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 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?

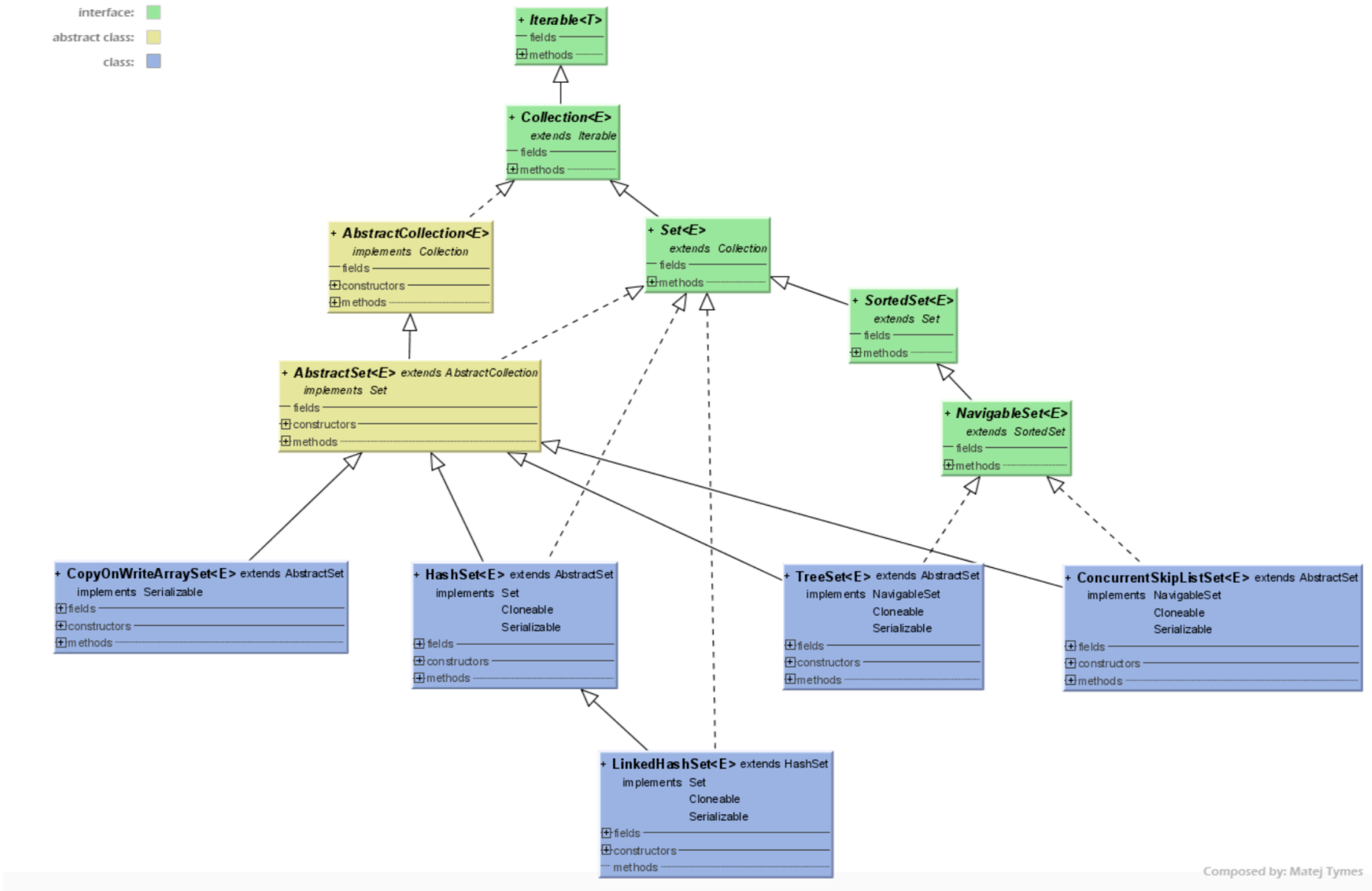
- A solução passa por passar a utilizar um outro mecanismo de tipo de dados existente em Java: as **interfaces**.

