Interfaces

- Como vimos atrás, as classes além de descreverem estrutura e comportamento, podem ser também vistas como tipos de dados
  - um tipo de dados representa o comportamento que é possível invocar num objecto desse tipo
- O tipo de dados básico de uma instância é o tipo da classe que a criou (através da invocação do construtor)

- Na declaração:

```
ClasseA a = new ClasseA();
ClasseB b = new ClasseB();
```

  - os objectos a e b são do tipo de dados que lhes é dado pela classe que os cria.

- Os tipos de dados das subclasses podem ser designados como subtipos e já sabemos que:

  - se ClasseA é superclasse de ClasseB

  - um objecto do tipo ClasseA pode ser substituído por um objecto do subtipo ClasseB sem alteração nas classes que manipulam objectos do tipo ClasseA.

- Este mecanismo de substituição é essencial para as construções polimórficas que estudamos anteriormente.

```
Mensagem c1 = new Carta("José Francisco", "Pedro Xavier", "Em anexo a proposta de compra.");
Mensagem c2 = new Carta("Produtos Estrela", "Joana Silva", "Junto enviamos factura.");
Mensagem s1 = new SMS("961234432", "929745228", "Estou à espera!");
Mensagem s2 = new SMS("911254535", "939541928", "Hoje não há aula...");
Mensagem e1 = new Email("anr", "jfc", "Teste POO", "Junto envio o enunciado.");
Mensagem e2 = new Email("a77721", "a55212", "Apontamentos", "Onde estão as fotocópias?");
Mensagem e3 = new Email("anr", "a43298", "Re: Entrega Projecto", "Recebido.");
```

- Claro que esta capacidade de substituição de tipos por subtipos só funciona no contexto de uma hierarquia

- Esta construção, ainda que muito poderosa, tem algumas limitações. A saber:

1. como fazer que classes não relacionadas hierarquicamente possam ser vistas como tendo um tipo comum?

2. como fazer para esconder, em determinados contextos, comportamento associado a um determinado tipo de dados?

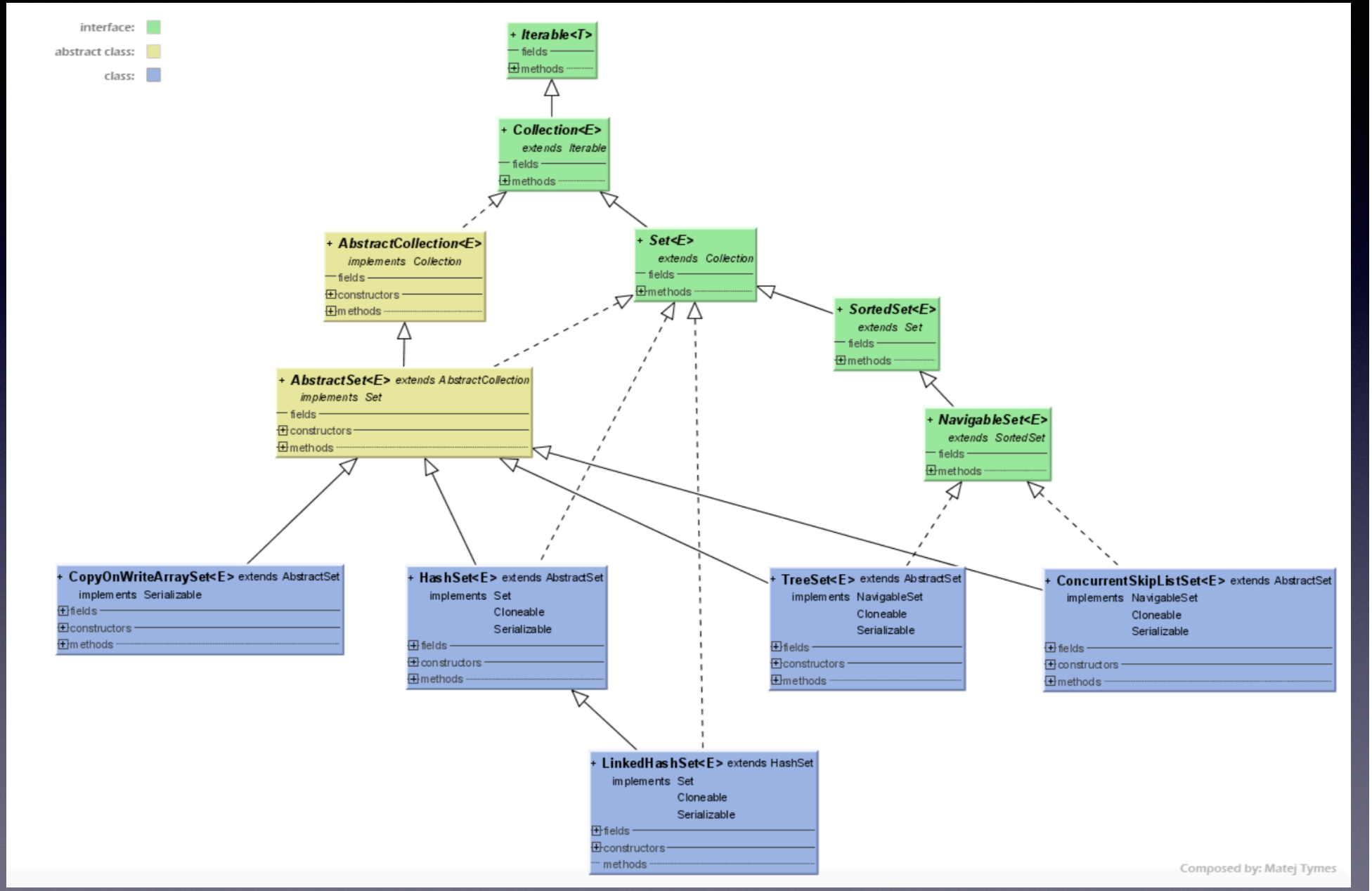
- A solução passa por passar a utilizar um outro mecanismo de tipo de dados existente em Java: as **interfaces**.

