Ficha de trabalho #5

Construtor de cópia
Associação
Método equals

Ver ficha#4: Atributos e métodos de instância e de classe; Construtor vazio; Constantes

Torna-se útil que as classes tenham mais um tipo de construtor, o **construtor cópia**, que recebe por parâmetro um objeto da mesma classe, e copia o valor de todos os atributos para o novo objeto que está a ser construído. Por exemplo:

```
public Ponto(Ponto p) {
    this.x = p.x;
    this.y = p.y;
}
```

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

```
public Reta(Reta r) {
    this.pontos = r.pontos;
}
```

Em Programação Orientada a Objetos pode haver 5 relações entre classes: associação, agregação, composição, dependência e herança.

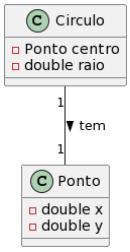
Uma **dependência** ocorre quando uma classe usa objetos de outra classe. Por exemplo, a classe que contém o main tem uma relação de dependência com as classes dos objetos que instancia.

Uma **associação** ocorre quando o tipo dos atributos de uma classe é outra classe. Uma associação tem associada uma **cardinalidade** como, por exemplo:

- Um-para-um (1 – 1): um dos atributos de uma classe é um objeto de outra classe
- Um-para-muitos (1 – *): um dos atributos de uma classe é um conjunto de objetos de outra

Um exemplo de uma **associação 1 – 1** é a classe Circulo, com um Ponto como centro.

```
public class Circulo {
    private Ponto centro;
    private double raio;

    public Circulo(Ponto centro, double raio) {
        this.centro = centro;
        this.raio = raio;
    }

    public Circulo(int xCentro, int yCentro, double raio) {
        this.centro = new Ponto(xCentro,yCentro);
        this.raio = raio;
    }

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



A classe Circulo tem 3 construtores:

1. Recebe um Ponto e o raio e constrói o círculo com centro no ponto e o raio definido.
2. Recebe as coordenadas x e y de um ponto e o raio e constrói um novo ponto para ser considerado o centro do círculo (com x e y) e atribui-lhe o raio
3. Recebe um circulo e constrói um novo circulo com os mesmos valores dos atributos do círculo que recebeu

Nota: A classe círculo tem vários construtores com o mesmo nome, mas com parâmetros diferentes. Isto é um exemplo de **polimorfismo** (vários métodos com o mesmo nome), em que os métodos estão ***overloaded***. O compilador distingue os métodos pela sua assinatura (número, tipo e ordem dos parâmetros), sabendo a qual deles nos referimos. O polimorfismo pode ocorrer em métodos que não sejam os construtores.

```
public class TesteCirculo {
    public static void main(String[] args) {
        Ponto centro1 = new Ponto(2,3);

        Circulo c1 = new Circulo(centro1, 1);
        System.out.println(c1);

        Circulo c2 = new Circulo(new Ponto(7,8), 3);
        System.out.println(c2);

        Circulo c3 = new Circulo(1, 4, 2);
        System.out.println(c3);
    }
}
```

Tendo isto em conta, podemos criar círculos de várias formas. O código à esquerda cria 3 círculos:

1. Circulo c1, com centro no ponto centro1 (definido acima) e raio de 1cm. Vai chamar o 1º construtor de círculo.
2. Circulo c2 com centro num novo ponto criado apenas para esse efeito e raio 3cm. Vai chamar o 1º construtor de círculo.
3. Circulo c3 com centro com coordenadas 1 e 4 e 2cm de raio. Vai chamar o 2º construtor de circulo.

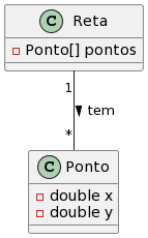
Como um Circulo contém um Ponto, podemos manipular o Ponto a partir do Circulo. Por exemplo, na classe Circulo podemos métodos para redefinir as coordenadas do Ponto centro. Ao chamar este método a partir do main, iríamos redefinir as coordenadas do centro do Circulo

