Flutter Internals

Introduction

- What is this document?
  - The goal of this document is to provide intuitive descriptions of Flutter's internals in an easily digestible Q&A format.
  - Descriptions are intended to be comprehensive (i.e., low-level) without becoming bogged down in implementation details (i.e., unmaintainable) or sacrificing clarity (i.e., cryptic).
  - This document strives to provide a "hawk's eye view" of Flutter's subsystems to help guide an investigation into the framework.
- Who is the audience of this document?
  - Contributors seeking to ramp up on how a particular subsystem works, developers looking to build intuition about Flutter internals, anyone looking for a guided tour through the codebase.
  - These notes were originally prepared to help "rekindle" intuition about Flutter without re-trawling through the documentation and code.
  - These notes may serve as a reference for other types of learning materials (such as a deep dive video series). I also hope to publish these notes as a standalone resource on the Flutter website.
  - Parts of this document may be suitable for "promotion" into the official API documentation to help improve clarity.
- Who can contribute to this document?
  - Anyone (thank you)! If there's a corner of the framework that you find confusing, please consider updating this FAQ with an intuitive explanation of how things fit together -- with references to relevant code. Please feel free to add yourself to the contributors line, above.
- Won't this document become stale?
  - Alas, this is certainly possible. While I've aimed to provide details that are precise without being too "fragile," a certain amount of staleness is unavoidable.
  - I'm committed to expanding and maintaining this document as part of my everyday work with Flutter; I see this as a way to remain on top of changes to the framework as things evolve.
  - Once polished, my hope is to release this document as a properly linked FAQ, complete with source. In this way, the community can help keep the content up to date.

- By telling a broader story than the API documentation, this FAQ should augment – not compete – with other documentation efforts.


Core

Framework

How is the app bootstrapped?

- `runApp` kicks off binding initialization by invoking the `WidgetsFlutterBinding` / `RenderingFlutterBinding.ensureInitialized` static method. This calls each binding's `initInstances` method, allowing each to initialize in turn.
  - This flow is built using mixin chaining: each of the concrete bindings (e.g., `WidgetsFlutterBinding`) extends `BaseBinding`, the superclass constraint shared by all binding mixins (e.g., `GestureBinding`). Consequently, common methods (like `BaseBinding.initInstances`) can be chained together via super invocations. These calls are linearized from left-to-right, starting with the superclass and proceeding sequentially through the mixins; this strict order allows later bindings to depend on earlier ones [?].
- `RendererBinding.initInstances` creates the `RenderView`, passing an initial `ViewConfiguration` (describing the size and density of the render surface). It then prepares the first frame (via `RenderView.prepareInitialFrame`); this schedules the initial layout and initial paint (via `RenderView.scheduleInitialLayout` and `RenderView.scheduleInitialPaint`; the latter creates the root layer, a `TransformLayer`). This marks the `RenderView` as dirty for layout and painting but does not actually schedule a frame.
  - This is important since users may wish to begin interacting with the framework (by initializing bindings via `BaseBinding.ensureInitialized`) before starting up the app (via `runApp`). For instance, a plugin may need to block on a backend service before it can be used.
- Finally, the `RendererBinding` installs a persistent frame callback to actually draw the frame (`WidgetsBinding` overrides the method invoked by this callback to add the build phase). Note that nothing will invoke this callback until the `Window.onDrawFrame` handler is installed. This will only happen once a frame has actually been scheduled.
- Returning to `runApp`, `WidgetsBinding.scheduleAttachRootWidget` asynchronously creates a `RenderObjectToWidgetAdapter`, a `RenderObjectWidget` that inserts its child (i.e., the app's root widget) into the provided container (i.e., the `RenderView`).
  - This asynchronicity is necessary to avoid scheduling two builds back-to-back; while this isn't strictly invalid, it is inefficient and may trigger asserts in the

framework.
    - If the initial build weren't asynchronous, it would be possible for intervening events to re-dirty the tree before the warm up frame is scheduled. This would result in a second build (without an intervening layout pass, etc.) when rendering the warm-up frame. By ensuring that the initial build is scheduled asynchronously, there will be no render tree to dirty until the platform is initialized.
    - For example, the engine may report user settings changes during initialization (via the `_updateUserSettingsData` hook). This invokes callbacks on the window (e.g., `Window.onTextScaleFactorChanged`), which are forwarded to all `WidgetsBindingObservers` (e.g., via `RendererBinding.handleTextScaleFactorChanged`). As an observer, `WidgetsApp` reacts to the settings data by requesting a rebuild.
- It then invokes `RenderObjectToWidgetAdapter.attachToRenderTree` to bootstrap and mount an element to serve as the root of the element hierarchy (`RenderObjectToWidgetElement`). If the element already exists, which will only happen if `runApp` is called again, its associated widget is updated (`RenderObjectToWidgetElement._newWidget`) and marked as needing to be built.
    - `RenderObjectToWidgetElement.updateChild` is invoked when this element is mounted or rebuilt, inflating or updating the child widget (i.e., the app's root widget) accordingly. Once a descendent `RenderObjectWidget` is inflated, the corresponding render object (which must be a `RenderBox`) will be inserted into the `RenderView` (via `RenderObjectToWidgetElement.insertChildRenderObject`). The resulting render tree is managed in the usual way going forward.
    - A reference to this element is stored in `WidgetsBinding.renderViewElement`, serving as the root of the element tree. As a `RootRenderObjectElement`, this element establishes the `BuildOwner` for its descendants.
- Finally, after scheduling the first frame (via `SchedulerBinding.instance.ensureVisualUpdate`, which will lazily install the frame callbacks), `runApp` invokes `SchedulerBinding.scheduleWarmUpFrame`, manually pumping the rendering pipeline. This gives the initial frame extra time to render as it's likely the most expensive.
- `SchedulerBinding.ensureFrameCallbacksRegistered` lazily installs frame callbacks as part of `SchedulerBinding.scheduleFrame`. Frames are typically scheduled in response to `PipelineOwner.requestVisualUpdate` (due to UI needing painting, layout, or a rebuild). Once configured, these callbacks (`Window.onBeginFrame`,

`Window.onDrawFrame`) are invoked once per frame by the engine, running transient and persistent processes, respectively. The latter is generally responsible for, e.g., ticking animations whereas the former runs the actual rendering pipeline.

How is a frame rendered?

- Once a frame is scheduled and callbacks are registered (via `SchedulerBinding.ensureFrameCallbacksRegistered`), the engine begins requesting frames automatically. The frame callbacks invoke handlers in response to these requests. In particular, `SchedulerBinding.drawFrame` processes persistent frame callbacks which are used to implement `Flutter`'s rendering pipeline. `WidgetsBinding.drawFrame` overrides `RendererBinding.drawFrame` to add the build process to this pipeline.
- The rendering pipeline builds widgets, performs layout, updates compositing bits, paints layers, and finally composites everything into a scene which it uploads to the engine (via `RenderView.compositeFrame`). Semantics are also updated by this process.
- `RenderView.compositeFrame` retains a reference to the root layer which it recursively composites using `Layer.buildScene`. This iterates through all layers that "`needsAddToScene`," which if true, composites a new frame. If false, previous invocations of `addToScene` will have stored an `EnglineLayer` in `Layer.engineLayer`, saving work ("retained rendering"); this is added using `SceneBuilder.addRetained`. Once a `Scene` is constructed, it is uploaded to the engine via `Window.render`.

How does the framework interact with the engine?

- The framework primarily interacts via the `Window` class, a dart interface with hooks into and out of the engine.
- The majority of the framework's flows are driven by frame callbacks invoked by the engine. Other flows are driven by gesture handling, platform messaging, and device messaging.

- Each binding serves as the singleton root of a subsystem within the framework; in several cases, bindings are layered to add functionality to more fundamental bindings (i.e., `WidgetsBinding` adds support for building to `RendererBinding`). All direct framework/engine interaction is managed via the bindings, with the sole exception of the `RenderView`.

What bindings are implemented?

- `GestureBinding` facilitates gesture handling across the framework, maintaining the gesture arena and pointer routing table.
  - Handles `Window.onPointerDataPacket`.
- `ServicesBinding` facilitates message passing between the framework and platform.
  - Handles `Window.onPlatformMessage`.
- `SchedulerBinding` manages a variety of callbacks (transient, persistent, post-frame, and non-rendering tasks), tracking lifecycle states and scheduler phases. It is also responsible for explicitly scheduling frames when visual updates are needed.
  - Handles `Window.onDrawFrame`, `Window.onBeginFrame`.
  - Invokes `Window.scheduleFrame`.
- `PaintingBinding` owns the image cache which efficiently manages memory allocated to graphical assets used by the application. It also performs shader warm up to avoid jank during drawing (via `ShaderWarmUp.execute` in `PaintingBinding.initInstances`). This ensures that the corresponding shaders are compiled at a predictable time.
- `SemanticsBinding` which is intended to manage the semantics and accessibility subsystems (at the moment, this binding mainly tracks accessibility changes emitted by the engine via `Window.onAccessibilityFeaturesChanged`).
- `RendererBinding` implements the rendering pipeline. Additionally, it retains the root of the render tree (i.e., the `RenderView`) as well as the `PipelineOwner`, an instance that tracks visual updates due to layout, painting, compositing, etc. The `RendererBinding` also responds to events that may affect the application's rendering (including semantic state until these handlers are moved to the `SemanticsBinding`).
  - Handles `Window.onSemanticsAction`, `Window.onTextScaleFactorChanged`, `Window.onMetricsChanged`, `Window.onSemanticsEnabledChanged`.

    - Invokes `Window.render`.
- `WidgetsBinding` augments the renderer binding with support for widget building (i.e., configuring the render tree based on immutable UI descriptions). It also retains the `BuildOwner`, an instance that facilitates rebuilding the render tree when configuration changes (e.g., a new widget is substituted). The `WidgetsBinding` also responds to events that might require rebuilding related to accessibility and locale changes (though these may be moved to the `SemanticsBinding` in the future).
    - Handles `Window.onAccessibilityFeaturesChanged`, `Window.onLocaleChanged`.
- `TestWidgetsFlutterBinding` supports the widget testing framework.

How do global keys work?

- `Element.inflateWidget` checks for a global key and uses it to find the original widget

Types

What types are used to describe positions?

- `OffsetBase` represents a 2-dimensional, axis-aligned vector. Subclasses are immutable and comparable using standard operators.
- `Offset` is an `OffsetBase` subclass that may be understood as a point in cartesian space or a vector. Offsets may be manipulated algebraically using standard operators; the “&” operator allows a `Rect` to be constructed by combining the offset with a `Size` (the offset identifies the rectangle’s top left corner). Offsets may also be interpolated.
- `Point` is a dart class for representing a 2-dimensional point on the cartesian plane.

What types are used to describe magnitudes?

- `Size` is an `OffsetBase` subclass that represents a width and a height. Geometrically, `Size` describes a rectangle with its top left corner coincident with the origin. Size includes a number of methods describing a rectangle with dimensions matching the current instance and a top left corner coincident with a specified offset. Sizes may be manipulated algebraically using standard operators; the “+” operator expands the size according to a provided delta (via `Offset`). Sizes may also be interpolated.
- `Radius` describes either a circular or elliptical radius. The radius is expressed as intersections of the x- and y-axes. Circular radii have identical values. Radii may be manipulated algebraically using standard operators and interpolated.

What types are used to describe regions?

- `Rect` is an immutable, 2D, axis-aligned, floating-point rectangle whose coordinates are relative to a given origin. A rectangle can be described in various ways (e.g., by its center, by a bounding circle, by offsets from its left, top, right, and bottom edges, etc) or constructed by combining an `Offset` and a `Size`. Rectangles can be inflated,

deflated, combined, intersected, translated, queried, and more. Rectangles can be compared for equality and interpolated.

- `RRect` augments a `Rect` with four independent radii corresponding to its corners. Rounded rectangles can be described in various ways (e.g., by offsets to each of its sides and one or more radii, by a bounding box fully enclosing the rounded rectangle with one or more radii, etc). Rounded rectangles define a number of sub-rectangles: a bounding rectangle (`RRect.outerRect`), a rectangle with identical sides and edges centered within the rounded corners (`RRect.middleRect`), tall and wide inner rectangles with height and width matching the rounded rectangle (`RRect.tallMiddleRect` , `RRect.wideMiddleRect`), and more. A rounded rectangle is said to describe a “stadium” if it possesses a side with no straight segment (e.g., entirely drawn by the two rounded corners). Rounded rectangles can be interpolated.

What types are used to describe coordinate spaces?

- `Axis` represents the X- or Y-axis (horizontal or vertical, respectively) relative to a coordinate space. The coordinate space can be arbitrarily transformed and therefore need not be parallel to the screen’s edges.
- `AxisDirection` applies directionality to an axis. The value represents the spatial direction in which values increase along the axis, with the origin being rooted at the opposite end (e.g., `AxisDirection.down` positions the origin at the top with positive values growing downward).
- `GrowthDirection` is the direction of growth relative to the current axis direction (e.g., how items are ordered along the axis). `GrowthDirection.forward` implies an ordering consistent with the axis direction (the first item is at the origin with subsequent items following). `GrowthDirection.reverse` is exactly the opposite (the last item is at the origin with preceding items following).
    - Growth direction does not flip the meaning of “leading” and “trailing,” it merely determines how children are ordered along a specified axis.
    - For a viewport, the origin is positioned based on the axis direction (e.g., `AxisDirection.down` positions the origin toward the top of the screen, `AxisDirection.up` positions the origin toward the bottom of the screen), with the growth direction determining how children are ordered at the origin. As a result,

both pieces of information are necessary to determine where a set of slivers should actually appear.

- `ScrollDirection` represents the user's scroll direction relative to the positive scroll offset direction (itself determined by axis direction and growth direction). Includes an idle state (`ScrollDirection.idle`).
    - Confusingly, this refers to the direction the content is moving on screen rather than where the user is scrolling (e.g., scrolling down a webpage causes the page's contents to move upward; this would be classified as `ScrollDirection.reverse` since this motion is opposite the axis direction).

What types are used to describe graphics?

- `Color` is a 32-bit immutable quantity describing alpha, red, green, and blue color channels. Alpha can be defined using an opacity value from zero to one. Colors may be freely interpolated and converted into a luminance value.
- `Shadow` represents a single drop shadow with a color, an offset from the casting element, and a blur radius characterizing the gaussian blur applied to the shadow.
- `Gradient` describes one or more smooth color transitions. Gradients may be interpolated and scaled; gradients may also be used to obtain a corresponding shader. Linear, radial, and sweep gradients are supported (via `LinearGradient` , `RadialGradient` , and `SweepGradient` , respectively). `TileMode` determines how a gradient paints beyond its defined bounds (a linear gradient defines a strip, a radial gradient defines a disc). Gradients may be clamped (e.g., hold their initial and final values), repeated (e.g., restarted at their bounds), or mirrored (e.g., restarted but with initial and final values alternating).

Messaging

How are messages passed between the framework and platform code?

- `ServicesBinding.initInstances` sets the global message handler (`Window.onPlatformMessage`) to `ServicesBinding.defaultBinaryMessenger` . This instance supports handler registration and message handling when messages arrive or are sent (`BinaryMessenger.setMessageHandler` , `BinaryMessenger.handlePlatformMessage`). It multiplexes the handler into channels using a channel name, an identifier shared by the framework and the platform.

What are the building blocks of messaging?

- `BinaryMessenger` multiplexes the global message handler via channel names, supporting handler registration and bidirectional binary messaging. Sending a message produces a future that resolves to the undecoded response.
- `MessageCodec` defines an interface to encode and decode byte data (`MessageCodec.encodeMessage` , `MessageCodec.decodeMessage`). A cross-platform binary codec is available (`StandardMessageCodec`) as well as a `JSON` -based codec (`JSONMessageCodec`). The platform must implement a corresponding codec natively.
- `MethodCodec` is analogous to `MessageCodec` , encoding and decoding `MethodCall` instances (which wrap a method name and a dynamic list of arguments). These codecs also pack and unpack results into envelopes to distinguish success and error outcomes.
- `BasicMessageChannel` provides a thin wrapper around `BinaryMessager` that uses the provided codec to encode and decode messages to and from raw byte data.
- `PlatformChannel` provides a thin wrapper around `BinaryMessager` that uses the provided method codec to encode and decode method invocations. Responses to incoming invocations are packed into envelopes indicating outcome; similarly, results from outgoing invocations are unpacked from their encoded envelope. These are returned as futures.

    - Success envelopes are unpacked and the result returned.
    - Error envelopes throw a `PlatformException`.
    - Unrecognized methods throw a `MissingPluginException` (except when using an `OptionalMethodChannel`).
- `EventChannel` is a helper that exposes a remote stream as a local stream. The initial subscription is handled by invoking a remote method called “listen” (via `PlatformChannel.invokeMethod`) which causes the platform to begin emitting a stream of envelope-encoded items. A top-level handler is installed (via `ServicesBinding.defaultBinaryMessenger.setMessageHandler`) to unpack and forward items to the output stream. When the stream ends for any reason, a remote method called “cancel” is invoked and the global handler cleared.
- `SystemChannels` is a singleton instance that provides references to messaging channels that are essential to the framework (`SystemChannels.system`, `SystemChannels.keyEvent`, etc).

Platform Integration

How does `Flutter` interact with the system clipboard?

- Clipboard and `ClipboardData`

How does `Flutter` interact with the host system?

How are device dimensions (insets, padding, overlays) conveyed?

How are device configurations (resolution, orientation) conveyed?

What are service extensions?

What are `SystemChannels`?

Conventions

`performAction` () is internal, action() is external.

`computeValue` () is internal, `getValue` () is external.

- Recursive calls should use the external variant.

`adoptChild` () and `dropChild` () from `AbstractNode` must be called after updating the actual child model to allow the framework to react accordingly.

Many widgets are comprised of lower-level widgets, generally prefixed with raw (e.g., `GestureDetector` and `RawGestureDetector` , `Chip` and `RawChip` , `MaterialButton` and `RawMaterialButton`). Text and `RichText` is a naming exception.

Data Model

Widgets

What are the widget building blocks?

- The main entry points are `RenderObjectWidget`, `StatefulWidget`, and `StatelessWidget`. Widgets that export data to one or more descendant widgets (via notifications or another mechanism) utilize `ProxyWidget` or its subclasses, `InheritedWidget` and `ParentDataWidget`.
- In general, widgets either directly or indirectly configure render objects; these indirect "component" widgets (e.g., stateful and stateless widgets) participate in a building process, whereas render object widgets manage an associated render object directly (creating it and updating it via `RenderObjectWidget.createRenderObject` and `RenderObjectWidget.updateRenderObject`, respectively).
- Certain widgets wrap (i.e., reproject) a child widget (`ProxyWidget`), introducing heritable state (`InheritedWidget`, `InheritedModel`) or setting parent data used by an enclosing widget (`ParentDataWidget`).
  - `ProxyWidget` notifies clients (via `ProxyElement.notifyClients`) in response to widget changes (via `ProxyElement.updated`, called by `ProxyElement.update`).
  - `ParentDataWidget` utilizes this flow to update the first descendant render object's parent data (via `ParentDataElement._applyParentData`, which calls `RenderObjectElement._updateParentData`); this will be triggered any time the widget is updated.
- `PreferredSizeWidget` is not widely used but an interesting example of adapting widgets to a specific need (e.g., an `AppBar` expressing a size preference).
- Numerous helper subclasses exist, most notably including `SingleChildRenderObjectWidget`, `MultiChildRenderObjectWidget`, `LeafRenderObjectWidget`. These provide storage for a child / children without implementing the underlying child model.
- Anonymous widgets can be created using `Builder` and `StatefulBuilder`.

How does building work?

- Every widget is associated with an element, a mutable object corresponding to the widget's instantiation within the tree. Only those widgets associated with `ComponentElement` (e.g., `StatelessWidget`, `StatefulWidget`, `ProxyWidget`) participate in the build process; `RenderObjectWidget` subclasses, generally associated with `RenderObjectElements`, do not (these simply update their render object when building). `ComponentElement` instances only have a single child, typically that returned by their widget's build method.
- When the element tree is first affixed to the render tree via `RenderObjectToWidgetAdapter.attachToRenderTree`, the `RenderObjectToWidgetElement` (itself a `RootRenderObjectElement`) assigns a `BuildOwner` for the element tree. The `BuildOwner` is responsible for tracking dirty elements (`BuildOwner.scheduleBuildFor`), establishing build scopes wherein elements can be rebuilt / descendent elements can be marked dirty (`BuildOwner.buildScope`), and unmounting inactive elements at the end of a frame (`BuildOwner.finalizeTree`). It also maintains a reference to the root `FocusManager` and triggers reassembly after a hot reload.
- When a `ComponentElement` is mounted (e.g., after being inflated), an initial build is performed immediately (via `ComponentElement._firstBuild`, which calls `ComponentElement.rebuild`).
- Later, elements can be marked dirty using `Element.markNeedsBuild`. This is invoked any time the UI might need to be updated reactively (or explicitly, in response to `State.setState`). This method adds the element to the dirty list and, via `BuildOwner.onBuildScheduled`, schedules a frame via `SchedulerBinding.ensureVisualUpdate`.
    - Other operations trigger a rebuild directly (i.e., without marking the tree dirty first). These include `ProxyElement.update`, `StatelessElement.update`, `StatefulElement.update`, and `ComponentElement.mount` (for the initial build, only).
    - In contrast, builds are scheduled by `State.setState`, `Element.reassemble`, `Element.didChangeDependencies`, `StatefulElement.activate`, etc.
    - Proxy elements use notifications to indicate when underlying data has changed. In the case of `InheritedElement`, each dependant's `Element.didChangeDependencies` is invoked which, by default, marks that element as dirty.

- Once per frame, `BuildOwner.buildScope` will walk the element tree in depth-first order, only considering those nodes that have been marked dirty. By locking the tree and iterating in depth first order, any nodes that become dirty while rebuilding must necessarily be lower in the tree; this is because building is a unidirectional process – a child cannot mark its parent as being dirty. Thus, it is not possible for build cycles to be introduced and it is not possible for elements that have been marked clean to become dirty again.
- As the build progresses, `ComponentElement.performRebuild` delegates to the `ComponentElement.build` method to produce a new child widget for each dirty element. Next, `Element.updateChild` is invoked to efficiently reuse or recreate an element for the child. Crucially, if the child's widget hasn't changed, the build is immediately cut off. Note that if the child widget did change and `Element.update` is needed, that child will itself be marked dirty, and the build will continue down the tree.
- Each `Element` maintains a map of all `InheritedElement` ancestors at its location. Thus, accessing dependencies from the build process is a constant time operation.
- If `Element.updateChild` invokes `Element.deactivateChild` because a child is removed or moved to another part of the tree, `BuildOwner.finalizeTree` will unmount the element if it isn't reintegrated by the end of the frame.

