com altura entre 1,60 e 1,70 e peso acima de 70kg.

(d) a quantidade de pessoas com idade entre 18 e 25.

(e) o nome do paciente mais velho.

(f) o nome da mulher mais baixa.

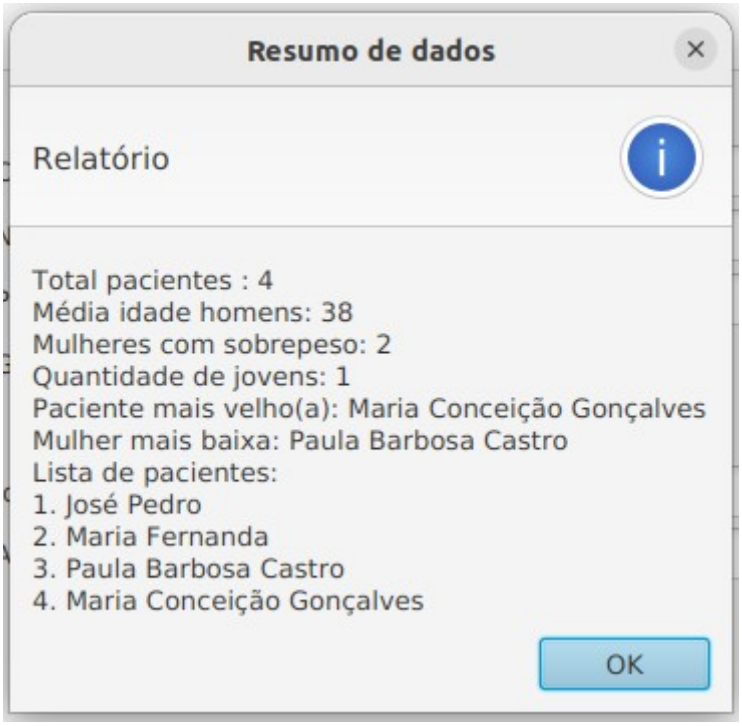
(g) a lista de todos os pacientes ordenados por código de identificação

A janela do Relatório pode ser criada como uma janela “Alert”.

Veja dicas em :

https://edencoding.com/new-windows-stage/

PROGRAMAÇÃO DE COMPUTADORES II



Obs:

- Lembre-se de separar as classes por responsabilidade!
- Utilize uma coleção ordenada para que os pacientes sejam armazenados em ordem conforme o código de identificação. Para isso, utilize o método estático sort() da classe Collection que precisa de um parâmetro do tipo List. Os elementos dessa coleção precisam implementar a interface Comparable e implementar o método compareTo(Object o).

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

PROGRAMAÇÃO DE COMPUTADORES II

`public interface` **`Comparable<T>`**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class `C` is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` of class `C`. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys `a` and `b` such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second `add` operation returns false (and the size of the sorted set does not increase) because `a` and `b` are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with equals. One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

```
{(x, y) such that x.compareTo(y) <= 0}.
```

The *quotient* for this total order is:

```
{(x, y) such that x.compareTo(y) == 0}.
```

It follows immediately from the contract for `compareTo` that the quotient is an *equivalence relation* on `C`, and that the natural ordering is a *total order* on `C`. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's `equals(Object)` method:

```
{(x, y) such that x.equals(y)}.
```