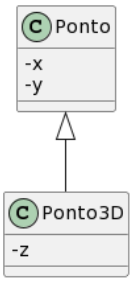
Ficha de trabalho #7

Herança
Construtores e métodos equals e toString da subclasse
Polimorfismo – sobreposição (override) e sobrecarga (overload)
Procura estática e dinâmica de métodos
Palavras reservadas instanceOf, getClass() e .class
Upcasting e downcasting
Classes e métodos abstratos

Um dos tipos de relações entre classes em POO é a **herança**, que acontece quando temos generalizações ou especializações entre as classes.



Aqui temos um exemplo de **herança**: a classe Ponto3D **herda** da classe Ponto, ou seja:

- Ponto é uma **generalização** de Ponto3D
- Ponto3D é uma **especialização** de Ponto.

Ponto é uma **superclasse** e Ponto3D é uma **subclasse**.

A classe Ponto3D, ao herdar da classe Ponto, vai herdar todos os seus atributos e métodos, podendo ter atributos e métodos específicos.

Podemos dizer que o Ponto3D **é um** Ponto com uma coordenada extra (a cota, z).

O código da classe Ponto3D é o seguinte:

```
public class Ponto3D extends Ponto{
    private double z;

    private static final double COTA_OMISSAO = 0.0;

    public Ponto3D() {
        super();
        this.z = COTA_OMISSAO;
    }

    public Ponto3D(double x, double y, double z) {
        super(x,y);
        this.z = z;
    }

    public Ponto3D(Ponto3D p) {
        super(p);
        this.z = p.z;
    }

    public Ponto3D clone() {
        return new Ponto3D(this);
    }

    public double getZ() {
        return z;
    }

    public void setZ(double z) {
        this.z = z;
    }

    @Override
    public boolean equals(Object obj) {
        if(!super.equals(obj)) {
            return false;
        }
        Ponto3D other = (Ponto3D) obj;
        return this.z == other.z;
    }

    @Override
    public String toString() {
        return "Ponto3D [z=" + z + "] (x, y)="+super.toString();
    }

    public double distanciaEuclideana(Ponto3D outro) {
        return Math.sqrt(
                Math.pow(this.getX() - outro.getX(), 2)
                + Math.pow(this.getY() - outro.getY(), 2)
                + Math.pow(this.z - outro.z, 2));
    }
}
```

A classe Ponto3D herda (palavra reservada **extends**) da classe Ponto e tem apenas um atributo extra (z).
Tem 3 **construtores**:

- Vazio, que chama o construtor vazio da superclasse com a instrução **super()** e define a cota por omissão
- Completo, que chama o construtor completo da superclasse com a instrução **super(x,y)** para preencher a parte correspondente a um Ponto (abcissa e ordenada) e depois define a cota
- Cópia, que chama o construtor de cópia da superclasse com a instrução **super(p)** para copiar a parte correspondente a um ponto e depois copia a cota

Tem métodos:

- Getter e setter da coordenada z
- **equals**, que compara primeiro as superclasses com a instrução super.equals(obj) e só depois o atributo específico
- **toString**, que mostra a descrição textual de um Ponto3D recorrendo à descrição textual da superclasse através da instrução super.toString()
- distanciaEuclideana, que calcula a distância entre dois pontos 3D. este método já tinha sido implementado na classe Ponto para calcular distâncias entre pontos com duas dimensões. Ao ser chamado, o java sabe qual é o método que deve ser executado, devido ao polimorfismo, e escolhe sempre o mais específico.

Polimorfismo significa “diversas formas”. Em POO, o polimorfismo permite a reutilização de código (métodos) por estes assumirem diversas formas. Há dois tipos de polimorfismo:

- **Sobrecarga** (***overload***): permitir, dentro da mesma classe, mais de um método com o mesmo nome, sendo distinguidos pela **procura estática** (durante a programação e a compilação) pela assinatura (número, tipo e ordem dos parâmetros).
- **Sobreposição** (***override***): permitir que numa subclasse se reescreva um método existente na superclasse, por ser necessário ter um comportamento diferente. Têm a mesma assinatura e, durante a execução (**procura dinâmica**), o Java irá escolher o método mais específico.