How do stateful widgets work?

- `StatefulWidget` is associated with `StatefulElement`, a `ComponentElement` that is almost identical to `StatelessElement`. The key difference is that the `StatefulElement` references the `State` of the corresponding `StatefulWidget`, and invokes lifecycle methods on that instance at key moments. For instance, when `StatefulElement.update` is invoked, the `State` instance is notified via `State.didUpdateWidget`.
- `StatefulElement` creates the associated `State` instance when it is constructed (i.e., in `StatefulWidget.createElement`). Then, when the `StatefulElement` is built for the first time (via `StatefulElement._firstBuild`, called by `StatefulElement.mount`), `State.initState` is invoked. Crucially, `State` instance and the `StatefulWidget` share the same element.

- Since `State` is associated with the underlying `StatefulElement`, if the widget changes, provided that `StatefulElement.updateChild` is able to reuse the same element (because the widget's runtime type and key both match), `State` will be preserved. Otherwise, the `State` will be recreated.

Why is changing tree depth expensive?

- Changing the depth necessarily causes a rebuild of the subtree, since there will never be an element match for the new child (since it wasn't a child previously).
- This requires `Element.inflateWidget` to be invoked, which causes a fresh build, which causes layout, paint, etc.
- Adding a `GlobalKey` to the previous child can mitigate this issue since `Element.updateChild` is able to reuse elements that are stored in the `GlobalKey` registry (allowing that subtree to simply be reinserted instead of rebuilt).

Elements

What are elements?

- The element tree is anchored in the `WidgetsBinding` and established via `runApp` / `RenderObjectToWidgetAdapter`. Widgets are immutable representations of UI configuration data. Widgets are “inflated” into `Element` instances, which serve as their mutable counterparts. Among other things, elements model the relationship between widgets (e.g., the widget tree), store state / inherited relationships, and participate in the build process.
- Many lifecycle events are triggered by changes to the element tree. In particular, all elements are associated with a `BuildOwner` that is responsible for tracking dirty elements and, during `WidgetsBinding.drawFrame`, re-building the widget / element tree. This drives lifecycle events within widget and state objects.
- Elements are initially mounted. They may then be updated multiple times. An element may be deactivated (by the parent via `Element.deactivateChild`); it can be activated within the same frame, else it will be unmounted.
- `Element.updateChild` is used to alter the configuration (widget) of a given child, potentially inflating a new element if none exists, the new and old widgets do not have the same type, or the widgets have different keys. Updating an element may update the element’s children.

What are the element building blocks?

- Elements are broken down into `RenderObjectElements` and `ComponentElements`. `RenderObjectElements` are responsible for configuring render objects and keeping the render object tree and widget tree in sync. `ComponentElements` do not directly manage render objects but instead produce intermediate nodes via mechanisms like `Widget.build`. Both are triggered by `Element.performRebuild` which is itself triggered by `BuildOwner.buildScope`.

How is the render tree managed by `RenderObjectElement`?

- Render object elements are responsible for managing an associated render object. `RenderObjectElement.update` will cause the associated render object to be updated to match a new configuration (widget). This render object may have children; however, there may be several intermediate elements between its render object element and any descendent render object elements (due to intervening component elements). Slot tokens are passed down the element tree so that the render object elements corresponding to these children can integrate with the parent render object. This is managed via `RenderObjectElement.insertChildRenderObject`, `RenderObjectElement.moveChildRenderObject`, and `RenderObjectElement.removeChildRenderObject`.
- Elements can be moved during a frame. Such elements are “forgotten” so that they are excluding from iteration / updates, with the actual mutation taking place when the element is added to its new parent.
- When updating, children must be updated, too. To avoid unnecessarily inflation (and potential loss of state), the new and old child lists are synchronized using a linear reconciliation scheme optimized for empty lists, matched lists, and lists with one mismatched region:

1. The leading elements and widgets are matched by key and updated
2. The trailing elements and widgets are matched by key with updates queued (update order is significant)
3. A mismatched region is identified in the old and new lists
4. Old elements are indexed by key
5. Old elements without a key are updated with null (deleted)
6. The index is consulted for each new, mismatched widget
7. New widgets with keys in the index update together (re-use)
8. New widgets without matches are updated with null (inflated)
9. Remaining elements in the index are updated with null (deleted)

What are the render object element building blocks?

- `LeafRenderObjectElement`, `SingleChildRenderObjectElement`, and `MultiChildRenderObjectElement` provide support for common use cases and correspond to the similarly named widget helpers. The multi-child and single-child variants integrate with `ContainerRenderObjectMixin` and `RenderObjectWithChildMixin`, respectively.
- These use the previous child (or null) as the slot identifier; this integrates nicely with `ContainerRenderObjectMixin` which supports an “after” argument when managing children.

How are components managed by `ComponentElement`?

- `ComponentElement.build` provides a hook for producing intermediate nodes in the element tree. `StatelessElement.build` invokes the widget’s build method, whereas `StatefulElement.build` invokes the state’s build method. Mounting and updating cause rebuild to be invoked. For `StatefulElement`, a rebuild may be scheduled spontaneously via `State.setState`. In both cases, lifecycle methods are invoked in response to changes to the element tree (for example, `StatefulElement.update` will invoke `State.didUpdateWidget`).
- If a component element rebuilds, the old element and new widget will still be paired, if possible. This allows `Element.update` to be used instead of `Element.inflateWidget`. Consequently, descendant render objects may be updated instead of recreated. Provided that the render object’s properties weren’t changed, this will likely circumvent the rest of the rendering process.

How do element dependencies work (inheritance)?

- Elements are automatically constructed with a hashmap of ancestor inherited widgets; thus, if a dependency is eventually added (via `Element.inheritFromWidgetOfExactType`), it is not necessary to walk up the tree. When elements express a dependency, that dependency is added to the element’s map

via `Element.inheritFromElement` and communicated to the inherited ancestor via `InheritedElement.updateDependency`.

- Locating ancestor render objects / widgets / states that do not utilize `InheritedElement` requires a linear walk up the tree.
- `InheritedElement` is a subclass of `ProxyElement`, which introduces a notification mechanism so that descendents can be notified after `ProxyElement.update` (via `ProxyElement.updated` and `ProxyElement.notifyClients`). This is useful for notifying dependant elements when dependencies change so that they can be marked dirty.

Render Tree

What are the render object building blocks?

- `RenderObject` provides the basic infrastructure for modeling hierarchical visual elements in the UI and includes a generalized protocol for performing layout, painting, and compositing. This protocol is unopinionated, deferring to subclasses to determine the inputs and outputs of layout, how hit testing is performed (though all render objects are `HitTestTargets`), and even how the render object hierarchy is modeled.
- Constraints represent the immutable inputs to layout. They must indicate whether they offer no choice (`Constraints.isTight`), are the canonical representation (`Constraints.isNormalized`), and provide "==" and `hashCode` implementations.
- `ParentData` is opaque data stored in a child `RenderObject` by its parent. The only requirement is that `ParentData` instances implement a `ParentData.detach` method to react to their render object being removed from the tree.

How is the render object lifecycle managed?

- The `RendererBinding` establishes a `PipelineOwner` as part of `RendererBinding.initInstances` (i.e., the binding's "constructor"). The `PipelineOwner` is analogous to the `BuildOwner`, tracking which render objects need compositing bits, layout, painting, and semantics updates. Objects mark themselves "dirty" by adding themselves to lists, e.g., `PipelineOwner._nodesNeedingLayout`, `PipeplineOwner._nodesNeedingPaint`, etc. All dirty objects are later cleaned by the corresponding "flush" method: `PipelineOwner.flushLayout`, `PipelineOwner.flushPaint`, etc. These methods form the majority of the rendering pipeline and are invoked every frame via `RendererBinding.drawFrame`.
- Whenever an object marks itself dirty, it will generally also invoke `PipelineOwner.requestVisualUpdate` to schedule a frame and update the UI. This calls the `PipelineOwner.onNeedVisualUpdate` handler which schedules a frame via `RendererBinding.ensureVisualUpdate` and `SchedulerBinding.scheduleFrame`. The rendering pipeline (as facilitated by `RendererBinding.drawFrame` or

`WidgetsBinding.drawFrame`) will invoke the `PipelineOwner`'s various flush methods to drive each stage.
- When the render object is attached to its `PipelineOwner`, it will mark the render object as dirty in all necessary dimensions (e.g., `RenderObject.markNeedsLayout`, `RenderObject.markNeedsPaint`). Generally, all are invoked since the internal flags that are consulted are initialized to true (e.g., `RenderObject._needsLayout`, `RenderObject._needsPaint`).

How is the render tree structured?

- The render tree consists of `AbstractNode` instances (`RenderObject` is a subclass). When a child is added or removed, `RenderObject.adoptChild` and `RenderObject.dropChild` must be called, respectively. When altering the depth of a child, `RenderObject.redepthChild` must be called to keep `RenderObject.depth` in sync.
- Every node also needs to use `RenderObject.attach` and `RenderObject.detach` to set and clear the pipeline owner (`RenderObject.owner`), respectively. Every `RenderObject` has a parent (`RenderObject.parent`) and an attached state (`RenderObject.attached`).
- Parent data is a `ParentData` subclass optionally associated with a render object. By convention, the child is not to access this data, though a protocol is free to alter this rule (but shouldn't). When adding a child, the parent data is initialized by calling `RenderObject.setupParentData`. The parent data may then be mutated in, e.g., `RenderObject.performLayout`.
- All `RenderObjects` implement the visitor pattern via `RenderObject.RenderObject.visitChildren` which invokes a `RenderObjectVisitor` once per child.

What is necessary when altering the child model?

- When adding or removing a child, call `RenderObject.adoptChild` and `RenderObject.dropChild` respectively.
- The parent's `RenderObject.attach` and `RenderObject.detach` methods must call their counterpart on each child.
- The parent's `RenderObject.redepthChildren` and `RenderObject.visitChildren` methods must recursively call `RenderObject.redepthChild` and the visitor argument on each child, respectively.

What are the render tree building blocks?

- `RenderObjectWithChildMixin` allocates storage for a single child within the object instance itself (`RenderObjectWithChildMixin.child`).
- `ContainerRenderObjectMixin` uses each child's parent data (which must implement `ContainerParentDataMixin`) to build a doubly linked list via `ContainerParentDataMixin.nextSibling` and `ContainerParentDataMixin.previousSibling` .
  - A variety of container-type methods are included: `ContainerRenderObjectMixin.insert` , `ContainerRenderObjectMixin.add` , `ContainerRenderObjectMixin.move` , `ContainerRenderObjectMixin.remove` , etc.
  - These adopt, drop, attach, and detach children appropriately. Additionally, when the child list changes, `RenderObject.markNeedsLayout` is invoked (as this may alter layout).

How do render objects manage layout?

- When a render object has been marked as dirty via `RenderObject.markNeedsLayout` , `RenderObject.layout` will be invoked with constraints as input and a size as output. Both the constraints and the size are implementation-dependent; the constraints must implement `Constraints` whereas the size is entirely arbitrary.
- If a parent depends on the child's geometry, it must pass the `parentUsesSize` argument to layout. The implementation of `RenderObject.markNeedsLayout` /

`RenderObject.sizedByParent` will need to call `RenderObject.markParentNeedsLayout` / `RenderObject.markParentNeedsLayoutForSizedByParentChange`, respectively.
- `RenderObjects` that solely determine their sizing using the input constraints must set `RenderObject.sizedByParent` to true and perform all layout in `RenderObject.performResize`.
- Layout cannot depend on position (typically stored using parent data). The position, if applicable, is solely determined by the parent. However, some `RenderObject` subtypes may utilize additional out-of-band information when performing layout. If this information changes, and the parent used it during the last cycle, `RenderObject.markParentNeedsLayout` must be invoked.

How do render objects manage hit testing?

- `RenderView`'s child is a `RenderBox`, which must therefore define `RenderBox.hitTest`. To add a custom `RenderObject` to the render tree, the top level `RenderView` must be replaced (which would be a massive undertaking) or a `RenderBox` adapter added to the tree. It is up to the implementer to determine how `RenderBox.hitTest` is adapted to a custom `RenderObject` subclass; indeed, that render object can implement any manner of `hitTest`-like method of its choosing. Note that all `RenderObjects` are `HitTestTargets` and therefore will receive pointer events via `HitTestTarget.pointerEvent` once registered via `HitTestEntry`.

How are render objects composited into layers?

- Render objects track a "`needsCompositing`" bit (as well as an "`alwaysNeedsCompositing`" bit and a "`isRepaintBoundary`" bit, which are consulted when updating "`needsCompositing`"). This bit indicates to the framework that a render object will paint into its own layer.
- This bit is not set directly. Instead, it must be marked dirty whenever it might possibly change (e.g., when adding or removing a child).

`RenderObject.markNeedsCompositingBitsUpdate` walks up the render tree, marking objects as dirty until it reaches a previously dirtied render object or a repaint boundary.
- Later, just before painting, `PipelineOwner.flushCompositingBits` visits all dirty render objects, updating the " `needsCompositing` " bit by walking down the tree, looking for any descendent that needs compositing. This walk stops upon reaching a repaint boundary or an object that always needs compositing. If any descendent node needs compositing, all nodes along the walk need compositing, too.
- It is important to create small "repaint sandwiches" to avoid introducing too many layers. [?]
- Composited render objects are associated with an optional `ContainerLayer` . For render objects that are repaint boundaries, `RenderObject.layer` will be `OffsetLayer` . In either case, by retaining a reference to a previously used layer, `Flutter` is able to better utilize retained rendering – i.e., recycle or selectively update previously rasterized bitmaps. This is possible because `Layers` preserve a reference to any underlying `EngineLayer` , and `SceneBuilder` accepts an " `oldLayer` " argument when building a new scene.

How do render objects handle transformations?

- Visual transforms are implemented in `RenderObject.paint` . The same transform is applied logically (i.e., when hit testing, computing semantics, mapping coordinates, etc) via `RenderObject.applyPaintTransform` . This method applies the child transformation to the provided matrix.
- `RenderObject.transformTo` chains paint transformation matrices mapping from the current coordinate space to the provided ancestor's coordinate space. [`WIP`]

What other protocols are implemented in the framework?

- `RenderBox` , `RenderSliver` .

Box Model

What are the render box building blocks?

- `RenderBox` models a box in 2D cartesian coordinates with a width, height, and offset. The origin of the box is the top-left corner (0,0), with the bottom-right corner corresponding to (width, height).
- `BoxParentData` stores a child's position in the parent's coordinate space. It is opaque to the child by convention.
- `BoxConstraints` provide cartesian constraints in the form of a maximum and minimum width and height between 0 and infinity, inclusive. Constraints are satisfied for all dimensions simultaneously within the given inclusive range.
  - Constraints are normalized when min is less or equal to max.
  - Constraints are tight when max and min are equal.
  - Constraints are loose when min is 0, even if max is also 0 (loose and tight).
  - Constraints are bounded when max is not infinite.
  - Constraints are unbounded when max is infinite.
  - Constraints are expanding when tightly infinite.
  - Constraints are infinite when min is infinite (max must also be infinite; thus, this is also expanding).
- `BoxHitTestResult` is an aggregator that collects the entries associated with a single hit test query. `BoxHitTestResult` includes several box-specific helpers, i.e., `BoxHitTestResult.addWithPaintOffset`, `BoxHitTestResult.addWithPaintTransform`, etc.
- `BoxHitTestEntry` represents a box that was hit during hit testing. It captures the position of the collision in local coordinates (`BoxHitTestEntry.localPosition`).

What are the render box tree building blocks?

- `RenderProxyBox` supports a single `RenderBox` child, delegating all methods to said child. `RenderShiftedBox` is identical, but offsets the child using `BoxParentData.offset` and requires a layout implementation. Both are built with `RenderObjectWithChildMixin`.

- `RenderObjectWithChildMixin` and `ContainerRenderObjectMixin` can both be used with a type argument of `RenderBox`. The `ContainerRenderObjectMixin` accepts a parent data type argument, as well; `ContainerBoxParentData` satisfies the type constraints and supports box parent data.
- `RenderBoxContainerDefaultsMixin` adds useful defaults to `ContainerRenderObjectMixin` for render boxes. This includes support for hit testing and painting children, as well as computing baselines.

How do render boxes handle layout?

- `RenderBox` layout is the same as `RenderObject` layout, with the input being `BoxConstraints` and the output being `RenderBox.size` (Size). Layout typically also sets the position (`BoxParentData.offset`) of any children, though the child may not read this data.
- The `RenderBox` protocol includes support for intrinsic dimensions (a size that is computed outside of the layout protocol) as well as baselines (the bottom of the box for vertical alignment). `RenderBox` instances track changes to these values and whether the parent has queried them; if so, when that box is marked as needing layout, the parent is marked as well.
- Marking a render box as needing layout implies that the box needs paint. Building a render box implies both.
- Render boxes can have non-box children. In this case, the constraints passed from the parent to the child will need to be adapted from `BoxConstraints` to the new protocol.

How do render boxes handle paint?

- Painting is the same as `RenderObject` painting. The offset provided corresponds to the origin of the render object in the canvas's coordinate space (which may not be the same).
- If the render box applies a transform when painting, including painting at a different offset than the one provided, `RenderBox.applyPaintTransform` must apply the same

transformation. `RenderBox.globalToLocal` and `RenderBox.localToGlobal` rely on this transform to map cartesian coordinates.
    - By default, `RenderBox.applyPaintTransform` will apply the child's offset to the matrix as a translation.

How do render boxes handle hit testing?

- All `RenderBoxes` must implement `RenderBox.hitTest`, which by default delegates to `RenderBox.hitTestChildren` and `RenderBox.hitTestSelf`, in that order. Note that items added to the `BoxHitTestResult` first are treated as being on top of those added later. All entries in the `BoxHitTestResult` are fed events from the corresponding event stream via `RenderBox.handleEvent`.

What are intrinsic dimensions?

- Conceptually, the intrinsic dimensions of a box are its natural dimensions – the size it "wants" to be. Given that this definition is subjective, it is left as an implementation detail what exactly this means for a given render object. Note that intrinsic dimensions are often defined in terms of child intrinsic dimensions and are therefore expensive to calculate (typically traversing an entire subtree).
- Intrinsic dimensions often do not match the dimensions resulting from layout (except when using the `IntrinsicHeight` and `IntrinsicWidth` widgets, which attempt to layout their child using its intrinsic dimensions).
    - Intrinsic dimensions are generally ignored unless one of these widgets are used (or another widget that explicitly incorporates intrinsic dimensions into its own layout).
- The render box model describes intrinsic dimensions in terms of minimum and maximum values for width and height.
    - Minimum intrinsic width is the smallest width before the box cannot paint correctly without clipping.
        - Intuition: making the box thinner would clip its contents.

        - If width is determined by height (ignoring constraints), the incoming height (which may be infinite, i.e., unconstrained) should be used. Otherwise, ignore the height.
    - Minimum intrinsic height is the same concept for height.
    - Maximum intrinsic width is the smallest width such that further expansion would not reduce minimum intrinsic height (for that width).
        - Intuition: making the box wider won't help fit more content.
        - If width is determined by height (ignoring constraints), the incoming height (which may be infinite, i.e., unconstrained) should be used. Otherwise, ignore the height.
    - Maximum intrinsic height is the same concept for height.
- The specific meaning of intrinsic dimensions depends on the implementation.
    - Text is width-in-height-out.
        - Max intrinsic width: the width of the string without line breaks (increasing the width would not shrink the preferred height).
        - Min intrinsic width: the width of the widest word (decreasing the width would clip the word or cause an invalid break).
        - Height intrinsics derived from width intrinsics for the given width.
    - Viewports ignore incoming constraints and aggregate child dimensions without clipping.
    - Aspect ratio boxes are entirely determined when one dimension is provided.
        - Use the incoming dimension to compute the other. If unconstrained, recurse.
    - When intrinsic dimensions cannot be computed or are too expensive, return zero.

What are baselines?

- Baselines are used to vertically align children independent of font size, padding, etc. When a real baseline isn't available, the bottom of the render box is used. Baselines are expressed as an offset from the box's y-coordinate; if there are multiple children, the first sets the baseline.
- Only a render box's parent can invoke `RenderBox.getDistanceToBaseline`, and only after that render box has been laid out in the parent's `RenderBox.performLayout` method.

Rendering

Layout

How is layout optimized (what are relayout boundaries)?

- If a child is not dirty and the constraints are unchanged, it needn't be laid out, cutting off the walk.
- If a parent does not use a child's size, it needn't be laid out with the child; the child will necessarily conform to the original input constraints, and thus nothing changes for the parent.
- If a parent passes tight constraints to its child, it needn't be laid out with the child; the child cannot change size.
- If a child is "`sizedByParent`" (the constraints solely determine the size), and the same constraints are passed, the parent needn't be laid out with the child; the child cannot change size.

How can UI be built in response to layout?

- `RenderObject.invokeLayoutCallback` allows a builder callback to be invoked during the layout process. Only subtrees that are still dirty may be manipulated which ensures that nodes are laid out exactly once. This allows children to be built on-demand during layout (needed for viewports) and nodes to be moved (needed for global key reparenting).
- Several invariants ensure that this is valid:
  - Building is unidirectional – information only flows down the tree, visiting each node once.
  - Layout visits each dirty node once.
  - Building below the current node cannot invalidate earlier layout; build information only flows down the tree.
  - Building after a node is laid out cannot invalidate that node; nodes are never revisited.
  - Therefore, it is safe to build within the unvisited subtree rooted at a given render object.

What are the layout primitives for single children?

