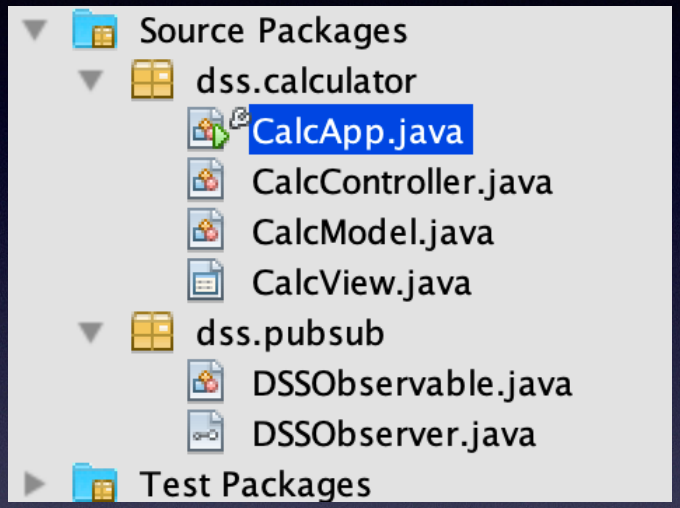
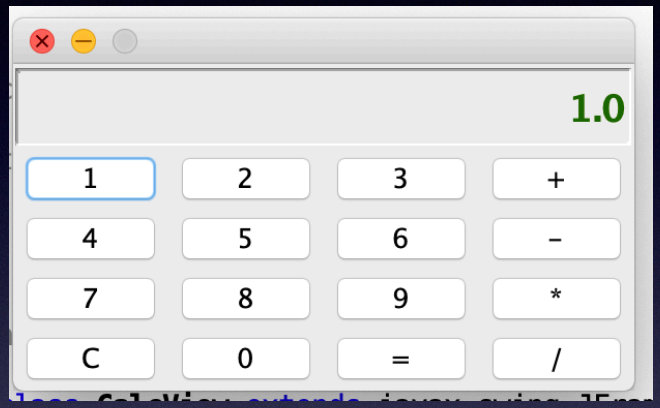
- Outro exemplo: no projecto,



- CalcApp é a classe que cria `Model`, `Controller` e `View` e coloca tudo a correr.

- Seja um exemplo de uma aplicação que é uma calculadora.



- A `View` tem o layout de uma calculadora (com botões e campo de texto)
- O `Model` implementa as operações (+,-,* e /)

- A classe principal (com o main)

```
public class CalcApp {

    private CalcApp() {
    }

    public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {

            @Override
            public void run() {
                CalcModel model = new CalcModel();
                CalcController controller = new CalcController(model);
                CalcView view = new CalcView(controller);
                /** view registada como observador do controller para poder actualizar o écran
                 * durante a construção do número no controller  */
                controller.addObserver(view);

                /** controller registado como observador do model para poder actualizar o valor após operações no model */
                model.addObserver(controller);
                view.run();
            }
        });
    }
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- Model:

```
public class CalcModel extends DSSObservable {
    private double value;

    public CalcModel() {
        this.value = 0;
    }

    public void add(double v) {
        this.value += v;
        this.notifyObservers(""+value);
    }

    public void subtract(double v) {
        this.value -= v;
        this.notifyObservers(""+value);
    }

    public void multiply(double v) {
        this.value *= v;
        this.notifyObservers(""+value);
    }

    public void divide(double v) {
        this.value /= v;
        this.notifyObservers(""+value);
    }

    public double getValue() {
        return this.value;
    }
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- Controller:

```
public class CalcController extends DSSObservable implements DSSObserver {

    private double screen_value;          // o valor que está a ser lido
    private char lastkey;                 // indica que se vai começar a "ler" um novo número
    private char opr;                     // memória com a operação a aplicar
    private CalcModel model;

    /** Creates a new instance of Calculadora */
    public CalcController(CalcModel model) {
        this.screen_value = 0;
        this.lastkey = ' ';
        this.opr = '=';
        this.model = model;
        /** o notifyObservers serve para comunicar o novo valor da calculadora */
        this.notifyObservers(this.screen_value);
    }

    public void processa(int d) {
        if (this.lastkey != 'd') {
            this.screen_value = d;
            this.lastkey = 'd';
        } else {
            this.screen_value = this.screen_value*10+d;
        }
        /** o notifyObservers serve para comunicar o novo valor da calculadora */
        this.notifyObservers(this.screen_value);
    }
```