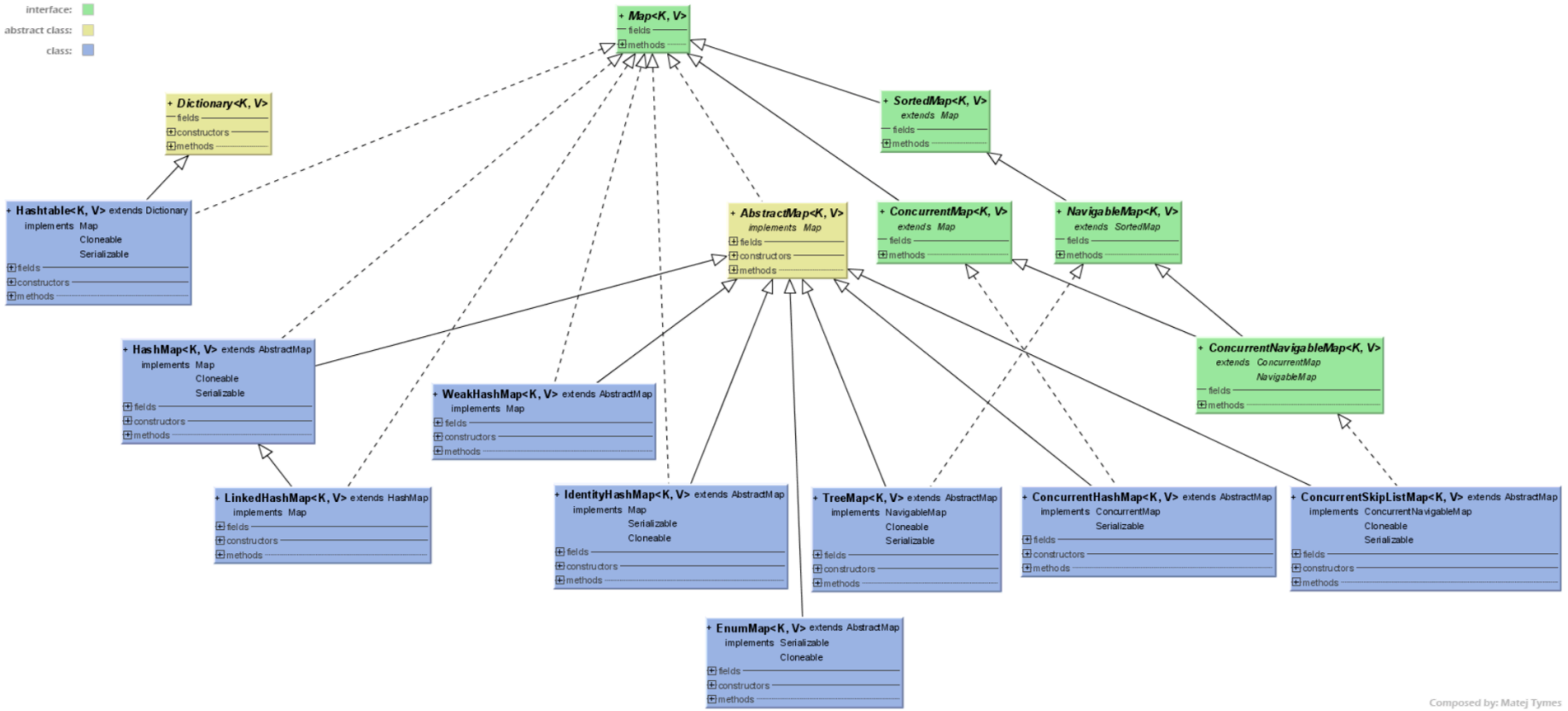
- Interfaces não são classes, mas são apenas declarações sintácticas de expressão de comportamento

