**Program Structure**

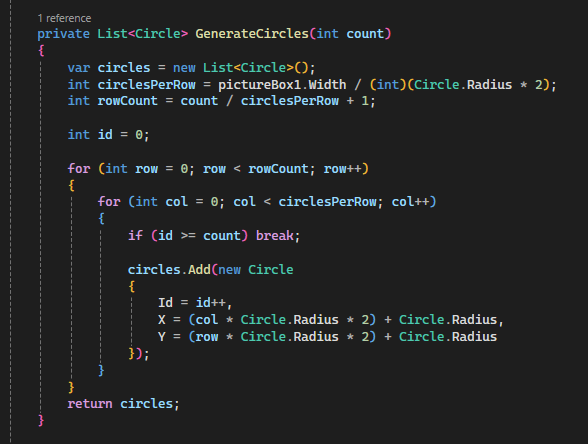
Structure of the program:

1. **Windows Forms GUI**:
   * User selects the number of workers (cbWorkerCount) and starts the simulation with a button (Start).
   * Circles are visualized in a PictureBox, and progress is displayed dynamically (lblWorkers).
2. **Core Classes**:
   * The **Circle** represents the painting targets, and the **Worker** represents the entities doing the work.
3. **Simulation Logic**:
   * Workers (Task.Run) dynamically pick unpainted circles and paint them, updating the shared state (IsPainted) and UI.

**Key Methods and Their Purpose**

**1. Generate Circles**

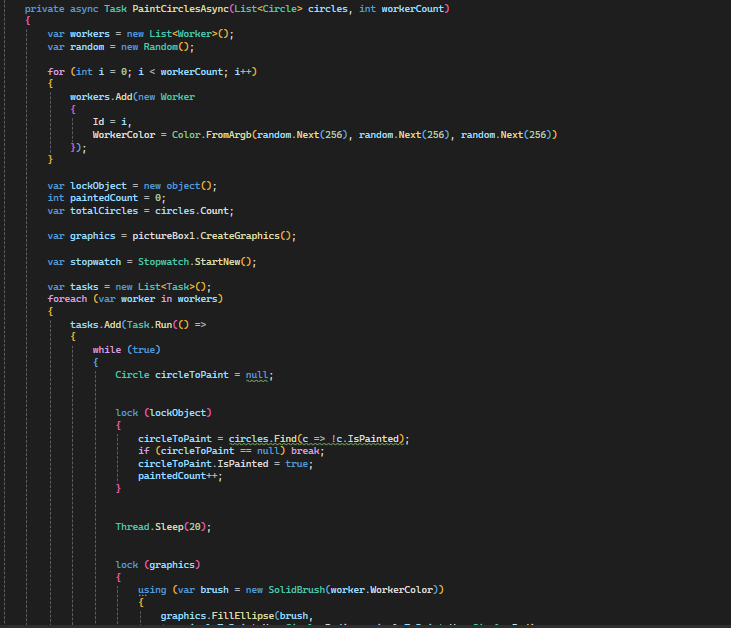
This method creates a grid of circles based on the PictureBox size and arranges them row by row. Each circle is represented by its X, Y coordinates and a radius of 10 pixels.

****

**2. Paint Circles**

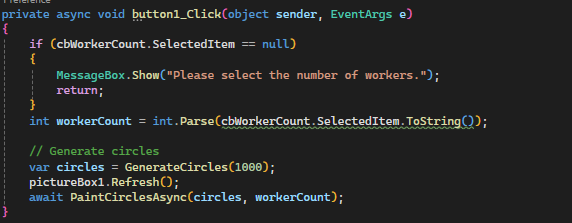
The painting process is handled in parallel by dynamically assigning tasks to workers. Each worker:

1. Selects the next unpainted circle.
2. Simulates painting with a delay of 20 milliseconds.
3. Updates the circle’s status and visualizes the painted circle on the PictureBox.

****

**3. User Interaction**

* **ComboBox (cbWorkerCount)**: Allows the user to select the number of workers.
* **Start Button (button1)**: Starts the simulation by:
  1. Generating 1,000 circles.
  2. Launching PaintCirclesAsync.

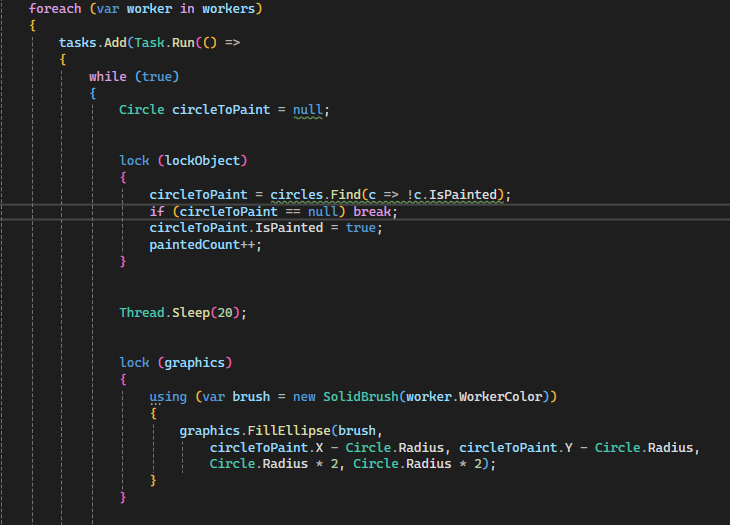
****

**Evaluation**

**a) Is this problem able to be parallelized?**

Yes, this problem is **highly parallelizable**. Multiple workers can paint circles simultaneously since each circle is independent of the others.