- `Container` provides a consistent interface to a number of more fundamental layout building blocks. Unless otherwise specified, each box is sized to match its child.
- `ConstrainedBox` applies an extra set of constraints in addition to the incoming constraints. The extra constraints are clamped to the incoming constraints (via `BoxConstraints.enforce`) and thus may not violate them. The box becomes as small as possible without a child.
- `SizedBox` delegates to `ConstrainedBox`, providing tight constraints for the specified dimensions. The result is that the child will be sized to these dimensions subject to the incoming constraints. The box is still sized even without a child.
- `UnconstrainedBox` lays out its child with unbounded constraints (in one or more dimensions). It then sizes itself to match the child subject to incoming constraints (i.e., to prevent any actual overflow). An alignment is applied if the child is a different size than its parent. If there would have been overflow, the child's painting is clipped accordingly. The box becomes as small as possible without a child.
- `OverflowBox` allows one or more components of the incoming constraints to be completely overwritten. The child is laid out using the altered constraints and therefore may overflow. An alignment is applied if the child is a different size than its parent. In all other ways, this box delegates to the child.
- `SizedOverflowBox` assumes a specific size, subject to the incoming constraints. Its child is laid out using the original constraints, however, and therefore may overflow the parent (e.g., if the incoming constraints permit a size of up to 100x100, and the parent specifies a size of 50x50, the child can select a larger size, overflowing its parent). Since this box has an explicit size, its given intrinsic dimensions matching this size.
- `FractionallySizedBox` sizes its child in proportion to the incoming constraints; it sizes itself to the child, but without violating the original constraints. In particular, the maximum constraints are multiplied by a corresponding factor to obtain a tight bound (if a factor isn't specified, that dimension's constraints are left unchanged). The child is laid out using the new constraints and may therefore overflow the parent. The parent is sized to the child subject to the incoming constraints, and therefore cannot overflow; note that painting and hit testing will not be automatically clipped. Intrinsic dimensions are defined as the child's intrinsic dimension scaled by the corresponding factor.
- `LimitedBox` applies a maximum width or height only when the corresponding constraint is unbounded. That is, when the incoming constraints are forwarded to the child, the maximum constraint in each dimension is set to a specific value if it is

otherwise unbounded. This is typically used to place expanding children into containers with infinite dimensions (e.g., viewports).

- `Padding` applies spacing around its child (via `EdgeInsetsGeometry`). This is achieved by providing the child with constraints deflated by the padding. The parent assumes the size of the child (which is zero if there is no child) inflated by the padding.
  - `EdgeInsetsGeometry` describes offsets in the four cardinal directions. `EdgeInsets` specifies the offsets with absolute positioning (e.g., left, top, right, bottom) whereas `EdgeInsetsDirectional` is text-orientation sensitive (e.g., start, top, end, bottom). Both allow offsets to be combined, subtracted, inflated, and deflated, as well as queried along an axis.
  - `Padding` defines intrinsic dimensions by reducing the incoming dimension by the total applicable padding before querying the child; the total padding is then added to the resulting dimension.
- `Transform` applies a transformation matrix during painting. It does not alter layout, delegating entirely to `RenderProxyBox` for these purposes. If the transform can solely be described as a translation, it delegates to its parent for painting, as well. Otherwise, it composites the corresponding transformation before painting its child. Optionally, this transform may be applied before hit testing (via `Transform.transformHitTests`).

What higher level widgets support layout of single children?

- `FittedBox` applies a `BoxFit` to its child. `BoxFit` specifies how a child box is to be inscribed within a parent box (e.g., `BoxFit.fitWidth` scales the width to fit the parent and the height to maintain the aspect ratio, possibly overflowing the parent; `BoxFit.scaleDown` aligns the child within the parent then scales to ensure that the child does not overflow). The child is laid out without constraint, then scaled to be as large as possible while maintaining the original aspect ratio and respecting the incoming constraints. The box fit determines a transformation matrix that is applied during painting.
  - The box fit is calculated by `applyBoxFit`, which considers each box's size without respect to positioning. The result is a `FittedSizes` instance, which describes a region within the child (the source) and a region within the parent (the destination). In particular, the source measures the portion of the child visible within the parent (e.g., if this is smaller than the child, the child will be clipped). The destination

measures the portion of the parent occupied by the child (e.g., if this is smaller than the parent, a portion of the parent will remain visible).
    - The sizes are transformed into rectangles using the current alignment (via `Alignment.inscribe`). Finally, a transformation is composed to translate and scale the child relative to the parent as per the calculated rectangles.
- `Baseline` aligns a child such that its baseline is coincident with its own (as specified). The child's baseline is obtained via `RenderBox.getDistanceToBaseline` and an offset calculated to ensure the child's baseline is coincident with the target line (itself expressed as an offset from the box's top). As a result, though the parent is sized to contain the child, the offset may cause the parent to be larger (e.g., if the child is shifted below the parent's top).
- `Align` allows a child to be aligned within a parent according to a relative position specified via `AlignmentGeometry`. During layout, the child is positioned according to the alignment (via `RenderAligningShiftedBox.alignChild`, which invokes `AlignmentGeometry.alongOffset`; this maps the alignment to logical coordinates using simple geometry). Note that the child must undergo layout beforehand since its size is needed to compute its center; the incoming constraints are loosened before being forwarded to the child. Next, the parent is sized according to whether a width or height factor was specified; if so, the corresponding dimension will be set to a multiple of that of the child (subject to the incoming constraints). Otherwise, it will expand to fill the constraints. If the incoming constraints are unbounded, the parent will instead shrinkwrap the child.
    - `Alignment` is a subclass of `AlignmentGeometry` that describes a point within a container using units relative to the container's size and center. The child's center is positioned such that it is coincident with this point. Each unit is measured as half the container's corresponding dimension (e.g., the container is effectively 2 units wide and 2 units tall). The center of the container corresponds to (0, 0), the top left to (-1, -1), and the bottom right to (1, 1). Units may overflow this range to specify different multiple of the parent's size.
    - `AlignmentDirectional` is identical but flips the x-axis for right-to-left languages.
    - `FractionalOffset` represents a relative position as a fraction of the container's size, measured from its top left corner. The child's top-left corner is positioned such that it is coincident with this point. The center of the container corresponds to (`0.5`, `0.5`), the top left to (0, 0), and the bottom right to (1, 1).
- `IntrinsicHeight` and `IntrinsicWidth` integrate the intrinsic dimension and layout protocols. Intrinsic dimensions reflect the natural size of an object; this definition varies

based on what the object actually represents. Prior to laying out, the child's intrinsic dimension is queried (e.g., via `RenderBox.getMaxIntrinsicWidth`). Next, the incoming constraints are tightened to reflect the resulting value. Finally, the child is laid out using the tight constraints. In this way, the child's intrinsic size is used in determining its actual size (the two will only match if permitted by the incoming constraints).

    - Boxes often define their intrinsic dimensions recursively. Consequently, an entire subtree may need to be traversed to produce one set of dimensions. This can quickly lead to polynomial complexity, particularly if intrinsic dimensions are computed several times during layout (which itself traverses the render tree).

What higher level widgets support layout of multiple children?

- `Stack`
- `LayoutBuilder` makes it easy to take advantage of interleaving the build / layout phase via callbacks. The callback has access to the incoming constraints (i.e., because the framework has completed building and is currently performing layout) but may still build additional UI (via `RenderObject.invokeLayoutCallback`). Once building is complete, layout continues through the subtree and beyond.
- `CustomSingleChildLayout` allows its layout and its child's layout to be customized without implementing a new render box. Deferring to a delegate, the associated `RenderCustomSingleChildLayout` invokes `SingleChildLayoutDelegate.getSize` to size the parent, then `SingleChildLayoutDelegate.getConstraintsForChild` to constrain the child, and finally `SingleChildLayoutDelegate.getPositionForChild` to position the child.
- `CustomMultiChildLayout` is similar, but supports multiple children. To identify children, each must be tagged with a `LayoutId`, a `ParentDataWidget` that stores an identifier in each child's `MultiChildLayoutParentData` (recall that render objects establish their children's parent data type). `MultiChildLayoutDelegate` behaves similarly to its single-child variant; override `MultiChildLayoutDelegate.performLayout` to size and position each child via `MultiChildLayoutDelegate.layoutChild` and

`MultiChildLayoutDelegate.positionChild`, respectively (the latter must be called once per child in any order).

- `ConstrainedLayoutBuilder`
- `LayoutBuilder`
- `SliverLayoutBuilder`

Painting

What are the painting building blocks?

- `Path` describes a sequence of potentially disjoint movements on a plane. Paths tracks a current point as well as one or more subpaths (created via `Path.moveTo`). Subpaths may be closed (i.e., the first and last points are coincident), open (i.e., the first and last points are distinct), or self intersecting (i.e., movements within the path intersect). Paths incorporate lines, arcs, beziers, and more; each operation begins at the current point and, once complete, defines the new current point. The current point begins at the origin. Paths can be queried (via `Path.contains`), transformed (via `Path.transform`), and merged (via `Path.combine`, which accepts a `PathOperation`).
  - `PathFillType` defines the criteria determining whether a point is contained by the path. `PathFillType.evenOdd` casts a ray from the point outward, summing the number of edge crossings; an odd count indicates that the point is internal. `PathFillType.nonZero` considers the path's directionality. Again casting a ray from the point outward, this method sums the number of clockwise and counterclockwise crossings. If the counts aren't equal, the point is considered to be internal.
- `Canvas` represents a graphical context supporting a number of drawing operations. These operations are captured by an associated `PictureRecorder` and, once finalized, transformed into a `Picture`. The `Canvas` is associated with a clip region (i.e., an area within which painting will be visible), and a current transform (i.e., a matrix to be applied to any drawing), both managed using a stack (i.e., clip regions and transforms can be pushed and popped as drawing proceeds). Any drawing outside of the canvas's culling box (`cullRect`) may be discarded; by default, however, affected pixels are retained. Many operations accept a `Paint` parameter which describes how the drawing will be composited (e.g., the fill, stroke, blending, etc).
  - Canvas exposes a rich API for drawing. The majority of these operations are implemented within the engine.
    - `Canvas.clipPath`, `Canvas.clipRect`, etc., refine (i.e., reduce) the clip region. These operations compute the intersection of the current clip region and the provided geometry to define a new clip region. The clip region can be anti-aliased to provide a gradual blending.

        - `Canvas.translate`, `Canvas.scale`, `Canvas.transform`, etc., alter the current transformation matrix (i.e., by multiplying it by an additional transform). The former methods apply standard transformations, whereas the latter applies an arbitrary 4x4 matrix (specified in column-major order).
        - `Canvas.drawRect`, `Canvas.drawLine`, `Canvas.drawPath`, etc., perform fundamental drawing operations.
        - `Canvas.drawImage`, `Canvas.drawAtlas`, `Canvas.drawPicture`, etc., copy pixels from a rendered image or recorded picture into the current canvas.
        - `Canvas.drawParagraph` paints text into the canvas (via `Paragraph._paint`).
        - `Canvas.drawVertices`, `Canvas.drawPoints`, etc., describe solids using a collection of points. The former constructs triangles from a set of vertices (Vertices) and a vertex mode (`VertexMode`); this mode describes how vertices are composed into triangles (e.g., `VertexMode.triangles` specifies that each sequence of three points defines a new triangle). The resulting triangles are filled and composited using the provided `Paint` and `BlendMode`. The latter paints a set of points using a `PointMode` describing how the collection of points is to be interpreted (e.g., as defining line segments or disconnected points).
    - The save stack tracks the current transformation and clip region. New entries can be pushed (via `Canvas.save` or `Canvas.saveLayer`) and popped (via `Canvas.restore`). The number of items in this stack can also be queried (via `Canvas.getSaveCount`); there is always at least one item on the stack. All drawing operations are subject to the transform and clip at the top of the stack.
        - All drawing operations are performed sequentially (by default or when using `Canvas.save` / `Canvas.restore`). If the operation utilizes blending, it will be blended immediately after completing.
        - `Canvas.saveLayer` allows drawing operations to be grouped together and composited as a whole. Each individual operation will still be blended within the saved layer; however, once the layer is completed, the composite drawing will be blended as a whole using the provided `Paint` and bounds.
            - For example, an arbitrary drawing can be made consistently translucent by first painting it using an opaque fill, and then blending the resulting layer with the canvas. If instead each component of the drawing were individually blended, overlapping regions would appear darker.
            - This is particularly useful for antialiased clip regions (i.e., regions that aren't pixel aligned). Without layers, any operations intersecting the clip

would needed to be antialiased (i.e., blended with the background). If a subsequent operation intersects the clip at this same point, it would be blended with both the background and the previous operation; this produces visual artifacts (e.g., color bleed). If both operations were combined into a layer and composited as a whole, only the final operation would be blended.
        - Note that though this doesn't introduce a new framework layer, it does cause the engine to switch to a new rendering target. This is fairly expensive as it flushes the `GPU`'s command buffer and requires data to be shuffled.
- Paint describes how a drawing operation is to be applied to the canvas. In particular, it specifies a number of graphical parameters including the color to use when filling or stroking lines (`Paint.color`, `Paint.colorFilter`, `Paint.shader`), how new painting is to be blended with old painting (`Paint.blendMode`, `Paint.isAntiAlias`, `Paint.maskFilter`), and how edges are to be drawn (`Paint.strokeWidth`, `Paint.strokeJoin`, `Paint.strokeCap`). Fundamental to most drawing is whether the result is to be stroked (e.g., drawn as an outline) or filled; `Paint.style` exposes a `PaintingStyle` instance specifying the mode to be used.
    - If stroking, `Paint.strokeWidth` is measured in logical pixels orthogonal to the path being painted. A value of `0.0` will cause the line to be rendered as thin as possible ("hairline rendering").
        - Any lines that are drawn will be capped at their endpoints according to a `StrokeCap` value (via `Paint.strokeCap`; `StrokeCap.butt` is the default and does not paint a cap). Caps extend the overall length of lines in proportion to the stroke width.
        - Discrete segments are joined according to a `StrokeJoin` value (via `Paint.strokeJoin`; `StrokeJoin.miter` is the default and extends the original line such that the next can be drawn directly from it). A limit may be specified to prevent the original line from extending too far (via `Paint.strokeMiterLimit`; once exceeded, the join reverts to `StrokeJoin.bevel`).
    - `ColorFilter` describes a function mapping from two input colors (e.g., the paint's color and the destination's color) to a final output color (e.g., the final composited color). If a `ColorFilter` is provided, it overrides both the paint color and shader; otherwise, the shader overrides the color.

    - `MaskFilter` applies a filter (e.g., a blur) to the drawing once it is complete but before it is composited. Currently, this is limited to a gaussian blur.
- Shader is a handle to a Skia shader utilized by the engine. Several are exposed within the framework, including `Gradient` and `ImageShader`. These are analogous, with the former generating pixels by smoothly blending colors and the latter reading them directly from an image. Both support tiling so that the original pixels can be extended beyond their bounds (a different `TileMode` may be specified in either direction); `ImageShader` also supports an arbitrary matrix to be applied to the source image.

Compositing

How are composited visuals represented?

- Scene is an opaque, immutable representation of composited UI. Each frame, the rendering pipeline produces a scene that is uploaded to the engine for rasterization via `Window.render`. A scene can also be rasterized into an `Image` via `Scene.toImage`. Generally, a `Scene` encapsulates a stack of rendering operations (e.g., clips, transforms, effects) and graphics (e.g., pictures, textures, platform views).
- `SceneBuilder` is lowest level mechanism (within the framework) for assembling a scene. `SceneBuilder` manages a stack of rendering operations (via `SceneBuilder.pushClipRect`, `SceneBuilder.pushOpacity`, `SceneBuilder.pop`, etc) and associated graphics (via `SceneBuilder.addPicture`, `SceneBuilder.addTexture`, `SceneBuilder.addRetained`, etc). As operations are applied, `SceneBuilder` produces `EngineLayer` instances that it both tracks and returns. These may be cached by the framework to optimize subsequent frames (e.g., updating an old layer rather than recreating it, or reusing a previously rasterized layer). Once complete, the final `Scene` is obtained via `SceneBuilder.build`.
  - Drawing operations are accumulated in a `Picture` (via `PictureRecorder`) which is added to the scene via `SceneBuilder.addPicture`.
  - External textures (represented by a texture ID established using the platform texture repository) are added to the scene via `SceneBuilder.addTexture`. This supports a "freeze" parameter so that textures may be paused when animation is paused.
  - Previously rasterized `EngineLayers` can be reused as an efficiency operation via `SceneBuilder.addRetained`. This is known as retained rendering.
- Picture is an opaque, immutable representation of a sequence of drawing operations. These operations can be added directly to a `Scene` (via `SceneBuilder.addPicture`) or combined into another picture (via `Canvas.drawPicture`). Pictures can also be rasterized (via `Picture.toImage`) and provide an estimate of their memory footprint.
- `PictureRecorder` is an opaque mechanism by which drawing operations are sequenced and transformed into an immutable `Picture`. Picture recorders are generally managed via a `Canvas`. Once recording is ended (via `PictureRecorder.endRecording`), a final `Picture` is returned and the `PictureRecorder` (and associated `Canvas`) are invalidated.

How are layers represented?

- Layer represents one slice of a composited scene. Layers are arranged into a hierarchy with each node potentially affecting the nodes below it (i.e., by applying rendering operations or drawing graphics). Layers are mutable and freely moved around the tree, but must only be used once. Each layer is composited into a `Scene` using `SceneBuilder` (via `Layer.addToScene`); adding the root layer is equivalent to compositing the entire tree (e.g., layers add their children). Note that the entire layer tree must be recomposited when mutated; there is no concept of dirty states.
  - `SceneBuilder` provides an `EngineLayer` instance when a layer is added. This handle can be stored in `Layer.engineLayer` and later used to optimize compositing. For instance, rather than repeating an operation, `Layer.addToScene` might pass the engine layer instead (via `Layer._addToSceneWithRetainedRendering`), potentially reusing the rasterized texture from the previous frame. Additionally, an engine layer (i.e., "`oldLayer`") can be specified when performing certain rendering operations to allow these to be implemented as inline mutations.
  - Layers provide support for retained rendering via `Layer.markNeedsAddToScene`. This flag tracks whether the previous engine layer can be reused during compositing (i.e., whether the layer is unchanged from the last frame). If so, retained rendering allows the entire subtree to be replaced with a previously rasterized bitmap (via `SceneBuilder.addRetained`). Otherwise, the layer must be recomposited from scratch. Once a layer has been added, this flag is cleared (unless the layer specifies that it must always be composited).
    - All layers must be added to the scene initially. Once added, the cached engine layer may be used for retained rendering. Other layers disable retained rendering altogether (via `Layer.alwaysNeedsAddToScene`). Generally, when a layer is mutated (i.e., by modifying a property or adding a child), it must be re-added to the scene.
    - If a layer must be added to the scene, all ancestor layers must be added, too. Retained rendering allows a subtree to be replaced with a cached bitmap. If a descendent has changed, this bitmap becomes invalid and therefore the ancestor layer must also be added to the scene (via `Layer.updateSubtreeNeedsAddToScene`).
      - Note that this property isn't enforced during painting; the layer tree is only made consistent when the scene is composited (via

`ContainerLayer.buildScene`).
        - A layer's parent must be re-added when it receives a new engine layer [?]
        - Generally, only container layers use retained rendering?
    - Metadata can be embedded within a layer for later retrieval (via `Layer.find` and `Layer.findAll`).
- `ContainerLayer` manages a list of child layers, inserting them into the composited scene in order. The root layer is a subclass of `ContainerLayer` (`TransformLayer`); thus, `ContainerLayer` is responsible for compositing the full scene (via `ContainerLayer.buildScene`). This involves making retained rendering state consistent (via `ContainerLayer.updateSubtreeNeedsAddToScene`), adding all the children to the scene (via `ContainerLayer.addChildrenToScene`), and marking the container as no longer needing to be added (via `Layer._needsAddToScene`). `ContainerLayer` implements a number of child operations (e.g., `ContainerLayer.append`, `ContainerLayer.removeAllChildren`), and multiplexes most of its operations across these children (e.g., `ContainerLayer.find`; note that later children are "above" earlier children).
    - `ContainerLayer` will use retained rendering for its children provided that they aren't subject to an offset (i.e., are positioned at the layer's origin). This is the entrypoint for most retained rendering in the layer tree.
    - `ContainerLayer.applyTransform` transforms the provided matrix to reflect how a given child will be transformed during compositing. This method assumes all children are positioned at the origin; thus, any layer offsets must be removed and expressed as a transformation (via `SceneBuilder.pushTransform` or `SceneBuilder.pushOffset`). Otherwise, the resulting matrix will be inaccurate.
- `OffsetLayer` is a `ContainerLayer` subclass that supports efficient repainting (i.e., repaint boundaries). Render objects defining repaint boundaries correspond to `OffsetLayers` in the layer tree. If the render object doesn't need to be repainted or is simply being translated, its existing offset layer may be reused. This allows the entire subtree to avoid repainting.
    - `OffsetLayer` applies any offset as a top-level transform so that descendant nodes are composited at the origin and therefore eligible for retained rendering.
    - `OffsetLayer.toImage` rasterizes the subtree rooted at this layer (via `Scene.toImage`). The resulting image, consisting of raw pixel data, is rendered into a bounding rectangle using the specified pixel ratio (ignoring the device's screen density).

- `TransformLayer` is a `ContainerLayer` subclass that applies an arbitrary transform; it also happens to (typically) be the root layer corresponding to the `RenderView`. Any layer offset (via `Layer.addToScene`) or explicit offset (via `OffsetLayer.offset`) is removed and applied as a transformation to ensure that `TransformLayer.applyTransform` behaves consistently.
- `PhysicalModelLayer` is the engine by which physical lighting effects (e.g., shadows) are integrated into the composited scene. This class incorporates an elevation, clip region, background color, and shadow color to cast a shadow behind its children (via `SceneBuilder.pushPhysicalShape`).
- `AnnotatedRegionLayer` incorporates metadata into the layer tree within a given bounding rectangle. An offset (defaulting to the origin) determines the rectangle's top left corner and a size (defaulting to the entire layer) represents its dimensions. Hit testing is performed by `AnnotatedRegionLayer.find` and `AnnotatedRegionLayer.findAll`. Results appearing deeper in the tree or later in the child list take precedence.
- `FollowerLayer` and `LeaderLayer` allow efficient transform linking (e.g., so that one layer appears to scroll with another layer). These layers communicate via `LayerLink`, passing the leader's transform (including `layerOffset`) to the follower. The follower uses this transform to appear where the follower is rendered. `CompositedTransformFollower` and `CompositedTransformTarget` widgets create and manage these layers.
- There are a variety of leaf and interior classes making up the layer tree.
  - Leaf nodes generally extend `Layer` directly. `PictureLayer` describes a single `Picture` to be added to the scene. `TextureLayer` is analogous, describing a texture (by ID) and a bounding rectangle; `PlatformViewLayer` is nearly identical, applying a view instead of a texture.
  - Interior nodes generally extend `ContainerLayer` or one of its subclasses. `OffsetLayer` is key to implementing efficient repainting. `ClipPathLayer`, `ClipRectLayer`, `ColorFilterLayer`, `OpacityLayer`, etc., apply the corresponding effect by pushing a scene builder state, adding all children (via `ContainerLayer.addChildrenToScene`, potentially utilizing retained rendering), then popping the state.
- `EngineLayer` and its various specializations (e.g., `OpacityEngineLayer`, `OffsetEngineLayer`) represent opaque handles to backend layers. `SceneBuilder`

produces the appropriate handle for each operation it supports. These may then be used to enable retained rendering and inline updating.