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

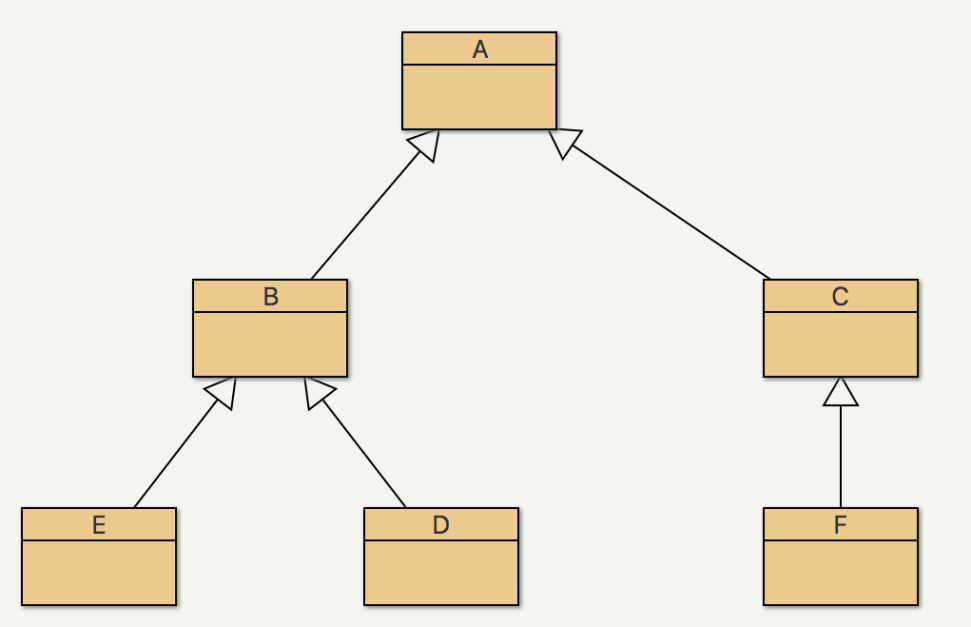
  - as classes decidem com que tipo de interfaces querem ser compatíveis

- 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
+ HashMap<K, V> extends AbstractMap
implements Map
Cloneable
Serializable
fields
constructors
methods
+ WeakHashMap<K, V> extends AbstractMap
implements Map
fields
constructors
methods
+ ConcurrentNavigableMap<K, V>
extends ConcurrentMap
NavigableMap
fields
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:



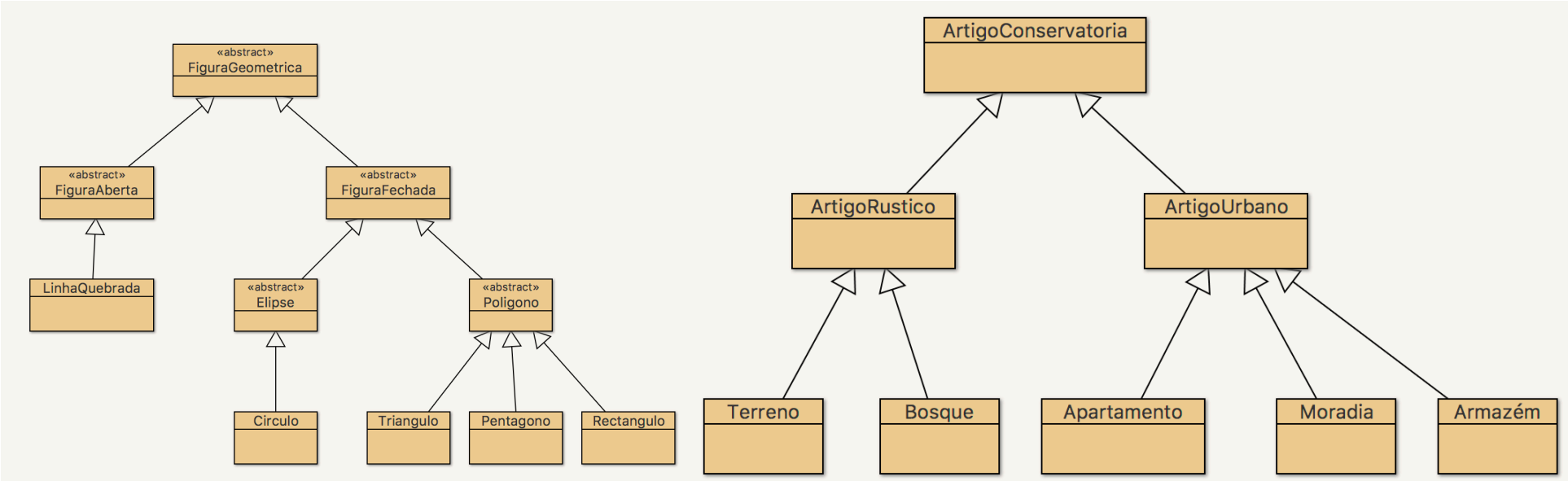
- Se quisermos representar uma colecção de objectos do tipo A (e dos seus subtipos) apenas precisamos de declarar:

  - 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 desejam 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

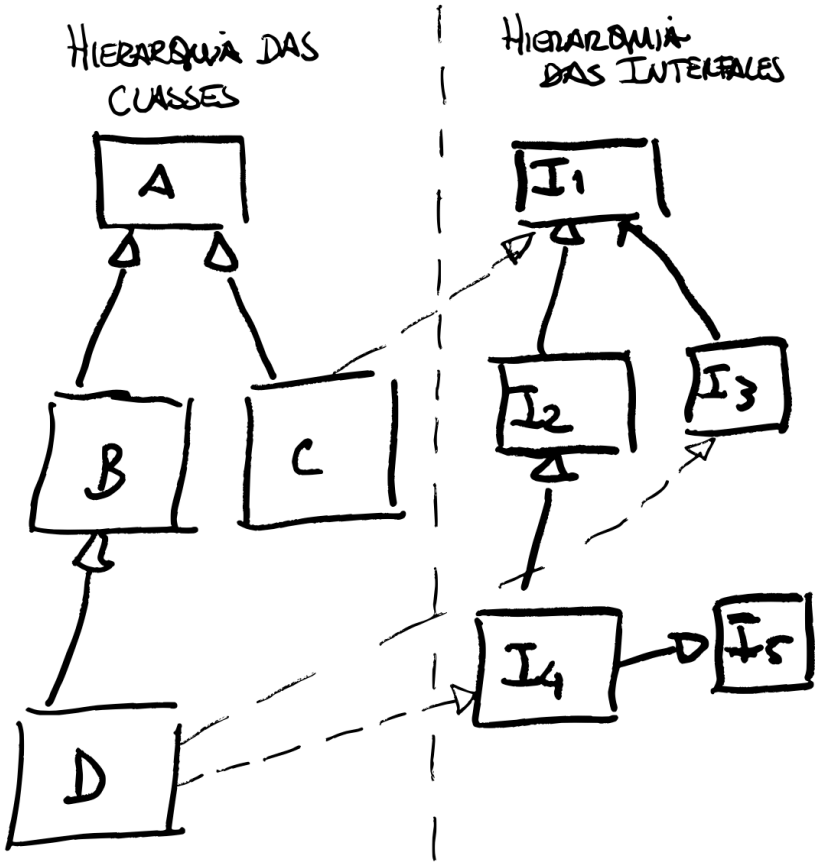
```
List<ArtigoConservatoria> propriedades = new ArrayList<>();
...
...
int numMensuravel = 0;
for(ArtigoConservatoria ac: this.propriedades)
  if (ac instanceof Mensuravel)
    numMensuravel++;
```

- 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)

