Custom Responsive Widgets Documentation

This document explains the usage and purpose of the custom responsive widgets located in `lib/custom_responsive_widgets.dart`. These widgets are designed to simplify building adaptive user interfaces in Flutter, responding to different screen sizes and orientations.

1. Core Concepts

1.1 `DeviceType` Enum

An enumeration that categorizes the current device based on screen width and orientation.

- `mobile`: Screen width less than 600 logical pixels, portrait orientation.
- `mobileLandscape`: Screen width less than 600 logical pixels, landscape orientation.
- `tablet`: Screen width between 600 and 1000 logical pixels, portrait orientation.
- `tabletLandscape`: Screen width between 600 and 1000 logical pixels, landscape orientation.
- `desktop`: Screen width 1000 logical pixels or greater.

1.2 Breakpoints

The thresholds used to determine the `DeviceType`:

- **Mobile Breakpoint**: `600.0` logical pixels.
- **Tablet Breakpoint**: `1000.0` logical pixels.

These can be adjusted within `custom_responsive_widgets.dart` if your design requires different thresholds.

1.3 `getDeviceType(BuildContext context)`

A utility function that returns the current `DeviceType` based on the `MediaQuery` of the provided `context`. This function is used internally by all custom responsive widgets.

2. Widget Reference

2.1 `ResponsiveVisibility`

Conditionally shows or hides its child widget based on the current `DeviceType` and orientation.

- **Purpose**: To render a widget only on specific device types or orientations, completely removing it from the widget tree and layout when not visible.
- **Properties**:
    - `child` (required `Widget`): The widget to control visibility for.
    - `showForMobile` (`bool`, default: `true`): Show on mobile portrait.
    - `showForMobileLandscape` (`bool`, default: `false`): Show on mobile landscape.
    - `showForTablet` (`bool`, default: `true`): Show on tablet portrait.
    - `showForTabletLandscape` (`bool`, default: `false`): Show on tablet landscape.
    - `showForDesktop` (`bool`, default: `true`): Show on desktop.

Usage Example:

```
ResponsiveVisibility(
  showForDesktop: true,
  showForMobile: false,
  showForTablet: false,
  child: Text('This text is only visible on desktop.'),
)
```

-

2.2 `ResponsiveSizedBox`

Sizes its child to a percentage of the screen width and/or height, or to a fixed size.

- **Purpose**: To create flexible boxes whose dimensions scale with the screen or are fixed.
- **Properties**:
    - `child` (required `Widget`): The widget to size.
    - `widthPercentage` (`double?`, 0.0-1.0): Width as a percentage of screen width.
    - `heightPercentage` (`double?`, 0.0-1.0): Height as a percentage of screen height.
    - `fixedWidth` (`double?`): Fixed width in logical pixels (overrides `widthPercentage`).
    - `fixedHeight` (`double?`): Fixed height in logical pixels (overrides `heightPercentage`).

Usage Example:

```
ResponsiveSizedBox(
  widthPercentage: 0.8, // 80% of screen width
```

```
  heightPercentage: 0.3, // 30% of screen height
  child: Container(color: Colors.blue),
)
```

-

2.3 `CustomPadding`

Applies padding to its child widget using a flexible property system.

- **Purpose**: Provides a more declarative way to specify padding.
- **Properties**:
  - `child` (required `Widget`): The widget to apply padding to.
  - `all` (`double?`): Padding on all sides (highest priority).
  - `horizontal` (`double?`): Horizontal padding (left and right).
  - `vertical` (`double?`): Vertical padding (top and bottom).
  - `top` (`double?`): Top padding.
  - `bottom` (`double?`): Bottom padding.
  - `left` (`double?`): Left padding.
  - `right` (`double?`): Right padding.
- **Property Priority**: `all` > (`horizontal` OR `vertical`) > (`top` AND/OR `bottom` AND/OR `left` AND/OR `right`).

Usage Example:
```
CustomPadding(
  all: 16.0,
  child: Text('Padded equally on all sides.'),
)

CustomPadding(
  horizontal: 24.0,
  top: 10.0, // horizontal and top will apply, vertical, bottom, right default to 0
  child: Text('Padded horizontally and top.'),
)
```

-

2.4 `ResponsiveText`

Displays text with a font size that adapts to the `DeviceType` and orientation.

- **Purpose**: Ensures text readability and aesthetic consistency across different screen sizes without manual `MediaQuery` checks.

- **Properties**:
  - `text` (required `String`): The text content.
  - `textStyle` (`TextStyle?`): An optional base text style to apply (font weight, color, etc.). The font size from this style will be overridden by the responsive font size.
  - `textAlign` (`TextAlign?`): How the text should be aligned.
  - `mobileFontSize` (`double?`): Font size for mobile portrait.
  - `mobileLandscapeFontSize` (`double?`): Font size for mobile landscape.
  - `tabletFontSize` (`double?`): Font size for tablet portrait.
  - `tabletLandscapeFontSize` (`double?`): Font size for tablet landscape.
  - `desktopFontSize` (`double?`): Font size for desktop.

