

# Interfaces

...

• • •

```
/**  
 * @param desig nome da turma.  
 */  
public TurmaSet(String desig) {  
    designação = desig;  
    alunos = new TreeSet<>(new ComparatorAlunoNome());  
}
```

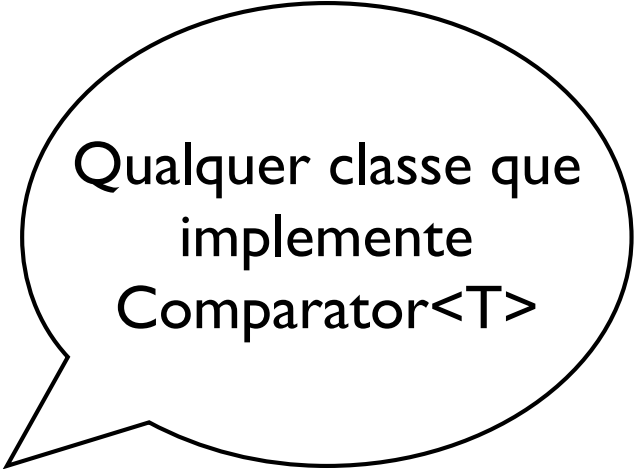
# TreeSet<T>

**public TreeSet<T>()**

Utiliza ordem natural de **T**

**public TreeSet<T>(Comparator<T> c)**

Utiliza o comparator **c**



Qualquer classe que  
implemente  
Comparator<T>

# Comparator<T>

Permitem definir diferentes critérios de ordenação

Implementam o método **int compare(T o1, T o2)**

Mesmas regras de **compareTo** aplicadas a **o1** e **o2**

```
/**
 * Comparator de Aluno - ordenação por número.
 *
 * @author José Creissac Campos
 * @version 20160403
 */

import java.util.Comparator;
public class ComparatorAlunoNum implements Comparator<Aluno> {
    public int compare(Aluno a1, Aluno a2) {
        int n1 = a1.getNumero();
        int n2 = a2.getNumero();

        if (n1==n2) return 0;
        if (n1>n2) return 1;
        return -1;
    }
}
```

```
/**
 * Comparator de Aluno - ordenação por nome.
 *
 * @author José Creissac Campos
 * @version 20160403
 */

import java.util.Comparator;
public class ComparatorAlunoNome implements Comparator<Aluno> {
    public int compare(Aluno a1, Aluno a2) {
        return a1.getNome().compareTo(a2.getNome());
    }
}
```

# Comparator<T>

Java 6...

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

# Interfaces Comparable<T> e Comparator<T>

## Interface Comparable<T>

### Method Summary

#### All Methods

#### Instance Methods

#### Abstract Methods

#### Modifier and Type

#### Method and Description

int

`compareTo(T o)`

Compares this object with the specified object for order.

## Interface Comparator<T>

### Method Summary

#### All Methods

#### Static Methods

#### Instance Methods

#### Abstract Methods

#### Default Methods

#### Modifier and Type

#### Method and Description

int

`compare(T o1, T o2)`

Compares its two arguments for order.

boolean

`equals(Object obj)`

Indicates whether some other object is "equal to" this comparator.

# Interfaces

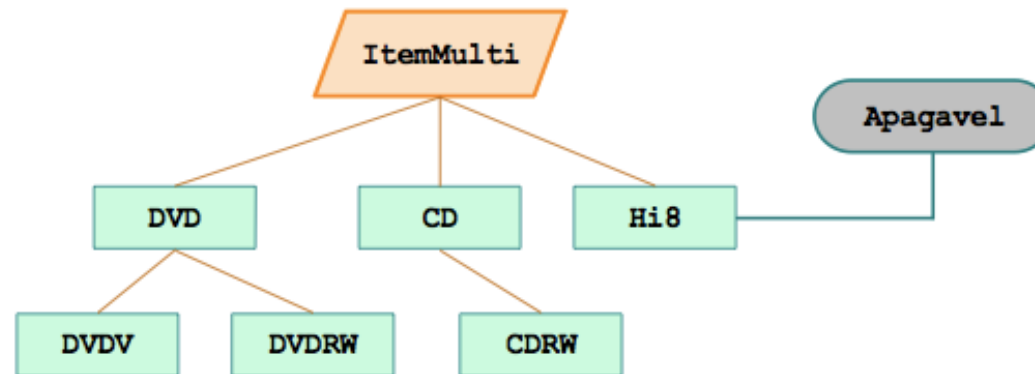
**Comparable<T>** e **Comparator<T>**  
*são interfaces*

