# Formas 2D

## 1 Objetivos

- Praticar o projeto de hierarquias de classes.
- Praticar o uso de classes abstratas.

## 2 Definição do problema

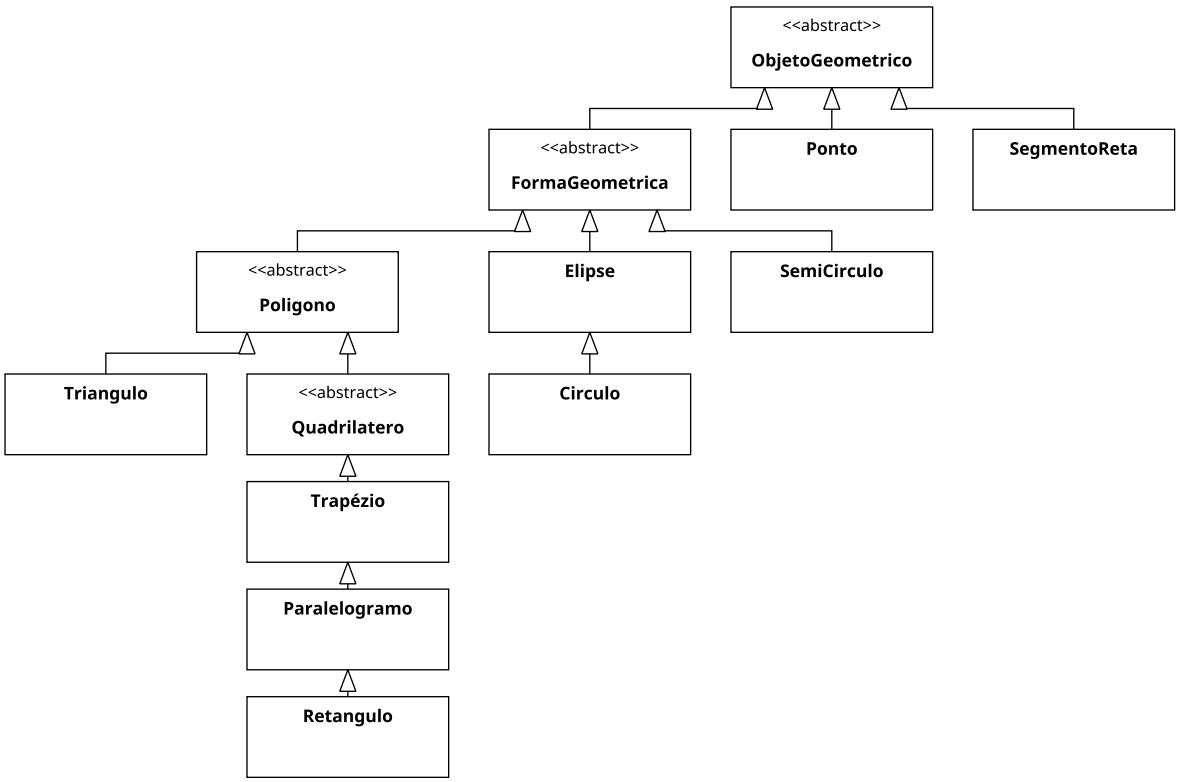



Figura 1: Diagrama de classes UML da hierarquia de objetos geométricos.

**Objetivo** Construir uma biblioteca de objetos geométricos utilizando os conceitos e técnicas de herança.

**Hierarquia de objetos** . A figura 1 representa todas as classes que devem ser definidas, bem como as relação de ascendência entre as classes. Por questão de espaço, os métodos não estão representados no diagrama, mas são detalhados a seguir. As propriedades de cada classe devem ser definidas de acordo com as especificações fornecidas.

