Explanation of PID Controller Code

Purpose of the PID Class

The provided code implements a **Proportional-Integral-Derivative (PID) controller** in Unity. A PID controller is a feedback mechanism used to control systems by minimizing errors between a desired and actual output. The controller calculates its output using three components:

- **Proportional (P):** Reacts to the current error.
- **Integral (I):** Reacts to the accumulated error over time.
- **Derivative (D):** Reacts to the rate of change of the error.

Class-Level Variables

- **PID Constants:**

$$\begin{aligned} k_P &- \text{Proportional constant} \\ k_I &- \text{Integral constant} \\ k_D &- \text{Derivative constant} \\ k_U &- \text{Scaling constant for the final output} \end{aligned}$$

- **State Variables:**

$$\begin{aligned} \text{lastErr} &- \text{The error from the previous computation (used for derivative calculation).} \\ \text{totErr} &- \text{The cumulative error (used for the integral term).} \\ \text{absError} &- \text{The sum of absolute errors (useful for debugging).} \end{aligned}$$

- **Buffers:**
  - `trashBuffer`: A buffer array to store intermediate results for debugging purposes.
  - `SampleTime`: The minimum interval for recalculating the PID output.
  - `accDTime`: Accumulated time since the last PID computation.
  - `lastU`: The last computed PID output.

Constructor

The constructor initializes the PID controller:

- Assigns constants k_P, k_I, k_D, and k_U.
- Sets up debugging variables and initializes the buffer.

Listing 1: PID Constructor

```
public PID(float kP, float kI, float kD, float kU) {
    myId = id++;
    this.kP = kP;
    this.kI = kI;
    this.kD = kD;
    this.kU = kU;

    trashBuffer = new float[4];
}
```

Key Methods

getTotError()

This method returns the total accumulated absolute error:

$$\text{absError} = \sum |\text{Error}|$$

Listing 2: getTotError Method

```
public float getTotError() {
    return absError;
}
```

getU(float Err, float deltaTime)

This is an overloaded version of the main computation function. It calculates the PID output using a default debugging buffer.

Listing 3: getU Method (Overloaded)

```
public float getU(float Err, float deltaTime) {
    return getU(Err, deltaTime, trashBuffer);
}
```

getU(float Err, float deltaTime, float[] values)

This is the core PID computation function. It calculates the PID output using the Proportional, Integral, and Derivative components.

Step-by-Step Logic

1. Check if enough time has passed since the last computation:

$$\text{if } \Delta t + \text{accDTime} > \text{SampleTime, continue.}$$

2. Calculate each PID component:

$$\begin{aligned}
\text{Proportional (P):} \quad & p_{\text{factor}} = \text{Error} \\
\text{Derivative (D):} \quad & d_{\text{factor}} = \frac{\text{Error} - \text{lastErr}}{\text{SampleTime}} \\
\text{Integral (I):} \quad & i_{\text{factor}} = \sum \text{Error} \quad (\text{totErr is updated.})
\end{aligned}$$

3. Calculate the final PID output:

$$u = k_P \cdot p_{\text{factor}} + k_D \cdot d_{\text{factor}} + k_I \cdot i_{\text{factor}}$$

4. Store intermediate values in the buffer for debugging:

$$\text{values}[0] = k_P \cdot p_{\text{factor}}, \quad \text{values}[1] = k_I \cdot i_{\text{factor}}, \quad \text{values}[2] = k_D \cdot d_{\text{factor}}, \quad \text{values}[3] = u \cdot k_U$$

5. Return the computed output, u.

Listing 4: getU Method (Main)

```
public float getU(float Err, float deltaTime, float[] values) {
    if (deltaTime + accDTime > SampleTime) {
        accDTime = deltaTime + accDTime - SampleTime;

        float pFactor = Err;
        float dFactor = (Err - lastErr);
        lastErr = Err;
        totErr += Err;
        float iFactor = totErr;
        absError += Mathf.Abs(Err);

        float u = (kP * pFactor + kD / SampleTime * dFactor + kI *
            SampleTime * iFactor) * kU;

        values[0] = kP * pFactor;
        values[1] = kI * iFactor;
        values[2] = kD * dFactor;
        values[3] = kU * u;

        lastU = u;
        return u;
    } else {
        accDTime += deltaTime;
        return lastU;
    }
}
```

Applications

This PID controller is suitable for:

- Stabilizing drones (controlling roll, pitch, or yaw).
- Controlling position or speed of moving objects.
- Any feedback control system requiring precision.

Example Usage

Listing 5: Example Usage

```
PID pidController = new PID(1.0f, 0.5f, 0.1f, 1.0f);
float error = 5.0f;
float deltaTime = 0.1f;

for (int i = 0; i < 10; i++) {
    float controlOutput = pidController.getU(error, deltaTime);
    Debug.Log("Control Output: " + controlOutput);
    error -= controlOutput * deltaTime;
}
```