What are the compositing building blocks?

- Canvas provides an interface for performing drawing operations within the context of a single `PictureRecorder`. That is, all drawing performed by a `Canvas` is constrained to a single `Picture` (and `PictureLayer`). Rendering and drawing operations spanning multiple layers are supported by `PaintingContext`, which manages `Canvas` lifecycle based on the compositing requirements of the render tree.
- `PaintingContext.repaintCompositedChild` is a static method that paints a render object into its own layer. Once nodes are marked dirty for painting, they will be processed during the painting phase of the rendering pipeline (via `PipelineOwner.flushPaint`). This performs a depth-first traversal of all dirty render objects. Note that only repaint boundaries are ever actually marked dirty; all other nodes traverse their ancestor chain to find the nearest enclosing repaint boundary, which is then marked dirty. The dirty node is processed (via `PaintingContext._repaintCompositedChild`), retrieving or creating an `OffsetLayer` to serve as the subtree's container layer. Next, a new `PaintingContext` is created (associated with this layer) to facilitate the actual painting (via `PaintingContext._paintWithContext`).
    - Layers can be attached and detached from the rendering pipeline. If a render object with a detached layer is found to be dirty (by `PipelineOwner.flushPaint`), the ancestor chain is traversed to find the nearest repaint boundary with an attached (or as yet uncreated) layer. This node is marked dirty to ensure that, when the layer is reattached, it will be repainted as expected (via `RenderObject._skippedPaintingOnLayer`).
- `PaintingContext` provides a layer of indirection between the paint method and the underlying canvas to allow render objects to adapt to changing compositing requirements. That is, a render object may introduce a new layer in some cases and re-use an existing layer in others. When this changes, any ancestor render objects will need to adapt to the possibility of a descendent introducing a new layer (via `RenderObject.needsCompositing`); in particular, this requires certain operations to be implemented by introducing their own layers (e.g., clipping). `PaintingContext`

manages this process and provides a number of methods that encapsulate this decision (e.g., `PaintingContext.pushClipPath`). `PaintingContext` also ensures that children introducing repaint boundaries are composited into new layers (via `PaintingContext.paintChild`). Last, `PaintingContext` manages the `PictureRecorder` provided to the `Canvas`, appending picture layers (i.e., `PictureLayer`) whenever a recording is completed.

- `PaintingContext` is initialized with a `ContainerLayer` subclass (i.e., the container layer) to serve as the root of the layer tree produced via that context; painting never takes place within this layer, but is instead captured by new `PictureLayer` instances which are appended as painting proceeds.
- `PaintingContext` may manage multiple canvases due to compositing. Each `Canvas` is associated with a `PictureRecorder` and a `PictureLayer` (i.e., the current layer) for the duration of its lifespan. Until a new layer must be added (e.g., to implement a compositing effect or because a repaint boundary is encountered), the same canvas (and `PictureRecorder`) is used for all render objects.
  - Recording begins the first time the canvas is accessed (via `PaintingContext.canvas`). This allocates a new picture layer, picture recorder, and canvas instance; the picture layer is appended to the container layer once created.
  - Recording ends when a new layer is introduced directly or indirectly; at this point, the picture is stored in the current layer (via `PictureRecorder.endRecording`) and the canvas and current layer are cleared. The next time a child is painted, a new canvas, picture recorder, and picture layer will be created.
  - Composited children are painted using a new `PaintingContext` initialized with a corresponding `OffsetLayer`. If a composited child doesn't actually need to be repainted (e.g., because it is only being translated), the `OffsetLayer` is reused with a new offset.
- `PaintingContext` provides a compositing naive API. If compositing is necessary, new layers are automatically pushed to achieve the desired effect (via `PaintingContext.pushLayer`). Otherwise, the effect is implemented directly using the `Canvas` (e.g., via `PictureRecorder`).
  - Pushing a layer ends the current recording, appending the new layer to the container layer. A `PaintingContext` is created for the new layer (if present, the layer's existing children are removed since they're likely outdated). If provided, a painting function is applied using the new context; any painting is

contained within the new layer. Subsequent operations with the original `PaintingContext` will result in a new canvas, layer, and picture recorder being created.

- `ClipContext` is `PaintingContext`'s base class. Its primary utility is in providing support for clipping without introducing a new layer (e.g., methods suitable for render objects that do not need compositing).

What do the "needs compositing" bits do?

- Compositing describes the process by which layers are combined by the engine during rasterization. Within the context of the framework, compositing typically refers to the allocation of render objects to layers with respect to painting. Needing compositing does not imply that a render object will be allocated its own layer; instead, it indicates that certain effects must be implemented by introducing a new layer rather than modifying the current one (e.g., applying opacity or clipping). For instance, if a parent first establishes a clip and then paints its child, a decision must be made as to how that clip is implemented: (1) as part of the current layer (i.e., `Canvas.clipPath`), or (2) by pushing a `ContainerLayer` (i.e., `ClipPathLayer`). The "needs compositing" bit is how this information is managed.
  - Conceptually, this is because this node might eventually paint a descendent that pushes a new layer (e.g., because it is a repaint boundary). Thus, any painting that might have non-local consequences must be implemented in a way that would work across layers (i.e., via compositing).
  - Note that this is different than from marking a render object as being a repaint boundary. Doing this causes a new layer to be introduced when painting the child (via `PaintingContext.paintChild`). This invalidates the needs compositing bits for all the ancestors of the repaint boundary. This might cause some ancestors to introduce new layers when painting, but only if they utilize any non-local operations

---

How are "needs compositing" bits updated?

- `RenderObject.markNeedsCompositingBitsUpdate` marks a render object as requiring a compositing bit update (via `PipelineOwner`. `_nodesNeedingCompositingBitsUpdate`). If a node is marked dirty, all of its ancestors are marked dirty, too. As an optimization, this walk may be cut off if the current node or the current node's parent is a repaint boundary (or the parent is already marked dirty).
    - If a node's compositing bits need updating, it's possible that it will now introduce a new layer (i.e., it will need compositing). If so, all ancestors will need compositing, too, since they may paint a descendent that introduces a new layer. As described, certain non-local effects will need to be implemented via compositing.
    - The walk may be cut off at repaint boundaries since all ancestors must already have been marked as needing compositing.
    - Adding or removing children (via `RenderObject.adoptChild` and `RenderObject.dropChild`) might alter the compositing requirements of the render tree. Similarly, changing the `RenderObject.alwaysNeedsCompositing` bits would require that the bits be updated.
- During the rendering pipeline, `PipelineOwner.flushCompositingBits` updates all dirty compositing bits (via `RenderObject._updateCompositingBits`). A given node needs compositing if (1) it's a repaint boundary (`RenderObject.isRepaintBoundary`), (2) it's marked as always needing compositing (`RenderObject.alwaysNeedsCompositing`), or (3) any of its descendants need compositing.
    - This process walks all descendants that have been marked dirty, updating their bits according to the above policy. Note that this will typically only walk the path identified when the compositing bits were marked dirty.
    - If a node's compositing bit is changed, it will be marked dirty for painting (as painting may now require additional compositing).
- If a render object always introduces a layer, it should toggle `alwaysNeedsCompositing`. If this changes (other than when altering children, which calls this automatically), `markNeedsCompositingBitsUpdate` should be called.

What are repaint boundaries?

- Certain render objects introduce repaint boundaries to the render tree (via `RenderObject.isRepaintBoundary`). These render objects are always painted into a

new layer, allowing them to paint separately from their parent. This effectively decouples the subtree rooted at the repaint boundary from previously painted nodes.
    - This is useful when part of the UI remains mostly static and part of the UI updates frequently. A repaint boundary helps to avoid repainting the static portion of the UI.
    - Render objects that are repaint boundaries are associated with an `OffsetLayer`, a special type of layer that can update its position without re-rendering.
- Render objects that form repaint boundaries are handled differently during painting (via `PaintingContext.paintChild`).
    - If the child doesn't need painting, the entire subtree is skipped. The corresponding layer is updated to reflect the new offset (via `PaintingContext._compositeChild`) and appended to the current parent layer.
    - If the child needs painting, the current offset layer is cleared or created (via `PaintingContext._repaintCompositedChild`) then used for painting (via `RenderObject._paintWithContext`).
    - Otherwise, painting proceeds through `RenderObject._paintWithContext`.
- Since repaint boundaries always push new layers, all ancestors must be marked as needing compositing. If any such node utilizes a non-local painting effect, that node will need to introduce a new layer, too.

How is painting managed given that children can push new layers?

- `PaintingContext.paintChild` manages this process by (1) ending any ongoing recording, (2) creating a new layer for the child, (3) painting the child in the new layer using a new painting context (unless it is a clean repaint boundary, which will instead be used as is), (4) appending that layer to the current layer tree, and (5) creating a new canvas and layer for any future painting with the original painting context.
- Methods in `PaintingContext` will consulting the compositing bits to determine whether to implement a behavior by inserting a layer or altering the canvas (e.g., clipping).

How can layers make painting more efficient?

- All render objects may be associated with a `ContainerLayer` (via `RenderObject.layer`). If defined, this is the last layer that was used to paint the render object (if multiple layers are used, this is the root-most layer).
  - Repaint boundaries are automatically assigned an `OffsetLayer` (via `PaintingContext._repaintCompositedChild`). All other layers must specifically set this field when painting.
  - Render objects that do not directly push layers must set this to null.
- `ContainerLayer` tracks any associated `EngineLayer` (via `ContainerLayer.engineLayer`). Certain operations can use this information to allow a layer to be updated rather than rebuilt.
  - Methods accept an "`oldLayer`" parameter to enable this optimization.
  - Stored engine layers can be reused directly (via `SceneBuilder.addRetained`); this is retained rendering.

How can the widget tree be manually rasterized?

- A layer can be rasterized to an image via `OffsetLayer.toImage`. A convenient way to obtain an image of the entire UI is to use this method on the `RenderView`'s own layer. To obtain a subset of the UI, an appropriate repaint boundary can be inserted and its layer used similarly.

How are clips, transforms, and other effects managed?

- There are two code paths. Layers that do not require compositing handle transforms and clips within their own rendering context (thus, the effects are limited to that single layer), via `Canvas`. Those that are composited, however, introduce these effects as special `ContainerLayers`, via `PaintingContext`.
  - A variety of useful `ContainerLayers` are included: `AnnotatedRegionLayer`, which is useful for storing metadata within sections of a layer, `BackdropFilterLayer`, which allows the background to be blurred or filtered, `OpacityLayer`, which allows opacity to be varied, and `OffsetLayer`, which is the key mechanism by which the repaint boundary optimization works.

- `Canvas.saveLayer` / `Canvas.restore` allow drawing commands to be grouped such that effects like blending, color filtering, and anti-aliasing can be applied once for the group instead of being incrementally stacked (which could lead to invalid renderings).

How are external textures composited?

- Textures are entirely managed by the engine and referenced by a simple texture ID. Layers depicting textures are associated with this ID and, when composited, integrated with the backend texture. As such, these are painted out-of-band by the engine.
- Texture layers are integrated with the render tree via `TextureBox`. This is a render box that tracks a texture ID and paints a corresponding `TextureLayer`. During layout, this box expands to fill the incoming constraints.

Semantics

Interaction

Gestures

How are hit tests performed?

- The `RendererBinding` is `HitTestable`, which implies the presence of a `hitTest` method. The default implementation defers to `RenderView`, which itself implements `hitTest` to visit the entire render tree. Each render object is given an opportunity to add itself to a shared `HitTestResult`.
- `GestureBinding.dispatchEvent` (via `HitTestDispatcher`) uses the `PointerRouter` to pass the original event to all render objects that were hit (`RenderObject` implements `HitTestTarget`, and therefore provides `handleEvent`). If a `GestureRecognizer` is utilized, the event's pointer is passed via `GestureRecognizer.addPointer` which registers with the `PointerRouter` to receive future events.
- Related events are sent to all original `HitTestTarget` as well as any routes registered with the `PointerRouter`. `PointerDownEvent` will close the gesture area, barring additional entries, whereas `PointerUpEvent` will sweep it, resolving competing gestures and preventing indecisive gestures from locking up input.

How are gestures captured and propagated?

- `Window.onPointerDataPacket` captures pointer updates from the engine and generates `PointerEvents` from the raw data. `PointerEventConverter` is utilized to map physical coordinates from the engine to logical coordinates, taking into account the device's pixel ratio (via `PointerEventConverter.expand`).

What does a gesture recognizer do?

- A pointer is added to the gesture recognizer by client code on `PointerDownEvent`. Gesture recognizers determine whether a pointer is allowed by overriding

`GestureRecognizer.isPointerAllowed`. If so, the recognizer subscribes to future pointer events via the `PointerRouter` and adds the pointer to the gesture arena via `GestureArenaManager.add`.
- The recognizer will process incoming events, outcomes from the gesture arena (`GestureArenaMember.accept` / `rejectGesture`), spontaneous decisions about the gesture (`GestureArenaEntry.resolve`), and other externalities. Typically, recognizers watch the stream of `PointerEvents` via `HitTestTarget.handleEvent`, looking for terminating events like `PointerDown`, or criteria that will cause acceptance / rejection. If the gesture is accepted, the recognizer will continue to process events to characterize the gesture, invoking user-provided callbacks at key moments.
- The gesture recognizer must unsubscribe from the pointer when rejecting or done processing, removing itself from the `PointerRouter` (`OneSequenceGestureRecognizer.stopTrackingPointer` does this).

What does the gesture arena do?

- The arena disambiguates multiple gestures in a way that allows single gestures to resolve immediately if there is no competition. A recognizer “wins” if it declares itself the winner or if it’s the last/sole survivor.

Can gesture recognizers be grouped together?

- A `GestureArenaTeam` combines multiple recognizers together into a group.
- Captained teams cause the captain recognizer to win when all unaffiliated recognizers reject or a constituent accepts.
- A non-captained team causes the first added recognizer to win when all unaffiliated recognizers reject. However, if a constituent accepts, that recognizer still takes the win.

What auxiliary classes support gesture handling?

- There are two major categories of gesture recognizers, multi-touch recognizers (i.e., `MultiTapGestureRecognizer`) that simultaneously process multiple pointers (i.e., tapping with two fingers will register twice), and single-touch recognizer (i.e., `OneSequenceGestureRecognizer`) that will only consider events from a single pointer (i.e., tapping with two fingers will register once).
- There is a helper "Drag" object that is used to communicate drag-related updates to other parts of the framework (like `DragScrollActivity`)
- There's a `VelocityTracker` that generates fairly accurate estimates about drag velocity using curve fitting.
- There are local and global `PointerRoutes` in `PointerRouter`. Local routes are used as described above; global routes are used to react to any interaction (i.e., to dismiss a tooltip).

Focus

What are the focus building blocks?

- `FocusManager`, stored in the `WidgetsBinding`, tracks the currently focused node and the most recent node to request focus. It handles updating focus to the new primary (if any) and maintaining the consistency of the focus tree by sending appropriate notifications. The manager also bubbles raw keyboard events up from the focused node and tracks the current highlight mode.
- `FocusTraversalPolicy` dictates how focus moves within a focus scope. Traversal can happen in a `TraversalDirection` (`TraversalDirection.left`, `TraversalDirection.up`) or to the next, previous, or first node (i.e., the node to receive focus when nothing else is focused). All traversal is limited to the node's closest enclosing scope. The default traversal policy is the `ReadingOrderTraversalPolicy` but can be overridden using the `DefaultFocusTraversal` inherited widget. `FocusNode.skipTraversal` can be used to allow a node to be focusable without being eligible for traversal.
- `FocusAttachment` is a helper used to attach, detach, and reparent a focus node as it moves around the focus tree, typically in response to changes to the underlying widget tree. As such, this ensures that the node's focus parent is associated with a context (`BuildContext`) that is an ancestor of its own context. `The` attachment instance delegates to the underlying focus node using inherited widgets to set defaults – i.e., when reparenting, `Focus.of` / `FocusScope.of` are used to find the node's new parent.
- `FocusNode` represents a discrete focusable element within the UI. Focus nodes can request focus, discard focus, respond to focus changes, and receive keyboard events when focused. Focus nodes are grouped into collections using scopes which are themselves a type of focus node. Conceptually, focus nodes are entities in the UI that can receive focus whereas focus scopes allow focus to shift amongst a group of descendant nodes. As program state, focus nodes must be associated with a `State` instance.
- `FocusScopeNode` is a focus node subclass that organizes focus nodes into a traversable group. Conceptually, a scope corresponds to the subtree (including nested scopes) rooted at the focus scope node. Descendent nodes add themselves to the nearest enclosing scope when receiving focus (`FocusNode._reparent` calls `FocusNode._removeChild` to ensure scopes forget nodes that have moved). Focus

scopes maintain a history stack with the top corresponding to the most recently focused node. If a child is removed, the last focused scope becomes the focused child; this process will not update the primary focus.

- Focus is a stateful widget that manages a focus node that is either provided (i.e., so that an ancestor can control focus) or created automatically. `Focus.of` establishes an inherited relationship causing the dependant widget to be rebuilt when focus changes. Autofocus allows the node to request focus within the enclosing scope if no other node is already focused.
- `FocusScope` is the same as above, but manages a focus scope node.
- `FocusHighlightMode` and `FocusHighlightStrategy` determine how focusable widgets respond to focus with respect to highlighting. By default the highlight mode is updated automatically by tracking whether the last interaction was touch-based. The highlight mode is either `FocusHighlightMode.traditional`, indicating that all controls are eligible, and `FocusHighlightMode.touch`, indicating that only those controls that summon the soft keyboard are eligible.

What is focus?

- Focus determines the UI elements that will receive raw keyboard events.
- The primary focused node is unique within the application. As such, it is the default recipient of keyboard events, as forwarded by the `FocusManager`.
- All nodes along the path from the root focus scope to the primary focused node have focus. Keyboard events will bubble along this path until they are handled.
- Moving focus (via traversal or by requesting focus) does not call `FocusNode.unfocus`, which would remove it from the enclosing scope's stack of recently focused nodes, also ensuring that a pending update doesn't refocus the node.
- Controls managing focus nodes may choose to render with a highlight in response to `FocusNode.highlightMode`.

What is the focus tree?

- The focus tree is a sparse representation of focusable elements in the UI consisting entirely of focus nodes. Focus nodes maintain a list of children, as well as assessors that return the descendants and ancestors of a node in furthest and closest order, respectively.
- Some focus nodes represent scopes which demarcate a subtree rooted at that node. Conceptually, scopes serve as a container for focus traversal operations.
- Both scopes and ordinary nodes can receive focus, but scopes pass focus to the first descendent node that isn't a scope. Even if a scope is not along the path to the primary focus, it tracks a focused child (`FocusScopeNode.focusedChild`) that would receive focus if it were.
- Scopes maintain a stack of previously focused children with the top of the stack being the first to receive focus if that scope receives focus (the focused child). When the focused child is removed, focus is shifted to the previous holder. If a focused child is a scope, it is called the scope's first focus.
- When a node is focused, all enclosing scopes ensure that their focused child is either the target node or a scope that's one level closer to the target node.
- As focus changes, affected nodes receive focus notifications that clients can use to update the UI. The focused node also receives raw keyboard events which bubble up the focus path.

How is focus attached to the widget tree?

- A `FocusAttachment` ensures a `FocusNode` is anchored to the correct parent when the focus tree changes (e.g., because the widget tree rebuilds). This ensures that the associated build contexts are consistent with the actual widget tree.
- As a form of program state, every focus node must be hosted by a `StatefulWidget` 's `State` instance. When state is initialized (`State.initState`), the focus node is attached to the current `BuildContext` (via `FocusNode.attach`), returning a `FocusAttachment` handle.
- When the widget tree rebuilds (`State.build` , `State.didChangeDependencies`), the attachment must be updated via `FocusAttachment.reparent` . If a widget is configured to use a new focus node (`State.didUpdateWidget`), the previous attachment must be detached (`FocusAttachment.detach` , which calls `FocusNode.unfocus` as needed) before attaching the new node. Finally, the focus

node must be disposed when the host state is itself disposed (`FocusNode.dispose` , which calls `FocusAttachment.detach` as needed).
- Reparenting is delegated to `FocusNode._reparent` , using `Focus.of` / `FocusScope.of` to locate the nearest parent node. If the node being reparented previous had focus, focus is restored through the new path via `FocusNode._setAsFocusedChild` .
- A focus node can have multiple attachments, but only one attachment may be active at a time.

How is focus managed?

- `FocusManager` tracks the root focus scope as well as the current (primary) and requesting (next) focus nodes. It also maintains a list of dirty nodes that require update notifications.
- As nodes request focus (`FocusNode.requestFocus`), the manager is notified that the node is dirty (`FocusNode._markAsDirty`) and that a focus update is needed (`FocusManager._markNeedsUpdate`), passing the requesting node. This sets the next focus node and schedules a microtask to actually update focus. This can delay focus updates by up to one frame.
- This microtask promotes the requesting node to primary (`FocusManager._applyFocusChange`), marking all nodes from the root to the incoming and outgoing nodes, inclusive, as being dirty. Then, all dirty nodes are notified that focus has changed (`FocusNode._notify`) and marked clean.
- The new primary node updates all enclosing scopes such that they describe a path toward it via each scope's focused child (`FocusNode._setAsFocusedChild`).

