POO

Classes e objetos

Prof. Alcides Calsavara
PUCPR

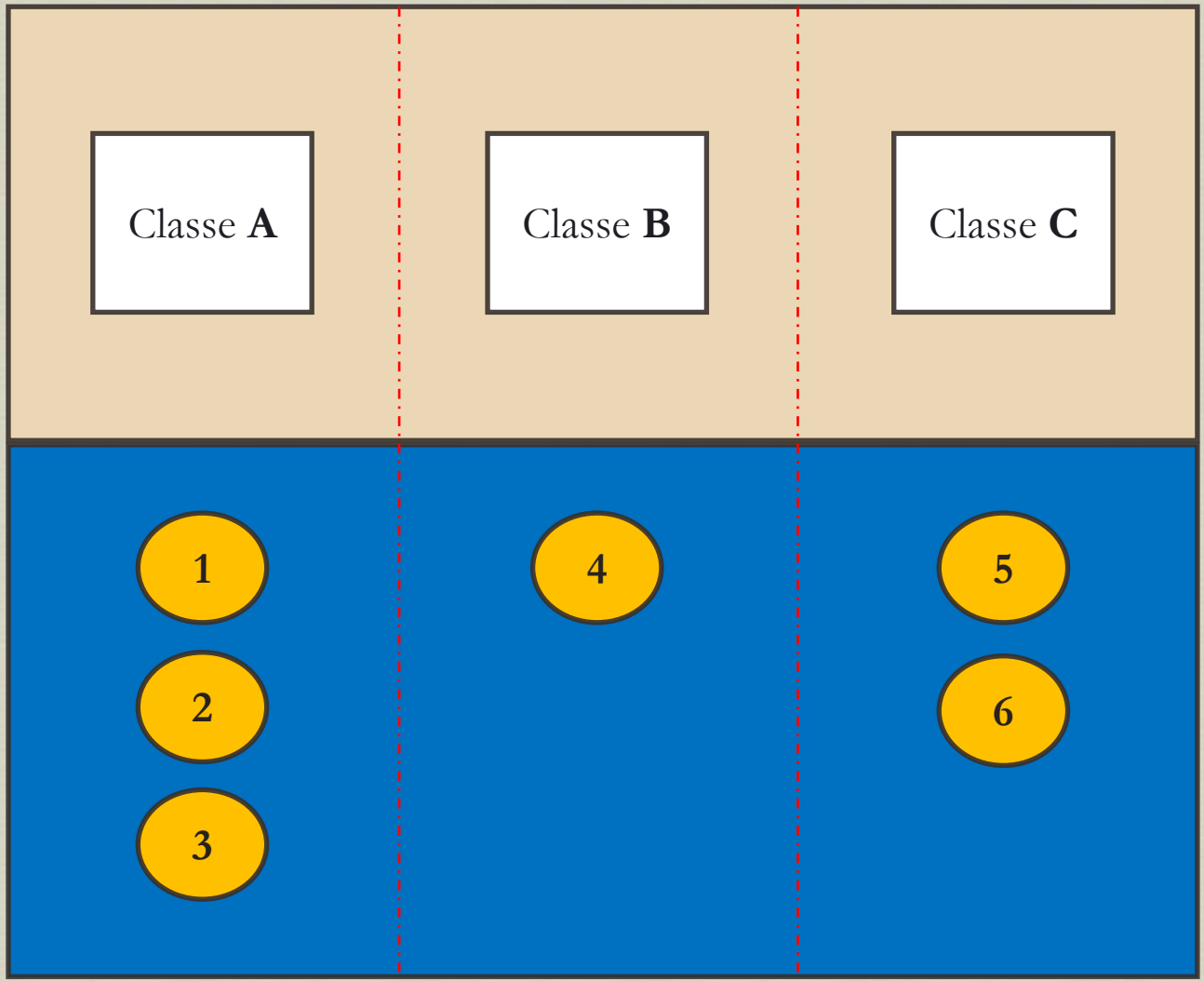
Conceitos

1. Objeto (instância de uma classe)
2. Instanciação (criação de um objeto)
3. Membros (atributos e métodos) de objetos
4. Estado de um objeto
5. Referência a um objeto
6. Chamada de método de um objeto

Classes e objetos

- O **código** de uma aplicação orientada a objetos consiste em um conjunto de **classes**.
- A **execução** de uma aplicação orientada a objetos é composta por um conjunto de **objetos** que interagem entre si.
- Um **objeto** é uma **instância** de uma **classe**.
- Uma **classe** define a estrutura e o comportamento de um tipo de objeto.

APLICAÇÃO ORIENTADA A OBJETOS



Composição de uma classe

Uma **classe** é composta por:

A. Um conjunto de **atributos**: define a estrutura dos objetos.

B. Um conjunto de **métodos**: define o comportamento dos objetos.

Método: uma "função" que pode ser aplicada a todo objeto da classe.

Atributo: uma "variável" definida no escopo da classe.

Um atributo pode ser acessado por **todos** os métodos da classe.

Cada objeto da classe possui o seu próprio valor para um atributo.

