Question 1
The program for Question 1 initializes a simple Swing window and places a custom drawing panel in the frame. The panel overrides paintComponent(Graphics g) so all rendering happens inside that method and is redrawn automatically when the window is resized or uncovered. The code computes sizes and positions from getWidth() and getHeight() so the drawing scales with the panel. Primitive drawing calls such as drawLine, `drawOval`, and `drawRect` are used to render the required shapes. Colors and stroke widths are chosen before painting each shape so different items get distinct appearance. Finally, the main method creates the JFrame, adds the panel, sets the default close operation and shows the window to start the GUI.
Question 2
Question 2's program follows the same Swing pattern but draws a different static composition (for example, a set of geometric shapes laid out with relative positioning). The panel computes positions as fractions of the panel size to keep the layout centered and responsive. The drawing code often uses loops for repeated elements (grid lines, repeated shapes) so the shape density adapts to STEPS or other parameters. Colors are set with setColor(...) and anti-aliasing can be enabled by casting to Graphics2D and setting RenderingHints. Input or constants drive any offsets, and the main routine creates and displays the frame that holds the panel. The net effect is a responsive rendering that looks correct at different window sizes.
Question 3
The Question 3 solution draws the "fanning lines" pattern by dividing each edge into a fixed number of equal increments and connecting corresponding points with drawLine. The panel computes stepW = width / STEPS and stepH = height / STEPS so the endpoints lie exactly on the edges regardless of window size. A single loop iterates the step index and, per iteration, computes four start/end points and issues four drawLine calls to create the mirrored fans in all corners.
Using Math.round for computed coordinates prevents rounding drift and keeps lines aligned to pixel coordinates. Because endpoints are computed from the live width/height, the whole pattern scales when the frame is resized. The main class simply constructs the frame, adds the panel, and shows it.
Question 4
Question 4 expands the single-corner fan into four symmetric fans so the pattern forms a diamond/rotated-square look. The implementation uses one loop to compute the coordinates for each step, then draws the four mirrored lines (left→bottom, right→bottom, right→top, left→top) per iteration. Each line calculation uses either i * stepH or `(i+1) * stepW` to place start and end points exactly as described in the exercise text. This produces the same nested curved envelope seen in the textbook when all four mirrored sets overlap. The code keeps all geometry relative to the panel size so the figure scales cleanly when the window changes. The main class for this question creates and displays the panel in a frame.
Question 5
Question 5 implements two different spirals in separate panels: a square (rectilinear) spiral and a circular spiral. The square spiral starts at the panel center and draws line segments in a repeating direction sequence (down, left, up, right), increasing the segment length after every two segments. This is done in a loop that updates (x,y) and draws drawLine(x,y,nx,ny) each iteration, so the spiral grows outward smoothly. The circular spiral was first implemented using semicircles but then replaced with a parametric polar sampler: $r = a + b\cdot\theta$ is sampled at small delta steps and consecutive points are joined with drawLine for a continuous Archimedean-like spiral. Both panels enable anti-aliasing by using Graphics2D and RenderingHints to produce smooth curves. Each spiral scales by

computing its step length or b parameter from the panel size so the drawings look right at different window sizes.

Question 6

The Question 6 solution draws twelve concentric circles centered on the panel. It computes the center as cx = getWidth()/2 and cy = getHeight()/2, then iterates radii 10, 20, …, 120 pixels. For each radius the code computes the upper-left of the bounding box (cx - r, cy - r) and calls `drawOval` (or `drawOval`/`fillOval`) with diameter `2*r`. The circles are drawn from smallest to largest (or vice versa) and are all centered by construction, so they remain concentric even when the panel is resized. The panel uses Graphics2D with anti-aliasing for smoother outlines. A small Main class creates the window and shows the panel.

Question 7

Question 7's panel draws ten filled shapes with randomized color, type, position, and size. A Random instance produces values for red/green/blue channels, widths and heights limited to half the panel dimensions, and (x,y) coordinates chosen so the entire shape stays inside the panel. Each iteration picks either a rectangle or an oval (via `random.nextBoolean()`), sets the fill color, and draws with `fillRect` or `fillOval`. To make results reproducible for demonstration, the panel can be seeded with a fixed seed (e.g., new Random(42)) so each run yields the same arrangement. The panel uses anti-aliasing hints for nicer rendering. The Main class constructs and displays the panel; resizing simply changes the bounds for future draws.

Question 8

Question 8 reads five integers using `JOptionPane.showInputDialog` and then displays them as horizontal bars. The `BarChartApp` main prompts the user five times, validating input (re-prompting on non-integer or negative input and treating Cancel as zero), collects the integers into an array, and passes them to the `BarChartPanel`. The panel finds the maximum value among the inputs and uses it to scale each bar's width to the available drawing area so the largest value fills the allotted width. Each bar is drawn as a filled rectangle with an adjacent numeric label; a set of colors is used to visually distinguish bars. The UI is responsive: the bar widths recompute from the panel width on repaint so resizing adjusts the chart automatically.

Question 9

Question 9 draws the shield by painting concentric filled circles (red → white → red → blue) centered in the panel and a white five-pointed star placed in the center. The program computes `maxRadius` from the smaller panel dimension and draws a sequence of `fillOval` calls with radii scaled as fractions of `maxRadius` to reproduce the ring widths in the reference image. The central star is created by computing 10 alternating outer/inner vertices using polar coordinates (angles every 36°) and building a `Polygon` that is then filled white. Anti-aliasing is enabled to produce smooth circular arcs and crisp star edges. The Main launcher opens a frame sized appropriately and places the `ShieldPanel` in it; because all coordinates are computed from the panel size the shield scales nicely when the window is resized.