Nota: no caso do Java Swing o código do controller e da View usualmente está no mesmo ficheiro. Mas podia ser separado usando a estratégia apresentada…

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- continuação…

```
public void processa(char opr) {
    switch (this.opr) {
        case '=': model.setValue(this.screen_value);
                  break;
        case '+': model.add(this.screen_value);
                  break;
        case '-': model.subtract(this.screen_value);
                  break;
        case '*': model.multiply(this.screen_value);
                  break;
        case '/': model.divide(this.screen_value); // Exercício: Acrescente tratamento d
                  break;
    };
    this.opr = opr;
    this.lastkey = opr;
}

public void clear() {
    model.reset();
    this.lastkey = ' ';
}

@Override
public void update(DSSObservable source, Object value) {
    this.screen_value = Double.parseDouble(value.toString());
}
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- A `View` (neste caso em Java Swing):

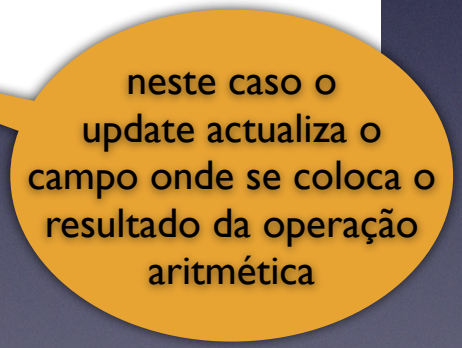
```
public class CalcView extends javax.swing.JFrame implements DSSObserver {

    /** A calculadora que vai fazer as contas... */
    private CalcController controller;

    /** Creates new form JCalculadora */
    public CalcView(CalcController ctl) {
        this.controller = ctl;
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    Generated Code

    private void opr_press(java.awt.event.ActionEvent evt) {
        // Add your handling code here:
        this.controller.processa(evt.getActionCommand().charAt(0));
    }
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software

- A `View`, implementa a interface Observer, e tem de fornecer um implementação para o método update:

```
/**
 * Método correspondente à interface Observer.
 * Este é o método que é invocado sempre que a calculadora efectua um
 * notifyObservers, actualiza o écran com o valor que vem como parâmetro
 */
public void update(DSSObservable o, Object arg) {
    this.screen.setText((arg.toString()));
}
```

(*) retirado de um exemplo da Uc de Desenvolvimento de Sistemas de Software



- Exemplo de uma aplicação com `View` em modo texto (exemplo dos Hotéis visto anteriormente):

```
public class HoteisIncApp {

    // A classe HoteisInc tem a 'lógica de negócio'.
    private HoteisInc logNegocio;

    // Menus da aplicação
    private Menu menuPrincipal, menuHoteis;

    /**
     * O método main cria a aplicação e invoca o método run()
     */
    public static void main(String[] args) {
        new HoteisIncApp().run();
    }
```

- Esta classe também implementa o `Controller`:

```
private void run() {
    System.out.println(this.logNegocio.toString());
    do {
        menuPrincipal.executa();
        switch (menuPrincipal.getOpcao()) {
            case 1: System.out.println("Escolheu adicionar");
                    break;
            case 2: //trataConsultarHotel();
            case 3: //outro método
        }
    } while (menuPrincipal.getOpcao()!=0); // A opção 0 é usada para sair
    try {
        this.logNegocio.guardaEstado("estado.obj");
    }
    catch (IOException e) {
        System.out.println("Ops! Não consegui gravar os dados!");
    }
    System.out.println("Até breve!...");
}
```

- A View:

```java
public class Menu {
    // variáveis de instância
    private List<String> opcoes;
    private int op;

    /**
     * Constructor for objects of class Menu
     */
    public Menu(String[] opcoes) {
        this.opcoes = Arrays.asList(opcoes);
        this.op = 0;
    }
```

- continuação…

```java
/**
 * Método para apresentar o menu e ler uma opção.
 *
 */
public void executa() {
    do {
        showMenu();
        this.op = lerOpcao();
    } while (this.op == -1);
}

/** Apresentar o menu */
private void showMenu() {
    System.out.println("\n *** Menu *** ");
    for (int i=0; i<this.opcoes.size(); i++) {
        System.out.print(i+1);
        System.out.print(" - ");
        System.out.println(this.opcoes.get(i));
    }
    System.out.println("0 - Sair");
}
```