Interfaces definem APIs (conjunto de métodos) que as classes que as implementam devem por sua vez implementar

Interfaces definem novos Tipos de Dados

# Porquê interfaces...

Uma hierarquia típica



```
public class Hi8 extends ItemMulti implements Apagavel {  
    //  
    private int minutos;  
    private double ocupacao;  
    private int gravacoes;  
    . . . .  
    // implementação de Apagavel  
    public void apaga() { ocupacao = 0.0; gravacoes = 0; }  
}
```

```
public interface Apagavel {  
    /**  
     * Apagar  
     */  
    public void apaga();  
}
```

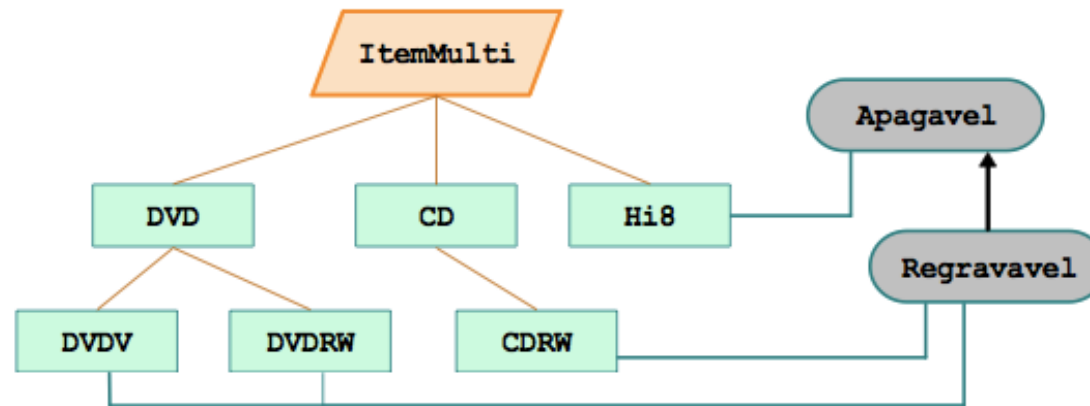


Qualquer instância de Hi8 é também do tipo Apagavel, ou seja:

```
Hi8 filme1 = new Hi8("A1", "2005", "obs1", 180, 40.0, 3);  
Apagavel apg1 = filme1;  
apg1.apaga();
```

no entanto, a uma instância de Hi8 que vemos como sendo um Apagavel, apenas lhe poderemos enviar métodos definidos nessa interface (i.e. nesse tipo de dados)

Temos também a possibilidade de ter vários tipos de dados válidos para diferentes objectos.



Por vezes, o que provavelmente acontece com **Regravavel**, a interface é apenas um marcador.

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

```
ItemMulti[] filmes = new ItemMulti[ 500] ;  
// código para inserção de filmes no array....  
int contaReg = 0;  
for(ItemMulti filme : filmes)  
    if (filme instanceof Regravavel) contaReg++  
out.printf("Existem %d items regraváveis.", contaReg);
```