Exemplo

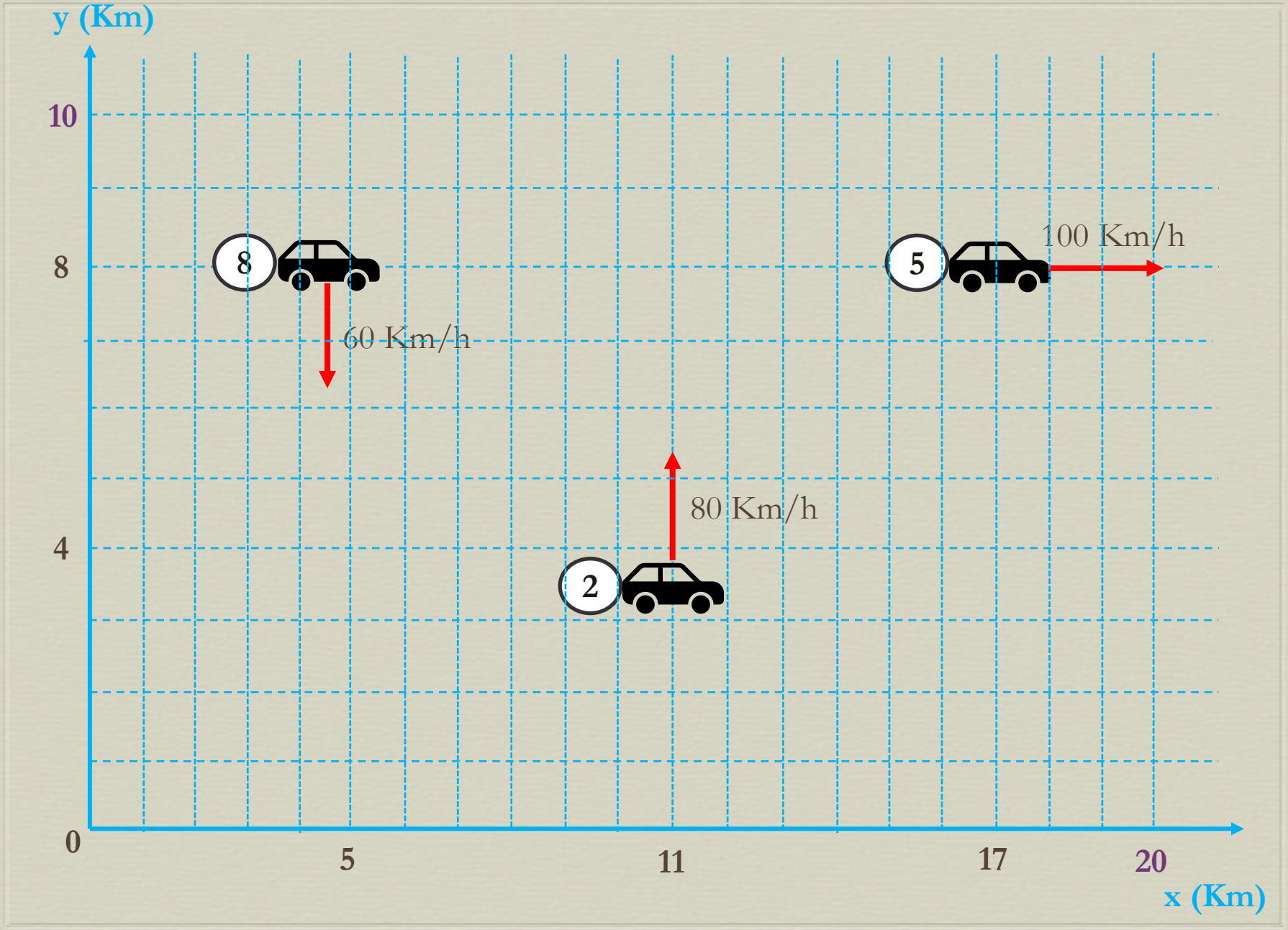
Um aplicativo de mobilidade que disponibiliza um conjunto de carros para fazer o deslocamento de pessoas.

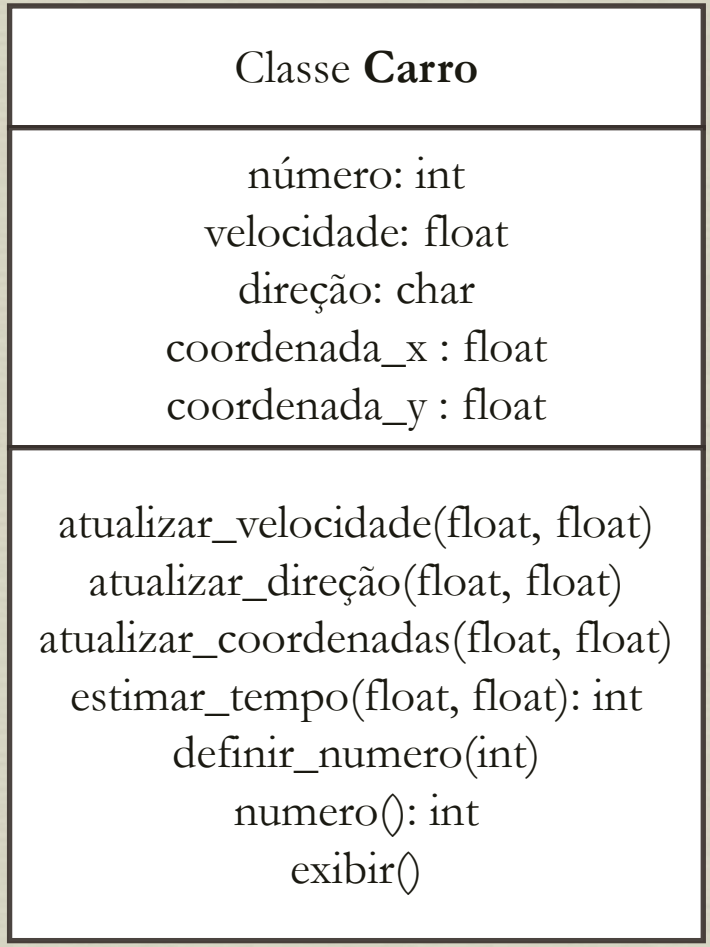
O aplicativo mantém os seguintes dados sobre cada carro:

a. identificação única: um número inteiro
b. posição no mapa: coordenadas (x, y)
c. velocidade: em Km/h
d. direção de movimento: Norte, Sul, Leste ou Oeste

As operações aplicáveis a um carro são:

a. atualizar as coordenadas
b. atualizar a velocidade
c. atualizar a direção do movimento (Norte, Sul, Leste, Oeste)
d. estimar o tempo para chegar em certo ponto do mapa


y (Km)
10
8
4
0
5
11
17
20
x (Km)
8
60 Km/h
5
100 Km/h
2
80 Km/h



ATRIBUTOS

MÉTODOS

:Carro
número: 5
velocidade: 100.0
direção: 'L'
coordenada_x: 17.0
coordenada_y: 8.0

:Carro
número: 8
velocidade: 60.0
direção: 'S'
coordenada_x: 4.5
coordenada_y: 8.0

:Carro
número: 2
velocidade: 80.0
direção: 'N'
coordenada_x: 11.0
coordenada_y: 3.5

Implementação de classes e objetos em Java

Qualificadores de Visibilidade e Proteção de Membros

- Usados para definir a visibilidade de cada membro (atributo ou método) de uma classe, isto é, para estabelecer o nível de acesso aos membros de uma classe.