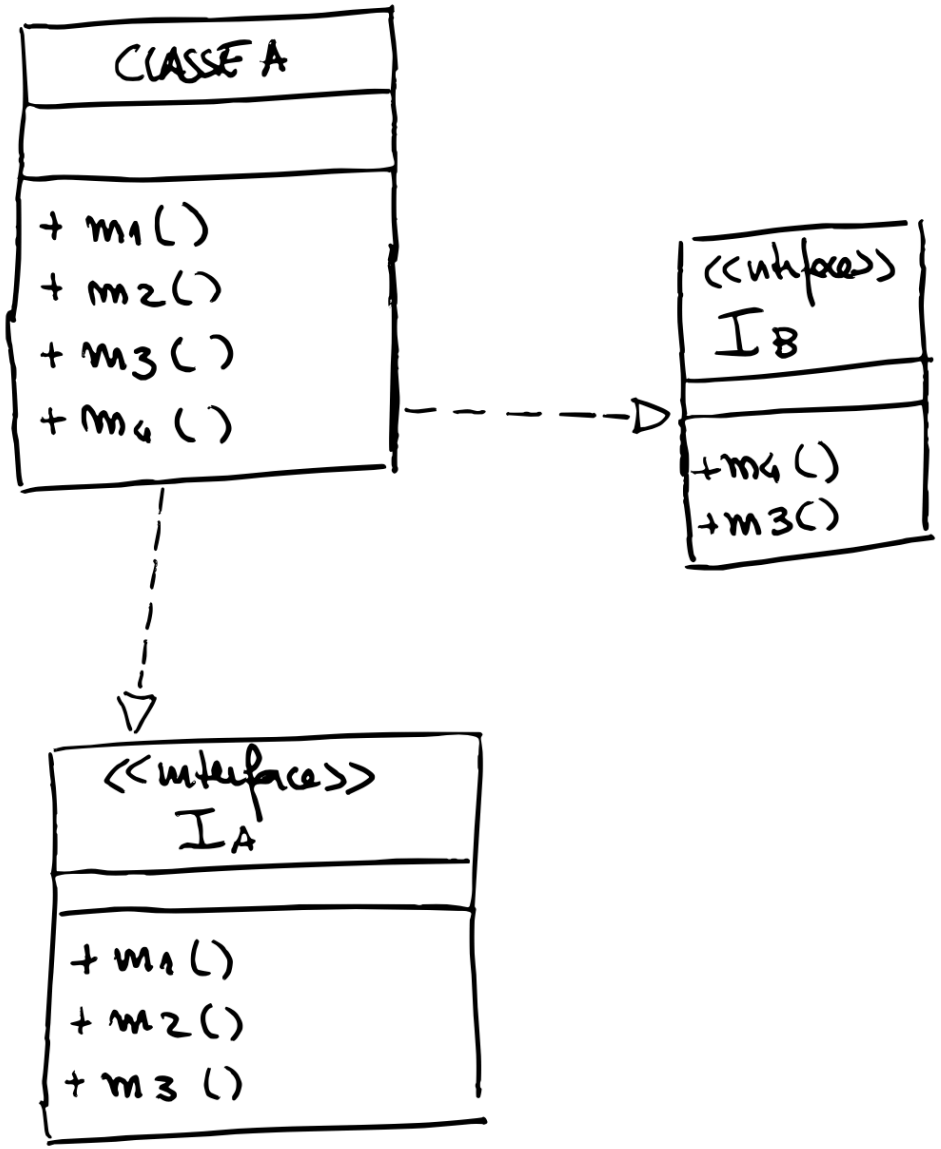
- existem duas hierarquias, que estão relacionadas, e o programador pode tirar partido disso.




```
I1 obj = new C();
I4 outro = new D();
I3 umdaOutro = new D();
```

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

- IA 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

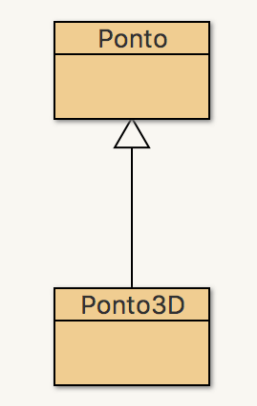
- As declarações constantes de uma interface constituem um contrato, isto é, especificam a forma do comportamento que as implementações oferecem

- 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 …