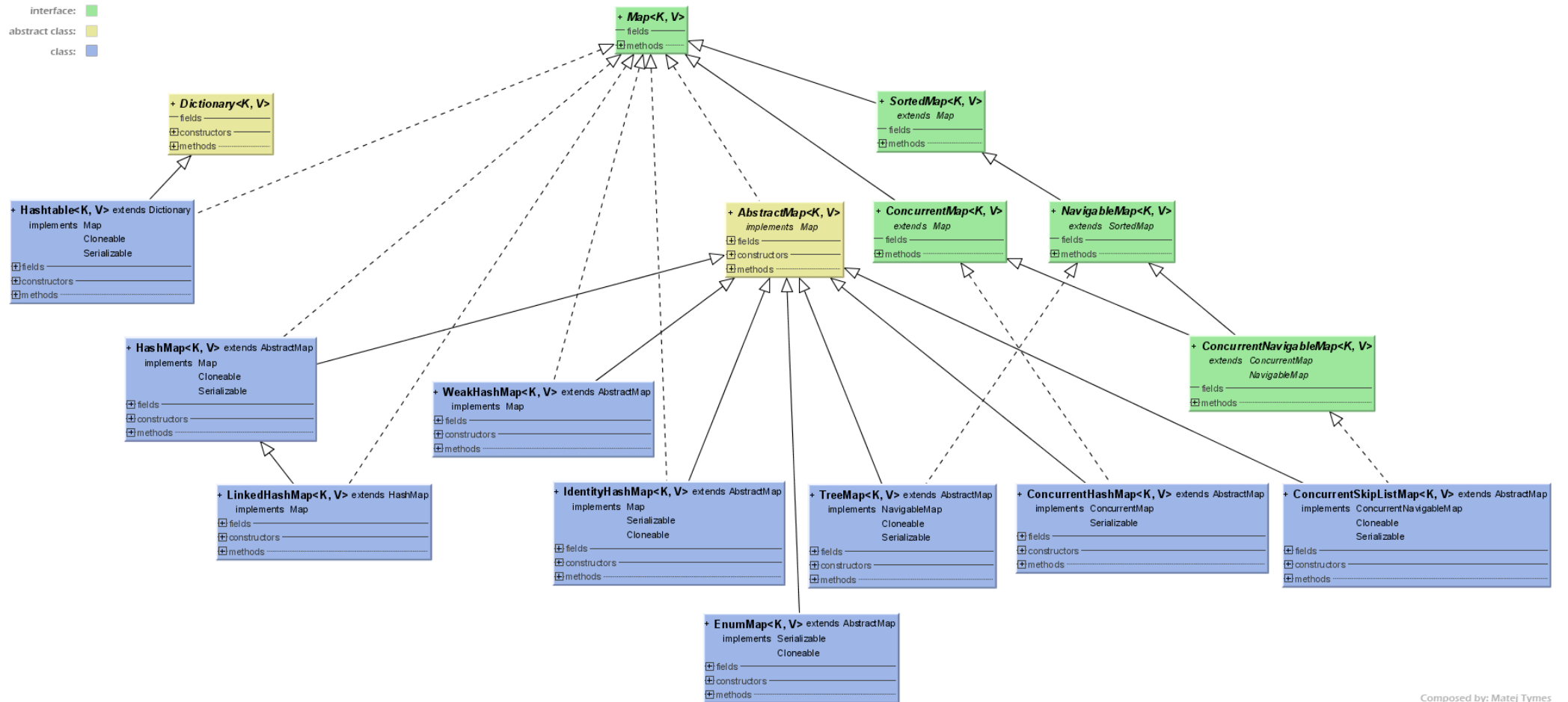
na expressão acima não se está a validar a classe, mas sim o tipo de dados estático

Do ponto de vista da concepção de arquitecturas de objectos, as interfaces são importantes para:

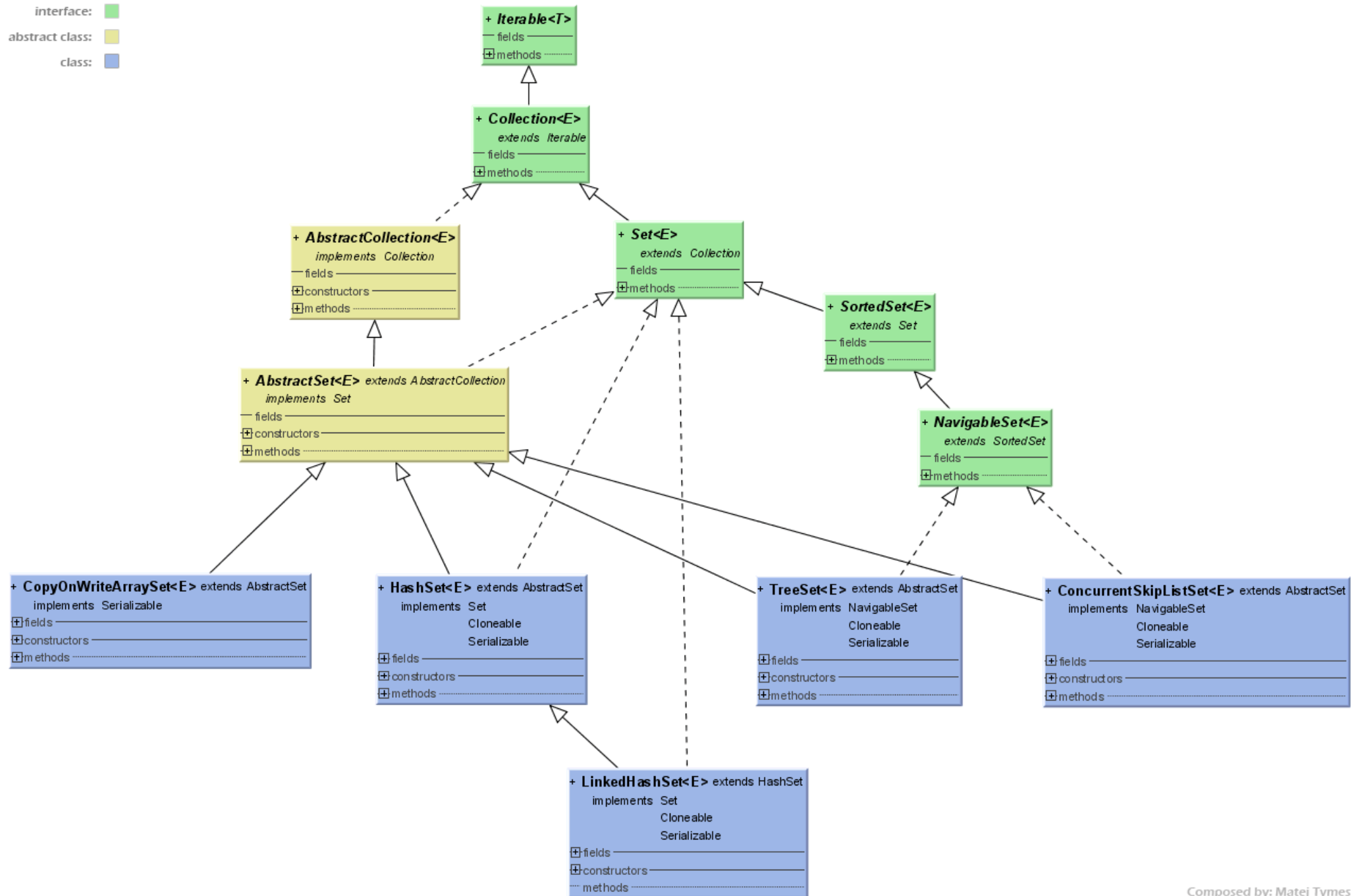
- reunirem similaridades comportamentais, entre classes não relacionadas hierarquicamente