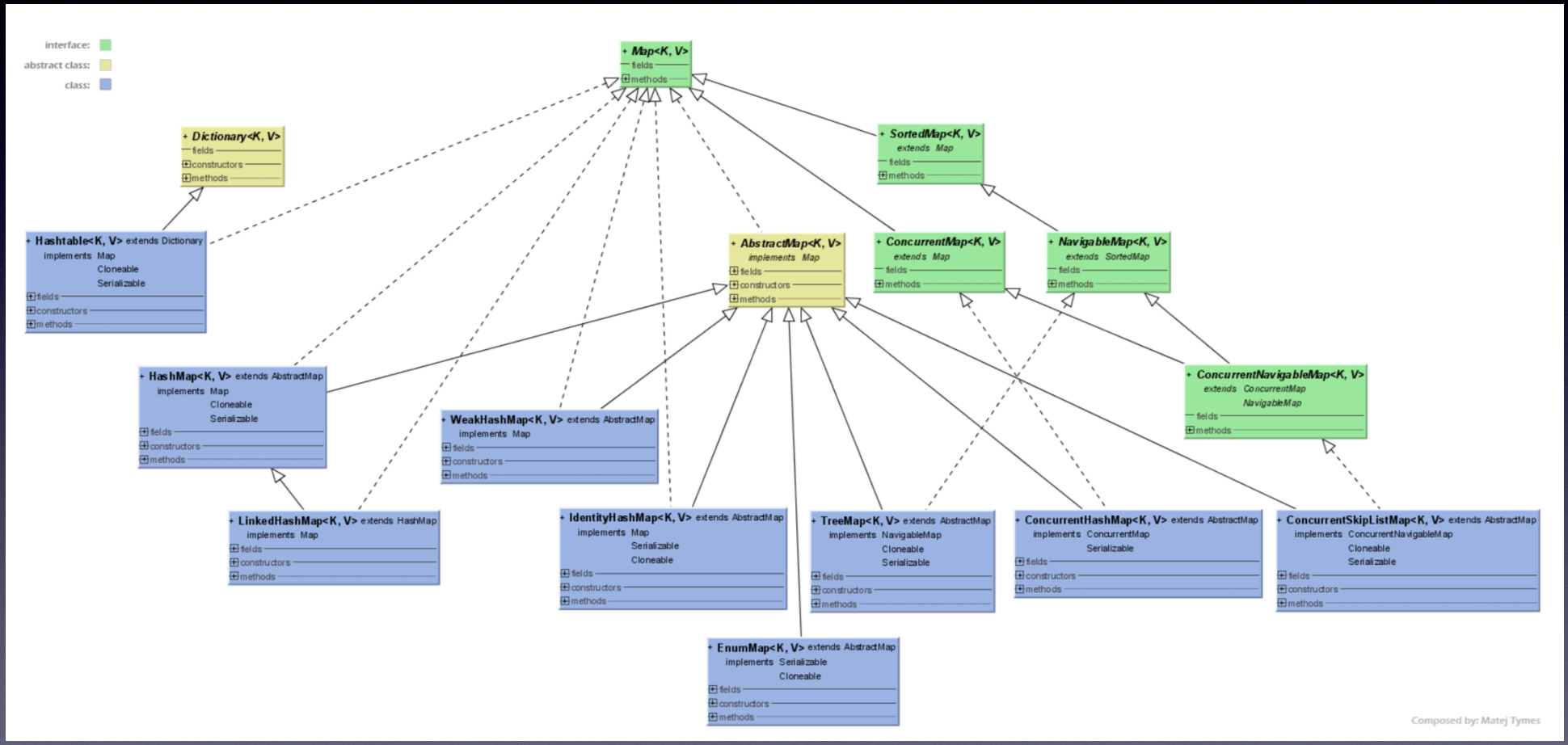
- Interfaces não são classes, mas são apenas declarações sintácticas de expressão de comportamento

  - não estão na mesma hierarquia das classes e possuem uma hierarquia própria de herança múltipla

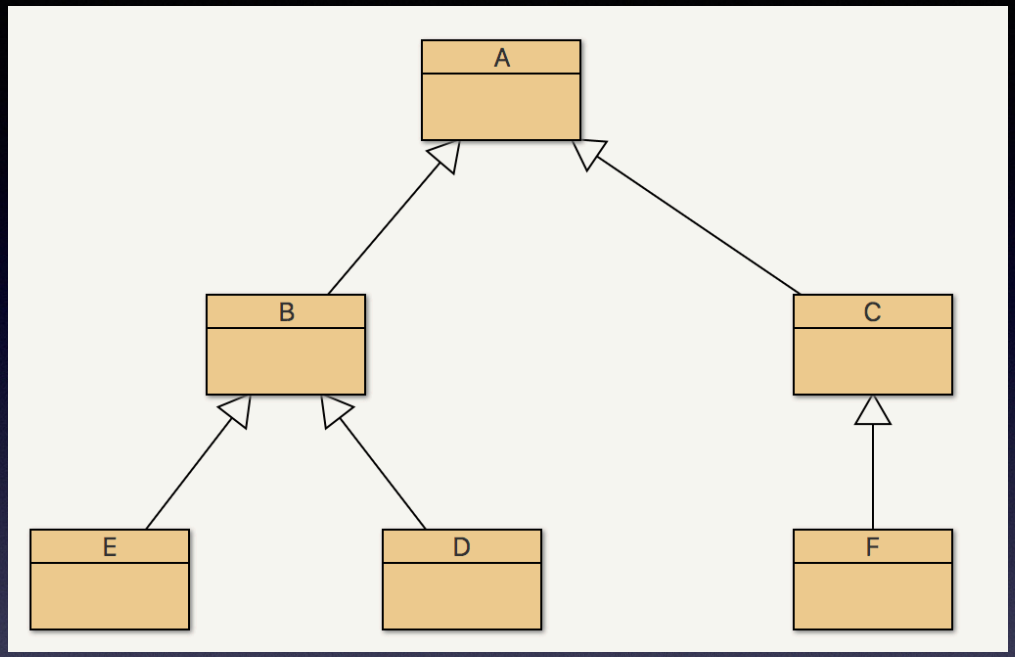
  - as classes decidem com que tipo de interfaces querem ser compatíveis

- Anteriormente já tivemos contacto com interfaces:
  - **`Set<T>`**, **`List<T>`**, **`Map<K,V>`** não são classes, mas tipos de dados declarados como interfaces
  - **`Comparator<T>`** é uma interface que representa o tipo de dados de todos os objectos que possuem (apenas!) o método `compare(...)` declarado
    - é uma functional interface


interface:
abstract class:
class:
+ Iterable<T>
fields
methods
+ Collection<E>
extends Iterable
fields
methods
+ AbstractCollection<E>
implements Collection
fields
constructors
methods
+ Set<E>
extends Collection
fields
methods
+ SortedSet<E>
extends Set
fields
methods
+ AbstractSet<E> extends AbstractCollection
implements Set
fields
constructors
methods
+ NavigableSet<E>
extends SortedSet
fields
methods
+ CopyOnWriteArraySet<E> extends AbstractSet
implements Serializable
fields
constructors
methods
+ HashSet<E> extends AbstractSet
implements Set
Cloneable
Serializable
fields
constructors
methods
+ TreeSet<E> extends AbstractSet
implements NavigableSet
Cloneable
Serializable
fields
constructors
methods
+ ConcurrentSkipListSet<E> extends AbstractSet
implements NavigableSet
Cloneable
Serializable
fields
constructors
methods
+ LinkedHashSet<E> extends HashSet
implements Set
Cloneable
Serializable
fields
constructors
methods
Composed by: Matej Tymes