- Visibilidade e qualificadores em Java:

  1. **público - `public` => permite acesso a partir de qualquer classe**
  2. **privado - `private` => acesso restrito à própria classe**
  3. **pacote** - `package`
  4. **protegido - `protected`**

Princípio de Encapsulamento

- Uma classe deve ocultar os seus detalhes de implementação das classes que a utilizam.
- Uma classe deve especificar quais dos seus membros (atributos e métodos) são visíveis e acessíveis por outras classes.
  - **atributos** devem ser de acesso restrito pela própria classe
  - **métodos** podem ser de acesso restrito pela própria classe ou de acesso livre por outras classes
- O conjunto de métodos de uma classe especificados como acessíveis por outras classes constitui a ***interface*** dessa classe.

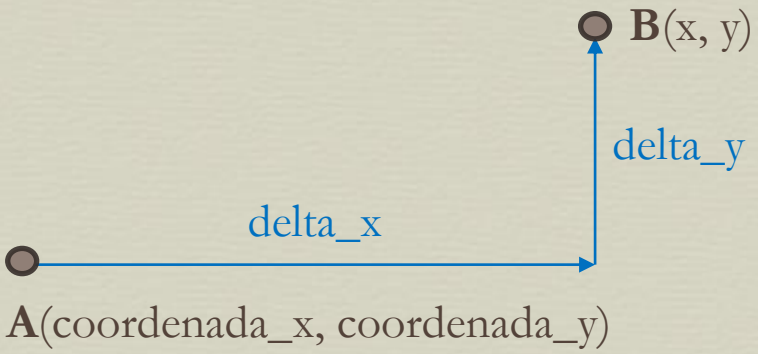
```
public class Carro {

    private int numero;
    private float velocidade;
    private char direcao;
    private float coordenada_x;
    private float coordenada_y;

    private void atualizar_velocidade(float x, float y) { ... }
    private void atualizar_direcao(float x, float y) { ... }

    public void atualizar_coordenadas(float x, float y) { ... }
    public int estimar_tempo(float x, float y) { ... }
    public void definir_numero(int n) { ... }
    public int numero() { ... }
    public void exibir() { ... }

}
```

ATRIBUTOS

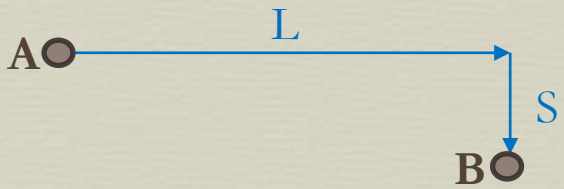
MÉTODOS

INTERFACE

```
private void atualizar_velocidade(float x, float y)
{
   float delta_x = Math.abs(x - coordenada_x);
   float delta_y = Math.abs(y - coordenada_y);
   float distancia_percorrida = delta_x + delta_y;
   velocidade = (distancia_percorrida / INTERVALO_PADRAO) * 3600;
   // 1 hora == 3600 segundos
}
```

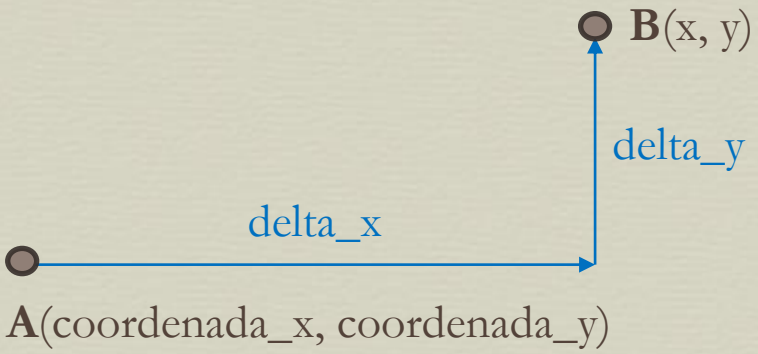


```
private void atualizar_direcao(float x, float y)
{
   switch( direcao )
   {
      case 'L', 'O':
         if (y < coordenada_y) direcao = 'S';
         else if (y > coordenada_y) direcao = 'N';
         break;
      case 'N', 'S':
         if (x < coordenada_x) direcao = 'O';
         else if (x > coordenada_x) direcao = 'L';
   }
}
```



```
public void atualizar_coordenadas(float x, float y)
{
    atualizar_velocidade(x, y);
    atualizar_direcao(x, y);
    coordenada_x = x;
    coordenada_y = y;
}
```

```
public int estimar_tempo(float x, float y)
{
   int tempo_estimado = 0; // em segundos
   float delta_x = Math.abs(x - coordenada_x);
   float delta_y = Math.abs(y - coordenada_y);
   float distancia_a_percorrer = delta_x + delta_y;
   tempo_estimado = (int) (distancia_a_percorrer / velocidade * 3600);
   return tempo_estimado;
}
```



```
public void definir_numero(int n)
{
    numero = n;
}

public int numero()
{
    return numero;
}
```

```
public void exibir()
{
   System.out.println("Carro " + numero + ":");

   System.out.println(String.format("Coordenadas atuais: (%.2f; %.2f)",
                              coordenada_x, coordenada_y));

   System.out.println(String.format("Velocidade: %.2f Km/h",
                              velocidade));

   System.out.println("Direção: " + direcao);

   System.out.println("----------------------------------------");
}
```