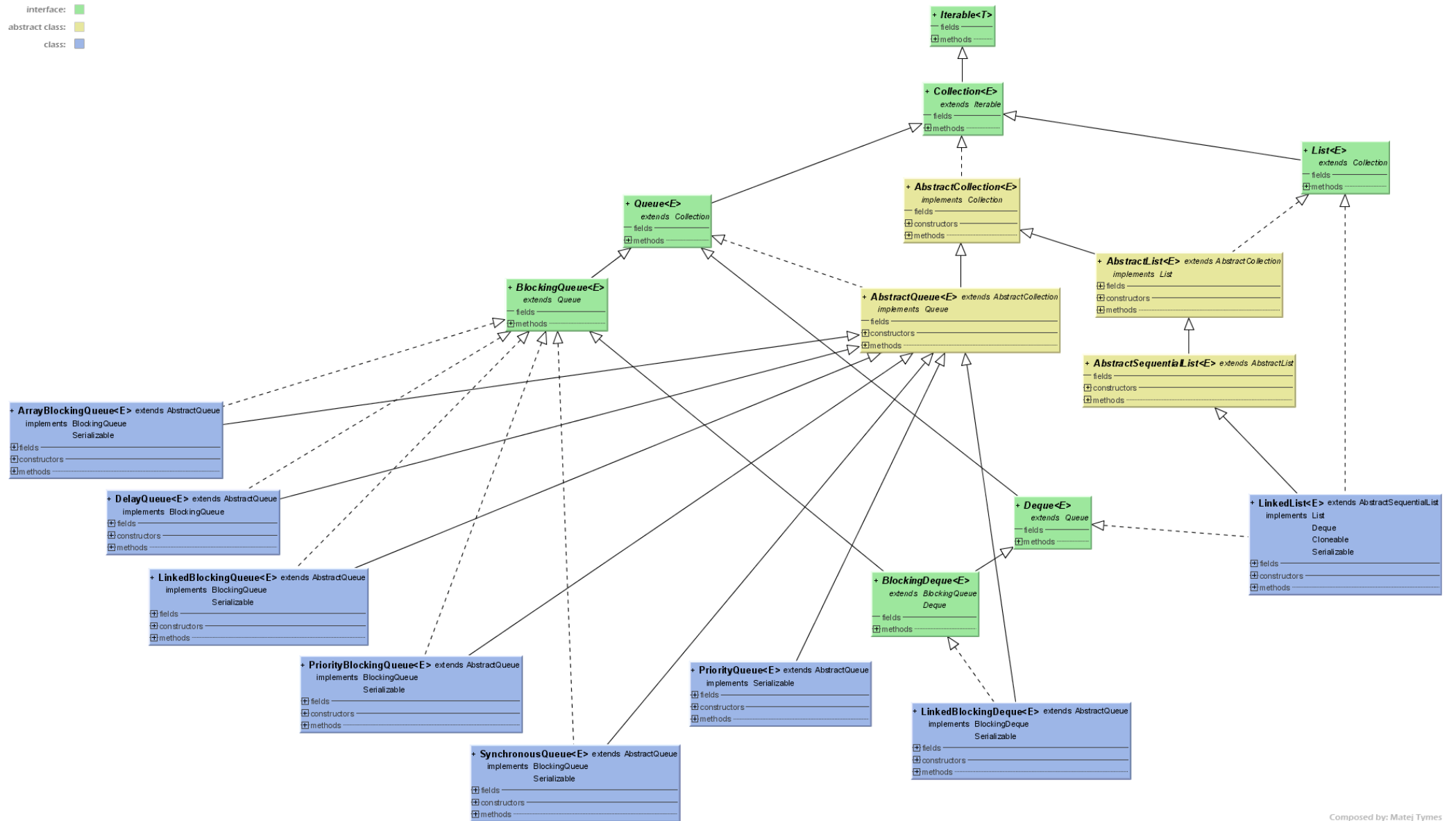
- definirem novos tipos de dados

- conterem a API comum a vários objectos, sem indicarem a classe dos mesmos (como no caso do JCF)