**Modelo geométrico simplificado** Todos os objetos geométricos seguem a mesma orientação que é ilustrada nos exemplos, ou seja, não há rotações diferentes das exibidas nas figuras. As exceções são `SegmentoReta` e `Circulo`, que podem sofrer rotações livremente. Em relação aos polígonos, quando um dos lados do polígono está na horizontal, significa que esse lado é paralelo ao eixo x; analogamente, se um dos lados está na vertical, significa que o lado é paralelo ao eixo y. Além disso, a ordenação dos pontos/vértices será sempre aquela que é documentada nos exemplos. Como referência para determinar alturas e larguras, considere as definições fornecidas pelos exemplos.

**Imutabilidade** Todos objetos devem ser imutáveis:

- Não devem possuir *setters* públicos para quaisquer propriedades;
- As propriedades devem ser `final`.

**Características comuns a todas as classes.** É importante garantir todas as condições para que o programa funcione corretamente:

- Todas as classes devem estar no pacote `main`;
- Todas as propriedades devem ser `private` ou `protected` (não definir propriedades como públicas);
- Todas as propriedades devem possuir método acessor (*getter*).

**Restrições das classes** Na seção "Detalhamento das APIs" são documentadas as restrições esperadas para cada elemento. Em particular, especifica-se:

i) Simplificações no respectivo conceito matemático (para facilitar a implementação);

ii) Se a classe é abstrata ou concreta;

iii) Para cada classe, os métodos de sua API.

Algumas implicações:

i) Os objetos geométricos possuem restrições nos seus valores:

- Todos os pontos devem ter coordenadas no intervalo real $[0, 1]$;
- Todas as medidas passadas como parâmetros dos métodos devem ser positivas.

As violações nas restrições de valores dos objetos devem ser sinalizadas via `IllegalArgumentException`;

ii) Você deve determinar o modificador de acesso mais adequado para cada método;

iii) Você deve determinar, com base nas assinaturas dos construtores, os atributos/propriedades das classes;

iv) Você deve determinar os modificadores de acesso ou de extensibilidade (i.e. `private`, `public`, `abstract`, `final`, etc.) mais adequados para cada método e cada propriedade.

v) É possível que a classe implemente mais métodos dos que os listados, i.e. quando deve implementar métodos da classe ascendente ou das suas interfaces.

**Reúso do código em hierarquias** Uma das motivações para definir hierarquias de classes é justamente melhorar o reúso de código, por razões que passam por legibilidade, desempenho, consistência, entre outras. Portanto, se as implementações dos métodos na sua hierarquia estão repetitivos, com vários trechos duplicados, a qualidade do código está baixa. O mesmo vale para a definição de propriedades das classes.

Possíveis problemas que merecem sua atenção:

i) Propriedades duplicadas nas classes mãe e filhas;

ii) Implementação de métodos praticamente iguais nas classes mãe e filhas (nesse caso, as filhas devem delegar parte do trabalho para a mãe);

iii) Classes irmãs com implementação de métodos praticamente iguais (nesse caso, esses métodos provavelmente precisam ser generalizados para a mãe);

iv) Métodos da classe mãe tomando decisões com base nos tipos das classes filhas. Isso é uma violação grave do princípio da extensibilidade[1], visto que uma classe mãe não deveria nem mesmo saber quem são suas filhas.

## Detalhamento das APIs

- **`ObjetoGeometrico`**. Classe abstrata.

  Implementação já está pronta.

- **`Ponto`**. Classe concreta.

  Implementação já está pronta.

- **`SegmentoReta`**. Classe concreta. Objeto formado pela ligação entre dois pontos.
  - **Construtor**:

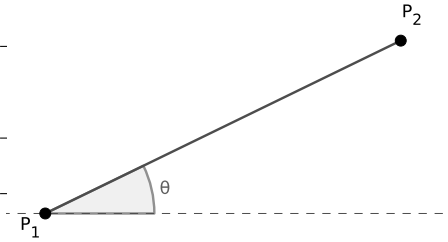
    ```
    SegmentoReta(Ponto p1, Ponto p2)
    ```

  - **Comprimento**:

    ```
    double comprimento()
    ```

  - **Coeficiente angular**:

    ```
    double coeficienteAngular()
    ```

    O coeficiente angular[a] mede a inclinação da reta em relação ao eixo x. Caso a reta seja paralela ao eixo y (coeficiente indefinido), retorna `Double.POSITIVE_INFINITY`.