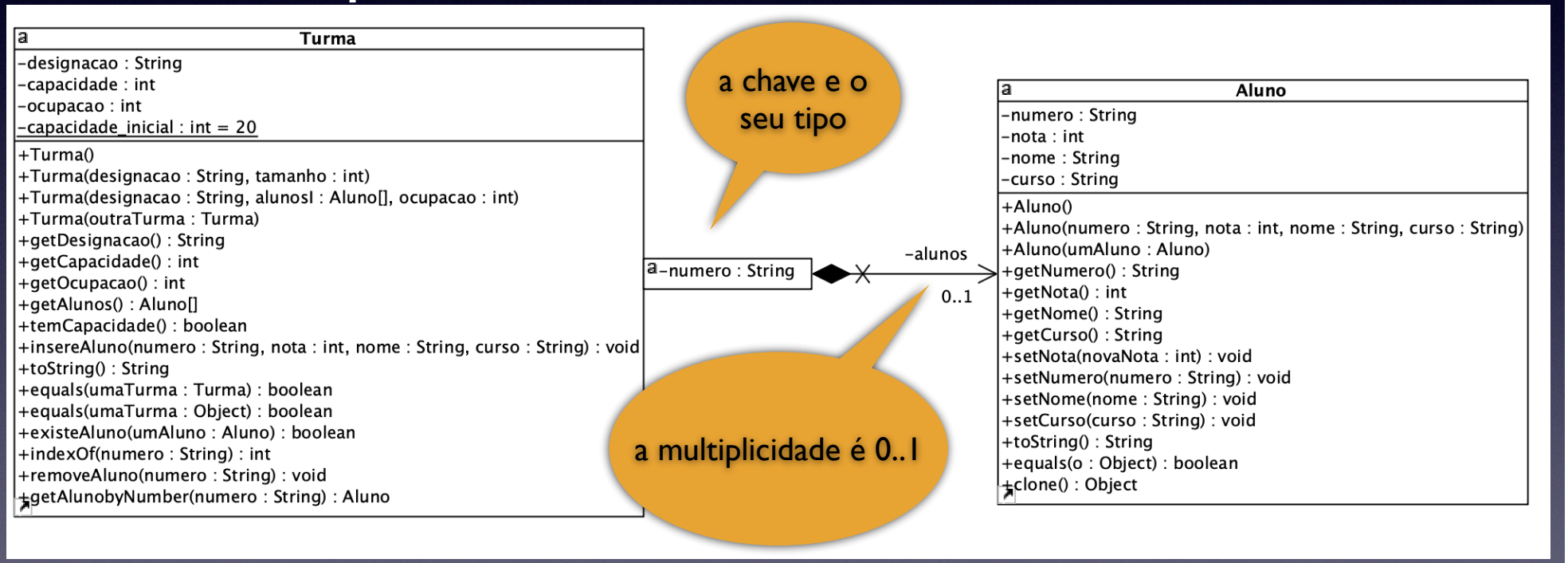
- Existem várias abordagens ao padrão MVC:
  - na comunidade não há um consenso claro de qual é a abordagem mais correcta (deve a view observar o model? a view deve interrogar directamente o model?)
  - o que se apresentou atrás é apenas uma visão de separação entre `Model`, `View` e `Controller`

- A definição base em que todos podemos concordar é que: os controllers fazem pedidos ao model e o model notifica os observadores quando o seu estado muda
  - idealmente devem ser os controllers os observadores do model, mas encontram-se abordagens em que é a view
- Mais importante é a noção de desacoplamento que a utilização de MVC e Observer proporcionam.