How is focus requested?

- When a node requests focus (`FocusNode.requestFocus`), every enclosing scope is updated to focus toward the requesting node directly or via an intermediate scope (`FocusNode._setAsFocusedChild`). The node is then marked dirty (

`FocusNode._markAsDirty`) which updates the manager's next node and requests an update.

- When a scope requests focus (`FocusScopeNode.requestFocus`), the scope tree is traversed to find the first descendent node that isn't a scope. Once found, that node is focused. Otherwise, the deepest scope is focused.
- The focus manager resolves focus via `FocusManager._applyFocusChange`, promoting the next node to the primary and sending notifications to all dirty nodes.

How is focus relinquished?

- When a node is unfocused, the enclosing scope "forgets" the node and the manager is notified (via `FrameManager._willUnfocusNode`).
  - If the node had been tracked as primary or next, the corresponding property is cleared and an update scheduled. If a next node is available, that node becomes primary. If there is no primary and no next node, the root scope becomes primary.
  - The deepest non-scope node will not be automatically focused. However, future traversal will attempt to identify the current first focus (`FocusTraversalPolicy.findFirstFocus`) when navigating.
- When a node is detached, if it had been primary, it is unfocused as above. It is then removed from the focus tree.
- When a node is disposed, the manager is notified (via `FrameManager._willDisposeFocusNode`) which calls `FrameManager._willUnfocusNode`, as above. Finally, it is detached.
- Focus updates are scheduled once per frame. As a result, state will not stack but resolve to the most recent request.

What is a keyboard token?

- Some controls display the soft keyboard in response to focus. The keyboard token is a boolean tracking whether focus was requested explicitly (via `FocusNode.requestFocus`) or assigned automatically due to another node losing focus.

- Controls that display the keyboard consume the token (`FocusNode.consumeKeyboardToken`), which returns its value and sets it to false. This ensures that the keyboard is shown exactly once in response to explicit user interaction.

Scrolling

Scrollable

What are the building blocks of scrolling?

- Scrollable provides the interaction model for scrolling without specifying how the actual viewport is managed (a `ViewportBuilder` must be provided). UI concerns are customized directly or via an inherited `ScrollConfiguration` that exposes an immutable `ScrollBehavior` instance. This instance is used to build platform-specific chrome (i.e., a scrolling indicator) and provides ambient `ScrollPhysics`, a class that describes how scrolling UI will respond to user gestures.
- `ScrollPhysics` is consulted throughout the framework to construct physics simulations for ballistic scrolling, to validate and adjust user interaction, to manage momentum across interactions, and to identify overscroll regions.
- `ScrollableState` connects the `Scrollable` to a `ScrollPosition` via a `ScrollController`. This controller is responsible for producing the `ScrollPosition` from a given `ScrollContext` and `ScrollPhysics`; it also provides the `initialScrollOffset`.
  - For example, `PageView` injects a page-based scrolling mechanism by having its `ScrollController` (`PageController`) return a custom scroll position subclass.
- The `ScrollContext` exposes build contexts for notifications and storage, a ticker provider for animations, and methods to interact with the scrollable; its analogous to `BuildContext` for an entire scrollable widget.
- `ScrollPosition` tracks scroll offset as pixels (reporting changes via `Listenable`), applies physics to interactions via `ScrollPhysics`, and through subclasses like `ScrollPositionWithSingleContext` (which implement `ScrollActivityDelegate` and makes concrete much of the actual scrolling machinery), starts and stops `ScrollActivity` instances to mutate the represented scroll position.
  - The actual pixel offset and mechanisms for reacting to changes in the associated viewport are introduced via the `ViewportOffset` superclass.
  - Viewport metrics are mixed in via `ScrollMetrics`, which redundantly defines pixel offset and defines a number of other useful metrics like the amount of content above and below the current viewport (`extentBefore`, `extentAfter`), the pixel offset corresponding to the top and bottom of the current viewport (

`minScrollExtent`, `maxScrollExtent`) and the viewport size (`viewportDimension`).
    - The scroll position may need to be corrected (via `ScrollPosition.correctPixels` [replaces pixels outright] / `ViewportOffset.correctBy` [applies a delta to pixels]) when the viewport is resized, as triggered by shrink wrapping or relayout. Every time a viewport (via `RenderViewport`) is laid out, the new content extents are checked by `ViewportOffset.applyContentDimensions` to ensure the offset won't change; if it does, layout must be repeated.
    - `ViewportOffset.applyViewportDimension` and `ViewportOffset.applyContentDimensions` are called to determine if this is the case; any extents provided represent viewport slack – how far the viewport can be scrolled in either direction beyond what is already visible. Activities are notified via `ScrollActivity.applyNewDimensions`().
        - The original pixel values corresponds to certain children being visible. If the dimensions of the viewport change, the pixel offset required to maintain that same view may change. For example, consider a viewport sized to a single letter displaying "A," "B," and "C" in a column. When "B" is visible, pixels will correspond to "A"'s height. Suppose the viewport expands to fit the full column. Now, pixels will be zero (no offset is needed). [?]
        - The same is true if the viewport's content changes size. Again, consider the aforementioned "A-B-C" scenario with "B" visible. Instead of the viewport changing size, suppose "A" is resized to be zero pixels tall. To keep "B" in view, the pixel offset must be updated (from non-zero to zero). [?]
- `ScrollController` provides a convenient interface for interacting with one or more `ScrollPositions`; in effect, it calls the corresponding method in each of its positions. As a `Listenable`, the controller aggregates notifications from its positions.
- `ScrollNotifications` are emitted by scrollable (by way of the active `ScrollActivity`). As a `LayoutChangedNotification` subclass, these are emitted after build and layout have already occurred, thus only painting can be performed in response without introduce jank.
    - Listening to a scroll position directly avoids the delay, allowing layout to be performed in response to offset changes. It's not clear why this is faster - both paths seem to trigger at the same time [?]

How is the scroll position updated in general?

- The `ScrollPositionWithSingleContext` starts and manages `ScrollActivity` instances via `drag`, `animateTo`, `jumpTo`, and more.
- `ScrollActivity` instances update the scroll position via `ScrollActivityDelegate`; `ScrollPositionWithSingleContext` implements this interface and applies changes requested by the current activity (`setPixels`, `applyUserOffset`) and starts follow-on activities (`goIdle`, `goBalastic`).
- Any changes applied by the activity are processed by the scroll position, then passed back to the activity which generates scroll notifications (e.g., `dispatchScrollUpdateNotification`).
- `DragScrollActivity`, `DrivenScrollActivity`, and `BallisticScrollActivity` apply user-driven scrolling, animation-driven scrolling, and physics-driven scrolling, respectively.
- `ScrollPosition.beginActivity` starts activities and tracks all state changes. This is possible because the scroll position is always running an activity, even when idle (`IdleScrollActivity`). These state changes generate scroll notifications via the activity.

How is the scroll position updated by dragging?

- The underlying `Scrollable` uses a gesture recognizer to detect and track dragging if `ScrollPhysics.shouldAcceptUserOffset` allows. When a drag begins, the `Scrollable`'s scroll position is notified via `ScrollPosition.drag`.
- `ScrollPositionWithSingleContext` implements this method to create a `ScrollDragController` which serves as an integration point for the `Scrollable`, which receives drag events, and the activity, which manages scroll state / notifications. The controller is returned as a `Drag` instance, which provides a mechanism to update state as events arrive.
- As the user drags, the drag controller forwards a derived user offset back to `ScrollActivityDelegate.applyUserOffset` (`ScrollPositionWithSingleContext`) which applies `ScrollPhysics.applyPhysicsToUserOffset` and, if valid, invokes

`ScrollActivityDelegate.setPixels`. This actually updates the scroll offset and generates scroll notifications.
- When the drag completes, a ballistic simulation is started via `ScrollActivityDelegate.goBallistic`. This delegates to the scroll position's `ScrollPhysics` instance to determine how to react.
- Interestingly, the `DragScrollActivity` delegates most of its work to the drag controller and is mainly responsible for forwarding scroll notifications.

How is the scroll position updated by `animateTo`?

- The `DrivenScrollActivity` is much more straightforward. It starts an animation controller which, on every tick, updates the current pixel value via `setPixels`. When animating, if the container over-scrolls, an idle activity is started. If the animation completes successfully, a ballistic activity is started instead.

How is scrolling behavior and state managed?

- The `ScrollPosition` writes the current scroll offset to `PageStorage` if `ScrollPosition.keepScrollOffset` is true.

How are the scrollable, the viewport, and any contained slivers associated?

- `ScrollView` is a base class that builds a scrollable and a viewport, deferring to its subclass to specify how its slivers are constructed. The subclass overrides `buildSlivers` to do this (`ScrollView.build` creates the `Scrollable`, which uses `ScrollView.buildViewport` as its `viewportBuilder`, which uses `ScrollView.buildSlivers` to obtain the sliver children).

Viewports

What are the general viewport building blocks?

- `ViewportOffset` is an interface that tracks the current scroll offset (`ViewportOffset.pixels`) and direction (`ViewportOffset.userScrollDirection` , which is relative to the positive axis direction, ignoring growth direction); it also offers a variety of helpers (e.g., `ViewportOffset.animateTo`). The offset represents how much content has been scrolled off screen, or more precisely, the number of logical pixels by which all children have been shifted opposite the viewport's scrolling direction (`Viewport.axisDirection`). For a web page, this would be how many pixels of content are above the browser's viewport. This interface is implemented by `ScrollPosition` , tying together viewports and scrollables. Pixels can be negative when scrolling before the center sliver. [?]

What are the viewport widget building blocks?

- Viewport is a layout widget that is larger on the inside. The viewport is associated with a scroll offset (`ViewportOffset`), an interface that is implemented by `ScrollPosition` and typically fulfilled by `ScrollPositionWithSingleContext` . As the user scrolls, this offset is propagated to descendant slivers via layout. Finally, slivers are repainted at their new offsets, creating the visual effect of scrolling.
- `ShrinkWrappingViewport` is a variant of viewport that sizes itself to match its children in the main axis (instead of expanding to fill the main axis).
- `NestedScrollViewViewport` is a specialized viewport used by `NestedScrollView` to coordinate scrolling across two viewports (supported by auxiliary widgets like `SliverOverlapAbsorberHandle`).
- `ScrollView` couples a viewport with a scrollable, deferring to its subclass to provide slivers; as such, the `ScrollView` provides a foundation for building scrollable UI. `CustomScrollView` accepts an arbitrary sliver list whereas `BoxScrollView` – and its subclasses `ListView` and `GridView` – apply a single layout model (e.g., list or grid) to a collection of slivers.

What are the viewport rendering building blocks?

- `RenderAbstractViewport` provides a common interface for all viewport subtypes. This allows the framework to locate and interact with viewports in a generic way (via `RenderAbstractViewport.of`). It also provides a generic interface for determining offsets necessary to reveal certain children.
- `RenderViewportBase` provides shared code for render objects that host slivers. By establishing an axis and axis direction, `RenderViewportBase` maps the offset-based coordinate space used by slivers into cartesian space according to a managed offset value (`ViewportOffset.pixels`). `RenderViewportBase.layoutChildSequence` serves as the foundation for sliver layout (and is typically invoked by `performLayout` in subclasses). `RenderViewportBase` also establishes the cache extent (the area to either side of the viewport that is laid out but not visible) as well as entry points for hit testing and painting.
- `RenderViewport` displays a subset of its sliver children based on its current viewport offset. A center sliver (`RenderViewport.center`) is anchored at offset zero. Slivers before center ("reverse children") grow opposite the axis direction (`GrowthDirection.reverse`) whereas the center along with subsequent slivers ("forward children") grow forward (`GrowthDirection.forward`); both groups are anchored according to the same axis direction (this is why both start from the same edge), though conceptually reverse slivers are laid out in the opposite axis direction (e.g., their "leading" and "trailing" edges are flipped).
  - The anchor point can be adjusted, changing the visual position of offset zero (`RenderViewport.anchor` is in the range [0, 1], with zero corresponding to the axis origin [?]).
  - Conceptually, children are ordered: RN-R0, center, FN-F0.
- `RenderShrinkWrappingViewport` similar to `RenderViewport` except sized to match the total extent of visible children within the bounds of incoming constraints.
- `RenderNestedScrollViewViewport`

What are the attributes of a viewport?

- Throughout this section, words like “main extent,” “cross extent,” “before,” “after,” “leading,” and “trailing” are used to eliminate spatial bias from descriptions. This is because viewports can be oriented along either axis (e.g., horizontal, vertical) with varying directionality (e.g., down, right). Moreover, the ordering of children along the axis is subject to the current growth direction (e.g., forward or reverse).
- Viewports have two sets of dimensions: outer and inner. The portion of the viewport that occupies space on screen has a main axis and cross axis extent (e.g., height and width); these are the viewport’s “outer”, “outside”, or “physical” dimensions. The inside of the viewport, which matches or exceeds the outer extent, includes all the content contained within the viewport; these are described using “inner”, “inside”, or “logical” dimensions. The inner edges correspond to the edges of the viewport’s contents; the outer edges correspond to the edges of the visible content. When otherwise unspecified, the viewport’s leading / trailing edges generally refer to its outer (i.e., physical) edges.
- The viewport is comprised of a bidirectional list of slivers. “Forward slivers” include a “center sliver,” and are laid out in the default axis direction (`GrowthDirection.forward`). “Reverse slivers” immediately precede the center sliver and are laid out opposite the axis direction (`GrowthDirection.reverse`). The line between forward and reverse slivers, at the center sliver’s leading edge, is called the “centerline,” coincident with the zero scroll offset. Within this document, the region within the viewport comprised of reverse slivers is called the “reverse region” with its counterpart being the “forward region.” `Note` that the viewport’s inner edges fully encompass both regions.
- The viewport positions the center sliver at scroll offset zero by definition. However, the viewport also tracks a current scroll offset (`RenderViewport.offset`) to determine which of its sliver children are in view and therefore should be rendered. This offset represents the distance between the center sliver’s leading edge (i.e., scroll offset zero) and the viewport’s outer leading edge, and increases opposite the axis direction.
    - Equivalently, this represents the number of pixels by which the viewport’s contents have been shifted opposite its axis direction.
    - For example, if the center sliver’s leading edge is aligned with the viewport’s leading edge, the offset would be zero. If its trailing edge is aligned with the viewport’s leading edge, the offset would be the sliver’s extent. If its leading edge is aligned with the viewport’s trailing edge, the offset would be the viewport’s extent, negated.

How do viewports manage parent data?

- There are two major classes of parent data used by slivers: `SliverPhysicalParentData` (used by `RenderViewport`) and `SliverLogicalParentData` (used by `RenderShrinkWrappingViewport`). These differ in how they represent the child's position. The former stores absolute coordinates from the parent's visible top left corner whereas the latter stores the distance from the parent's zero scroll offset to the child's nearest edge. Physical coordinates are more efficient for children that must repaint often but incur a cost during layout. Logical coordinates optimize layout at the expense of added cost during painting.
- Viewports use two subclasses to support multiple sliver children, `SliverPhysicalContainerParentData` and `SliverLogicalContainerParentData`. These are identical to their superclasses (where the "parent" isn't a sliver but the viewport itself), mixing in `ContainerParentDataMixin<RenderSliver>`.

When might the center sliver not appear at the leading edge?

- The center sliver may be offset by `RenderSliver.centerOffsetAdjustment` (added to the current `ViewportOffset.pixels` value). This effectively shifts the zero scroll offset (e.g., to visually center the center sliver).
- The zero scroll offset can itself be shifted by a proportion of the viewport's main extent via `RenderViewport.anchor`. Zero positions the zero offset at the viewport's leading edge; one positions the offset at the trailing edge (and `0.5` would position it at the midpoint).
- These adjustments are mixed into the calculation early on (see `RenderViewport.performLayout` and `RenderViewport._attemptLayout`). Conceptually, it is easiest to ignore them other than to know that they shift the centerline's visual position.
- The center sliver may also paint itself at an arbitrary offset via `SliverGeometry.paintOrigin`, though this won't actually move the zero offset.

What are some of the quirks of viewport layout?

- Forward and reverse slivers are laid out separately and are generally isolated from one another [?]. Reverse slivers are laid out first, then forward slivers. Reverse slivers and forward slivers share the same axis direction (i.e., the generated constraints reference the same `SliverConstraints.axisDirection`), though reverse sliver calculations (e.g., for painting or layout offset) effectively flip this direction. Thus, it is most intuitive to think of reverse slivers as having their leading and trailing edges flipped, etc.
- Layout offset is effectively measured from the viewport's outer leading edge to the nearest edge of the sliver (i.e., the offset is relative to the viewport's current view).
    - More accurately, a sliver's layout offset is measured from the zero scroll offset of its parent which, for a viewport, coincides with the centerline. However, since layout offset is iteratively computed by summing layout extents (in `RenderViewportBase.layoutChildSequence`) and these extents are zero unless a sliver is visible, this formal definition boils down to the practical definition described above. [?]
    - This property explains why `RenderViewportBase.computeAbsolutePaintOffset` is able to produce paint offsets trivially from layout offsets (this is surprising since layout offsets are ostensibly measured from the zero scroll offset whereas paint offsets are measured from the box's top left corner).
    - Even though layout offsets after the trailing edge are approximate (due to an implementation detail of `RenderViewportBase.layoutChildSequence`), this direct mapping remains safe as out-of-bounds painting will be clipped.
- Scroll offset can be interpreted in two ways.
    - When considering a sliver or viewport in isolation, scroll offset refers to a one dimensional coordinate space anchored at the object's leading edge and extending toward its trailing edge.
    - When considering a sliver in relation to a parent container, scroll offset represents the first offset in the sliver's coordinate space that would be visible in its parent (e.g. offset zero implies that the leading edge of the sliver is visible; offset N implies that all except the leading N pixels are visible -- if N is greater than the sliver's extent, some of those pixels are empty space). Conceptually, a scroll offset represents how far a sliver's leading precedes the viewport.
        - When zero, the sliver has been fully scrolled after the leading edge (and possibly after the trailing edge).
        - When less than the sliver's scroll extent, a portion of the sliver precedes the leading edge.
        - When greater, the sliver is entirely before the leading edge.

- Scroll extent represents how much space a sliver might consume; it need only be accurate when the constraints would permit the sliver to fully paint (i.e., the desired paint extent fits within the remaining paintable space).
    - Slivers preceding the leading edge or appearing within the viewport must provide valid scroll extents if they might conceivably be painted.
    - Slivers beyond the trailing edge may approximate their scroll extents since no pixels remain for painting.
- Overlap is the pixel offset (in the main axis direction) necessary to fully "escape" any earlier sliver's painting. More formally, this is the distance from the sliver's current position to the first pixel that hasn't been painted on by an earlier sliver. Typically, this pixel is after the sliver's offset (e.g., because a preceding sliver painted beyond its layout extent). However, in some cases, the overlap can be negative indicating that the first such pixel is before the sliver's offset (i.e., earlier than the sliver's offset).
- All slivers preceding the viewport's trailing edge receive unclamped values for the remaining paintable and cacheable extents, even if those slivers are located far offscreen. Slivers implement a variety of layout effects and therefore may consume visible (or cacheable) pixels at their discretion.

What other services does the viewport provide?

- Viewports support the concept of maximum scroll obstruction (`RenderViewportBase.maxScrollObstructionExtentBefore`), a region of the viewport that is covered by "pinned" slivers and that effectively reduces the viewport's scrollable bounds. This is a secondary concept used only when computing the scroll offset to reveal a certain sliver (`AbstractRenderViewport.getOffsetToReveal`).
- Viewports provide a mechanism for querying the paint and (approximate) scroll offsets of their children. The implementation depends on the type of viewport; efficiency may also be affected by the parent model (e.g., `RenderViewport` uses `SliverPhysicalContainerParentData` , allowing paint offsets to be returned immediately).

How are viewport children ordered?

- A logical index is assigned to all children. The center child is assigned zero; subsequent children (forward slivers) are assigned increasing indices (e.g., 1, 2, 3) whereas preceding children (reverse slivers) are assigned decreasing indices (e.g., -1, -2, -3).
- Children are stored sequentially (R reverse slivers + the center sliver + F forward slivers), starting with the “last” reverse sliver (-R), proceeding toward the “first” reverse sliver (-1), then the center sliver (0), then ending on the last forward sliver (F+1).
    - The first child (`RenderViewport.firstChild`) is the “last” reverse sliver.
- Viewports define a painting order and a hit-test order. Reverse slivers are painted from last to first (-1), then forward slivers are painted from last to first (center). Hit-testing proceeds in the opposite direction: forward slivers are tested from the first (center) to the last, then reverse slivers are tested from the first (-1) to the last.

What do shrink-wrapping viewports do differently?

- Whereas an ordinary viewport expands to fill the main axis, shrink-wrapping viewports are sized to minimally contain their children. Consequently, as the viewport is scrolled, its size will change to accommodate the visible children which may require layout to be repeated. Shrink-wrapping viewports do not support reverse children.
- The shrink-wrapping viewport uses logical coordinates instead of physical coordinates since it performs layout frequently.

What is a scroll offset correction?

- A pixel offset directly applied to the `ViewportOffset.pixels` value allowing descendent slivers to adjust the overall scrolling position. This is done to account for errors when estimating overall scroll extent for slivers that build content dynamically.
    - Such slivers often cannot measure their actual extent without building and laying out completely first. Doing this continuously would be prohibitively slow and thus relative positions are used (i.e., position is reckoned based on where neighbors are without necessarily laying out all children).
- The scroll offset correct immediately halts layout, propagating to the nearest enclosing viewport. The value is added directly to the viewport’s current offset (e.g., a positive

correction increases `ViewportOffset.pixels`, translating content opposite the viewport's axis direction – i.e., scrolling forward).
    - Conceptually, this allows slivers to address logical inconsistencies that arise due to, e.g., estimating child positions by determining a scroll offset that would have avoided the problem, then reattempting layout using this new offset.
    - Adjusting the scroll offset will not reset other child state (e.g., the position of children in a `SliverList`); thus, when such a sliver requests a scroll offset correction, the offset selected is one that would cause any existing, unaltered state to be consistent.
    - For example, a `SliverList` may not have enough room for newly revealed children when scrolling backwards (e.g., because the incoming scroll offset indicates an inadequate number of pixels preceding the viewport's leading edge). The `SliverList` calculates the scroll offset correction that would have avoided this logical inconsistency, adjusting it (and any affected layout offsets) to ensure that children appear at the same visual location in the viewport.
- If a chain of corrections occurs, layout will eventually fail.