Simulação do aplicativo

```
Crie um conjunto de carros

Repita para sempre:

   1. Para cada carro:
         A. Escolha um par de coordenadas (x,y) aleatoriamente
         B. Atualize as coordenadas do carro para (x,y)

   2. Escolha um destino aleatório (x,y)

   3. Para cada carro:
         Estime o tempo para chegar ao destino (x,y)

   4. Aguarde um tempo padrão
```

```
public class Xuber {
    private static float MAX_X = 20;
    private static float MAX_Y = 10;
    private static final int INTERVALO_PADRAO = 1200;
    // tempo em segundos entre atualizações da posição
    private static final int FATOR_SIMULACAO = 60;
    // redução na escala do tempo de simulação

    public static void main(String[] args) { ... }

    private static float x() { ... }

    private static float y() { ... }

    private static void aguardar_intervalo_padrao() { ... }
}
```

```
public static void main(String[] args)
{
    Carro carro_A = new Carro();
    Carro carro_B = new Carro();
    Carro carro_C = new Carro();

    carro_A.definir_numero(5);
    carro_B.definir_numero(8);
    carro_C.definir_numero(2);

    ...
```

```
public static void main(String[] args)
{
   ...
   int iteracao = 1;
   do {
      System.out.println("\n========================================== ITERAÇÃO " + iteracao);

      carro_A.atualizar_coordenadas( x(), y() );
      carro_B.atualizar_coordenadas( x(), y() );
      carro_C.atualizar_coordenadas( x(), y() );

      carro_A.exibir();
      carro_B.exibir();
      carro_C.exibir();

      float destino_x = x();
      float destino_y = y();
      System.out.println(String.format("Tempo em segundos para chegar no destino (%.2f; %.2f)",
            destino_x, destino_y));
      System.out.println("Carro " + carro_A.numero() + ": " +
            carro_A.estimar_tempo(destino_x, destino_y));
      System.out.println("Carro " + carro_B.numero() + ": " +
            carro_B.estimar_tempo(destino_x, destino_y));
      System.out.println("Carro " + carro_C.numero() + ": " +
            carro_C.estimar_tempo(destino_x, destino_y));

      aguardar_intervalo_padrao();
      iteracao++;

   } while (true);
}
```

Atualizar as coordenadas de cada carro

Estimar o tempo para cada carro chegar num certo destino

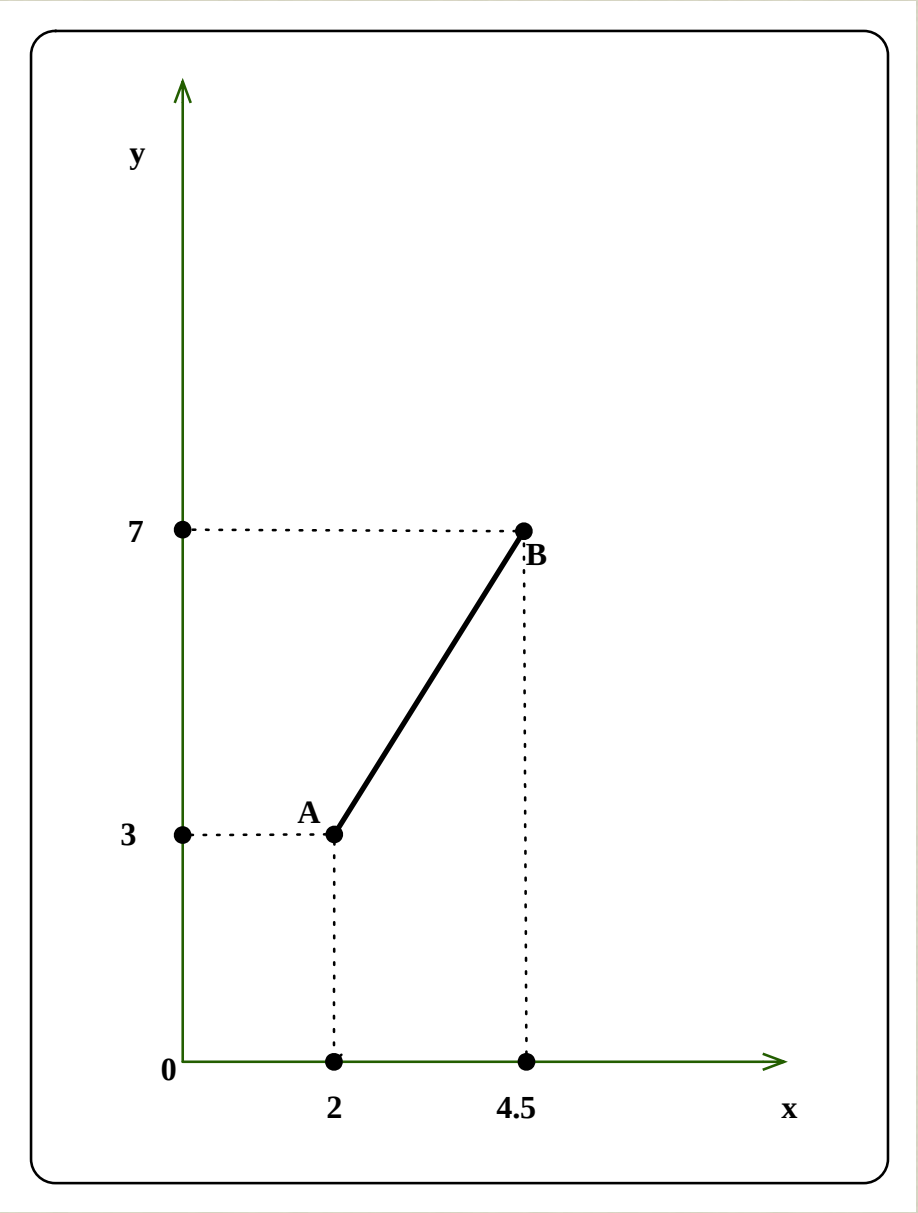
Aguardar um tempo até a próxima iteração

```
private static float x()
{
   double aleatorio = Math.random(); // valor entre 0 e 1
   float valor_x = (float) aleatorio * MAX_X;
   return valor_x;
}

private static float y()
{
   double aleatorio = Math.random(); // valor entre 0 e 1
   float valor_y = (float) aleatorio * MAX_Y;
   return valor_y;
}
```

```
private static void aguardar_intervalo_padrao()
{
  try { Thread.sleep(INTERVALO_PADRAO * 1000 / FATOR_SIMULACAO); }
  catch (Exception e) { }
}
```