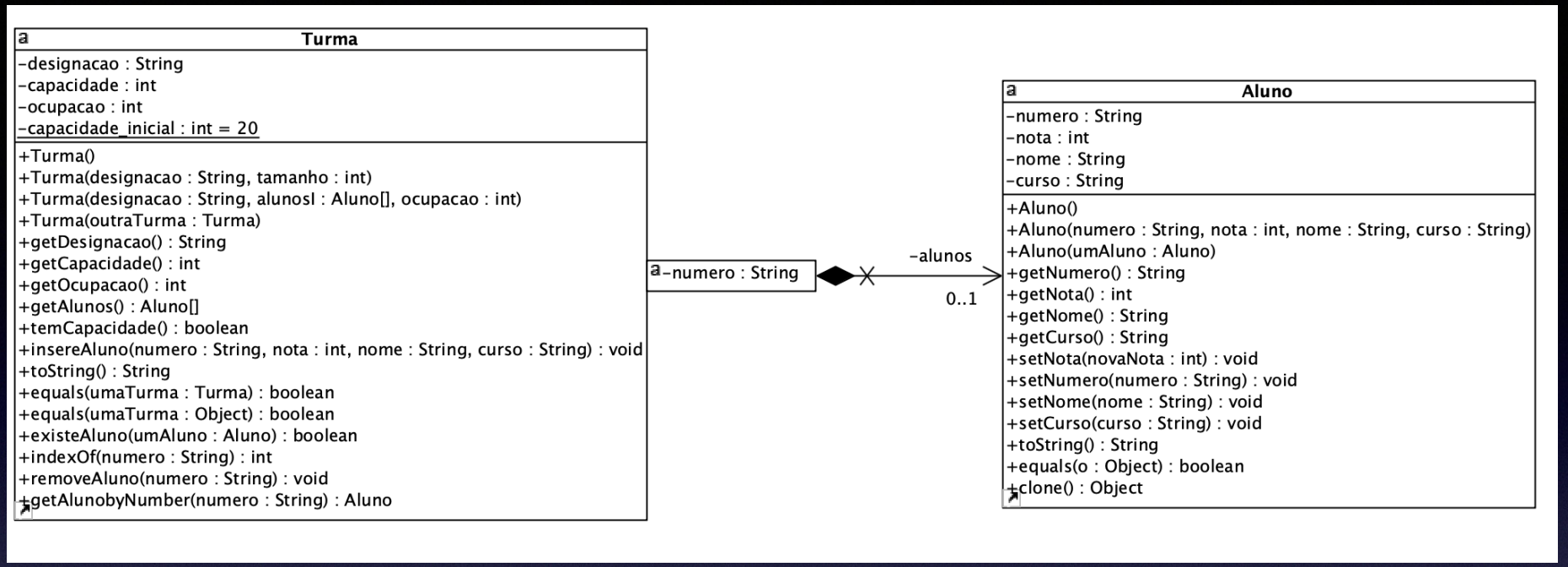
Notação do Diagrama de Classe UML

- Em relação ao diagrama de classe UML que temos vindo a utilizar é necessário acrescentar mais informação:
  - como descrever apropriadamente mapeamentos
  - como descrever o que é abstracto

- Na descrição do Map vamos indicar a chave e o seu tipo e a classe dos objectos que fazem parte dos valores.