interface:
abstract class:
class:
+ Map<K, V>
fields
methods
+ Dictionary<K, V>
fields
constructors
methods
+ SortedMap<K, V>
extends Map
fields
methods
+ Hashtable<K, V> extends Dictionary
implements Map
Cloneable
Serializable
fields
constructors
methods
+ AbstractMap<K, V>
implements Map
fields
constructors
methods
+ ConcurrentMap<K, V>
extends Map
fields
methods
+ NavigableMap<K, V>
extends SortedMap
fields
methods
+ ConcurrentNavigableMap<K, V>
extends ConcurrentMap
NavigableMap
fields
methods
+ HashMap<K, V> extends AbstractMap
implements Map
Cloneable
Serializable
fields
constructors
methods
+ WeakHashMap<K, V> extends AbstractMap
implements Map
fields
constructors
methods
+ LinkedHashMap<K, V> extends HashMap
implements Map
fields
constructors
methods
+ IdentityHashMap<K, V> extends AbstractMap
implements Map
Serializable
Cloneable
fields
constructors
methods
+ TreeMap<K, V> extends AbstractMap
implements NavigableMap
Cloneable
Serializable
fields
constructors
methods
+ ConcurrentHashMap<K, V> extends AbstractMap
implements ConcurrentMap
Serializable
fields
constructors
methods
+ ConcurrentSkipListMap<K, V> extends AbstractMap
implements ConcurrentNavigableMap
Cloneable
Serializable
fields
constructors
methods
+ EnumMap<K, V> extends AbstractMap
implements Serializable
Cloneable
fields
constructors
methods
Composed by: Matej Tymes

- Consideremos as seguinte classes:

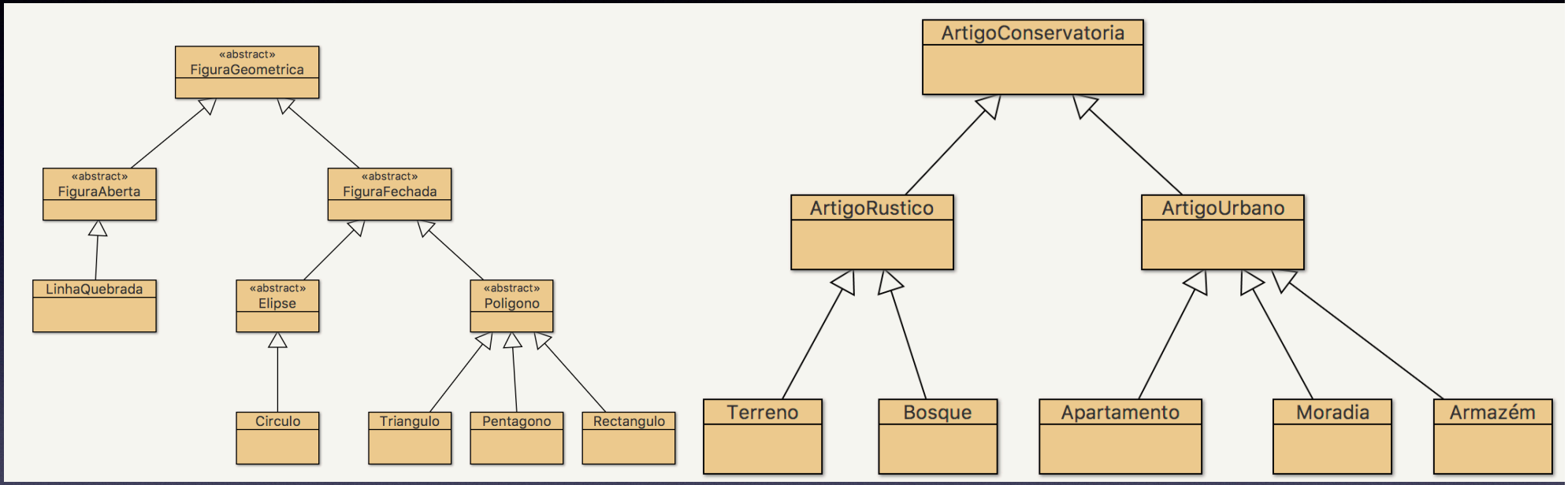


- Se quisermos representar uma colecção de objectos do tipo A (e dos seus subtipos) apenas precisamos de declarar:
  - Collection<A>

- Mas o que acontece se quisermos apenas ter numa colecção objectos do tipo D e F?
  - solução?: aceitamos objectos do tipo A e validamos depois se são D ou F? ☠
  - solução: criamos um tipo de dados comum só a D e F!
    - mas como o fazemos na construção hierárquica anterior?
    - altera-se a hierarquia?

- Reescrever a hierarquia quase nunca é uma hipótese viável porque isso tem (mesmo muito!) impacto nas classes que usam objectos desta hierarquia
  - é impossível fazer isso de forma silenciosa!
  - pretendemos ter programação genérica e extensível, mas também estável e sem impactos nas aplicações já existentes.

- Outro exemplo:



- todas estas classes possuem o método `area()` e `perimetro()`, mas como é que as compatibilizamos por forma a serem tratadas de forma comum?

- Criando uma interface, por exemplo chamada `Aravel`, que contenha a declaração do comportamento desejado (tipo de dados)

```
public interface Aravel {
   public double area();
   public double perimetro();
}
```

- tornando agora possível ter classes que queiram ser compatíveis com este novo tipo de dados.

- As classes que queiram ser compatíveis com a interface devem fazer:

```
public class Terreno extends ArtigoRustico implements Aravel {
```

- Uma classe pode implementar mais do que uma interface.

```
public class Terreno extends ArtigoRustico implements Aravel,Mensuravel {
```

```
public interface Mensuravel {
  public double valorMercado();
}
```

- Uma instância de Terreno pode ser vista como sendo:
  - do tipo de dados `Terreno` e tem acesso a todos os métodos
  - do tipo de dados `Aravel` e tem acesso *apenas* a `area()` e `perimetro()`
  - do tipo de dados `Mensuravel` e tem acesso *apenas* a `valorMercado()`

Elementos de uma interface

- A declaração de interface pode conter:
  - um conjunto de métodos abstractos
  - um conjunto de constantes
  - métodos concretos, designados por *default methods*
  - métodos de classe concretos, *static methods*

Hierarquia das Interfaces

- As interfaces podem estar colocadas numa estrutura hierárquica, que ao invés da das classes é uma hierarquia múltipla
  - uma interface pode herdar de mais do que uma interface
  - nessas situações é preciso ter atenção à herança de métodos com a mesma assinatura
    - `NomeInterface.nomeMetodo()`

Default Methods

- Até à versão 8 do Java as interfaces apenas podiam declarar assinaturas de métodos, que as classes implementadoras deviam tornar concretos.
- A partir dessa distribuição foi possível acrescentar métodos completamente codificados e que podem ser utilizados pelas classes que implementem a interface.

- Consideremos a interface `Aravel` definida anteriormente. O que acontece que acrescentarmos mais um método abstracto à interface?
  - as classes que a implementam passam a dar erro de compilação, porque…
  - …falta implementar um método!

- Com a utilização dos *default methods* as classes que já implementam a interface não precisam de ser alteradas.
- As classes implementadoras que desejem passam a poder utilizar esse método
- Se a interface `Aravel` passar agora a ter um método default:

```
public interface Aravel {
  public double area();
  public double perimetro();
  public default Ponto pontoCentral(...) { <<codigo>> }
}
```

- A classe Terreno que implementa Aravel tem de implementar os métodos `area()` e `perimetro()` e pode utilizar o método `pontoCentral(...)` já codificado.