```
public void redefinirXCentro(double val) {
    this.centro.setX(val);
}

public void redefinirYCentro(double val) {
    this.centro.setY(val);
}
```

```
public class TesteCirculo {
    public static void main(String[] args) {
        Circulo c2 = new Circulo(new Ponto(7,8), 3);
        System.out.println(c2);
        c2.redefinirXCentro(10);
        c2.redefinirYCentro(20);
        System.out.println(c2);
    }
}
```

Um exemplo de uma **associação 1- *** é a classe Reta, que tem um array de Pontos onde são guardados os pontos inicial e final da reta. Neste caso, temos uma associação 1 – 2 (uma reta tem associados 2 pontos).

```
public class Reta {
    private Ponto[] pontos;

    public Reta(Ponto[] pontos) {
        this.pontos = new Ponto[2];
        this.pontos[0] = pontos[0];
        this.pontos[1] = pontos[1];
    }

    public Reta(Ponto inicio, Ponto fim) {
        this.pontos = new Ponto[2];
        this.pontos[0] = inicio;
        this.pontos[1] = fim;
    }

    public Reta() {
        this.pontos = new Ponto[2];
    }

    public Reta(Reta r) {
        this.pontos = r.pontos;
    }

    ...
}
```



A classe Reta tem 4 construtores:

1. Recebe um array de pontos, cria o array para guardar os seus pontos (com tamanho 2 porque só precisamos dos pontos inicial e final) e atribui os pontos inicial e final
2. Recebe dois pontos, cria o array para guardar os seus pontos e atribui os pontos inicial e final
3. Cria apenas o array de tamanho 2. Implica que iremos necessitar, por exemplo, dos dois métodos abaixo para definir os pontos inicial e final da reta
4. Copia o array de pontos da outra reta para a reta que estamos a criar

Como uma reta contém dois pontos, podemos também manipular esses pontos a partir da reta.

O objetivo do método **equals** é verificar se dois objetos são iguais. Em todas as classes que criarmos iremos precisar do método equals. Este método irá comparar os objetos e todos os atributos dos objetos para verificar se são iguais. Por exemplo:

```
class Ponto {
  // ...
  public boolean equals(Object outro) {
    if (this == outro)
        return true;
    if (outro == null || getClass() != outro.getClass())
        return false;
    Ponto outroPonto = (Ponto) outro;
    return x == outroPonto.x && y == outroPonto.y;
  }
  // ...
}
```

```
class Circulo {
  // ...
  public boolean equals(Object obj) {
      if (this == obj)
          return true;
      if (obj == null || getClass() != obj.getClass())
          return false;
      Circulo other = (Circulo) obj;
      return this.centro.equals(other.centro)
        && this.raio == other.raio;
  }
  // ...
}
```