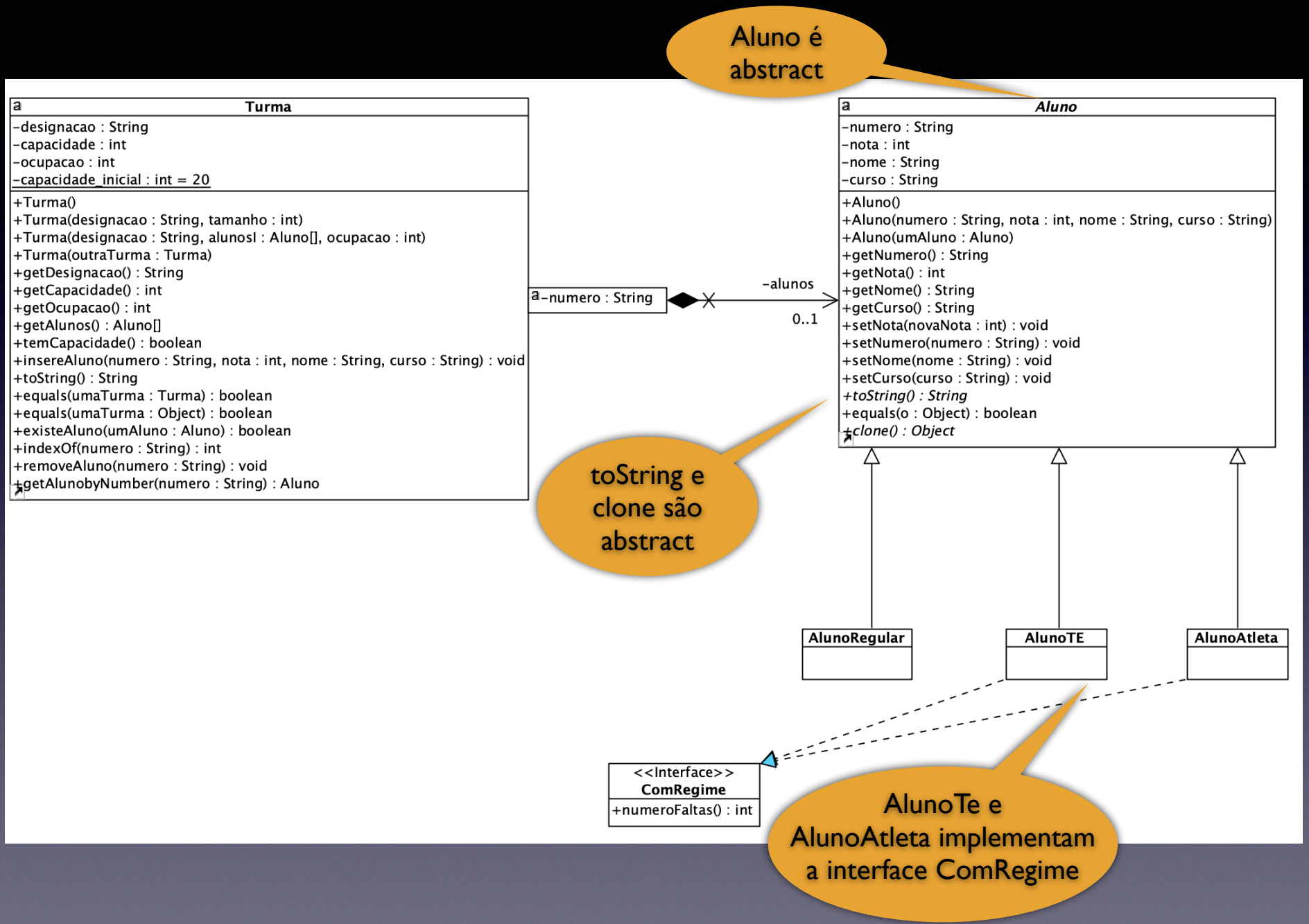
- se a chave existir temos acesso a uma instância de Aluno, caso contrário a zero!



- dá origem a:

```
public class Turma {
  private String designacao;
  private int capacidade;
  private int ocupacao;
  private static final int capacidade_inicial = 20;

  private Map<String,Aluno> alunos;

  ...
```

- Em UML as definições que são abstractas são anotadas a *itálico* ou em alguns editores como utilizando a notação `<<abstract>>` (já não faz parte da norma…)


Aluno é abstract
Turma
-designacao : String
-capacidade : int
-ocupacao : int
-capacidade_inicial : int = 20
+Turma()
+Turma(designacao : String, tamanho : int)
+Turma(designacao : String, alunosI : Aluno[], ocupacao : int)
+Turma(outraTurma : Turma)
+getDesignacao() : String
+getCapacidade() : int
+getOcupacao() : int
+getAlunos() : Aluno[]
+temCapacidade() : boolean
+insereAluno(numero : String, nota : int, nome : String, curso : String) : void
+toString() : String
+equals(umaTurma : Turma) : boolean
+equals(umaTurma : Object) : boolean
+existeAluno(umAluno : Aluno) : boolean
+indexOf(numero : String) : int
+removeAluno(numero : String) : void
+getAlunobyNumber(numero : String) : Aluno
-numero : String
-alunos
0..1
Aluno
-numero : String
-nota : int
-nome : String
-curso : String
+Aluno()
+Aluno(numero : String, nota : int, nome : String, curso : String)
+Aluno(umAluno : Aluno)
+getNumero() : String
+getNota() : int
+getNome() : String
+getCurso() : String
+setNota(novaNota : int) : void
+setNumero(numero : String) : void
+setNome(nome : String) : void
+setCurso(curso : String) : void
+toString() : String
+equals(o : Object) : boolean
+clone() : Object
toString e clone são abstract
AlunoRegular
AlunoTE
AlunoAtleta
<<Interface>>
ComRegime
+numeroFaltas() : int
AlunoTe e AlunoAtleta implementam a interface ComRegime