Viewport Layout

How does a viewport layout its children?

- `RenderViewport` kicks off layout via `RenderViewport.performLayout`.
  - Out-of-band data is incorporated into layout, supplementing constraints and geometry.
    - `minScrollExtent`: total scroll extent of reverse slivers, negated.
    - `maxScrollExtent`: total scroll extent of forward slivers.
    - `hasVisualOverflow`: whether clipping is needed when painting.
  - `RenderViewport.performLayout` applies the center sliver adjustment and repeatedly attempts layout until layout succeeds or a maximum number of attempts is reached.
    - Main and cross axis extents are derived from width and height given the viewport's axis.
    - Attempt to layout all children (`RenderViewport._attemptLayout`), passing the current scroll offset (`ViewportOffset.pixels`) including any visual adjustment of the center sliver (`RenderSliver.centerOffsetAdjustment`).
    - If the attempt fails due to a scroll offset correction being returned, that correction is applied (`ViewportOffset.pixels` is adjusted additively, without notifying listeners) and layout re-attempted.
    - If the attempt succeeds, the viewport must verify that the final content dimensions would not invalidate the viewport's current pixel offset (via `ViewportOffset.applyContentDimensions`); if it does, layout must be re-attempted.
      - The out-of-band minimum and maximum scroll extents are transformed into "slack" values, representing how much the viewport can be scrolled in either direction from the zero offset, excluding pixels visible in the viewport.
      - As an example, if layout was attempted assuming a large viewport offset, but layout indicates that there is no slack on either side of the viewport, this would be considered an invalid state – the viewport shouldn't have been scrolled so far. [?]
      - The typical implementation of this method (`ScrollPositionWithSingleContext.applyContentDimensions`) never

requests re-layout. Intead, the current scroll activity is given an opportunity to react to any changes in slack (`ScrollActivity.applyNewDimensions`).

- `RenderViewport._attemptLayout` lays out the reverse slivers then, if successful, the forward slivers; else, it returns a scroll offset correction. Before attempting layout, this method clears stale layout information (e.g., out-of-band state) and calculates a number of intermediate values used to determine the constraints passed to the first sliver in each sequence (i.e., the arguments to `RenderViewportBase.layoutChildSequence`).
  - All out-of-band data is cleared and the intermediate viewport metrics calculated.
    - `centerOffset` : the offset from the viewport's outer leading edge to the centerline (i.e., from the current scroll offset to the zero scroll offset).
      - Negative when scrolled into the forward region, positive when scrolled into the reverse region.
      - If the anchor (i.e., visual adjustment of the centerline) is non-zero, applies an additional offset proportional to the main extent. This offset affects all other metrics, too.
    - `reverseDirectionRemainingPaintExtent` : total visible pixels available for painting reverse slivers.
    - `forwardDirectionRemainingPaintExtent` : total visible pixels available for painting forward slivers.
    - `fullCacheExtent` : the viewport's visual extent plus the two adjacent cache extents (`RenderViewportBase.cacheExtent`).
      - Subsequent definitions refer to the region defined by the leading cache extent, the visible viewport, and the trailing cache extent as the "cacheable viewport."
    - `centerCacheOffset` : the offset from the leading edge of the cacheable viewport to the centerline.
    - `reverseDirectionRemainingCacheExtent` : total visible and cacheable pixels available for painting reverse slivers.
    - `forwardDirectionRemainingCacheExtent` : total visible and cacheable pixels available for painting forward slivers.
  - If present, reverse slivers are laid out via `RenderViewportBase.layoutChildSequence` . The parameters to this method serve as initial values for the first reverse sliver; as layout progresses, they are incrementally adjusted for each subsequent sliver.

- In the following descriptions, since reverse slivers are effectively laid out in the opposite axis direction, the meaning of “leading” and “trailing” are flipped. That is, what was previously the trailing edge of the viewport / sliver is now described as its leading edge.
- child: the initial reverse sliver (immediately preceding the center sliver).
- `scrollOffset` : the amount the first reverse sliver has been scrolled before the leading edge of the viewport.
  - If the forward region is visible, this is always zero. This is because the initial reverse sliver must necessarily be after the leading edge.
- overlap: zero, since nothing has been painted.
- `layoutOffset` : the offset to apply to the sliver’s layout position (i.e., to push it past any pixels earmarked for other slivers), effectively measured from the viewport’s leading edge (though technically measured from the centerline). Only increased by slivers that are on screen.
  - If the forward region is not visible, this is always zero. This is because layout may start from the leading edge unimpeded.
  - If the forward region is visible, this is `forwardDirectionRemainingPaintExtent` ; those pixels are earmarked for forward sliver layout (which, being visible, will reduce the available space in the viewport). Thus, the first reverse sliver must be “bumped up” by a corresponding amount.
    - If the forward region is partially visible, the above represents how much of the visible viewport has been allocated for forward layout.
    - If the forward region is solely visible, the above still applies (i.e., layout offset must be adjusted by `forwardDirectionRemainingPaintExtent` which, in this case, is the full extent).
    - No adjustment beyond this is necessary because the forward region appears after the reverse region and can therefore only interact with reverse layout by consuming paint extent.
- `remainingPaintExtent` : total visible pixels available for reverse slivers (`reverseDirectionRemainingPaintExtent`).
- advance: `childBefore` to iterate backwards.
- `remainingCacheExtent` : total visible and cacheable pixels available for painting (`reverseDirectionRemainingCacheExtent`).

    - `cacheOrigin`: how far before the leading edge is cacheable (always negative). Forward region not eligible.
- Forward slivers are laid out via `RenderViewportBase.layoutChildSequence`. The parameters to this method serve as initial values for the center sliver. These values are the opposite of the reverse values, with a handful of exceptions.
    - "Leading" and "trailing" once again refer to their intuitive definitions.
    - child: the center sliver (i.e., the first forward sliver).
    - `scrollOffset`: the amount the centerline has been scrolled before the leading edge of the viewport.
        - If the reverse region is visible, this is always zero. This is because the center sliver must necessarily be after the leading edge.
    - overlap: number of pixels needed to jump from the starting layout position to the first pixel not painted on by an earlier sliver (before or after the sliver's leading edge).
        - If reverse slivers were laid out, the reverse region is treated as a self-contained block that fully consumes its portion of the viewport. Thus, there can be no overlap. [?]
        - If reverse slivers were not laid out, the initial overlap is how far the centerline is from the viewport's leading edge (i.e., the full extent of the reverse region, negated). Intuitively, a negative overlap indicates that the first clean pixel is before the center sliver's leading edge and not after.
            - `RenderViewportBase.layoutChildSequence` tracks the furthest offset painted so far (`maxPaintOffset`). Passing a negative value ensures that the reverse region (e.g., due to overscroll) is captured by this calculation; else it would remain empty.
            - For example, failing to do this might cause a pinned app bar to separate from the top of the viewport during overscroll.
    - `layoutOffset`: the offset to apply to the sliver's layout position (i.e., to push it past any pixels earmarked for other slivers), effectively measured from the viewport's leading edge (though technically measured from the centerline). Only increased by slivers that are on screen.
        - If the reverse region is not visible, this is always zero. This is because layout may start from the leading edge unimpeded.

        - If the reverse region is visible, the layout offset must be bumped by the full extent of the reverse region – not just the visible portion (note that this differs from the reverse case). Forward slivers are laid out after reverse slivers, and so must account for the full amount of potentially consumed layout.
            - If the reverse region is partially visible, the offset used is `forwardDirectionRemainingPaintExtent`. This represents how much of the visible viewport has been allocated for reverse layout.
            - If the reverse region is solely visible, the offset used is the full distance from the viewport's outer leading edge to the centerline. That is, the entire reverse region is considered as having been used for layout.
            - Though this is an upper-bound approximation, this value is acceptable since forward slivers won't be painted (they are offscreen by assumption). Consequently, layout offsets within the forward region serve mainly for ordering, not precise positioning. [?]
    - `remainingPaintExtent`: total visible pixels available for forward slivers (`forwardDirectionRemainingPaintExtent`).
        - Note that slivers preceding the leading edge will receive the full value even if they are off screen; it is up to the sliver's interpretation of its scroll offset to determine whether it consumes pixels.
    - advance: `childAfter` to iterate forwards.
    - `remainingCacheExtent`: total visible and cacheable pixels available for painting (`forwardDirectionRemainingCacheExtent`).
    - `cacheOrigin`: how far before the leading edge is cacheable (always negative). Reverse region not eligible.

How are child sequences laid out?

- `RenderViewportBase.layoutChildSequence` is the engine that drives sliver layout. It processes a sequence of children one by one, passing initial constraints and adjusting layout state as geometry is produced.

- Layout state is initialized. The initial layout offset is captured and the effective scroll direction computed (this direction will be flipped to account for reverse growth).
  - `maxPaintOffset` : the layout offset corresponding to the farthest pixel painted so far. Adjusted by incoming overlap (e.g., to allow slivers to avoid overlapping with earlier slivers; or, when negative, to allow slivers to fill empty space due to overscroll).
- For every child, invoke `RenderSliver.performLayout` , passing an appropriate `SliverConstraint` instance.
  - `scrollDirection` : the effective scroll direction.
  - `scrollOffset` : the incoming (e.g., center sliver's) scroll offset, decremented by all preceding slivers' scroll extents. Negative values are clamped at zero since fully exposed slivers have a zero scroll offset.
    - Conceptually, this is the amount that the current sliver's leading edge precedes the viewport's leading edge as reckoned using earlier slivers' extents.
  - overlap: the difference between the farthest pixel painted thus far (`maxPaintOffset`) and the current layout offset.
    - Conceptually, this is the offset needed to jump to the nearest clean pixel from the sliver's layout position. Can be negative.
  - `precedingScrollExtent` : sum of all preceding slivers' scroll extents, including those that are off screen.
    - Note that a sliver may consume infinite scroll extent, so this quantity may be infinite.
  - `remainingPaintExtent` : the original paintable extent decreased by the total layout extent consumed so far (paint extent is not used because painting can overlap whereas layout cannot).
    - Conceptually, this is how much the sliver may choose to paint starting from its scroll offset.
  - `parentUsesSize` : this is always true; if a sliver must lay out, so too must the viewport.
  - `correctedCacheOrigin` (intermediate value, negative): ensure the cache origin falls within the sliver, clamping it to the sliver's leading edge.
  - `cacheExtentCorrection` (intermediate value, negative): how much of the cache region could not be filled by the sliver (i.e., because it precedes its leading edge).

        - `cacheOrigin` : the corrected cache origin.
            - Even if the sliver significantly precedes the viewport's leading edge (i.e., has a large positive scroll offset), the cache origin is added to the sliver's scroll offset (i.e., the offset needed to reach the viewport's leading edge from the sliver's leading edge) and will therefore correctly index into the leading cache region.
        - `remainingCacheExtent` : the original cacheable extent (the leading and trailing cache regions plus the viewport's main extent) decreased by the total cache extent consumed so far.
            - Conceptually, this is how much the sliver may choose to paint starting from the corrected cache origin. Always larger than `remainingPaintExtent` .
            - This is adjusted by `cacheExtentCorrection` to discount the portion of the cache that is inaccessible to the sliver. The sliver is effectively starting further into the cacheable region, not bumping the cacheable region out to ensure that the cacheable region ends at the same place.
    - If layout returns an offset correction, control returns to the caller which, in most cases, applies the correction and reattempts layout.
    - Layout state is mutated to account for the geometry produced by the sliver.
        - `effectiveLayoutOffset` : the layout offset as adjusted by the sliver's `SliverGeometry.paintOrigin` (negative values shift the offset toward the leading edge; the default is zero).
            - Conceptually, this is where the sliver has been visually positioned in the viewport. Note that it will still consume `SliverGeometry.layoutExtent` pixels at its original layout offset and not the effective layout offset.
        - `maxPaintOffset` : update if the farthest pixel painted by this sliver (the effective layout offset increased by `SliverGeometry.paintExtent`) exceeds the previous maximum.
        - `scrollOffset` : decrease by the scroll extent of the sliver. Unlike `sliverScrollOffset` , which is clamped at zero, this quantity is permitted to become negative (representing the proportion of total scroll extent after the viewport's leading edge).
        - `layoutOffset` : increase by the layout extent of the sliver. Layout extent must be less than paint extent, which itself must be less than remaining paint extent. Thus, only those slivers consuming space in the visible viewport will increase layout offset.

        - Out of band data is updated to reflect the total forward and reverse scroll extent encountered so far (`_maxScrollExtent` and `_minScrollExtent` , respectively), as well as whether clipping is needed (`_hasVisualOverflow`).
    - Update cache state to if any cache was consumed (i.e., `SliverGeometry.cacheExtent` > 0).
        - `remainingCacheExtent` : reduce the cacheable extent by the number of pixels that were either consumed or unreachable due to preceding the sliver's leading edge (`cacheExtentCorrection`).
        - `cacheOrigin` : if the leading cacheable extent has not been exhausted, update the origin to point to the next eligible pixel; this offset must be negative. Otherwise, set the cache origin to zero. Subsequent slivers may still attempt to fill the trailing cacheable extent if space permits.
    - Update the sliver's parent data to reflect its position.
        - If the sliver precedes the viewport's leading edge or is otherwise visible, the effective layout offset correctly represents this sliver's position.
        - Otherwise, construct an approximate position by adding together the initial layout offset and the proportion of total scroll extent after the viewport's leading edge.
            - The latter term is equivalent to the initial scroll offset decreased by the total scroll extent processed so far. Once negative, this value represents how far slivers have extended into the visible portion of the viewport (and beyond).
            - Layout offsets only increase for slivers that are painted (and have a `SliverGeometry.layoutExtent` that is non-zero); thus, only the portion of scroll extent falling within the visible region can possibly increment layout offset. As a result, the above quantity represents an upper-bound offset that can be used to reliably order, if not paint, offscreen slivers.
        - Viewports that use physical parent data (e.g., `RenderViewport`) must compute the absolute paint offset to store. Otherwise, the layout offset may be stored directly.

How do shrink-wrapping viewports handle layout?

- The out-of-band data tracked by shrink-wrapping viewports is extended.

    - `_shrinkWrapExtent`: the total maximum paint extent (`SliverGeometry.maxPaintExtent`) of all slivers laid out so far. Maximum paint extent represents how much a sliver could paint without constraint. Thus, this extent represents the amount of space needed to fully paint all children.
    - There is no minimum extent since there are no reverse slivers.
- There are several major differences when performing layout.
    - `RenderShrinkWrappingViewport.performLayout` is the entrypoint to viewport layout, repeating layout until there are no offset corrections and final dimensions are deemed valid.
        - If there are no children, the viewport expands to fill the cross axis, but is as small as possible in the main axis; it then returns. Else, the extents are initialized to be as big as the constraints permit.
        - Layout is attempted (`RenderShrinkWrappingViewport._attemptLayout`) using the unadulterated viewport offset (`ViewportOffset.pixels`). Any center offset adjustment is ignored. If layout produces a scroll offset correction, that correction is applied, and layout reattempted.
        - Effective extent is computed by clamping the maximum extent needed to paint all children (`_shrinkWrapExtent`) to the incoming constraints.
        - Final layout dimensions (inner and outer) are verified. If either would invalidate the viewport offset (`ViewportOffset.pixels`), layout must be reattempted.
            - The outer dimension is set to the effective extent and verified via `ViewportOffset.applyViewportDimensions`.
            - The inner dimension is verified via `ViewportOffset.applyContentDimensions`, which requires “slack” values representing how far before and after the zero offset the viewport can scroll. Since there are no reverse slivers, the minimum extent is zero. The maximum extent is the difference between the total scrollable extent and the total paintable extent of all children (clamped to zero). That is, how much additional scrolling is needed even after all children are painted.
        - Last, the viewport is sized to match its effective extent, clamped by the incoming constraints.
    - `RenderShrinkWrappingViewport._attemptLayout` is a thin wrapper around `RenderViewportBase.layoutChildSequence` that configures the layout parameters passed to the first child in the sequence.
        - child: the first child, since there are no reverse slivers.

- `scrollOffset` : incoming offset, clamped above zero (i.e., the amount the viewport is scrolled).
- `scrollOffset` : incoming offset, clamped below zero (i.e., the amount the viewport is over-scrolled).
- `layoutOffset` : always zero.
- `remainingPaintExtent` : the main axis extent (i.e., the full viewport)
- advance: `childAfter` to iterate forwards.
- `remainingCacheExtent` : the main axis extent plus two cache extents.
- `cacheOrigin` : `cacheExtent` pixels before the layout offset.


Slivers

Sliver Model

What are the sliver building blocks?

- `RenderSliver` is the base class used by render objects that implement scrolling effects. Layout utilizes `SliverConstraints` to produce `SliverGeometry` (`RenderSliver.geometry`); painting is unchanged, with the origin positioned at the top-left corner. Hit testing is implemented similarly to `RenderBox` , relying instead on extents within the viewport's main axis.
- `RenderSliverHelpers` incorporates a variety of helpers, including support for implementing `RenderObject.applyPaintTransform` and `RenderObject.hitTestChildren` for slivers with box children.
- `RenderSliverSingleBoxAdapter`
- `RenderSliverToBoxAdapter` adapts a render box to the sliver protocol, allowing it to appear alongside and within other slivers.

How do slivers manage children?

- Most slivers contain box children. Some, like `RenderSliverPadding` , have a single sliver child. Few slivers have multiple sliver children (generally, this is the purview of viewports).
- Some slivers manage children dynamically. This is quite complicated and discussed in a subsequent section.

What helpers do slivers provide?

- `RenderSliver.calculateCacheOffset` and `RenderSliver.calculatePaintOffset` accept a region (defined by a "from" and "to" relative to the sliver's zero offset) and determine the contained extent that is visible. This value is derived from the current scroll offset and remaining paint/cache extent from the provided constraints.

    - The result is only meaningful for slivers that paint exactly as much as is scrolled into view. For example, if a sliver paints more than it was scrolled (e.g., a pinned sliver that always paints the same amount), this calculation will be invalid.
- `RenderSliver.getAbsoluteSizeRelativeToOrigin` : the size of the sliver in relation to the sliver's leading edge rather than the canvas's origin. This size is derived from the cross axis and paint extents; the paint extent is negated if the sliver's contents precede its leading edge from the perspective of the canvas.
- `RenderSliver.childScrollOffset` : the distance from the parent's zero scroll offset to a child's zero scroll offset; unaffected by scrolling. Defaults to returning zero (i.e., child and parent are aligned).
- `RenderSliver.childMainAxisPosition` : the distance from the parent's visible leading edge to the child's visible leading edge, in the axis direction. If the actual leading edge is not visible, the edge defined by the intersection of the sliver with the viewport's leading edge is used instead. The child must be visible, though the parent need not be (e.g., a persistent header that pins its child even when the anchoring sliver is scrolled away).
    - Slivers that contain box children cannot measure distance using the child's visible edge since box children are viewport naive. As a result, the box's actual leading edge is always used.
- `RenderSliver.childCrossAxisPosition` : for vertical axes, this is the distance from the parent's left side to the child's left side. For horizontal axes, this is the distance from parent's top side to the child's top side.

What new properties do slivers introduce?

- `RenderSliver.geometry` is analogous to `RenderBox.size` and contains the output of layout.
- `RenderSliver.centerOffsetAdjustment` is an offset applied to the viewport's center sliver (e.g., to center it). Implicitly shifts neighboring slivers. Positive offsets shift the sliver opposite the axis direction.

How is sliver parent data managed?

- There are two options for encoding parent data (i.e., a sliver's position within its parent): using absolute coordinates from the parent's visible top left corner (via `SliverPhysicalParentData.paintOffset`) or using a delta from the parent's zero scroll offset to the nearest child edge (via `SliverLogicalParentData.layoutOffset`). The former optimizes painting at the expense of layout whereas the latter makes the opposite trade-off.
    - When thinking about paint offsets, it's crucial to note the distinction between a sliver's top-left corner and its visible top-left corner. If a sliver straddles the leading edge in a downward-scrollable viewport, its children are positioned relative to the top-left corner of the viewport, not the top-left corner that precedes the leading edge. This is always the edge that is nearest the top-left corner regardless of axis or growth direction.
    - Logical offsets are relative to the parent's zero scroll offset. This is either the parent's leading or trailing edge, depending on growth direction.

How do slivers perform hit testing?

- Slivers follow a similar model to `RenderBox`, tracking whether a point is contained within the current sliver or one of its descendents. The current sliver determines whether this is the case by first testing its children and then itself (`RenderSliver.hitTest` invokes `RenderSliver.hitTestChildren` then `RenderSliver.hitTestSelf`).
- If a hit is detected, the sliver adds a `SliverHitTestEntry` to the `SliverHitTestResult`, and returns true. The sliver will then receive subsequent pointer events via `RenderSliver.handleEvent`.
    - `SliverHitTestEntry` associates a sliver with the coordinate that was hit.
    - `SliverHitTestResult` is a `HitTestResult` with a convenience method for mapping to sliver coordinates (`SliverHitTestResult.addWithAxisOffset`). This adapts a main and cross axis position in the parent's coordinate space to the child's coordinate space by subtracting the offset to the child. A paint offset is also used since `HitTestResult` is also responsible for mapping pointer events, which use canvas coordinates. [?]
- Hit testing will only be performed once layout has completed; painting will not have occurred yet.

- Coordinates are expressed as main and cross axis positions in the sliver's own coordinate space. Any positions provided to children must be mapped into that child's coordinate space.
    - Positions are relative to the current scroll offset; that is, a main axis position of zero corresponds to the first visible offset within the sliver, not the leading edge of the sliver.
    - The interactive region is defined by the current hit test extent anchored at the current scroll offset. The sliver's full cross axis extent is considered interactive, unless the default hit testing behavior is overridden.
    - Slivers may paint anywhere in the viewport using a non-zero paint origin. Special care must be taken to transform coordinates if this is the case. For instance, a sliver that contains a sliver child that always paints a button at the viewport's trailing edge will filter out events that lie outside of its own interactive bounds unless it is explicitly written to handle its child's special painting behavior.

How do slivers represent layout constraints?