Exemplo:
Elementos Geométricos

Uma classe pode possuir um atributo que seja uma referência para um objeto de alguma classe. Por exemplo, um segmento de reta é definido por dois pontos no plano. Na figura abaixo, está desenhado um segmento de reta entre os pontos **A** e **B**, sendo **A** com coordenadas (2, 3) e **B** com coordenadas (4.5, 7).


y
7
B
A
3
0
2
4.5
x

Os elementos geométricos dessa figura podem ser representados por objetos instanciados a partir das classes **Ponto** e **Segmento_de_Reta**, descritas a seguir. Note que a classe **Ponto** possui os atributos **x** e **y** do tipo **double** para representar as suas coordenadas, enquanto a classe **Segmento_de_Reta** possui os atributos **p1** e **p2** do tipo **Ponto** para representar os dois pontos que definem o segmento de reta. Note, também, que o método construtor da classe **Segmento_de_Reta** recebe, como parâmetros, duas referências para objetos da classe **Ponto**. De fato, qualquer método de uma classe (e, não apenas o construtor) pode ter um parâmetro cujo tipo é uma classe.

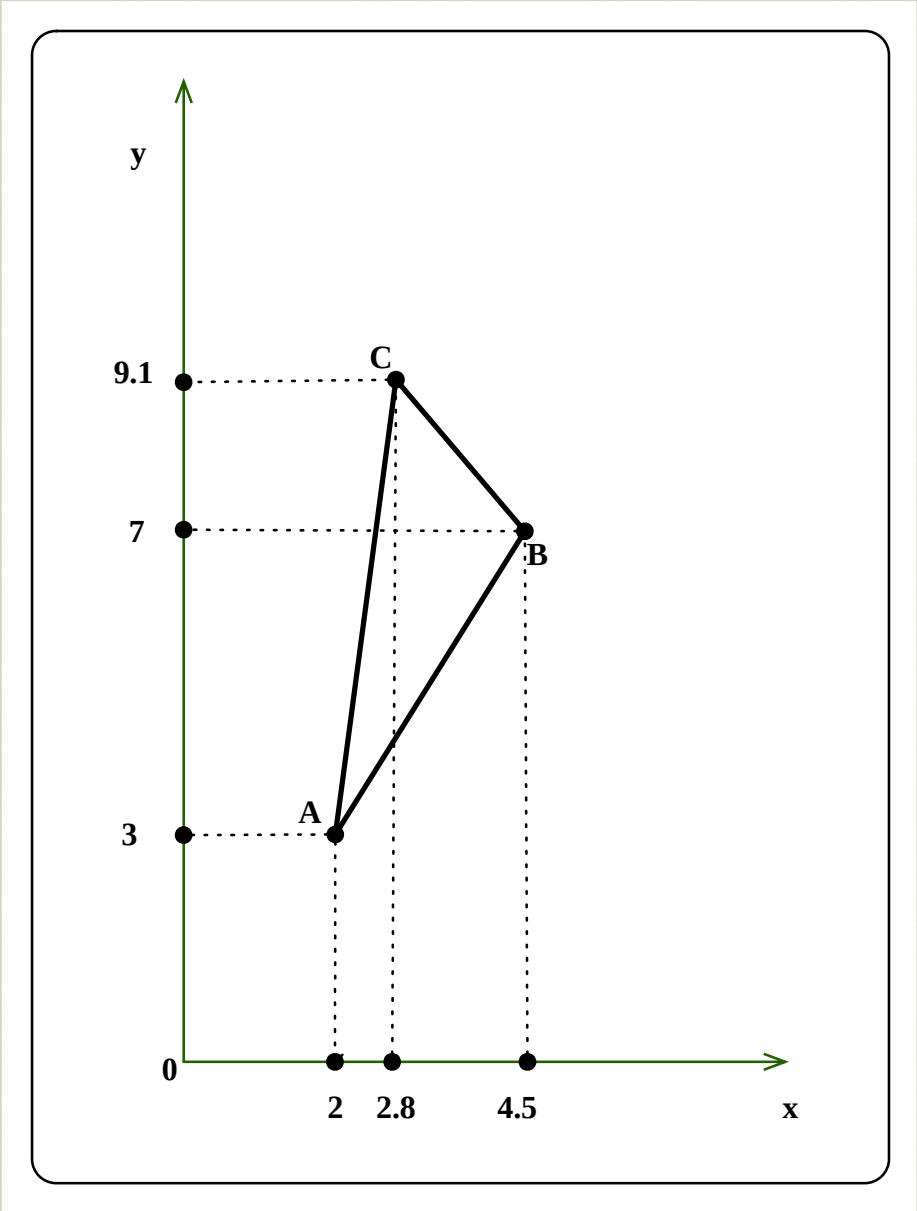
```java
public class Ponto {
    private double x;
    private double y;
    public Ponto(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public String texto() {
        String t = "(" + x + ", " + y + ")";
        return t;
    }
}
```

```
public class Segmento_de_Reta {
   private Ponto p1;
   private Ponto p2;
   public Segmento_de_Reta(Ponto p1, Ponto p2) {
      this.p1 = p1;
      this.p2 = p2;
   }
   public void desenhe() {
      System.out.println("Segmento de reta entre " +
                  p1.texto() + " e " + p2.texto());
      // aqui deve ser usado um pacote gráfico para
      // desenhar o segmento de reta
   }
}
```