- As classes que não conhecem a implementação de `pontoCentral(...)` continuam inalteradas e a funcionar devidamente.

- Os default methods permitem adicionar funcionalidade às interfaces já existentes, sem alterar as relações de implementação já existentes
- os default methods foram criados para permitirem acrescentar funcionalidade à API das Collections
- Se a classe tiver um método com a mesma assinatura que um default method, prevalece a definição da classe!

Métodos static em interfaces

- É também possível criar métodos static na definição das interfaces.
  - as instâncias das classes que implementam a interface tem acesso a estes métodos
  - invocáveis da forma `Interface.metodo()`

Classes abstractas e interfaces

- A escolha de utilização de uma classe abstracta ou uma interface é por vezes uma decisão difícil.
  - Uma classe abstracta pode ter variáveis de instância enquanto uma interface não pode.
  - Uma classe abstracta e uma interface podem declarar métodos abstractos para outras classes implementarem.

- No entanto, uma classe abstracta não obriga as subclasses a implementá-los. Se não o fizerem as subclasses ficam como abstractas. Nas interfaces, se não o fizerem, dá erro de compilação.

- Uma interface é um tipo de dados (uma API) que pode ser utilizada por qualquer classe. A definição da classe abstracta só é possível dentro da hierarquia.

- Em Java tem sido dada uma importância acrescida às interfaces como mecanismo de extensibilidade.

… e ainda

- A verificação de tipo pode ser feita da mesma forma que fazemos para as classes, com instanceof

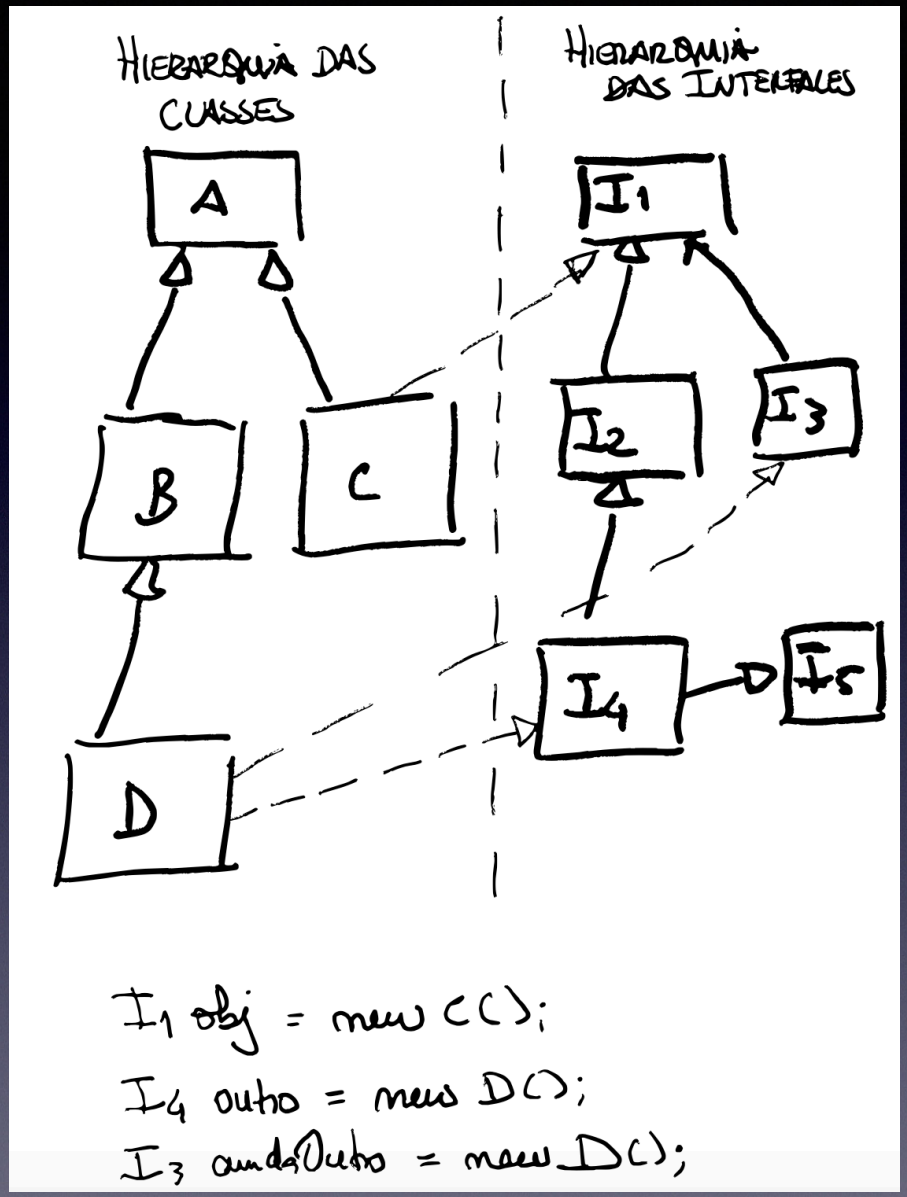
```
List<ArtigoConservatoria> propriedades = new ArrayList<>();
...
...
int numMensuravel = 0;
for(ArtigoConservatoria ac: this.propriedades)
  if (ac instanceof Mensuravel)
    numMensuravel++;
```

- na expressão acima não se está a validar a classe, mas sim o tipo de dados estático

- Do ponto de vista da concepção de arquitecturas de objectos, as interfaces são importantes para:
  - reunirem similaridades comportamentais, entre classes não relacionadas hierarquicamente
  - definirem novos tipos de dados
  - conterem a API comum a vários objectos, sem indicarem a classe dos mesmos (como no caso do JCF)