Podemos criar e utilizar os vários pontos de várias formas (por exemplo, no main):

```
public class TestePonto3D {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(1,2);
        Ponto p2 = new Ponto(3,4);

        System.out.println(Ponto.distanciaEuclideana(p1, p2));
        System.out.println(Ponto.distanciaEuclideana(1, 2, 3, 4));
    }
}
```

Criar pontos e mostrar distâncias entre pontos.

Como o método distanciaEuclideana tem várias implementações com diferentes assinaturas na classe Ponto, através do **polimorfismo** por **sobrecarga**, o java sabe qual é o método que tem de chamar (pelos parâmetros).

```
public class TestePonto3D {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(1,2);
        Ponto p2 = new Ponto(3,4);
        Ponto3D p3 = new Ponto3D(1,2,3);
        Ponto3D p4 = new Ponto3D(4,5,6);
        ArrayList<Ponto> pontos = new ArrayList<>();
        pontos.add(p1);
        pontos.add(p2);
        pontos.add(p3);
        pontos.add(p4);

        for(Ponto p: pontos) {
            System.out.println(p);
        }
    }
}
```

Criar pontos e pontos 3D, criar um ArrayList de pontos e adicionar todos os pontos ao ArrayList (Ponto3D é subclasse de Ponto – Ponto3D é Ponto –, pelo que pode ser adicionado a um ArrayList do tipo Ponto).

Percorrer a lista de pontos e mostrá-los.

Como o método toString está implementado na superclasse e na subclasse, através do **polimorfismo** por **sobreposição**, o java sabe qual deles tem de chamar (o mais específico)

```
public class TestePonto3D {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(1,2);
        Ponto p2 = new Ponto(3,4);
        Ponto3D p3 = new Ponto3D(1,2,3);
        Ponto3D p4 = new Ponto3D(4,5,6);

        ArrayList<Ponto> pontos = new ArrayList<>();
        pontos.add(p1); pontos.add(p2); pontos.add(p3); pontos.add(p4);

        for(Ponto p: pontos) {
            if(p instanceof Ponto3D) {
                System.out.println(p);
            }
        }
    }
}
```

Ao percorrer a lista de pontos, para cada ponto vai verificar se é uma instância da classe Ponto3D (palavra reservada **instanceOf**) e só mostra os objetos do tipo Ponto3D.

Como Ponto3D é subclasse de Ponto, se tivéssemos instanceOf Ponto, iria mostrar todos os pontos

```
public class TestePonto3D {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(1,2);
        Ponto p2 = new Ponto(3,4);
        Ponto3D p3 = new Ponto3D(1,2,3);
        Ponto3D p4 = new Ponto3D(4,5,6);

        ArrayList<Ponto> pontos = new ArrayList<>();
        pontos.add(p1); pontos.add(p2); pontos.add(p3); pontos.add(p4);

        for(Ponto p: pontos) {
            if(p.getClass() == Ponto.class) {
                System.out.println(p);
            }
        }
    }
}
```

Ao percorrer a lista de pontos, para cada ponto vai verificar se a sua classe (método **getClass()**) é Ponto (**Ponto.class**) e só mostra os objetos do tipo Ponto (excluindo Ponto3D, que têm classe diferente)

Como há uma relação entre a superclasse e a subclasse, podemos a qualquer momento ter necessidade de converter objetos de um dos tipos para outro, para podermos por exemplo chamar métodos específicos a um dos tipos. Utilizamos:

- **Downcasting** à conversão de um objeto da superclasse num objeto da subclasse (descida na hierarquia).
- **Upcasting** à conversão de um objeto da subclasse num objeto da superclasse (subida na hierarquia)

```
public class TestePonto3D {
    public static void main(String[] args) {
        Ponto p1 = new Ponto(1,2);
        Ponto p2 = new Ponto(3,4);
        Ponto3D p3 = new Ponto3D(1,2,3);
        Ponto3D p4 = new Ponto3D(4,5,6);

        ArrayList<Ponto> pontos = new ArrayList<>();
        pontos.add(p1); pontos.add(p2); pontos.add(p3); pontos.add(p4);

        for(Ponto p: pontos) {
            if(p instanceof Ponto3D) {
                Ponto3D p3d = (Ponto3D)p;
                System.out.println(p3d.getZ());
            }
        }
    }
}
```