Assim, é possível escrever o código abaixo da classe **DesenhaReta** que, quando executado, cria dois objetos para representar os pontos **A** e **B** e, em seguida, cria um objeto para representar o segmento de reta **r** entre esses dois pontos e, finalmente, invoca o método **desenhe** desse segmento de reta. (Note que o método **desenhe** não gera uma representação gráfica; simplesmente, imprime uma mensagem equivalente. Para que o exemplo fique completo, é preciso utilizar algum pacote gráfico, por exemplo Swing, para desenhar objetos gráficos.)

```
public class DesenhaReta {
   public static void main(String args[]) {
      Ponto A = new Ponto( 2, 3 );
      Ponto B = new Ponto( 4.5, 7 );
      Segmento_de_Reta r =
                     new Segmento_de_Reta(A, B);
      r.desenhe();
   }
}
```

É possível desenhar muitas figuras geométricas com o uso de segmentos de reta. Por exemplo, um triângulo, que é definido por três pontos **A**, **B** e **C**, é delimitado por três segmentos de reta: um segmento de **A** a **B**, outro de **B** a **C** e, finalmente, outro de **C** a **A**, conforme mostra a figura abaixo.


y
9.1
C
7
B
A
3
0
2
2.8
4.5
x

Dessa forma, usando-se a classe **Segmento_de_Reta** para o desenho de um triângulo, pode ser criada a classe **Triangulo**, descrita a seguir.

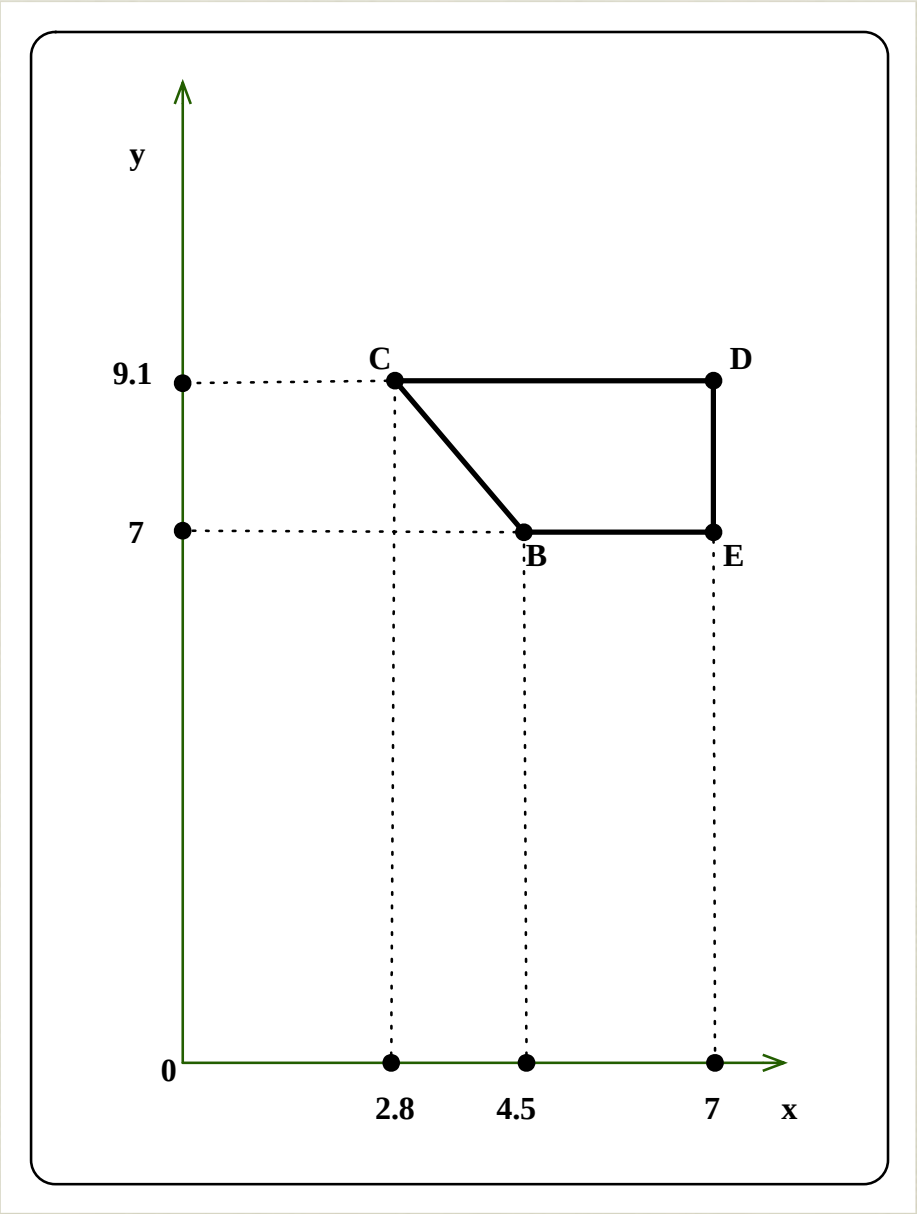
```
public class Triangulo {
   private Ponto p1;
   private Ponto p2;
   private Ponto p3;
   public Triangulo(Ponto p1, Ponto p2, Ponto p3) {
      this.p1 = p1;
      this.p2 = p2;
      this.p3 = p3;
   }
   public void desenhe() {
      Segmento_de_Reta r1 = new Segmento_de_Reta( p1, p2 );
      Segmento_de_Reta r2 = new Segmento_de_Reta( p2, p3 );
      Segmento_de_Reta r3 = new Segmento_de_Reta( p3, p1 );
      r1.desenhe();
      r2.desenhe();
      r3.desenhe();
   }
}
```

Note que a classe **Triangulo** possui os atributos **p1**, **p2** e **p3** do tipo **Ponto** para representar os três pontos que definem um triângulo. O método **desenhe** da classe **Triangulo** cria os objetos referenciados por **r1**, **r2** e **r3** que representam os três segmentos de reta que delimitam o triângulo, de acordo com os pontos **p1**, **p2** e **p3**. Cada um desses segmentos desenha a si próprio quando o seu método **desenhe** é chamado. Note, ainda, que as referências **r1**, **r2** e **r3** para objetos da classe **Segmento_de_Reta** são variáveis locais do método **desenhe**.