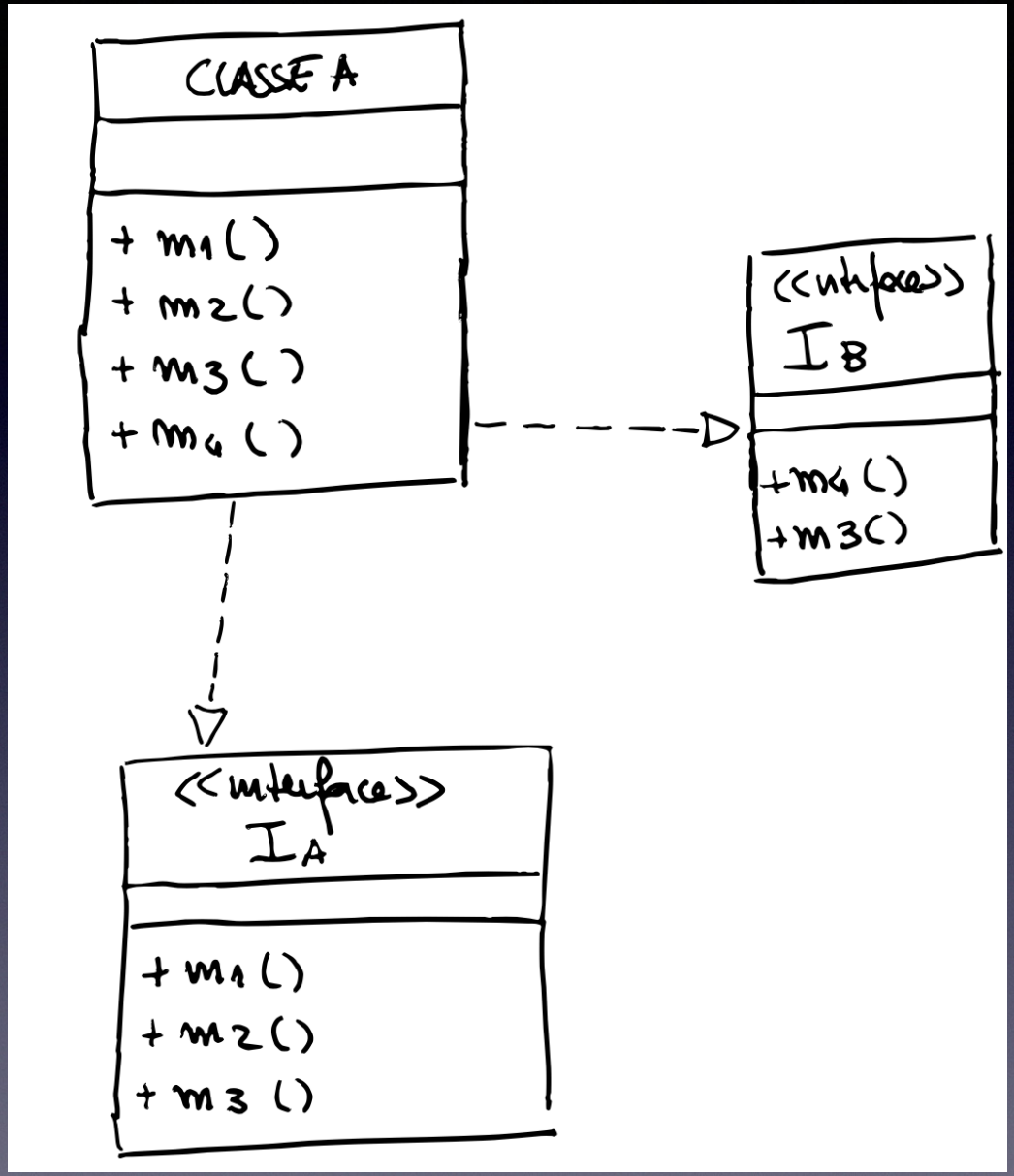
- Uma classe passa a ter duas vistas (ou classificações) possíveis:
  - é subclasse, por se enquadrar na hierarquia normal de classes, tendo um mecanismo de herança simples de estado e comportamento
  - é subtipo, por se enquadrar numa hierarquia múltipla de definições de comportamento (abstracto ou já implementado cf. default methods)

- existem duas hierarquias, que estão relacionadas, e o programador pode tirar partido disso.



- as interfaces podem ser utilizadas para agrupar comportamento e limitar o acesso ao conjunto de métodos de uma classe

- I_A apenas disponibiliza os métodos `m1`, `m2` e `m3`



finalmente...

- Classes podem implementar múltiplas interfaces
- As interfaces podem:
  - incluir métodos **static**
  - fornecer implementações por omissão dos métodos (os métodos **default**)
- Functional Interface
  - uma interface com um único método abstracto (e qualquer número de métodos default)
  - Instâncias criadas com expressões lambda e com referências a métodos ou construtores

Em resumo...

- As interfaces Java são especificações de tipos de dados. Especificam o conjunto de operações a que respondem objectos desse tipo
- Uma instância de uma classe é imediatamente compatível com:
  - o tipo da classe
  - o tipo da interface (se estiver definido)

Interfaces

- permitem relacionar, sob a figura de um novo tipo de dados, objectos de classes não relacionadas hierarquicamente
  - possibilita manter uma hierarquia de herança e ter uma outra hierarquia de tipos de dados

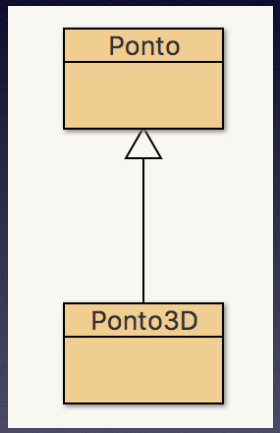
- possibilitam que um mesmo objecto possa apresentar-se sob a forma de diferentes tipos de dados, exibindo comportamentos diferentes, consoante o que seja pretendido
  - permite esconder a natureza do objecto e fazer a sua *tipagem* de acordo com as necessidades do momento
  - permite limitar os métodos que, em determinado momento, podem ser invocados num objecto

- (muito relevante) possibilitam que o código possa ser desenvolvido apenas com o recurso à interface e sem saber qual a implementação
  - principal razão para termos utilizado List<E>, Set<E>, Map<K,V>
  - o programador apenas necessita saber o comportamento oferecido e pode construir os seus programas em função disso

- As declarações constantes de uma interface constituem um contrato, isto é, especificam a forma do comportamento que as implementações oferecem
- Por forma a fazer programas apenas precisamos de saber isso e ter bem documentado o que cada método faz.
- Em função disso podemos fazer o programa e os programas de teste.

Tipos Parametrizados

- Consideremos novamente a hierarquia dos pontos:



- e considere-se que pretendemos manipular colecções de pontos

- Como sabemos uma lista de pontos, `List<Ponto>` pode conter instâncias de Ponto, Ponto3D ou outras subclasses destas classes.
- no entanto, essa lista não é o supertipo das listas de subtipos de Ponto!!
- …, porque a hierarquia de `List<E>` não tem a mesma estruturação da hierarquia de E

- Consideremos a classe ColPontos que tem uma lista de Ponto.

```
public class ColPontos {

  private List<Ponto> meusPontos;

  public ColPontos() {
    this.meusPontos = new ArrayList<Ponto>();

  }

  public void setPontos(List<Ponto> pontos) {
    this.meusPontos = pontos.stream().map(Ponto::clone).collect(Collectors.toList());
  }
```

- Seja agora uma classe que utiliza uma instância de ColPontos

```
public class TesteColPontos {
  public static void main(String[] args) {
    Ponto3D r1 = new Ponto3D(1,1,1);
    Ponto3D r2 = new Ponto3D(10,5,3);
    Ponto3D r3 = new Ponto3D(4,16,7);

    ArrayList<Ponto3D> pontos = new ArrayList<>();

    ColPontos colecao = new ColPontos();
    colecao.setPontos(pontos);
```

List<Ponto3D> não pode ser vista como List<Ponto>!!

```
incompatible types:
java.util.ArrayList<Ponto3D> cannot be
converted to java.util.List<Ponto>
```

lass compiled - no syntax errors

- O tipo das Lists de Ponto e das Lists dos seus subtipos declara-se como:
  `List<? extends Ponto>`

- O tipo das Lists de superclasses de Ponto declara-se como:
  `List<? super Ponto>`

- Será necessário alterar o código dos métodos para permitir esta compatibilidade de tipos:

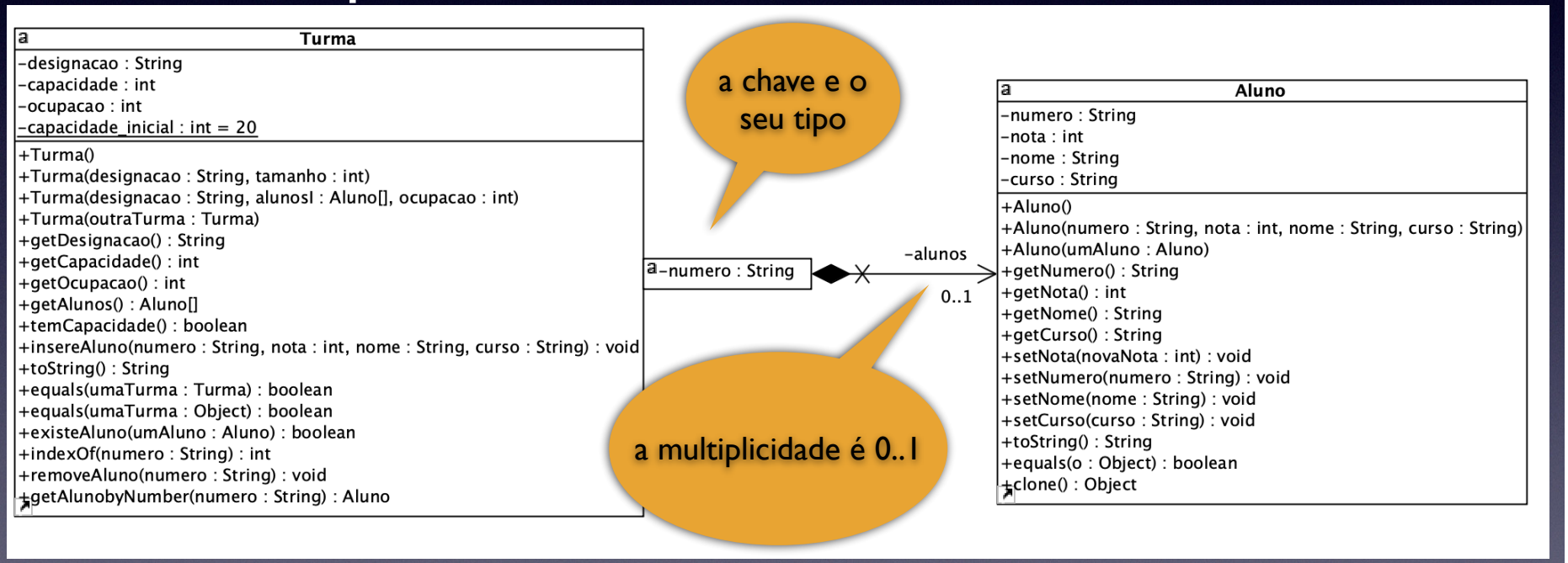
```
public void setPontos(List<? extends Ponto> pontos) {
  this.meusPontos = pontos.stream().map(Ponto::clone).collect(Collectors.toList());
}
```

- Collection<? extends Ponto> =
  - Collection<Ponto3D> ou
  - Collection<PontoComCor> ou …

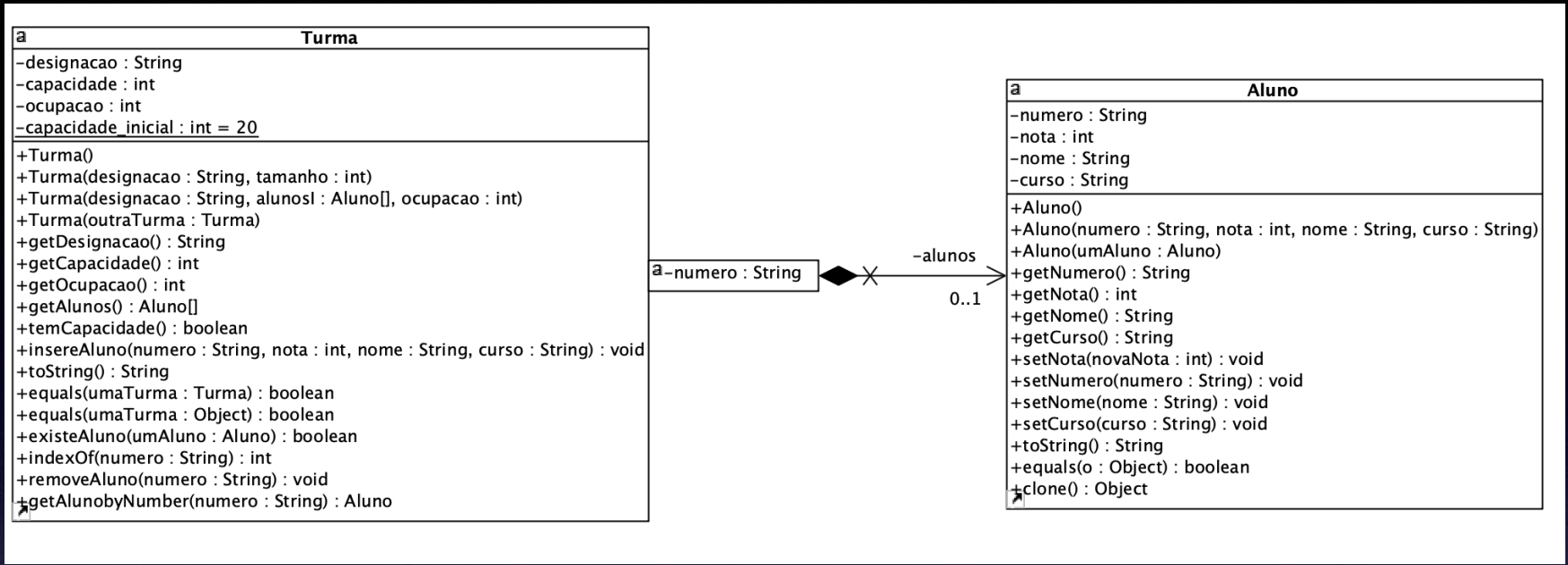
Notação do Diagrama de Classe UML

- Em relação ao diagrama de classe UML que temos vindo a utilizar é necessário acrescentar mais informação:
  - como descrever apropriadamente mapeamentos
  - como representar implementação de interfaces
  - 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.