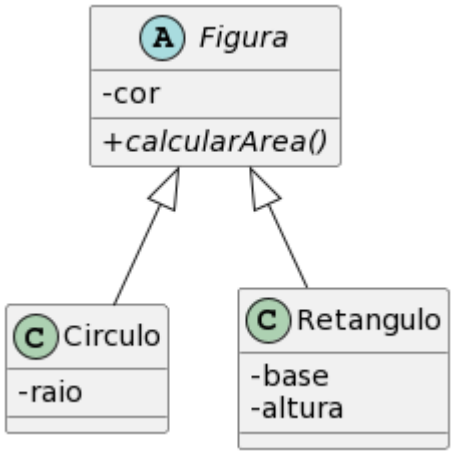
Exemplo de **downcasting** de um ponto pertencente à lista de pontos para Ponto3D, de forma a ficar acessível o método getZ, específico do Ponto3D.

Nota: só é possível efetuar a conversão caso o objeto seja mesmo uma instância da subclasse, pelo que a verificação (com **instanceOf**) deve ser **sempre** efetuada.

```
public class TestePonto3D {
    public static void main(String[] args) {
        Ponto3D p3 = new Ponto3D(1,2,3);
        Ponto p = (Ponto)p3;
        System.out.println(p.getX());
    }
}
```

Exemplo de **upcasting** de um Ponto3D para Ponto. Ficam apenas acessíveis os métodos de Ponto, e os métodos de Ponto3D deixam de o ser (exceto os métodos com override – ex: toString – , porque a procura dinâmica de métodos continua a ter o mesmo resultado).

Em algumas situações, pode não fazer sentido que todas as classes sejam instanciáveis (que seja permitido criar objetos de todas as classes). Podemos nesse caso recorrer à utilização de classes abstratas, que poderão ter várias implementações com características diferentes.



Neste exemplo, temos uma **classe abstrata** Figura, porque não faz sentido criar uma figura sem saber de que tipo é. A figura tem o atributo cor e um método que, neste caso, é um **método abstract**, porque será implementado em cada uma das subclasses (porque para cada subclasse tem um comportamento diferente, dado que diferentes figuras têm diferentes fórmulas para cálculo da área). O objetivo dos métodos abstract é obrigar a implementar um comportamento nas subclasses. Neste caso, para todas as figuras é preciso haver uma forma de calcular a área

```java
public abstract class Figura {
    private String cor;

    private static final String COR_OMISSAO = "Branco";

    public Figura() {
        this.cor = COR_OMISSAO;
    }

    public Figura(String cor) {
        this.cor = cor;
    }

    public Figura(Figura f) {
        this.cor = f.cor;
    }

    public String getCor() {
        return cor;
    }

    public void setCor(String cor) {
        this.cor = cor;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Figura other = (Figura) obj;
        return this.cor.equals(other.cor);
    }

    public String toString() {
        return "Figura de cor " + this.cor;
    }

    public abstract double calcularArea();
}
```

A classe Figura é uma classe abstract com o atributo cor.

Tem 3 construtores:

- Vazio, que define a cor por omissão
- Completo, que define a cor através do valor passado por parâmetro
- Cópia, que copia uma figura

E os métodos:

- Getter e setter de cor
- Equals, que determina se a figura é igual a outro objeto, comparando as cores
- toString, que retorna a descrição textual da figura
- calcularArea, a definição de um método abstract que terá de ser implementado nas subclasses

