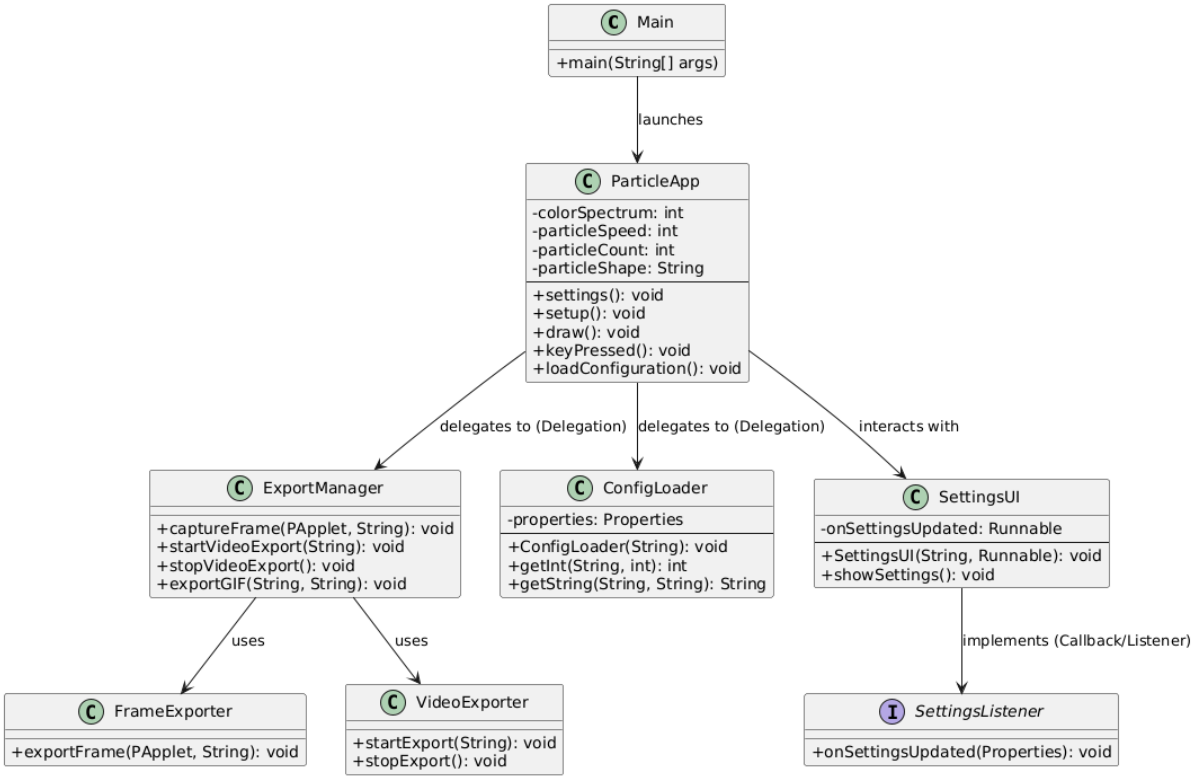
UML Diagram


Main
+main(String[] args)
launches
ParticleApp
-colorSpectrum: int
-particleSpeed: int
-particleCount: int
-particleShape: String
+settings(): void
+setup(): void
+draw(): void
+keyPressed(): void
+loadConfiguration(): void
delegates to (Delegation)
delegates to (Delegation)
interacts with
ExportManager
+captureFrame(PApplet, String): void
+startVideoExport(String): void
+stopVideoExport(): void
+exportGIF(String, String): void
ConfigLoader
-properties: Properties
+ConfigLoader(String): void
+getInt(String, int): int
+getString(String, String): String
SettingsUI
-onSettingsUpdated: Runnable
+SettingsUI(String, Runnable): void
+showSettings(): void
uses
uses
implements (Callback/Listener)
FrameExporter
+exportFrame(PApplet, String): void
VideoExporter
+startExport(String): void
+stopExport(): void
SettingsListener
+onSettingsUpdated(Properties): void