- Sliver constraints can be converted to box constraints via `SliverConstraints.asBoxConstraints`. This method accepts minimum and maximum bounds for the sliver's main extent (e.g., if the main axis is horizontal, this would correspond to width). The cross axis constraint is always tight.
- Constraints provide information about scrolling state from the perspective of the sliver being laid out so that it can select appropriate geometry.
    - `SliverConstraints.axis`, `SliverConstraints.axisDirection`, `SliverConstraints.crossAxisDirection`: the current orientation and directionality of the main and cross axes.
    - `SliverConstraints.growthDirection`: how a sliver's contents are ordered relative to the main axis direction (i.e., first-to-last vs. last-to-first).
    - `SliverConstraints.normalizedGrowthDirection`: the growth direction that would produce the same child ordering assuming a "standard" axis direction (`AxisDirection.down` or `AxisDirection.right`). This is the opposite of `SliverConstraints.growthDirection` if axis direction is up or left.
    - `SliverConstraints.cacheOrigin`: how far before the current scroll offset to begin painting to support caching (i.e., the number of pixels to "reach back" from

the first visible offset in the sliver). Always between -`SliverConstraints.scrollOffset` and zero.

- `SliverConstraints.overlap` : a delta from the current scroll offset to the furthest pixel that has not yet been painted by an earlier sliver, in the axis direction. For example, if an earlier sliver painted beyond its layout extent, this would be the offset to “escape” that painting. May be negative if the furthest pixel painted precedes the current sliver (e.g., due to overscroll).
- `SliverConstraints.precedingScrollExtent` : the total scrolling distance consumed by preceding slivers (i.e., the sum of all preceding scroll extents). Scroll extent is approximate for slivers that could not fully paint in the available space; thus, this value may be approximate. If a preceding sliver is built lazily (e.g., a list that builds children on demand) and has not finished building, this value will be infinite. It may also be infinite if a preceding sliver is infinitely scrollable.
- `SliverConstraints.remainingPaintExtent` : the number of pixels available for painting in the viewport. May be infinite (e.g., for shrink-wrapping viewports). Zero when all pixels have been painted (e.g., when a sliver is beyond the viewport’s trailing edge).
  - Whether this extent is actually consumable by the sliver depends on the sliver’s intended behavior.
- `SliverConstraints.remainingCacheExtent` : the number of pixels available for painting in the viewport and cache regions, starting from the scroll offset adjusted by the cache origin. Always larger than remaining paint extent.
  - Whether this extent is actually consumable by the sliver depends on the sliver’s intended behavior.
- `SliverConstraints.scrollOffset` : the offset of the first visible part of the sliver, as measured from the sliver’s leading edge in the axis direction. Equivalently, this is how far the sliver’s leading edge precedes its parent’s leading edge.
  - All slivers after the leading edge have a scroll offset of zero, including those after the trailing edge.
  - The scroll offset may be larger than the sliver’s actual extent if the sliver is far before the parent’s leading edge.
- `SliverConstraints.userScrollDirection` : the `ScrollDirection` in relation to the axis direction and the content’s logical ordering (as determined by growth direction).
  - Note that this describes the direction in which the content is actually moving on the user’s screen, not the intuitive direction that the user is scrolling (e.g.,

scrolling down a webpage translates the content up; thus, the scroll direction would be `ScrollDirection.reverse`. If the content were reversed within that web page, it would be `ScrollDirection.forward`).

- `SliverConstraints.viewportMainAxisExtent`: the number of pixels that the viewport can display in its main axis.

How do slivers represent geometry?

- Slivers define several extents (i.e., regions within the viewport) with scroll, layout, and paint being particularly important. Scroll extent corresponds to the amount of scrolling necessary to go from the sliver's leading edge to its trailing edge. Layout extent corresponds to the number of pixels physically occupied by the sliver in the visible viewport. Paint extent corresponds to the number of pixels painted on by the sliver in the visible viewport. Paint extent and layout extent typically go to zero as the sliver is scrolled out of the viewport, whereas scroll extent generally remains fixed.
- A sliver's actual position within its parent (sometimes called its "layout position") is encoded in one of two ways: as an absolute paint offset that can be directly used when painting or as a layout offset relative to the parent's zero scroll offset (`SliverPhysicalParentData` and `SliverLogicalParentData`, respectively). Though a sliver may have a particular position, it is free to paint anywhere in the viewport.
- Sliver geometry captures several such extents as well as the sliver's current relationship with the viewport.
  - `SliverGeometry.paintOrigin`: where to begin painting relative to the sliver's current layout position, expressed as an offset in the axis direction (e.g., a negative value would precede the current position).
    - The sliver still consumes layout extent at its original layout position. Subsequent layout is unaffected by the paint origin.
    - Taken into account when determining the next sliver's overlap constraint, particularly if this sliver paints further than any preceding sliver.
  - `SliverGeometry.paintExtent`: the number of contiguous pixels consumed by this sliver in the visible region. Measured from the sliver's layout position, as adjusted by the paint origin (`SliverGeometry.paintOrigin`, typically zero).
    - Paint extent ≤ remaining paint extent: it is inefficient to paint out of bounds.
    - Typically, but not necessarily, goes to zero when scrolled out of the viewport.

        - The paint extent combined with the paint origin determines the next sliver's overlap constraint.
    - `SliverGeometry.maxPaintExtent` : the number of pixels this sliver could paint in an unconstrained environment (i.e., a shrink-wrapping viewport). Approximate.
    - `SliverGeometry.layoutExtent` : the number of contiguous pixels consumed by this sliver in the visible region for layout (i.e., the amount of space this sliver will physically occupy in the viewport). Measured from the sliver's layout position regardless of paint origin. Layout positions are advanced by summing layout extents.
        - Layout extent ≤ paint extent: cannot take up more space than is painted.
        - Defaults to paint extent.
        - Typically, but not necessarily, goes to zero when scrolled out of the viewport. Can still take up space even when logically offscreen.
    - `SliverGeometry.scrollExtent` : the amount of scrolling needed to reach this sliver's trailing edge from its leading edge. Used to calculate all subsequent sliver's scroll offsets.
        - Approximate, except when sliver is able to fully paint.
        - Typically, but not necessarily, constant.
        - Typically, but not necessarily, an upper bound on paint extent.
    - `SliverGeometry.cacheExtent` : the number of contiguous pixels consumed by this sliver in the cacheable region. Measured from the sliver's layout position, as adjusted by the cache origin (i.e., the offset into the available cache region from the sliver's current scroll offset).
        - Layout extent ≤ paint extent ≤ cache extent: the cacheable viewport includes the visible region; thus, any visible pixels consumed count toward the sliver's cache extent.
        - Slivers that paint at positions other than their layout position will need to calculate how much of the cache region was actually consumed.
    - `SliverGeometry.hitTestExtent` : the number of contiguous pixels painted by this sliver that are interactive in the visible region. Measured from the sliver's layout position. Does not take the paint origin into account.
        - Hit test extent ≤ paint extent: can only handle events in painted region.
        - Defaults to paint extent.
        - Typically, but not necessarily, goes to zero when scrolled out of the viewport.
        - Subject to interpretation by `RenderSliver.hitTest` which might, for instance, take into account paint origin; if applicable, the parent sliver must be written to

forward hit events that fall outside of its own interactive bounds.
    - `SliverGeometry.maxScrollObstructionExtent` : only meaningful for slivers that affix themselves to the leading or trailing edges of the viewport (e.g., a pinned header), effectively reducing the amount of content visible within the viewport (i.e., the viewport's outer dimension). If applicable, this is the extent of that reduction, in pixels.
    - `SliverGeometry.hasVisualOverflow` : whether a clip is necessary due to visual overflow. The viewport will add a clip if any of its children report visual overflow.
    - `SliverGeometry.visible` : whether this sliver should be painted. Defaults to true when paint extent is greater than zero.
    - `SliverGeometry.scrollOffsetCorrection` : a correction indicating that the incoming scroll offset was invalid (e.g., [?]) and therefore must be corrected. If set, no further geometry need be returned as layout will be recomputed from scratch. Must be propagated to the containing viewport.

How is a box embedded within a sliver?

- `SliverToBoxAdapter` is a single-child render object widget that incorporates a `RenderSliverToBoxAdapter` into the render tree.
- `RenderSliverSingleBoxAdapter` lays the groundwork for managing a single render box child.
    - Children are positioned using physical coordinates (`SliverPhysicalParentData`); these coordinates are measured from the parent's visible top-left corner to the child's visible top-left corner.
    - It provides a hook for updating the child's position using the parent's constraints and geometry (`RenderSliverSingleBoxAdapter.setChildParentData`). This calculation takes into account that render boxes may be partially off screen and must therefore be offset during painting to ensure that the correct slice of the box is visible.
        - For instance, assuming a typical downward-oriented viewport, a non-zero scroll offset indicates how much of the parent is currently above the top of the viewport. To paint the correct slice of the viewport, the render box must be translated above the canvas by that same amount.

        - To crystallize this, consider an upward-oriented viewport. The child's paint offset is zero as its parent is scrolled into view (the box's visible top-left corner is coincident with its parent's visible top-left corner). Once the sliver has been fully scrolled across the viewport and begins to scroll off the viewport's top edge, its parent's first visible top-left corner effectively becomes the top of the viewport and remains there as scrolling continues (i.e., this is now the top-left visible corner of the sliver). Therefore, to correctly paint the box shifting out of view, it must be offset to account for the shifted frame of reference.
    - The child's main axis position is calculated using the box's actual edge since render box is viewport naive. Thus, the box's main axis position may be negative if it precedes the viewport's leading edge.
    - Hit testing and painting trivially wrap the box's own implementation.
- `RenderSliverToBoxAdapter` extends `RenderSliverSingleBoxAdapter` and does the actual work of adapting the sliver layout protocol to the render box equivalent.
    - Boxes receive tight cross axis constraints and unbounded main axis constraints (i.e., the box is free to select its main dimension without constraint).
    - This extent serves as the sliver's scroll extent and max paint extent (i.e., because the size of the box is precisely how much this sliver can scroll and paint).
    - The adapter calculates how much of the desired extent would be visible given the sliver's current scroll offset and the remaining space in the viewport (via `RenderSliver.calculatePaintOffset`).
    - This quantity is used as the paint, layout, and hit-testing extents (i.e., because this is how much of the box is visible, consuming space, and interactive, respectively).
    - If the box cannot be rendered in full (i.e., there isn't enough room in the viewport or the sliver isn't past the leading edge), a clip is applied by indicating that there is visual overflow.

What are some examples of basic slivers?

- `RenderSliverPadding`: adds padding around a child sliver.
    - Lays out child with the incoming constraints modified to account for padding.
        - `SliverGeometry.scrollExtent`: decreased to reflect leading padding pushing content toward the viewport's leading edge.

        - `SliverGeometry.cacheOrigin` : offset to reflect padding consuming cache space.
        - `SliverGeometry.remainingPaintExtent` , `SliverGeometry.remainingCacheExtent` : reduced by any leading padding that is visible.
        - `SliverGeometry.crossAxisExtent` : reduced by the total cross axis padding.
    - Calculate the padding sliver's geometry using the child's geometry.
        - `SliverGeometry.paintExtent` : either the total visible main axis padding plus the child's layout extent or, if larger, the leading padding plus the child's paint extent (this would allow the child's painting to overflow trailing padding). Clamped to remaining paint extent.
        - `SliverGeometry.scrollExtent` :the child's scroll extent plus main axis padding.
        - `SliverGeometry.layoutExtent` : the total visible main axis padding plus the child's layout extent. Clamped to the paint extent (i.e., cannot consume more space than is painted).
        - `SliverGeometry.hitTestExtent` : the total visible main axis padding plus the child's paint extent or, if larger, the leading padding plus the child's hit testing extent (allowing the child's hit testing region to overflow).
    - Finally, a position is selected according to axis direction, growth direction, and the leading padding that is currently visible.
- `RenderSliverFillRemaining` is a single box adapter that causes its child to fill the remaining space in the viewport. Subsequent slivers are never seen as this sliver consumes all usable space.
    - This sliver calculates the remaining paint extent in the main axis, ensuring that this is passed as a tight constraint to the box child (unlike `RenderSliverToBoxAdapter` which leaves the main extent unconstrained). The child is subsequently laid out.
    - The sliver then calculates its own geometry.
        - `SliverGeometry.scrollExtent` : the full extent of the viewport. This is an upper bound since it isn't actually possible to scroll beyond this sliver: it consumes all viewport space.
        - `SliverGeometry.paintExtent` , `SliverGeometry.layoutExtent` : matches the extent of the child that fits within the viewport.
    - Finally, the box is positioned (using the incoming constraints and calculated geometry) such that it is correctly offset during painting if it cannot fit in the

viewport.

Persistent Headers

How do persistent headers work?

- `RenderSliverPersistentHeader` provides a base class for implementing persistent headers within a viewport, adding support for varying between a minimum and maximum extent during scrolling. Some subclasses introduce pinning behavior, whereas others allow the header to scroll into and out of out of view. `SliverPersistentHeader` encapsulates this behavior into a stateless widget, delegating configuration (and child building) to a `SliverPersistentHeaderDelegate` instance.
  - Persistent headers can be any combination of pinnable and floating (or neither); those that float, however, can also play a snapping animation. All persistent headers expand and contract in response to scrolling; only floating headers do so in response to user scrolling anywhere in the viewport.
    - A floating header reappears whenever the user scrolls its direction. The header expands to its maximum extent as the user scrolls toward it, and shrinks as the user scrolls away.
    - A pinned header remains at the top of the viewport. Unless floating is also enabled, the header will only expand when approaching its actual position (e.g., the top of the viewport).
    - Snapping causes a floating header to animate to its expanded or contracted state when the user stops scrolling, regardless of scroll extent.
  - Persistent headers contain a single box child and track a maximum and minimum extent. The minimum extent is typically based on the box's intrinsic dimensions (i.e., the object's natural size). As per the render box protocol, by reading the child's intrinsic size, the persistent header will be re-laid out if this changes.
  - `RenderSliverPersistentHeader` doesn't configure parent data or determine how to position its child along the main axis. It does, however, provide support for hit testing children (which requires subclasses to override `RenderSliverPersistentHeader.childMainAxisPosition`). It also paints its child without using parent data, computing the child's offset in the same way as `RenderSliverSingleBoxAdapter` (i.e., to determine whether its viewport-naive box is offset on the canvas if partially scrolled out of view).
  - Two major hooks are exposed:

- `RenderSliverPersistentHeader.updateChild` : supports updating the contained box whenever the persistent header lays out or the box itself changes size. This is provided two pieces of layout information.
  - Shrink offset is the delta between the current and maximum extents (i.e., how much more room there is to grow). Always positive.
  - Overlaps content is true if the header's leading edge is not at its layout position in the viewport.
- `RenderSliverPersistentHeader.layoutChild` : invoked by subclasses to updates then lay out children within the largest visible portion of the header (between maximum and minimum extent). Subclasses provide a scroll offset, maximum extent, and overlap flag, all of which may differ from the incoming constraints.
  - Shrink offset is set to the scroll offset (how far the header is before the viewport's leading edge) and is capped at the max extent. Conceptually, this represents how much of the header is off screen.
  - If the child has changed size or the shrink offset has changed, the box is given an opportunity to update its appearance (via `RenderSliverPersistentHeader.updateChild`). This is done within a layout callback since, in the common case, the child is built dynamically by the delegate. This is an example of interleaving build and layout.
    - This flow is facilitated by the persistent header widgets (e.g., `_SliverPersistentHeaderRenderObjectWidget`), which utilize a custom render object element (`_SliverPersistentHeaderElement`).
    - When the child is updated (via `RenderSliverPersistentHeader.updateChild`), the specialized element updates its child's element (via `Element.updateChild`) using the widget produced by the delegate's build method.
  - Finally, the child is laid out with its main extent loosely constrained to the portion of the header that's visible – with the minimum extent as a lower bound (i.e., so that the box is never forced to be smaller than the minimum extent).

How is the expanding / contracting scrolling effect implemented?

- `RenderSliverScrollingPersistentHeader` expands to its maximum extent when scrolled into view, and shrinks to its minimum extent before being scrolled out of view.
  - Lays out the child in the largest visible portion of the header (up to the maximum extent), then returns its own geometry.
    - `SliverGeometry.scrollExtent` : always the max extent since this is how much scrolling is needed to scroll past the header.
    - `SliverGeometry.paintOrigin` : paints before itself if there's a negative overlap (e.g., to fill empty space in the viewport from overscroll).
    - `SliverGeometry.paintExtent` , `SliverGeometry.layoutExtent` : the largest visible portion of the header, clamped to the remaining paint extent.
    - `SliverGeometry.maxPaintExtent` : the maximum extent; it's not possible to provide more content.
  - Tracks the child's position across the scroll (i.e., the distance from the header's leading edge to the child's leading edge). The child is aligned with the trailing edge of the header to ensure it scrolls into view first.
    - Calculated as the portion of the maximum extent in view (which is negative when scrolled before the leading edge), minus the child's extent after layout.

How does pinning work?

- `RenderSliverPinnedPersistentHeader` is similar to its scrolling sibling, but remains pinned at the top of the viewport regardless of offset. It also avoids overlapping earlier slivers (e.g., useful for building stacking section labels).
  - The pinned header will generally have a layout offset of zero when scrolled before the viewport's leading edge. As a consequence, any painting that it performs will be coincident with the viewport's leading edge. This is how pinning is achieved.
    - Recall that layout offset within a viewport only increases when slivers report a non-zero layout extent. Slivers generally report a zero layout extent when they precede the viewport's leading edge (i.e., when viewport scroll offset exceeds their extent); thus, when the header precedes the viewport's leading edge, it will likely have a zero layout offset. Additionally, recall that layout offset is used when the viewport paints its children. Since the header will have a zero layout offset, at least in the case of `RenderViewport` , the sliver will be painted at the viewport's painting origin.

        - If an earlier, off-screen sliver consumes layout, this will bump out where the header paints. This might cause strange behavior.
        - If another pinned header precedes this one, it will not consume layout. However, by utilizing the overlap offset, the current header is able to avoid painting on top of the preceding header.
    - Next, it lays out the child in the largest visible portion of the header (up to the maximum extent), then returns its own geometry.
        - `SliverGeometry.scrollExtent` : always the max extent since this is how much scrolling is needed to scroll past the header (though, practically speaking, it can never truly be scrolled out of view).
        - `SliverGeometry.paintOrigin` : always paints on the first clean pixel to avoid overlapping earlier slivers.
        - `SliverGeometry.paintExtent` : always paints the entire child, even when scrolled out of view. Clamped to remaining paint extent.
        - `SliverGeometry.layoutExtent` : the pinned header will consume the portion of its maximum extent that is actually visible (i.e., it only takes up space when its true layout position falls within the viewport). Otherwise (e.g., when pinned), it occupies zero layout space and therefore may overlap any subsequent slivers. Clamped to remaining paint extent.
        - `SliverGeometry.maxScrollObstructionExtent` : the minimum extent, since this is the most that the viewport can be obscured when pinned.
    - The child is always positioned at zero as this sliver always paints at the viewport's leading edge.

How does floating work?

- `RenderSliverFloatingPersistentHeader` is similar to its scrolling sibling, but reattaches to the viewport's leading edge as soon as the user scrolls in its direction. It then shrinks and detaches if the user scrolls away.
    - The floating header tracks scroll offset, detecting when the user begins scrolling toward the header. The header maintains an effective scroll offset that matches the real scroll offset when scrolling away, but that enables floating otherwise.
        - It does this by jumping ahead such that the sliver's trailing edge (as measured using the effective offset and maximum extent) is coincident with the

viewport's leading edge. This is the "floating threshold." `All` subsequent scrolling deltas are applied to the effective offset until the user scrolls the header before the floating threshold. At this point, normal behavior is resumed.

- `RenderSliverFloatingPersistentHeader.performLayout` detects the user's scroll direction and manages an effective scroll offset. The effective scroll offset is used for updating geometry and painting, allowing the header to float above other slivers.
  - The effective scroll offset matches the actual scroll offset the first time layout is attempted or whenever the user is scrolling away from the header and the header isn't currently floating.
  - Otherwise, the header is considered to be floating. This occurs when the user scrolls toward the header, or the header's actual or effective scroll offset is less than its maximum extent (i.e., the header's effective trailing edge is at or after the viewport's leading edge).
    - When floating first begins (i.e., because the user scrolled toward the sliver), the effective scroll offset jumps to the header's maximum extent. This effectively positions its trailing edge at the viewport's leading edge.
    - As scrolling continues, the delta (i.e., the change in actual offset) is applied to the effective offset. As a result, geometry is updated as though the sliver were truly in this location.
    - The effective scroll offset is permitted to become smaller, allowing the header to reach its maximum extent as it scrolls into view.
    - The effective scroll offset is also permitted to become larger such that the header shrinks. Once the header is no longer visible, the effective scroll offset jumps back to the real scroll offset, and the header is no longer floating.
  - Once the effective scroll offset has been updated, the child is laid out using the effective scroll offset, the maximum extent, and an overlap flag.
    - The header may overlap content whenever it is floating (i.e., its effective scroll offset is less than its actual scroll offset).
  - Finally, the header's geometry is computed and the child is positioned.
- `RenderSliverFloatingPersistentHeader.updateGeometry` computes the header's geometry as well as the child's final position using the effective and actual scroll offsets.
  - `SliverGeometry.scrollExtent` : always the max extent since this is how much scrolling is needed to scroll past the header.

        - `SliverGeometry.paintOrigin` : always paints on the first clean pixel to avoid overlapping earlier slivers.
        - `SliverGeometry.paintExtent` : the largest visible portion of the header (using its effective offset), clamped to the remaining paint extent.
        - `SliverGeometry.layoutExtent` : the floating header will consume the portion of its maximum extent that is actually visible (i.e., it only takes up space when its true layout position falls within the viewport). Otherwise (e.g., when floating), it occupies zero layout space and therefore may overlap any subsequent slivers. Clamped to remaining paint extent.
        - `SliverGeometry.maxScrollObstructionExtent` : the maximum extent, since this is the most that the viewport can be obscured when floating.
    - The child's position is calculated using the sliver's effective trailing edge (i.e., as measured using the sliver's effective scroll offset). When the sliver's actual position precedes the viewport's leading edge, its layout offset will typically be zero, and thus the sliver will paint at the viewport's leading edge.