Como é uma classe não instanciável (é abstract e não faz sentido haver objetos do tipo Figura sem ser de um dos subtipos), não tem método clone()

```java
public class Retangulo extends Figura {
    private double base;
    private double altura;

    private static final double BASE_OMISSAO = 0;
    private static final double ALTURA_OMISSAO = 0;

    public Retangulo() {
        super();
        this.base = BASE_OMISSAO;
        this.altura = ALTURA_OMISSAO;
    }

    public Retangulo(String cor, double base, double altura) {
        super(cor);
        this.base = base;
        this.altura = altura;
    }

    public Retangulo(Retangulo r) {
        super(r);
        this.base = r.base;
        this.altura = r.altura;
    }

    public double getBase() {
        return base;
    }

    public double getAltura() {
        return altura;
    }

    public void setBase(double base) {
        this.base = base;
    }

    public void setAltura(double altura) {
        this.altura = altura;
    }

    @Override
    public boolean equals(Object obj) {
        if (!super.equals(obj))
            return false;
        Retangulo other = (Retangulo) obj;
        return this.altura == other.altura
                && this.base == other.base;
    }

    @Override
    public String toString() {
        return super.toString()
          + " é um retangulo de base=" + base
          + " e altura=" + altura;
    }

    @Override
    public double calcularArea() {
        return this.base * this.altura;
    }
}
```

A classe Retangulo herda da classe Figura e tem os atributos base e altura.

Tem 3 construtores:

- Vazio, que chama o construtor vazio da superclasse e define a base e a altura por omissão
- Completo, que chama o construtor completo da superclasse para definir a cor através do valor passado por parâmetro, e define a base e a altura através dos valores passados por parâmetro
- Cópia, que copia um retângulo, chamando o construtor cópia da superclasse para copiar a parte correspondente à Figura

E os métodos:

- Getters e setters de base e altura
- Equals, que determina se o retângulo é igual a outro objeto, verificando primeiro se as superclasses são iguais e passando só depois aos atributos específicos de Retangulo
- toString, que retorna a descrição textual do retangulo, chamando primeiro o toString da superclasse e acrescentando-lhe a informação especifica
- calcularArea, implementação do método abstract de acordo com as propriedades do retângulo

```java
public class Circulo extends Figura {
    private double raio;

    private static final double RAIO_OMISSAO = 0;

    public Circulo() {
        super();
        this.raio = RAIO_OMISSAO;
    }

    public Circulo(String cor, double raio) {
        super(cor);
        this.raio = raio;
    }

    public Circulo(Circulo c) {
        super(c);
        this.raio = c.raio;
    }

    public double getRaio() {
        return raio;
    }

    public void setRaio(double raio) {
        this.raio = raio;
    }

    @Override
    public boolean equals(Object obj) {
        if (!super.equals(obj))
            return false;
        Circulo other = (Circulo) obj;
        return this.raio == other.raio;
    }

    @Override
    public String toString() {
        return super.toString()
                + " é um círculo de raio=" + raio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * Math.pow(raio, 2);
    }
}
```

A classe Circulo herda da classe Figura e tem o atributo raio.

Tem 3 construtores:

- Vazio, que chama o construtor vazio da superclasse e define o raio por omissão
- Completo, que chama o construtor completo da superclasse para definir a cor através do valor passado por parâmetro, e o raio através do valor passado por parâmetro
- Cópia, que copia um retângulo, chamando o construtor cópia da superclasse para copiar a parte correspondente à Figura

E os métodos:

- Getter e setter de raio
- Equals, que determina se o circulo é igual a outro objeto, verificando primeiro se as superclasses são iguais e passando só depois aos atributos específicos de Circulo
- toString, que retorna a descrição textual do circulo, chamando primeiro o toString da superclasse e acrescentando-lhe a informação especifica
- calcularArea, implementação do método abstract de acordo com as propriedades do circulo

```java
public class TesteFiguras {
    public static void main(String[] args) {
        Retangulo r1 = new Retangulo("azul",1,2);
        Circulo c1 = new Circulo("verde",1);

        ArrayList<Figura> figuras = new ArrayList<>();
        figuras.add(r1);
        figuras.add(c1);

        for(Figura f: figuras) {
            String frase = f + " de área: " + f.calcularArea();
            if(f instanceof Retangulo) {
                frase += ", porque ";
                frase += ((Retangulo)f).getBase();
                frase += " * " ;
                frase += ((Retangulo)f).getAltura();
                frase += " = ";
                frase += f.calcularArea();
            } else if(f instanceof Circulo) {
                frase += ", porque ";
                frase += Math.PI;
                frase += " * " ;
                frase += ((Circulo)f).getRaio();
                frase += " * " ;
                frase += ((Circulo)f).getRaio();
                frase += " = ";
                frase += f.calcularArea();
            }
            System.out.println(frase);
        }
    }
}
```

No main podemos criar círculos e retangulos, adicioná-los a uma lista de Figuras (porque tanto Circulo como Retangulo são figuras).

