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

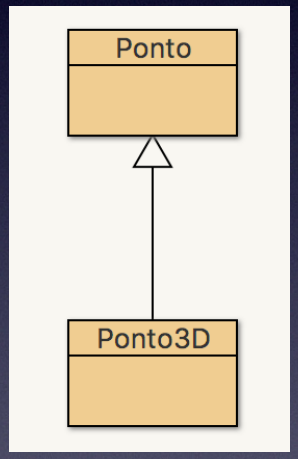
- possibilitam que um mesmo objecto possa apresentar-se sob a forma de diferentes tipos de dados, exibindo comportamentos diferentes, consoante o que seja pretendido
  - permite esconder a natureza do objecto e fazer a sua *tipagem* de acordo com as necessidades do momento
  - permite limitar os métodos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