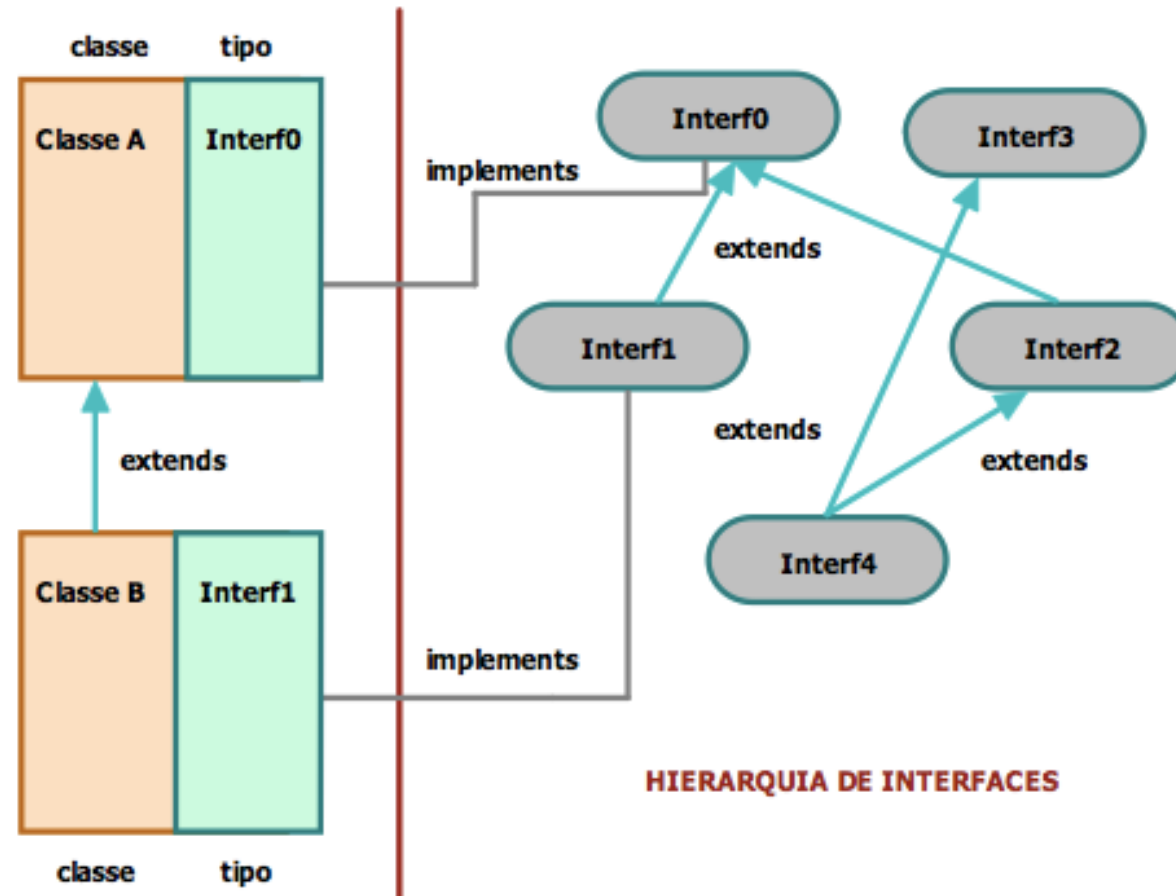


interface: ■  
 abstract class: ■  
 class: ■





O modelo geral é assim:



onde coexistem as noções de classe e interface, bem assim como as duas hierarquias

