Classes aninhadas (*Nested Classes*)

A nomenclatura correta para os tipos de classes aninhadas em Java:

- **Classe aninhada estática** (*static nestes class*): são usadas sem uma instância da classe externa e apenas acessam membros estáticos de sua externa. O objetivo é obter um encapsulamento da estrutura representada.
- **Classe interna** não-estática: são classes aninhadas (*inner class*) que acessam todos os membros de sua externa.
- **Classe interna local**: definidas dentro de métodos ou blocos (*local inter class*), podem apenas ser acessadas dentro do método/bloco com acesso à dados da classe externa.
- **Classe interna anônima** (*anonymous inner class*): não têm nome e são criadas para extender uma superclasse ou interface, para um único uso (única instância) local.

O quadro abaixo ilustra o acesso à classe externa (*outer instante*) e o usos comuns para cada tipo:

Type	Static	Access Outer Instance?	Common Use
1. Static Nested Class	✅ Yes	❌ No	Utility/helper classes
2. Inner Class	❌ No	✅ Yes	Encapsulation
3. Local Class	❌ No	✅ Yes (if final/implicitly final)	Short-lived logic
4. Anonymous Class	❌ No	✅ Yes	One-time use, inline logic

Exercícios:

1. Observe o código abaixo:
   a. Sem implementar esse código, anote as saídas que você supõe para: `x`, `this.x` e `ShadowTest.this.x`.
   b. Em seguida, verifique sua resposta, codificando a classe.
   c. Qual é o tipo desse aninhamento de classes?

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

2. Suponha um sistema de *e-commerce* simples. Use uma classe interna não estática (Item) para representar os itens em um Pedido. O Item tem acesso aos detalhes do Pedido (como o ID do pedido), o que reforça o relacionamento de que um item pertence a um pedido específico.

3. Imagine um utilitário de conversão de unidades. Nesse caso, a classe externa será a *MeasurementUtility* a qual contem uma classe aninhada estática, a *Conversion*, para agrupar funcionalidades relacionadas à conversão de unidades de medidas. As funcionalidades são tais como: *meterstoCentimeters*(), c*entimeterstoMeters*(), *millimeterstoNanometers*(), etc.. Implemente uma versão da MeasurementUtility em Java, com as funcionalidade e tipos de conversões que desejar.

4. Pensando na API Java, tente absorver a hierarquia de classes sob a classe `java.awt.geom.Point2D` .Desenho o diagrama UML dessa solução e busca justificar a implementação utilizada.

5. Entenda o papel da `Interface Comparator<T>` da API JavaSE. Ela serve para realizar a comparação de objetos de tipos de classes (não naturalmente ordenáveis) como a classe `Pedido` da questão 2. Proponha um exemplo de código de classe anônima que implemente `Comparator<T>` e seja gerada na chamada (um dos argumentos) de um dos métodos: Collections.sort() ou Arrays.sort(). Você pode testar com qualquer classe.