Usage Example:

```
ResponsiveText(
  text: 'Adaptive Title',
  mobileFontSize: 18.0,
  desktopFontSize: 28.0,
  textStyle: TextStyle(fontWeight: FontWeight.bold, color: Colors.deepPurple),
)
```

-

2.5 `ResponsiveConstraintBox`

Applies minimum and maximum width/height constraints to its child based on the `DeviceType` and orientation.

- **Purpose**: Prevents widgets from becoming too small or too large, ensuring they stay within desirable bounds on different screens.
- **Properties**:
  - `child` (required `Widget`): The widget to constrain.
  - `minWidth...`, `maxWidth...`, `minHeight...`, `maxHeight...`: Specific min/max values for each `DeviceType` and orientation. (e.g., `minWidthMobile`, `maxWidthDesktop`, `maxHeightTabletLandscape`).

Usage Example:

```
ResponsiveConstraintBox(
  maxWidthDesktop: 800.0, // Max width 800px on desktop
  minWidthMobile: 300.0,  // Min width 300px on mobile
  child: Container(color: Colors.green, child: Text('Constrained Box')),
)
```

-

2.6 `ResponsiveSpacer`

Provides dynamic spacing between widgets, adapting its width or height based on the `DeviceType` and orientation.

- **Purpose**: To manage responsive gaps in `Row` or `Column` layouts.
- **Properties**:
    - `widthMobile`, `heightMobile`, etc.: Width/height values for different `DeviceType` and orientations.

Usage Example:

```
Row(
  children: [
    Text('Item 1'),
    ResponsiveSpacer(widthMobile: 10.0, widthDesktop: 30.0), // Smaller gap on mobile, larger on desktop
    Text('Item 2'),
  ],
)
```

-

2.7 `ResponsiveBuilder`

A powerful generic widget that provides the current `DeviceType` and `Orientation` to its builder function, allowing for highly customized UI rendering logic.

- **Purpose**: When `ResponsiveVisibility` isn't enough and you need to build entirely different widget trees or apply complex logic based on the responsive context.
- **Properties**:
    - `builder` (required `ResponsiveWidgetBuilder`): A function that takes `(BuildContext context, DeviceType deviceType, Orientation orientation)` and returns a `Widget`.

Usage Example:

```
ResponsiveBuilder(
  builder: (context, deviceType, orientation) {
    if (deviceType == DeviceType.desktop) {
      return MyDesktopLayout();
    } else if (deviceType == DeviceType.mobile && orientation == Orientation.portrait) {
      return MyMobilePortraitLayout();
    } else {
      return MyTabletOrMobileLandscapeLayout();
```

```
  }
 },
)
```

-

2.8 `ResponsiveValue<T>`

A utility class that helps you select a specific value (of any type `T`) based on the current `DeviceType` and orientation.

- **Purpose**: To retrieve dynamic values for properties like padding, margins, counts, colors, or animation durations, without manually writing `switch` statements every time.
- **Properties**:
  - `mobile`, `mobileLandscape`, `tablet`, `tabletLandscape`, `desktop`: Values of type `T` for each responsive category.
- **Method**:
  - `get(BuildContext context)`: Returns the appropriate value based on the current device and orientation.

Usage Example:

```
final double buttonHeight = ResponsiveValue<double>(
  mobile: 40.0,
  tablet: 50.0,
  desktop: 60.0,
).get(context)!; // Use ! if you're sure a value will be present for the current context

final Color primaryColor = ResponsiveValue<Color>(
  mobile: Colors.red,
  desktop: Colors.blue,
).get(context)!;
```

-

2.9 `ResponsiveLayoutGrid`

Creates a `GridView` whose number of columns adapts to the `DeviceType` and orientation.

- **Purpose**: To build responsive grid layouts common in dashboards, galleries, or product listings.
- **Properties**:
  - `children` (required `List<Widget>`): The list of widgets to display in the grid.

    - `mobileColumns`, `mobileLandscapeColumns`, `tabletColumns`, `tabletLandscapeColumns`, `desktopColumns` (`int`, defaults provided): The number of columns for each responsive category.
    - `mainAxisSpacing` (`double?`): Spacing between rows.
    - `crossAxisSpacing` (`double?`): Spacing between columns.
    - `childAspectRatio` (`double?`): The ratio of the cross-axis extent to the main-axis extent of each child.

Usage Example:

```
ResponsiveLayoutGrid(
  mobileColumns: 1,
  tabletColumns: 2,
  desktopColumns: 3,
  mainAxisSpacing: 10.0,
  crossAxisSpacing: 10.0,
  children: [
    // Your grid items here
    Container(color: Colors.red, child: Center(child: Text('Item 1'))),
    Container(color: Colors.green, child: Center(child: Text('Item 2'))),
    // ... more items
  ],
)
```

-

3. General Usage

1. **Save the file**: Place the combined code in your `lib` folder (e.g., `lib/cwp.dart`).

Import: In any `.dart` file where you want to use these widgets, add the import:
import 'package:your_app_name/custom_responsive_widgets.dart'; // Adjust 'your_app_name' as needed

2.
3. **Ensure `MaterialApp`**: Always ensure your application's root widget is a `MaterialApp` (or `WidgetsApp`). All these responsive widgets rely on `MediaQuery` which is provided by `MaterialApp`.

By leveraging these widgets, you can build powerful, adaptable Flutter applications with cleaner, more maintainable code!