Assim, é possível escrever o código abaixo da classe **DesenhaTriangulo** que, quando executado, cria um objeto da classe **Triangulo** referenciado por **t** para representar um triângulo com vértices nos pontos **A**, **B** e **C** e, em seguida, invoca o método **desenhe** desse triângulo que, então, desenha os três segmentos de reta correspondentes.

```
public class DesenhaTriangulo {
   public static void main(String args[]) {
      Ponto A = new Ponto( 2, 3 );
      Ponto B = new Ponto( 4.5, 7 );
      Ponto C = new Ponto( 2.8, 9.1 );
      Triangulo t = new Triangulo( A, B, C );
      t.desenhe();
   }
}
```

De forma similar, também é possível desenhar um quadrilátero desenhando-se os quatros segmentos de reta que o delimitam. Na figura abaixo, está desenhado um quadrilátero (neste caso, um trapézio) definido pelos pontos **B**, **C**, **D** e **E**. Note que há quatro segmentos de reta: um de **B** a **C**, outro de **C** a **D**, outro de **D** a **E** e, finalmente, outro de **E** a **B**.


y
9.1
7
0
C
D
B
E
2.8
4.5
7
x

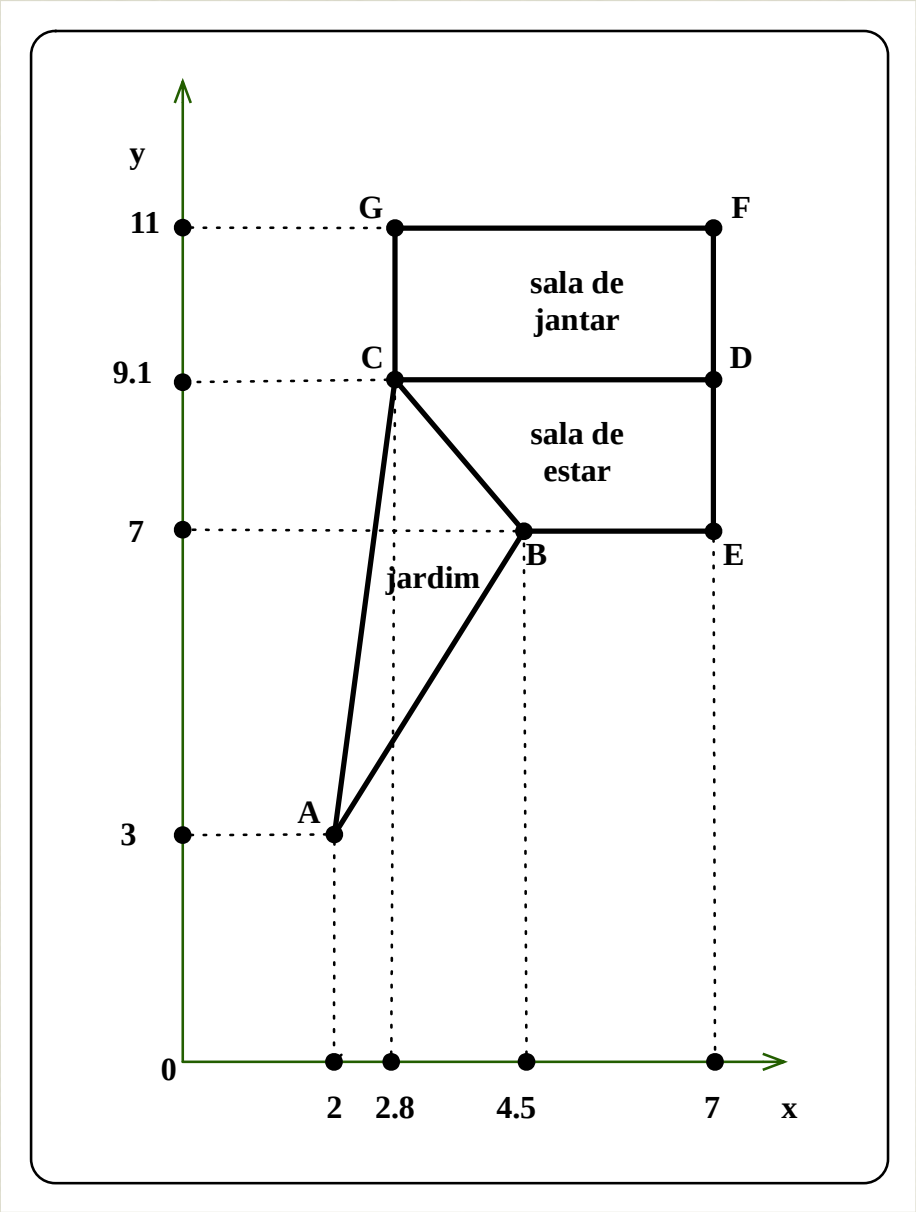
Um quadrilátero pode ser representado por meio de um objeto da classe **Quadrilatero**, descrita abaixo. Note que, comparado com a classe **Triangulo**, a única diferença significativa na classe **Quadrilatero** é a inclusão de um quarto ponto (**p4**) como atributo.

```
public class Quadrilatero {
   private Ponto p1;
   private Ponto p2;
   private Ponto p3;
   private Ponto p4;
   public Quadrilatero(Ponto p1, Ponto p2, Ponto p3, Ponto p4) {
      this.p1 = p1;
      this.p2 = p2;
      this.p3 = p3;
      this.p4 = p4;
   }
   public void desenhe() {
      Segmento_de_Reta r1 = new Segmento_de_Reta(p1, p2);
      Segmento_de_Reta r2 = new Segmento_de_Reta(p2, p3);
      Segmento_de_Reta r3 = new Segmento_de_Reta(p3, p4);
      Segmento_de_Reta r4 = new Segmento_de_Reta(p4, p1);
      r1.desenhe();
      r2.desenhe();
      r3.desenhe();
      r4.desenhe();
   }
}
```