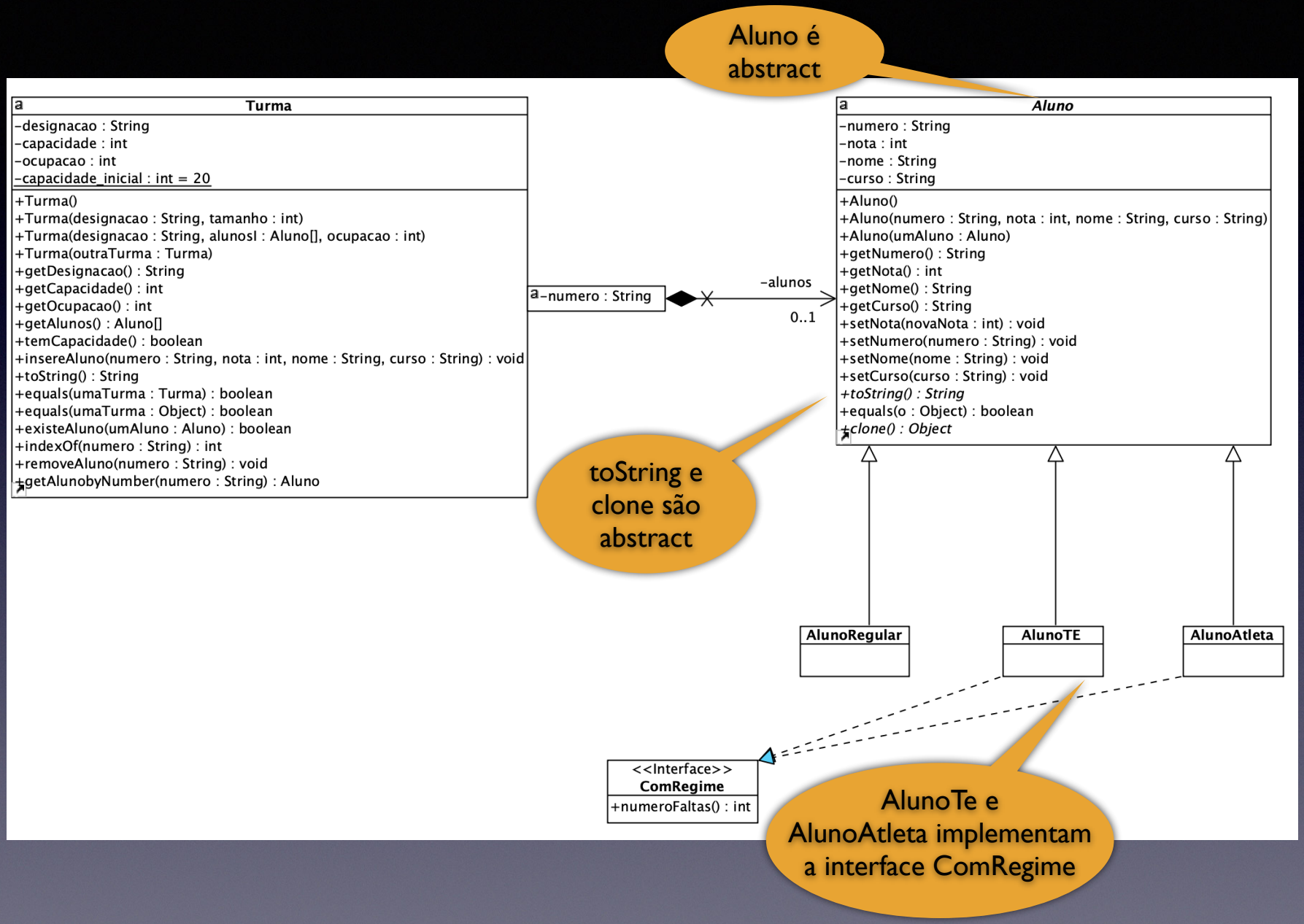
- se a chave existir temos acesso a uma instância de Aluno, caso contrário a zero!



- 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

O package java.util.function

- Este package possui várias interfaces funcionais que foram adicionadas e cujo objectivo é poderem ser utilizadas para parametrizar as classes através da injecção de comportamento.
- São interfaces funcionais cuja definição contém apenas um método.
  - que é abstracto e será instanciado pelo programador.

(*) seguindo a estrutura apresentada em Functional Interfaces in Java, Ralph Lecessi, 2019

- Considerem-se os quatros tipos básicos de entidades deste package:

Model	Has Arguments	Returns a Value	Description
Predicate	yes	boolean	Tests argument and returns true or false.
Function	yes	yes	Maps one type to another.
Consumer	yes	no	Consumes input (returns nothing).
Supplier	no	yes	Generates output (using no input).

- Estas definições são utilizadas criando-se uma função de um destes tipos de dados e definindo-a utilizando uma expressão lambda.

Predicate<T>

- A interface `Predicate` faz a avaliação de uma condição associada a um valor de entrada que é de um tipo genérico.
- O método devolve true se a condição for verdade, falso caso contrário

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

- O programador associa depois um tipo de dados quando define o predicado:

```
Predicate<Integer> p1;
```

- e faz a associação da expressão que representa a condição. Exemplo:

```
p1 = x -> x > 7;
```

- testando a veracidade com a invocação de test

```
if (p1.test(9))
  System.out.println("Predicate x > 7 é verdade para x == 9.");
```

- Claro que uma mais valia desta abordagem é poder passar para um método um predicado
  - invocar o teste do predicado dentro do método

```
public static void result(Predicate<Integer> p, Integer arg) {
  if (p.test(arg))
    System.out.println("O predicado é true para " + arg);
  else
    System.out.println("O predicado é falso para " + arg);
}
```

```
public static void main(String[] args) {
   Predicate<Integer> p1 = x -> x == 5;

   result(p1,5);
   result(y -> y%2 == 0, 5);
}
```

- Num exemplo de uma das aulas práticas em que temos uma turma como contendo um Map<Aluno>, podemos definir um método mais geral que permita identificar os alunos que satisfazem determinado predicado:

```
/**
 * Passar um predicado para um método, possibilitando assim a parametrização
 * de comportamento através de um parâmetro.
 * Este método devolve a lista dos alunos que satisfazem o predicado p
 */

public List<Aluno> alunosqueRespeitamP(Predicate<Aluno> p) {
  return this.alunos.values().
              stream().
              filter(a -> p.test(a)).
              map(Aluno::clone).
              collect(Collectors.toList());
}
```

generaliza-se o mecanismo de filtragem

- Definindo agora os predicados podemos obter diversos filtros de informação:
  - não tendo que repetir código
  - parametrizando o comportamento de filtragem pretendido

```
Predicate<Aluno> p = a -> a instanceof AlunoTE;

List<Aluno> alunosTE = t.alunosqueRespeitamP(p);
for (Aluno a: alunosTE)
  System.out.println(a.toString());
```

Function<T,R>

- Esta interface funcional aceita dois tipos de dados, sendo que recebe um parâmetro `T` e devolve um resultado do tipo `R`.

```
@FunctionalInterface
public interface Function<T, R>
{
    R apply(T t);
    ...
}
```

- o método `apply` transforma o objecto do tipo `T` numa resposta do tipo `R`.