**Code for Parallel Execution**: Each worker is represented as a task (Task.Run), enabling parallelism:

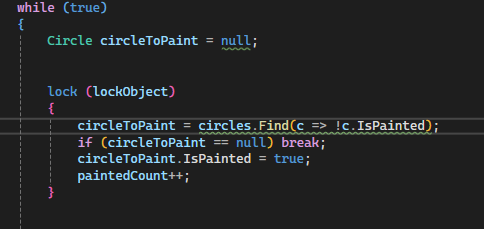


**b) How would the problem be partitioned?**

The problem is **partitioned dynamically**:

* Circles are shared among all workers.
* Workers pick the next available unpainted circle dynamically from the shared circles list.

**Code for Partitioning**: Each worker selects a circle dynamically from the shared list:

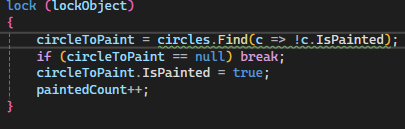


**c) Are communications needed?**

Yes, communications are needed in two ways:

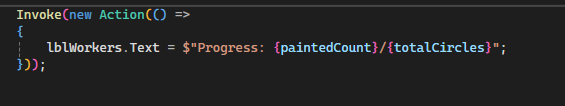
1. **Between Workers and Shared Data (Circles)**:
   * Workers communicate implicitly by accessing and updating the shared circles list.

**Code for Accessing Shared Data**:



1. **Between Workers and the UI**:
   * Progress is updated on the UI (lblWorkers.Text) from worker threads.

**Code for Updating UI**: UI updates are performed using Invoke to ensure thread safety:



**d) Are there any data dependencies?**

Yes, the **shared list of circles (circles)** has a dependency:

* Only unpainted circles can be selected.
* Once a circle is painted, no other worker should paint it again.

**e) Are there synchronization needs?**

Yes, there are two main synchronization needs:

1. **Shared Resource Access**:
   * Workers need synchronized access to the circles list to ensure no two workers paint the same circle.
2. **UI Updates**:
   * Since worker threads cannot directly update the UI, updates are synchronized using Invoke.

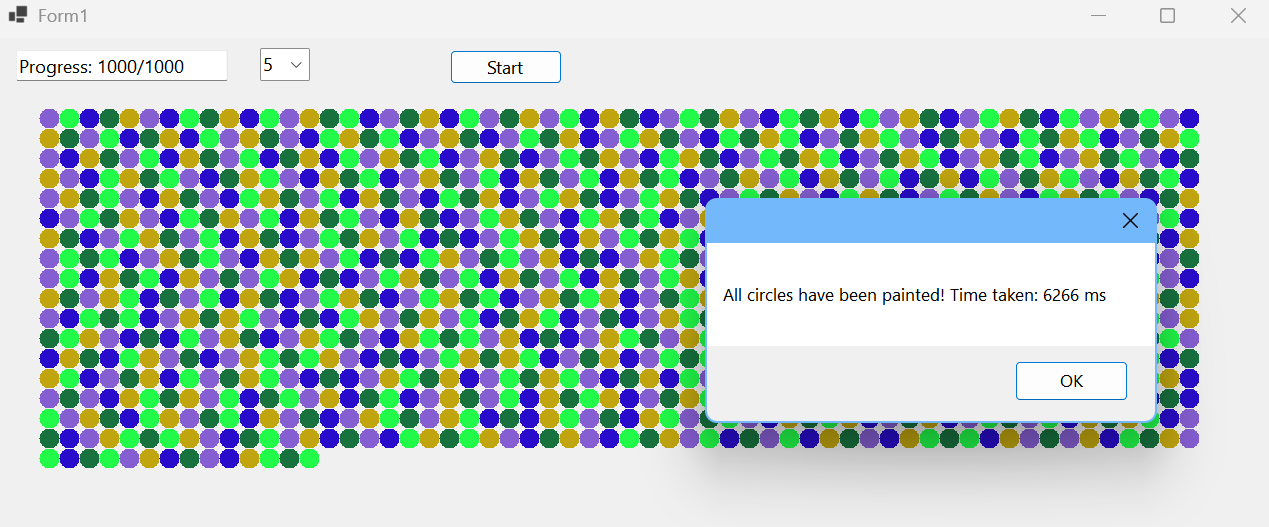
**f) Will load balancing be a concern?**

Load balancing is **not a significant concern** in this design because:

* Workers dynamically pick unpainted circles.
* Even if one worker finishes early, others continue until all circles are painted.

**Code for Dynamic Load Balancing**: The dynamic assignment ensures workers always have tasks until all circles are painted:

**The Result**



The program successfully executes the task of painting 1,000 circles arranged in a grid layout. The workload distribution across threads is efficient, and each thread (worker) paints circles dynamically. The program visualizes the painted circles with unique colors for each worker, clearly showcasing the progress in real-time.

**The result for K = 5:** The execution time is **6.266 seconds (6,266 ms)**. The workload is distributed dynamically, ensuring that all circles are painted without any idle workers. As we can observe, the circles are painted row by row, with workers dynamically selecting available tasks. The distribution of workload appears balanced across the threads, as there is no visible clustering of one worker's color. This demonstrates that the computational speed between threads is consistent, ensuring smooth execution. Additionally, the progress bar reflects accurate tracking of the task, confirming that all 1,000 circles are successfully painted.