```
class Reta {
  // ...
  public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null || getClass() != obj.getClass())
        return false;
    Reta other = (Reta) obj;
    for(int i=0; i<this.pontos.length; i++) {
        if(!this.pontos[i].equals(other.pontos[i])){
            return false;
        }
    }
    return true;
  }
  // ...
}
```

1. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar pessoas.

   1.1. Analisar a classe Data fornecida e seguir os passos no final da ficha para “Reutilizar a classe Data em vários projetos”

   1.2. Para cada pessoa pretendemos guardar o nome (String) e a data de nascimento (Data). Devem ser implementados os construtores:
   - Pessoa()
   - Pessoa(String nome, Data dataNascimento)
   - Pessoa(String nome, int diaNascimento, int mesNascimento, int anoNascimento)
   - Pessoa(Pessoa p)

   Devem também ser implementados os getters e setters, toString, equals e o método:
   - calcularDiferenca que recebe uma Pessoa e retorna a diferença entre as datas de nascimento das pessoas

   1.3. Criar uma classe TestePessoa para testar as funcionalidades de Pessoa:
   - Criar pelo menos duas pessoas, com datas de nascimento diferentes
   - Mostrar a diferença entre as datas de nascimento das pessoas
   - Criar uma terceira pessoa com os mesmos dados da primeira
   - Verificar se a primeira pessoa é igual à terceira
   - Comentar o método equals e voltar a verificar

2. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar livros.

   2.1. Para cada livro pretendemos guardar o título (String), autor (Pessoa, à semelhança do ex 1), data de publicação (Data, à semelhança do ex 1), editor (String), e preço (double). Devem ser implementados os construtores:
   - Livro()
   - Livro(String titulo, Pessoa autor, Data dataPublicacao, String editor, double preco)
   - Livro(String titulo, String nomeAutor, Data dataNascimentoAutor, int anoPublicacao, int mesPublicacao, int diaPublicacao, String editor, double preco)
   - Livro(Livro l)

   Devem também ser implementados os getters e setters, toString, equals e os métodos:
   - precoComDesconto que retorna o preço ao aplicar uma percentagem de desconto

   2.2. Criar uma classe TesteLivro para testar as funcionalidades implementadas
   - Criar dois livros
   - Criar um terceiro livro com os mesmos dados do primeiro
   - Ver o preço dos livros
   - Verificar se o primeiro livro é igual ao terceiro
   - Comentar o método equals na classe Livro e voltar a verificar

3. Pretende-se criar um programa em Java, seguindo as regras da POO e com documentação recorrendo a JavaDoc, para representar atletas inscritos num ginásio, à semelhança do ex 3 da ficha#4.

   3.1. Para cada atleta pretendemos guardar, para além da informação da ficha anterior, um cartão, cuja informação é: código (inteiro, gerado automaticamente e sequencialmente) e o nº de entradas (int), nº de saídas (int) e deve ter os construtores completo, vazio e de cópia, getters, setters, equals e toString e os métodos:
   - registarEntrada, que incrementa o valor do número de entradas
   - registarSaida, que incrementa o valor do número de saídas
   - validarEntradasSaidas, que verifica se o número de entradas é igual ao de saídas

   3.2. A classe Atleta deve ter os construtores:
   - Atleta(String nome, int idade, char genero, double peso, double altura, Cartao cartao)
   - Atleta(String nome, int idade, char genero, double peso, double altura, int nrEntradasCartao, int nrSaidasCartao)
   - Atleta(String nome, int idade, char genero, double peso, double altura)
   - Atleta()
   - Atleta(Atleta a)

   Devem ainda ser implementados os getters e setters e os métodos toString e equals e os seguintes métodos:
   - registarEntrada, que aumenta em uma unidade o número de entradas do cartão
   - registarSaida, que aumenta em uma unidade o número de saídas do cartão
   - validarEntradasSaidas(), que valida as entradas/saídas do cartão

   3.3. Implementar uma classe de teste para testar as funcionalidades implementadas
   - Criar um atleta com um novo cartão
   - Mostrar os dados do atleta
   - Registar duas entradas e duas saídas do atleta
   - Mostrar se o número de entradas e saídas são iguais
   - Registar mais uma entrada do atleta
   - Voltar a mostrar se o número de entradas e saídas são iguais

Reutilizar a classe Data em vários projetos:

1. Criar um projeto chamado Utilitarios
   1.1. Dentro do projeto criar um package chamado utilitários
   1.2. Dentro do package criar uma classe chamada Data e inserir o código fornecido
2. No novo projeto onde quisermos utilizar a Data (ex: no projeto pessoas):
   2.1. Clicar com o botão direito do rato no projeto
   2.2. Escolher a opção “Build Path” seguida de “Configure Build Path…”
   2.3. Na tab “Projects” escolher “Classpath” e clicar em “Add…”
   2.4. Selecionar o projeto Utilitarios e clicar em “OK”
   2.5. Clicar em “Apply and Close”
   2.6. Eliminar o ficheiro “module-info.java”
   2.7. Nas classes onde se usar a Data vai aparecer um erro “Data cannot be resolved to a type”: clicar no indicador do erro e selecionar e opção “Import ‘Data’ (utilitários)”
   2.8. Todas as classes que estiverem no projeto Utilitarios ficam acessíveis