- Na definição da Function é necessário associar os tipos de dados e depois definir o seu comportamento através de uma lambda expression

```
Function<String, Integer> f;
f = x -> Integer.parseInt(x);
```

- a utilização faz-se pela invocação na Function do método `apply`

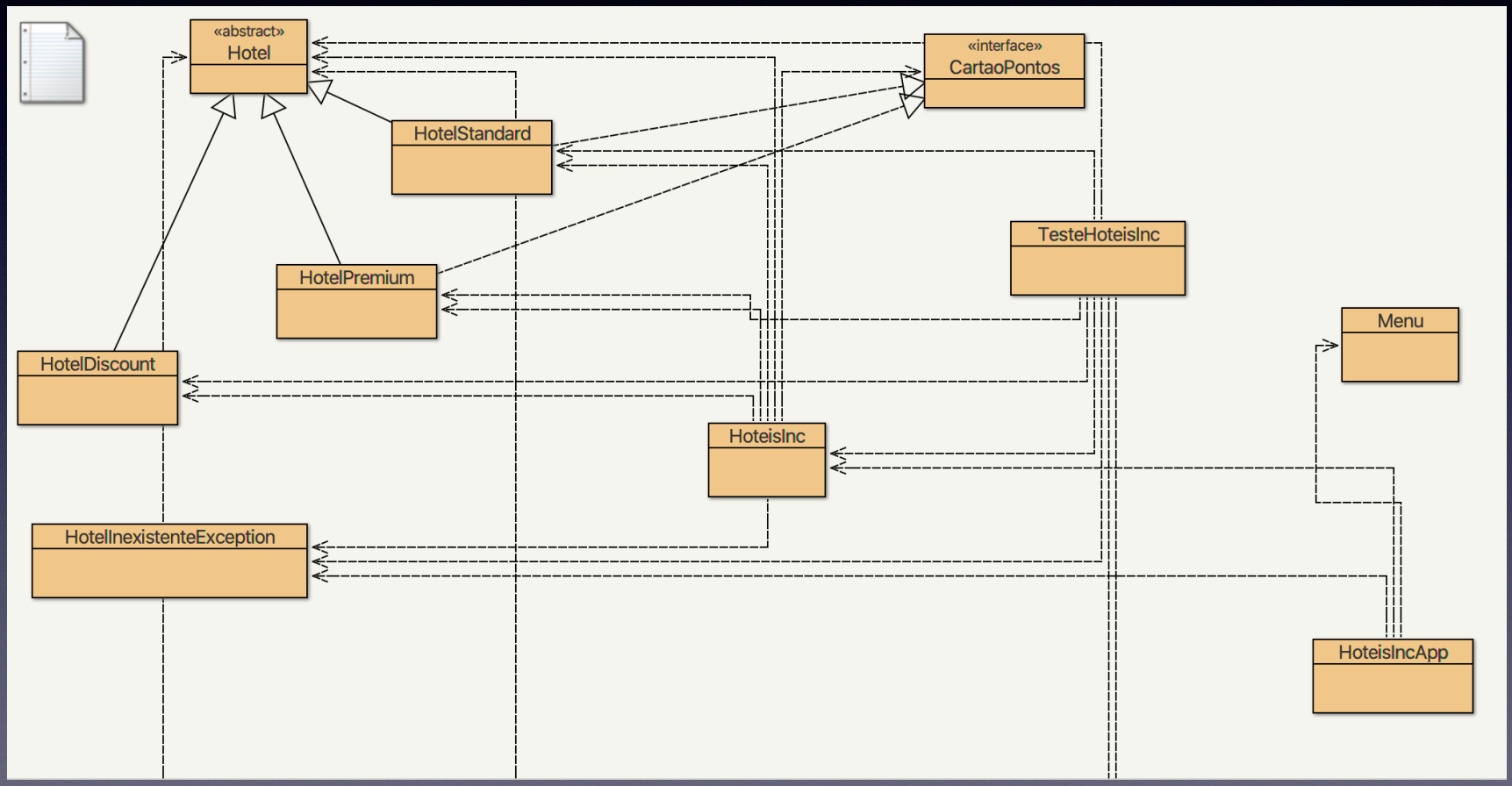
```
Integer i = f.apply("100");
System.out.println(i);

OUTPUT:

100
```

- Como vimos para os predicados é possível passar estas Function como parâmetro

```
public class Transformer {

  private static <T,R> R transform(T t, Function<T,R> f) {
    return f.apply(t);
  }


  public static void main(String[] args) {

    Function<String,Integer> fsi = x -> Integer.parseInt(x);
    Function<Integer,String> fis = x -> Integer.toString(x);

    Integer i = transform("100", fsi);
    String s = transform(200, fis);
    System.out.println(i);
    System.out.println(s);
  }

}
```

- Podemos utilizar a declaração de Function
  - para tornar mais genéricos todos os métodos que fazem travessias a colecções e aplicam uma função
  - evita-se a repetição de código
  - tal é possível derivado do facto de que é permitido passar uma expressão lambda como parâmetro

- Seja o seguinte projecto:



- Consideremos que queremos fazer diferentes métodos:
  - obter o preço de todos os hotéis da cadeia de hotéis
  - listar o nome de todos os hotéis
  - listar o número de estrelas de todos os hotéis
- Todos estes métodos vão ter um código muito semelhante.

- Pode ser definido um método que aplique a função a todos os objectos do tipo Hotel

```java
/**
 * Método que recebe uma Function<T,R> e aplica a todos os
 * hóteis existentes.
 */

public <R> List<R> devolveInfoHoteis(Function<Hotel,R> f) {
  List<R> res = new ArrayList<>();
  for(Hotel h: this.hoteis.values())
    res.add(f.apply(h));


  return res;
}
```

- E, de cada vez, que seja necessário aplicar um novo tipo de selecção de informação criamos uma `Function`

- Exemplo:

```
Function<Hotel,Double> fpreco = h -> h.precoNoite();
Function<Hotel,String> fnome = h -> h.getNome();


List<Double> precos = osHoteis.devolveInfoHoteis(fpreco);
for(Double d: precos)
  System.out.println(d.toString());

List<String> nomes = osHoteis.devolveInfoHoteis(fnome);
for(String d: nomes)
  System.out.println(d.toString());
```

- Existe também a possibilidade de definir funções binárias, do tipo Function<T,U,R>

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
    R apply(T t, U u);
}
```

- apesar de terem tipos T e U, poderão representar o mesmo tipo de dados.

Consumer<T>

- Esta interface é utilizada para processamento de informação
  - não devolve resultado, é `void`, e aplica o método `accept` a todos os objectos

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);
    ...
}
```

- Podemos também generalizar muitos métodos que fazem travessias e modificam os visitados

```
/**
 * Método que recebe uma Consumer<T> e aplica a todos os
 * hóteis existentes.
 */

public void aplicaTratamento(Consumer<Hotel> c) {
  this.hoteis.values().forEach(h -> c.accept(h));
}
```

```
Consumer<Hotel> downgradeEstrelas = h -> h.setEstrelas(h.getEstrelas()-1);
osHoteis.aplicaTratamento(downgradeEstrelas);
```

Supplier<T>

- Supplier é uma interface que é utilizada para gerar informação. Não tem parâmetros e devolve um resultado do tipo T.

```
@FunctionalInterface
public interface Supplier<T>
{
     T get();
}
```

- Pode ser utilizada por exemplo para permitir criar métodos que fazem *pretty printing* de informação.
  - Criam-se expressões de pretty printing e aplicam-se a todos os objectos
  - Com isto evita-se estar sempre a alterar o método toString.