    [a] `https://pt.wikipedia.org/wiki/Declive`

  - **Teste de paralelismo**

    ```
    boolean paralelo(SegmentoReta s)
    ```

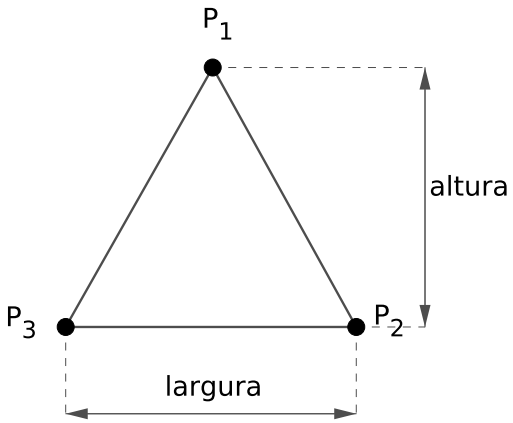
[1] `https://en.wikipedia.org/wiki/Extensibility`

Verifica se a linha onde passa o segmento de reta atual é paralela à linha onde passa o segmento de reta `s`.

- **`FormaGeometrica`**. Classe abstrata.
  - **Largura**:
    ```
    abstract double largura()
    ```
  - **Área**:
    ```
    abstract double area()
    ```
  - **Altura**:
    ```
    abstract double altura()
    ```
  - **Perímetro**:
    ```
    abstract double perimetro()
    ```

- **`Polígono`**. Classe abstrata.
  - **Construtor**:
    ```
    Poligono(Ponto[] pontos)
    ```

- **`Triangulo`**. Classe concreta.
  - **Construtor**:
    ```
    Triangulo(Ponto p1, Ponto p2,
              Ponto p3)
    ```




- **`Quadrilatero`**. Classe abstrata.
  - **Construtor**:
    ```
    public Quadrilatero(Ponto p1, Ponto p2, Ponto p3, Ponto p4)
    ```

- **`Trapezio`**. Classe concreta.

  Forma geométrica com ao menos um par de lados paralelos. Cada par é formado por duas bases, considere como bases os segmentos $\overline{P_1P2}$ e $\overline{P_3P_4}$, i.e. esses segmentos sempre serão paralelos. A base menor pode ser qualquer dos dois segmentos, idem para a base maior. Caso as bases tenham o mesmo comprimento, considerar $\overline{P_1P2}$ como base menor e $\overline{P_3P_4}$ como base maior.

– **Construtor**

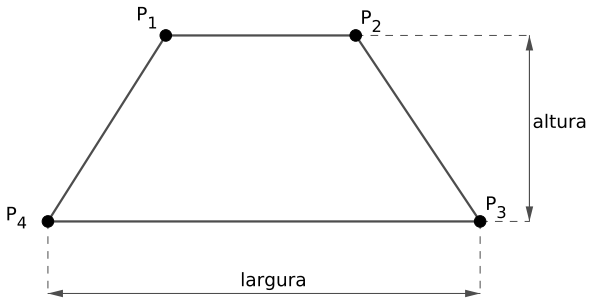
```
Trapezio(Ponto p1, Ponto p2, Ponto
   p3, Ponto p4)
```

– **Base menor**. Retorna o segmento de reta formado pelos pontos da base menor.

```
SegmentoReta baseMenor()
```

– **Base maior**. Retorna o segmento de reta formado pelos pontos da base maior.

```
SegmentoReta baseMaior()
```




– **Teste de existência**. Verifica se é possível existir trapézio com os pontos dados.