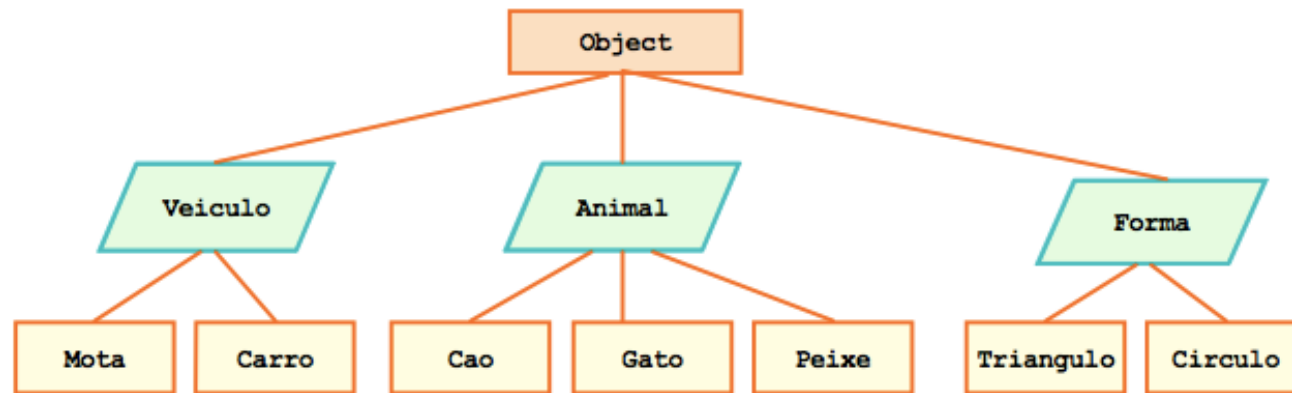


Uma classe passa a ter duas vistas (ou classificações) possíveis:

é subclasse, por se enquadrar na hierarquia normal de classes, tendo um mecanismo de herança simples de estado e comportamento

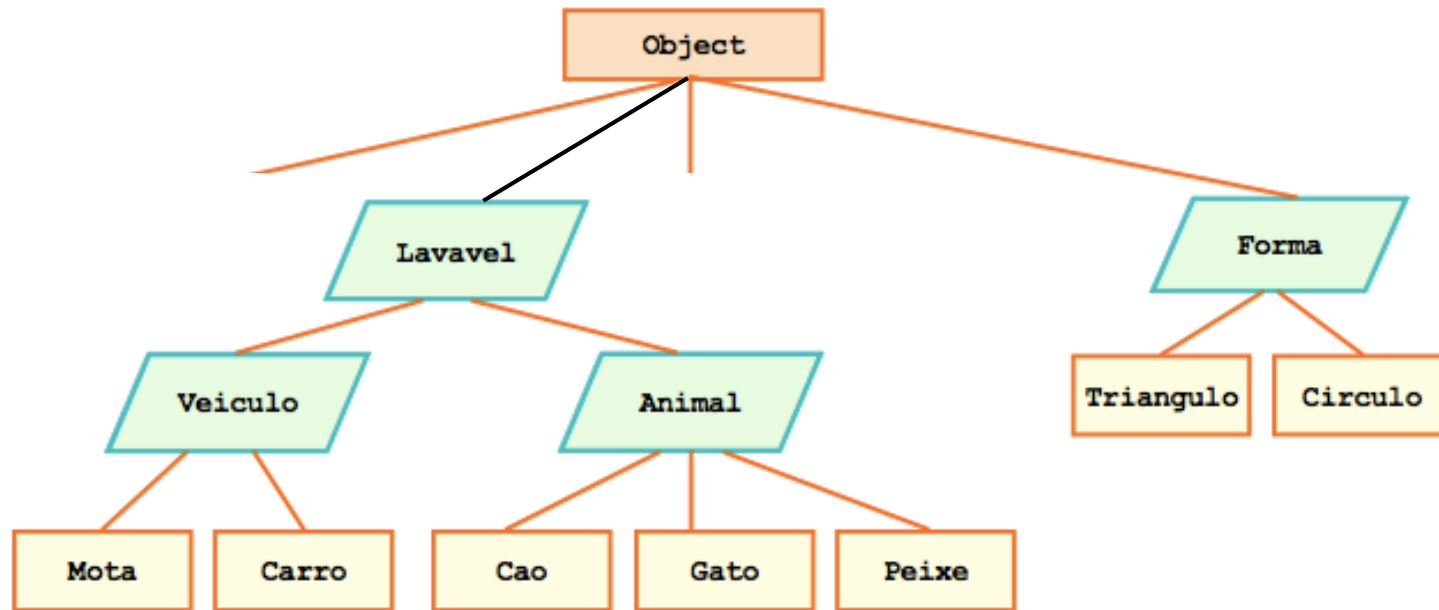
é subtipo, por se enquadrar numa hierarquia múltipla de definições de comportamento abstracto (puramente sintático)

Existem situações que apenas são possíveis de satisfazer considerando as duas hierarquias.



se fosse importante saber os objectos desta hierarquia que poderiam ser lavados, então...