PlantUML Code

File Manager | Default Diagram | Samples | Refresh

```
@startuml
skinparam classAttributeIconSize 0

class Main {
    + main(String[] args)
}

class ParticleApp {
    - colorSpectrum: int
    - particleSpeed: int
    - particleCount: int
    - particleShape: String
    --
    + settings(): void
    + setup(): void
    + draw(): void
    + keyPressed(): void
    + loadConfiguration(): void
}

class ExportManager {
    + captureFrame(PApplet, String): void
    + startVideoExport(String): void
    + stopVideoExport(): void
    + exportGIF(String, String): void
}

class FrameExporter {
    + exportFrame(PApplet, String): void
}

class VideoExporter {
    + startExport(String): void
    + stopExport(): void
}

class ConfigLoader {
    - properties: Properties
    --
    + ConfigLoader(String): void
    + getInt(String, int): int
    + getString(String, String): String
}

class SettingsUI {
    - onSettingsUpdated: Runnable
    --
    + SettingsUI(String, Runnable): void
    + showSettings(): void
}

interface SettingsListener {
    + onSettingsUpdated(Properties): void
}

Main --> ParticleApp : launches
ParticleApp --> ExportManager : delegates to (Delegation)
ExportManager --> FrameExporter : uses
ExportManager --> VideoExporter : uses
ParticleApp --> ConfigLoader : delegates to (Delegation)
ParticleApp --> SettingsUI : interacts with
SettingsUI --> SettingsListener : implements (Callback/Listener)

@enduml
```

Key Classes and Core Methods in the UML Diagram

The UML diagram highlights 8 key classes and interfaces critical to the project. Here's a breakdown of the selected classes, their core methods, and the logic behind the diagram structure:

1. Main

- Purpose: Serves as the entry point for the application.
- Core Method:

  main(String[] args): Launches the application by creating an instance of ParticleApp.
- Reason for Inclusion: Every application needs an entry point, and Main provides a simple but essential role in starting the program.

2. ParticleApp

- Purpose: The central class that orchestrates the particle visualization, configuration, and interactions.
- Core Methods:

1. settings(): Configures the Processing canvas size.
2. setup(): Initializes the visualization setup, including configurations and export options.
3. draw(): Renders particle visualizations based on user-defined settings.
4. keyPressed(): Handles user input for actions like exporting frames or videos.
5. loadConfiguration(): Loads configuration values using ConfigLoader.

- Reason for Inclusion: As the main application logic hub, this class interacts with all other components.

3. ExportManager

- Purpose: Handles exporting functionalities such as capturing frames, creating GIFs, and exporting videos.
- Core Methods:

1. captureFrame(PApplet, String): Captures a single frame and saves it as an image.
2. startVideoExport(String): Starts recording frames for video export.
3. stopVideoExport(): Stops video recording and finalizes the file.

4. exportGIF(String, String): Creates a GIF from captured frames.

- Reason for Inclusion: This class encapsulates all exporting logic, demonstrating the Delegation Pattern by handling tasks delegated by ParticleApp.

4. FrameExporter

- Purpose: A helper class used by ExportManager to handle frame exports.
- Core Method:

  exportFrame(PApplet, String): Saves individual frames as image files.
- Reason for Inclusion: Highlights the Delegation Pattern within ExportManager.

5. VideoExporter

- Purpose: A helper class used by ExportManager to manage video exporting.
- Core Methods:
  1. startExport(String): Initializes video recording.
  2. stopExport(): Finalizes and saves the video.
- Reason for Inclusion: Like FrameExporter, this class showcases the Delegation Pattern in action.

6. ConfigLoader

- Purpose: Loads and manages configuration properties.
- Core Methods:
  1. ConfigLoader(String): Constructor that initializes the configuration loader with a properties file.
  2. getInt(String, int): Retrieves integer values from the configuration.
  3. getString(String, String): Retrieves string values from the configuration.
- Reason for Inclusion: Essential for managing user-defined settings, directly used by ParticleApp.

7. SettingsUI

- Purpose: Provides a graphical interface for updating visualization settings.
- Core Methods:
  1. SettingsUI(String, Runnable): Constructor that links the UI to configuration updates.

2. showSettings(): Displays the settings interface for user interaction.

- Reason for Inclusion: Demonstrates the Callback/Listener Pattern through its interaction with SettingsListener.

8. SettingsListener

- Purpose: Defines the interface for handling configuration updates.
- Core Method:

  onSettingsUpdated(Properties): Triggered when settings are updated.

- Reason for Inclusion: Central to the Callback/Listener Pattern, ensuring real-time updates between SettingsUI and ParticleApp.

Logic Behind the UML Diagram

Relationships:

- The Delegation Pattern is highlighted by the relationships between ParticleApp, ExportManager, FrameExporter, and VideoExporter.
- The Callback/Listener Pattern is demonstrated through the interaction between SettingsUI and SettingsListener.

Vertical Hierarchy:

The diagram follows a top-to-bottom flow, making it intuitive to trace relationships starting from Main.

Focus on Core Functionality:

Only classes that directly impact the application's core features (visualization, exporting, and configuration) are included.

Compact Representation:

Methods and attributes are listed concisely to avoid clutter, focusing on functionality.