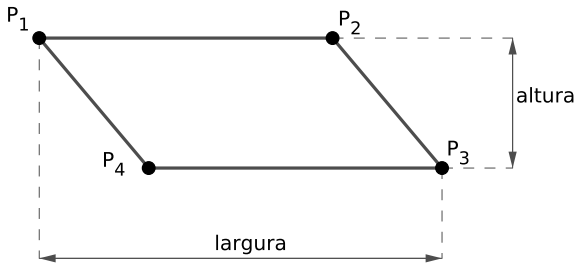
```
static boolean existe(Ponto p1,
   Ponto p2, Ponto p3, Ponto p4)
```

- **`Paralelogramo`**. Classe concreta.

  Forma geométrica com dois pares de lados paralelos.

  – **Construtor:**

```
public Paralelogramo(Ponto p1,
                     Ponto p2,
                     Ponto p3,
                     Ponto p4)
```




  – **Teste de existência:**

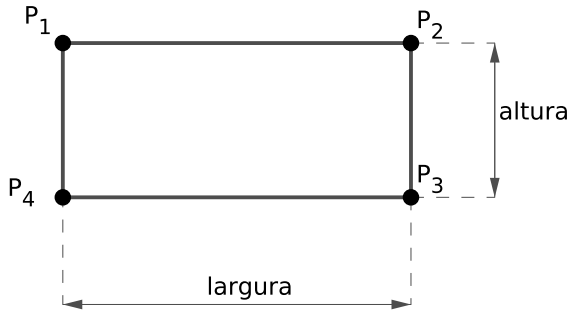
  Verifica se os pontos formam um paralelogramo.

```
static boolean existe(Ponto p1,
                      Ponto p2,
                      Ponto p3,
                      Ponto p4)
```

- **`Retangulo`**. Classe concreta.

– **Construtor**:

```
Retangulo(Ponto p1,
          Ponto p2,
          Ponto p3,
          Ponto p4)
```




– **Testa se é um quadrado**:

```
boolean quadrado()
```

– **Teste de existencia**:

```
static boolean existe(Ponto p1,
                      Ponto p2,
                      Ponto p3,
                      Ponto p4)
```

- **`Elipse`**. Classe concreta.

  Considere que a largura é sempre maior ou igual que a altura.

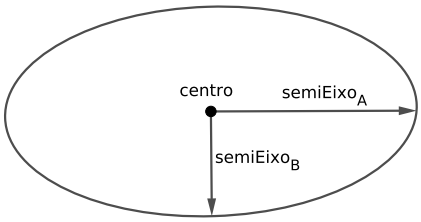
  – **Construtor**:

  ```
  Elipse(Ponto centro, double semiEixoA,
     double semiEixoB)
  ```

  Não há ordem pré-estabelecida entre os semi-eixos passados como argumento, i.e., qual representa o menor e qual representa o maior deve ser determinado internamente no objeto.

  – **Semi-eixo menor**:

  ```
  double getSemiEixoMenor()
  ```

  Retorna o valor do semi-eixo menor.

  – **Semi-eixo maior**:

  ```
  double getSemiEixoMaior()
  ```

  Retorna o valor do semi-eixo maior.

  – **Circunferência**:

  ```
  double circunferencia()
  ```

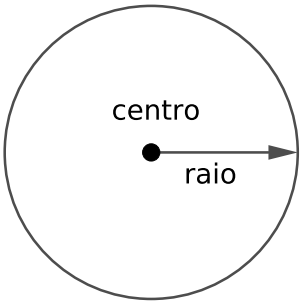
  Sinônimo de perímetro.

  – Para uma *aproximação* da circunferência da elipse, utilize a fórmula $2\pi\sqrt{\frac{a^2+b^2}{2}}$, onde $a$ e $b$ são os semi-eixos.

- **`Circulo`**. Classe concreta.

– **Construtor**:

```
Circulo(Ponto centro, double raio)
```




– **Circunferência**:

```
double circunferencia()
```

Nota: não usar a mesma fórmula (aproximada) da elipse, pois aqui temos uma fórmula exata.

- **`SemiCirculo`**. Classe concreta.

– **Construtor**:

```
SemiCirculo(Ponto centro,
            double raio)
```