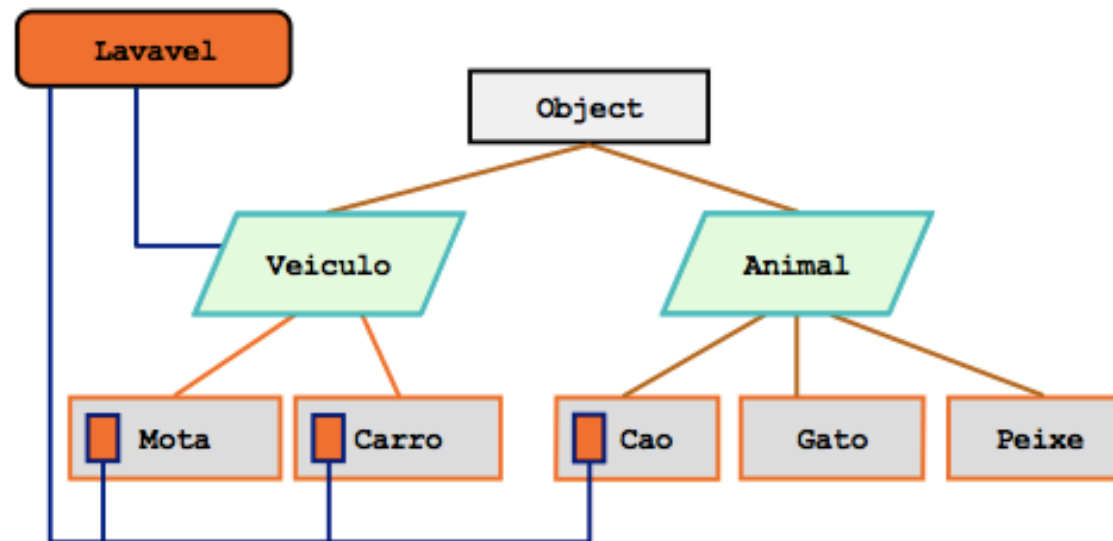
Podíamos pensar em...



no entanto, esta solução obrigaria objectos não “laváveis”, a ter um método `aLavar ( )`

Com a utilização de ambas as hierarquias poderemos ter:

```
public interface Lavavel {  
    public void aLavar();  
}
```



# 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 vs. Classes Abstractas

Propriedades	Interfaces	Classes Abstractas
<b>Herança Múltipla</b>	Uma classe pode implementar várias interfaces	Uma classe só pode "estender" uma classe abstracta
<b>Implementação</b>	Não aceita nenhum código	O código que se pretender, cf. as necessidades da concepção
<b>Constantes</b>	Apenas <code>static final</code> . Podem ser usadas pelas classes implementadoras sem qualificadores	De classe ou de instância e código de inicialização se necessário
<b>Relação <i>is-a</i></b>	Em geral não descrevem a identidade principal de uma classe, mas capacidades periféricas aplicáveis a outras classes	Descreve uma identidade básica de todos os seus descendentes
<b>Adicionar funcionalidade</b>	Não deve ser feito pois implicaria refazer todas as implementações da interface. Usar antes subinterfaces	Sem problema para as subclasses desde que o método seja codificado na classe abstracta
<b>Homogeneidade</b>	Classes que as implementam são homogéneas na API	Subclasses homogéneas em API e que partilham algum código, por exemplo, classes de dados

# no entanto...

Em Java8 as interfaces podem:

- incluir métodos **static**

- fornecer implementações por omissão dos métodos (*keyword* **default**)

Functional Interfaces (Java8)

- uma interface com um único método abstracto (e qualquer número de métodos **default**)

- Possibilitam o uso de expressões lambda e de referências a métodos ou construtores