O código abaixo da classe **DesenhaQuadrilatero** cria um objeto **q** para representar o quadrilátero mostrado na figura e, em seguida, invoca o método **desenhe** para esse objeto.

```
public class DesenhaQuadrilatero {
   public static void main(String args[]) {
      Ponto B = new Ponto( 4.5, 7 );
      Ponto C = new Ponto( 2.8, 9.1 );
      Ponto D = new Ponto(7, 9.1);
      Ponto E = new Ponto( 7, 7);
      Quadrilatero q = new Quadrilatero( B, C, D, E );
      q.desenhe();
   }
}
```

Agora, vamos supor que um projetista de plantas de imóveis residenciais queira desenhar as peças de uma casa, como a ilustrada na figura abaixo, que possui uma sala de estar em forma retangular (um quadrilátero), uma sala de jantar em forma trapezoidal (outro quadrilátero) e um jardim triangular.


y
11
9.1
7
3
0
G
F
C
D
B
E
A
sala de
jantar
sala de
estar
jardim
2
2.8
4.5
7
x

A representação desse projeto pode ser feito por meio de objetos das classes **Triangulo** e **Quadrilatero**: um objeto da classe **Triangulo** para representar o jardim, um objeto da classe **Quadrilatero** para representar a sala de estar e outro objeto da classe **Quadrilatero** para representar a sala de jantar. O código da classe **ProjetistaJunior** descrita abaixo cria esses objetos, intanciando-os diretamente das classes **Triangulo** e **Quadrilatero**.

```
public class ProjetistaJunior {
   public static void main(String args[]) {
      Ponto A = new Ponto( 2, 3 );
      Ponto B = new Ponto( 4.5, 7 );
      Ponto C = new Ponto( 2.8, 9.1 );
      Triangulo jardim = new Triangulo( A, B, C );
      jardim.desenhe();

      Ponto D = new Ponto(7, 9.1);
      Ponto E = new Ponto(7, 7);
      Quadrilatero sala_estar = new Quadrilatero( B, C, D, E );
      sala_estar.desenhe();

      Ponto F = new Ponto(7, 11);
      Ponto G = new Ponto(2.8, 11);
      Quadrilatero sala_jantar = new Quadrilatero( C, D, F, G );
      sala_jantar.desenhe();
   }
}
```

Outra maneira de organizar esse código é fazer com o que os objetos das classes **Triangulo** e **Quadrilatero** sejam criados por um intermediário, responsável por gerenciar essa criação. Assim, o código do projetista passa a fazer uso do intermediário toda vez que quiser criar um novo objeto. Tal intermediário pode ser implementado por meio da classe **Gerador_de_Poligonos**, descrita a seguir.

```
public class Gerador_de_Poligonos {
   public Triangulo gere_triangulo(Ponto p1, Ponto p2, Ponto p3) {
      Triangulo t = new Triangulo(p1, p2, p3);
      return t;
   }
   public Quadrilatero gere_quadrilatero(Ponto p1, Ponto p2,
                                         Ponto p3, Ponto p4) {
      Quadrilatero q = new Quadrilatero(p1, p2, p3, p4);
      return q;
   }
}
```

A classe **Gerador_de_Poligonos** possui um método específico para gerar cada tipo de polígono, sendo que cada método retorna uma referência do tipo de objeto correspondente: o método **gere_triangulo** retorna uma referência (**t**) para um objeto da classe **Triangulo**, enquanto o método **gere_quadrilatero** retorna uma referência (**q**) para um objeto da classe **Quadrilatero**.

Assim, o código referente ao projetista pode ser reescrito conforme mostra abaixo a classe **ProjetistaSenior**.

```
public class ProjetistaSenior {
   public static void main(String args[]) {

      Gerador_de_Poligonos gerador = new Gerador_de_Poligonos();

      Ponto A = new Ponto( 2, 3 );
      Ponto B = new Ponto( 4.5, 7 );
      Ponto C = new Ponto( 2.8, 9.1 );
      Triangulo jardim = gerador.gere_triangulo( A, B, C );
      jardim.desenhe();

      Ponto D = new Ponto(7, 9.1);
      Ponto E = new Ponto(7, 7);
      Quadrilatero sala_estar = gerador.gere_quadrilatero( B, C, D, E );
      sala_estar.desenhe();

      Ponto F = new Ponto(7, 11);
      Ponto G = new Ponto(2.8, 11);
      Quadrilatero sala_jantar = gerador.gere_quadrilatero( C, D, F, G );
      sala_jantar.desenhe();
   }
}
```

Note que, agora, é criado um objeto da classe **Gerador_de_Poligonos** referenciado como **gerador** e, com uso desse, são criados os demais objetos que representam as peças da casa.

Como está o exemplo, não há uma vantagem significativa na introdução do intermediário para criação dos objetos gráficos, a não ser tornar o código mais elegante. Em aplicações reais, por outro lado, essa abordagem é frequentemente utilizada, pois permite operações mais avançadas em um único ponto, tais como verificar regras de segurança para a criação de objetos, contabilizar a criação de objetos, manter um histórico de objetos criados, realizar operações em grupos de objetos (por exemplo, remover todos os quadriláteros do desenho).