Podemos então percorrer a lista das figuras e, recorrendo a downcasting e ao polimorfismo por sobreposição, mostrar diferentes informações para as diferentes figuras.

1. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar colaboradores de uma empresa. Para a **empresa** é necessário guardar o nome e NIF e uma lista de colaboradores. Para todos os **colaboradores** pretendemos guardar o nome, NIF, data de nascimento e morada. Deve ser possível calcular o vencimento de qualquer colaborador, sabendo que diferentes tipos de colaboradores têm diferentes fórmulas de cálculo de vencimento. Para os **colaboradores à comissão** é necessário guardar o salário base e a comissão e o seu vencimento é a soma do salário base com a comissão. Para os **colaboradores à hora** é necessário guardar o número de horas e o valor por hora (igual para todos os colaboradores à hora, de momento no valor de 11,5€) e o seu vencimento é o produto entre o número de horas e o valor por hora. Pretende-se que o programa permita:
   - Criar uma empresa e 2 trabalhadores de cada tipo e adicioná-los à lista de trabalhadores
   - Mostrar todos os trabalhadores e os seus vencimentos
   - Mostrar os trabalhadores agrupados por tipo

2. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar automóveis num stand. Para o **stand** é necessário guardar o nome e a morada e a lista de automóveis. Para todos os **automóveis** pretendemos guardar a marca, o modelo, a cor, o número de portas e o preço e pretendemos saber autonomia (em km) do depósito/bateria, sabendo que diferentes tipos de automóveis têm diferentes formas de cálculo da autonomia. Para **automóveis a combustível** precisamos de guardamos também o tipo de combustível, o consumo (em L/100km), a capacidade do depósito e a potência do motor; a autonomia em km é o quociente entre a capacidade do depósito e o consumo do automóvel, multiplicado por 100. Para **automóveis elétricos** guardamos também a capacidade da bateria em KWH e o consumo (em KW/10km); a autonomia é o quociente entre a capacidade e o consumo multiplicado por 10. Pretende-se que o programa permita:
   - Criar um stand e 2 automóveis de cada tipo e adicioná-los à lista de automóveis
   - Mostrar todos os automóveis juntamente com a sua autonomia
   - Mostrar os automóveis agrupados por tipo

3. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar produtos à venda numa mercearia. Sobre a **mercearia** pretendemos guardar apenas o nome e a lista de produtos disponíveis, que poderão ser vendidos à unidade, a peso, ou por volume. Para todos os **produtos** necessitamos de saber o seu nome e de ter um método de cálculo do preço, dependente da quantidade vendida, e que tem diferentes fórmulas de cálculo para diferentes tipos de produtos. Para os **produtos à unidade** necessitamos de saber o preço unitário e o preço da embalagem (o preço da embalagem é igual para todos os produtos vendidos à unidade e tem um valor inicial de 0,50€); o preço final (a quantidade vendida é sempre 1) é a soma do preço do produto e da embalagem. Para os **produtos a peso** é necessário saber o preço por kg; o preço final é o produto da quantidade vendida pelo preço por kg. Para os **produtos por volume** temos de guardar o preço por litro e o tipo de vasilhame (plástico adiciona 0,50€ ao preço e vidro acrescenta 1,00€); o preço final é o produto da quantidade vendida pelo preço por litro, acrescido do preço do vasilhame. Pretende-se que o programa permita:
   - Criar uma mercearia e 2 produtos de cada tipo e adicioná-los ao catálogo da mercearia
   - Mostrar todos os produtos juntamente com o seu preço (considerar 1 unidade, 1Kg e 1L)
   - Mostrar os produtos, agrupados por tipo de produto