How does pinning and floating work together?

- `RenderSliverFloatingPinnedPersistentHeader` is identical to its parent (`RenderSliverFloatingPersistentHeader`) other than in how it calculates its geometry and child's position. Like its parent, the header will reappear whenever the user scrolls toward it. However, the header will remained pinned at the viewport's leading edge even when the user scrolls away.
    - The calculated geometry is almost identical to its parent. The key difference is that the pinned, floating header always paints at least its minimum extent (room permitting). Additionally, its child is always positioned at zero since painting always occurs at the viewport's leading edge.
        - `SliverGeometry.scrollExtent` : always the max extent since this is how much scrolling is needed to scroll past the header (though it will continue to paint itself).
        - `SliverGeometry.paintOrigin` : always paints on the first clean pixel to avoid overlapping earlier slivers.
        - `SliverGeometry.paintExtent` : the largest visible portion of the header (using its effective offset). Never less than the minimum extent (i.e., the child will

always be fully painted), never more than the remaining paint extent. Unlike the non-floating pinned header, this will vary so that the header visibly grows to its maximum extent.

- `SliverGeometry.layoutExtent` : the pinned, floating header will consume the portion of its maximum extent that is actually visible (i.e., it only takes up space when its true layout position falls within the viewport). Otherwise (e.g., when floating), it occupies zero layout space and therefore may overlap any subsequent slivers. Clamped to the paint extent, which may be less than the remaining paint extent if still growing between minimum and maximum extent.
- `SliverGeometry.maxScrollObstructionExtent` : the maximum extent, since this is the most that the viewport can be obscured when floating.

Container Slivers

How do slivers with multiple children manage parent data?

- `SliverMultiBoxAdaptorParentData` extends `SliverLogicalParentData` to support multiple box children (via `ContainerParentDataMixin<RenderBox>`) as well as the keep alive protocol (via `KeepAliveParentDataMixin`), allowing certain children to be cached when not visible. In addition to tracking each child's layout offset and neighbors in the child list, this parent data also contains an integer index assigned by the manager.
- `SliverGridParentData` is a subclass of `SliverMultiBoxAdaptorParentData` that also tracks the child's cross axis offset (i.e., the distance from the child's left or top edge from the parent's left or top edge, depending on whether the axis is vertical or horizontal, respectively).

What are the building blocks for slivers with multiple children?

- `RenderSliverMultiBoxAdaptor` provides a base class for slivers that manage multiple box children to efficiently fill remaining space in the viewport. Generally, subclasses will create children dynamically based on the current scroll offset and remaining paint extent. A manager (the `RenderSliverBoxChildManager`) provides an interface to create, track, and remove children in response to layout; this allows a clean separation between how children are laid out and how they are produced. Subclasses implement layout themselves, delegating to the superclass as needed. Only visible and cached children are associated with the render object at any given time; cached children are managed using the keep alive protocol (via `RenderSliverWithKeepAliveMixin`).
    - Conceptually, this adaptor adds visibility pruning to the sliver protocol. Ordinarily, all slivers are laid out as the viewport is scrolled -- including those that are off screen. Using this adaptor, children are created and destroyed only as they become visible and invisible. Once destroyed, children are removed from the render tree, eliminating unnecessary building and layout.
    - Additional constraints are imposed on the children of this render object:

        - Children cannot be removed once they've been laid out in a given pass.
        - Children can only be added by the manager, and then only with a valid index (i.e., an unused index).
- `RenderSliverBoxChildManager` provides a bidirectional interface for managing, creating, and removing box children. This interface serves to decouple the `RenderSliverMultiBoxAdaptor` from the source of children.
    - The manager uses parent data to track child indices. These are integers that provide a stable, unique identifier for children as they are created and destroyed.
- `SliverMultiBoxAdaptorElement` implements the `RenderSliverBoxChildManager` interface to link element creation and destruction to an associated widget's `SliverChildDelegate`. This delegate typically provides (or builds) children on demand.
- `RenderSliverFixedExtentBoxAdaptor` is a subclass of `RenderSliverMultiBoxAdaptor` that arranges a sequence of box children with equal main axis extent. This is more efficient since the adaptor may reckon each child's position directly (i.e., without layout). Children that are laid out are provided tight constraints.

What are the building blocks for managing children?

- `SliverChildDelegate` is a subclass that assists in generating children dynamically and destroying those children when they're no longer needed (it also supports destruction mitigation to avoid rebuilding expensive subtrees). Children are provided on demand (i.e., lazily) using a build method that accepts an index and produces the corresponding child.
- `SliverChildListDelegate` is a subclass of `SliverChildDelegate` that provides children using an explicit list. This negates the benefit of only building children when they are in view. This delegate may also wrap produced children in `RenderBoundary` and `AutomaticKeepAlive` widgets.
- `SliverChildBuilderDelegate` is a subclass of `SliverChildDelegate` that provides children by building them on demand. This improves performance by only building those children that are currently visible. This delegate may also wrap produced children in `RenderBoundary` and `AutomaticKeepAlive` widgets.

What are the widget building blocks for dynamic slivers?

- `SliverMultiBoxAdaptorWidget`
- `SliverChildDelegate`

How does keep alive work?

- Keep alive is a mechanism for retaining the render object, element, and state associated with an item when that item might otherwise have been destroyed (i.e., because it is scrolled out of view).
- `RenderSliverMultiBoxAdaptor` incorporates `RenderSliverWithKeepAliveMixin` to ensure that its associated parent data incldues the `KeepAliveParentDataMixin`. This mixin introduces "`keepAlive`" / "`keptAlive`" flags that form the basis of all caching decisions (the former indicates whether keep alive behavior is requested; the latter indicates whether the item is currently being kept alive).
- The "`keepAlive`" flag signals that the associated child is to be retained even when it would have been destroyed. This flag is altered by `KeepAlive`, a parent data widget associated with the `KeepAliveParentDataMixin`.
  - If an item is to be marked as keep alive, no additional work is necessary; the item must have previously been alive, else it would have been destroyed.
  - If an item is to be unmarked as keep alive, its parent must perform layout so that the item may be cleaned up (i.e., if it would have been destroyed).
- The `KeepAlive` widget can be applied out of turn (i.e., `KeepAlive.debugCanApplyOutOfTurn` returns true). This implies that the associated element can alter the child's parent data without triggering a rebuild (via `ParentDataElement.applyWidgetOutOfTurn`). This allows parent data to be altered in the middle of building, layout, and painting.
  - This is useful when handling lists since their children may request to be kept alive during building (via `AutomaticKeepAlive`). Since enabling this feature will never invalidate the parent's own layout (see discussion above), this is always safe. The benefit is that the bit can be flipped without requesting another frame.
- `AutomaticKeepAlive` manages an immediate `KeepAlive` child based on the `KeepAliveNotifications` emitted by descendent widgets.

    - These notifications indicate that the subtree is to be kept alive (and must therefore use the `KeepAliveParentDataMixin`).
    - A handler associated with the notification is invoked by the client to indicate that it may be released. If the client is deactivated (i.e., removed from the tree), it must call the handler; else, the widget will be leaked. The handler must first be called if a new `KeepAliveNotification` is generated.
    - `AutomaticKeepAliveClientMixin` provides helpers to help `State` subclasses appropriately manage `KeepAliveNotifications`.
- The "`keepAlive`" flag is honored by `RenderSliverMultiBoxAdaptor`, which maintains a keep alive bucket and ensures that the associated children remain in the render tree but are not actually rendered.

What services must the child manager provide?

- `RenderSliverBoxChildManager.childCount`: the manager must provide an accurate child count if there are finite children.
- `RenderSliverBoxChildManager.didAdoptChild`: invoked when a child is adopted (i.e., added to the child list) or needs a new index (e.g., moved within the list). This is the only place where the child's index is set.
    - Note that any operation that might change the index (e.g., `RenderSliverMultiBoxAdaptor.move`, which is called indirectly by `RenderObjectElement.updateChild` when the child order changes) will always call `RenderSliverBoxChildManager.didAdoptChild` to update the index.
- `RenderSliverBoxChildManager.didStartLayout`, `RenderSliverBoxChildManager.didFinishLayout`: invoked at the beginning and end of layout.
    - In `SliverMultiBoxAdaptorElement`, the latter invokes a corresponding method on the delegate providing the first and last indices that are visible in the list.
- `RenderSliverBoxChildManager.setDidUnderflow`: indicates that the manager could not fill all remaining space with children. Generally invoked unconditionally at the beginning of layout (i.e., without underflow), then again if there is space left over in the viewport (i.e., with underflow).
    - Used to determine whether additional children will affect what's visible in the list.

- `RenderSliverBoxChildManager.estimateMaxScrollOffset` : estimates the maximum scroll extent that can be consumed by all visible children. Provided information about the children current in view (first and last indices and leading and trailing scroll offsets).
  - In `SliverMultiBoxAdaptorElement` , this defers to the associated widget. If no implementation is provided, it multiples the average extent (calculated for all visible children) by the overall child count to obtain an approximate answer.
  - Leading and trailing scroll offset describe the leading and trailing edges of the children, even if the leading and trailing children aren't entirely in view.
- `RenderSliverBoxChildManager.createChild` : returns a child with the given index after incorporating it into the render object (creates the initial child if a position isn't specified). May cache previously built children.
  - In `SliverMultiBoxAdaptorElement` , this method utilizes `Element.updateChild` to update or inflate a new child using the widget tree built by a `SliverChildDelegate` .
    - All children are cached once built. This cache is cleared whenever the list is rebuilt since the associated delegate may have changed [?].
  - If a child is updated (e.g., because it was found in the cache), it will already have the correct index.
  - If the child is inflated, the index will need to be set (via `RenderSliverBoxChildManager.didAdoptChild`).
    - `Element.inflateWidget` causes the render object (`RenderSliverMultiBoxAdaptor`) to create and mount the child, which inserts it into a slot associated with the child's index (via `SliverMultiBoxAdaptorElement.insertChildRenderObject`). This invokes `RenderSliverMultiBoxAdaptor.insert` to update the render object's child list and adopt the new child. This calls back into the manager, which sets the index appropriately (the index is stored in a member variable).
- `RenderSliverBoxChildManager.removeChild` : removes a child, generally in response to "garbage collection" when the child is no longer visible; also used to destroy a child that was previously kept alive.
  - In `SliverMultiBoxAdaptorElement` , this method invokes `Element.updateChild` with the new widget set to null, causing the child to be deactivated and detached. This invokes `SliverMultiBoxAdaptorElement.removeChildRenderObject` using the child's index as the slot identifier, which calls `RenderSliverMultiBoxAdaptor.remove` to update the render object's child list.

Once updated, the child render object is dropped and removed. If the child had been kept alive, it is removed from the keep alive bucket, instead.

How are children assigned indices?

- Concrete indices are provided to the child manager when producing children (e.g., building via the child delegate). These are computed incrementally, starting from zero. Sequential indices are computed based on items currently in view and where the child is added (before or after the current children). Negative indices are valid.
- The only method that updates the index stored in `SliverMultiBoxAdaptorParentData` is `RenderSliverBoxChildManager.didAdoptChild`.
- This method is invoked whenever the render object adds a child to its child list. It is not invoked when the keep alive bucket is altered (i.e., it only considers the effective child list).
- Changes are generally driven by the widget layer and the `SliverChildDelegate` in particular. As the list's manager (and a convenient integration point between the render tree and the widget), `SliverMultiBoxAdaptorElement` facilitates the process.
  - `SliverMultiBoxAdaptorElement` tracks when children are created and destroyed (either explicitly or during a rebuild), setting `SliverMultiBoxAdaptorElement._currentlyUpdatingChildIndex` to the intended index just before the render tree is updated. The index also serves as the slot for overridden `RenderObjectElement` operations.
  - Updates pass through the element to the `RenderSliverMultiBoxAdaptor` (according to `Flutter`'s usual flow), which invokes `RenderSliverBoxChildManager.didAdoptChild` any time the index might change.
  - Finally, the currently updating index is written to the child's parent data by the manager.

How does a multi-box adaptor manage its children?

- The multi-box adaptor effectively has two child lists: the effective child list, as implemented by the `ContainerRenderObjectMixin`, and a keep alive bucket, managed

separately.

- The keep alive bucket associates indices to cached render box children. The children in this list are kept alive, and therefore attached to the render tree, but are not otherwise visible or interactive.
    - Keep alive children are not included in the effective child list. Therefore, the helpers provided by `ContainerRenderObjectMixin` do not interact with these children (though a few operations have been overridden).
    - Keep alive children remain attached to the render tree in all other respects required by the render object protocol (they are adopted, dropped, attached, detached, etc).
- Both child models (i.e., the child list and the keep alive bucket) are considered for non-rendering operations. The keep alive bucket is ignored when iterating for semantics, notifying the manager when a child is adopted, hit testing, painting, and performing layout.
- Child render boxes are obtained via `RenderSliverMultiBoxAdaptor._createOrObtainChild`, which consults the keep alive bucket before requesting the manager to provide a child. Note that the manager incorporates an additional caching layer (i.e., to avoid rebuilding children unless the entire list has been rebuilt).
    - If the child is found in the keep alive bucket, it is first dropped (i.e., because it is being moved to a different child model) and then re-inserted into the child list. The list is marked dirty because its children have changed.
    - If the manager provides the child, it also inserts it into the render object's child list.
- Child render boxes are destroyed by `RenderSliverMultiBoxAdaptor._destroyOrCacheChild`, which consults the keep alive flag before requesting the manager to remove the child.
    - If the child has the keep alive flag enabled, it is first removed from the child list and then readopted (i.e., switched to the keep alive child model). The manager is not notified of this adoption. Last, the list is marked dirty because its children have changed.
    - If the manager destroys the child, it also removes it from the render object's child list.
- The child list may be manipulated using the usual render object container methods. Insert, remove, remove all, and move have been overridden to support both child models (i.e., keep alive and container).
    - `RenderSliverMultiBoxAdaptor.insert` cannot be used to add children to the keep alive bucket.

- The initial child is bootstrapped via `RenderSliverMultiBoxAdaptor.addInitialChild` to allow the initial layout offset and starting index to be specified. The child is sourced from the keep alive bucket or, if not found, the manager. If no child can be provided, underflow is indicated. Importantly, this child is not laid out.
- Subsequent children are inserted and laid out via `RenderSliverMultiBoxAdaptor.insertAndLayoutLeadingChild` and `RenderSliverMultiBoxAdaptor.insertAndLayoutChild`. The former method positions children at the head of the child list whereas the latter positions them at its tail.
    - Newly inserted leading children become `RenderSliverMultiBoxAdaptor.firstChild`; non-leading children may or may not become `RenderSliverMultiBoxAdaptor.lastChild` depending on where they're inserted.
    - If the child was successfully inserted, it is then laid out with the provided constraints. Otherwise, underflow is indicated.
- Children that are no longer visible are removed (or shifted to the keep alive bucket) via `RenderSliverMultiBoxAdaptor.collectGarbage`. This method destroys a number of children at the start and end of the list. It also cleans up keep alive overhead, and so must always be called during layout.
    - Iterates the keep alive bucket to find children that no longer have the keep alive flag set (recall that toggling the flag doesn't force a rebuild). Such children must be destroyed by the manager; this is what will actually remove them from the keep alive bucket (via `RenderSliverMultiBoxAdaptor.remove`).

How is the multi-box adaptor rebuilt?

- Any time the adaptor is rebuilt (e.g., because the delegate has changed), all children must also be rebuilt. In some cases, additional children may be built, too.
    - Rebuilding may update or remove existing children, causing the list to relayout.
    - Relayout will cause additional children to be built if there is space in the viewport and the new delegate can provide them.
- Recall that updating or creating a render object (e.g., in response to `Element.updateChild`) will modify fields in that render object. Most render objects will schedule layout, painting, etc., in response to such changes.

- The element associated with the multi-box adaptor (`SliverMultiBoxAdaptorElement`) maintains an ordered cache of all child elements and widgets that have been built since the list itself was rebuilt. This list includes keep alive entries.
    - This cache must be recreated whenever the list is rebuilt since the delegate may itself produce different children.
- Rebuilding attempts to preserve children (i.e., by matching keys) while gracefully handling index changes. Children are updated with the rebuilt widget so that unchanged subtrees can be reused.
    - A candidate set of child elements is computed by assigning all the old elements their original indices.
    - If a child is being moved (as determined by `SliverChildDelegate.findIndexByKey`), that child is unconditionally assigned its new index.
        - If an earlier item was assigned that index, it will be overwritten and deactivated.
    - All candidate children are built and updated (via `SliverMultiBoxAdaptorElement.updateChild`). This will attempt to reuse elements where possible. If a candidate cannot be built (i.e., because the delegate does not build anything at that index), it is deactivated and removed.
        - The element and widget cache is repopulated with those children that are successfully built.
        - If at any point the adaptor's child list is mutated (i.e., because a child is added or removed), it will be dirtied for layout, painting, and compositing.
        - If a child is being kept alive (i.e., not visible but otherwise retained), care is taken to avoid using when manipulating the child list (e.g., it never serves as the "after" argument when inserting children).
    - Last, if the list underflowed during the most recent layout attempt, the element will attempt to build one additional child (using the next available index). If this produces a child, it will be inserted into its parent's child list, scheduling a relayout. This, in turn, will provide the delegate an opportunity to build additional children to fill the remaining space in the viewport.
    - The child associated with the greatest index is tracked as `SliverMultiBoxAdaptorElement._currentBeforeChild` . This serves as the "after" argument whenever the child list is manipulated, preserving the index order.

How does the multi-box adaptor paint its children?

- The child's position relative to the visible portion of the viewport is calculated using the incoming scroll offset and the child's layout position in the list. These quantities are mapped to a concrete offset from the top-left corner of the canvas, taking into account the viewport's axis direction. If the child intersects with the viewport, it is painted at the resulting offset.

How are children laid out when their extent is fixed?

- `RenderSliverFixedExtentList` is a trivial `RenderSliverFixedExtentBoxAdaptor` subclass that lays out a sequence of children using a fixed main axis extent in order and without gaps (cross axis extent is determined from the incoming constraints).
- Calculations are reckoned directly (using index and item extent). Indices start at zero and increase along the sliver's axis direction. Items are laid out at the position corresponding to the item's index multiplied by item extent. Max scroll extent is trivially the number of children multiplied by item extent. Two calculations, however, are a bit confusing:
  - `RenderSliverFixedExtentBoxAdaptor.getMaxChildIndexForScrollOffset`: returns the maximum index for the child at the current scroll offset or earlier.
    - Conceptually, this computes the maximum index of all children preceding or intersecting the list's leading edge (i.e., including off-screen children).
  - `RenderSliverFixedExtentBoxAdaptor.getMinChildIndexForScrollOffset`: returns the minimum index for the child at the current scroll offset or later.
    - Conceptually, this computes the minimum index for all children exceeding or intersecting the list's leading edge (i.e., including on-screen children).
  - These are identical except when children do not intersect the leading edge (i.e., the leading edge is coincident with the seam between children).
- Layout is performed sequentially, with new children retrieved from the manager as space allows or no more children are available. Depending on the `SliverChildDelegate` in use, this might lead to the interleaving of build and layout phases (i.e., if children are built on demand rather than provided up front). Indices are assigned sequentially as layout progresses and relevant lifecycle methods within the manager are invoked (e.g., `RenderSliverBoxChildManager.didStartLayout`).

- Target first and last indices are computed based on the children that would be visible given the scroll offset and the remaining paint extent.
  - Shrink wrapping viewports have infinite extent. In this case, there is no last index.
- Any children that were visible but are now outside of the target index range are garbage collected (via `RenderSliverMultiBoxAdaptor.collectGarbage`). This also cleans up any expired keep alive children.
- If an initial child hasn't been attached to the list, it is created but not laid out (via `RenderSliverMultiBoxAdaptor.addInitialChild`). This establishes the initial index and layout offset as all other children are positioned relative to this one.
  - If this fails, layout concludes with scroll extent and maximum paint extent set to the estimated maximum extent. All other geometry is zero.
- If indices have become visible that precede the first child's index, the corresponding children are built and laid out (via `RenderSliverMultiBoxAdaptor.insertAndLayoutLeadingChild`).
  - These children will need to be built since they could not have been attached to the list by assumption.
  - If one of these children cannot be built, layout halts with a scroll offset correction. Though this correction is currently computed incorrectly, the intent is to scroll the viewport such that the leading edge is coincident with the earliest, successfully built child.
- Identify the child with the largest index that has been built and laid out so far. This is the trailing child.
  - If there were leading children, this will be the leading child adjacent to the initial child. If not, this is the initial child itself (which is now laid out if necessary).
- Lay out every remaining child until there is no more room (i.e., the target index is reached) or no more children (i.e., a child cannot be built). Update the trailing child as layout progresses.
  - The trailing child serves as the "after" argument when inserting children (via `RenderSliverMultiBoxAdaptor.insertAndLayoutChild`).
  - The children may have already been attached to the list. If so, the child is laid out without being rebuilt.
  - Layout offsets are calculated directly using the child's index and the fixed extent.
- Compute the estimated maximum extent using the first and last index that were actually reified as well as the enclosing leading and trailing scroll offsets.

- Return the resulting geometry:
  - `SliverGeometry.scrollExtent` : estimated maximum extent (this is correct for fixed extent lists).
  - `SliverGeometry.maxPaintExtent` : estimated maximum extent (this is the most that can be painted).
  - `SliverGeometry.paintExtent` : the visible portion of the range defined by the leading and trailing scroll offsets.
  - `SliverGeometry.hasVisualOverflow` : true if the target index wasn't reached or the list precedes the viewport's leading edge (i.e., incoming scroll offset is greater than zero).
- If the list was fully scrolled, it will not have had an opportunity to lay out children. However, it is still necessary to report underflow to the manager.

Dynamic Slivers

What are a few common types of dynamic slivers?

- `RenderSliverList` positions children of varying extents in a linear array along the main axis. All positioning is relative to adjacent children in the list. Since only visible children are materialized, and earlier children may change extent, positions are occasionally corrected to maintain a consistent state.
- `RenderSliverGrid` positions children in a two dimensional arrangement determined during layout by a `SliverGridDelegate`. The positioning, spacing, and size of each item is generally static, though the delegate is free to compute an arbitrarily complex layout. The current layout strategy does not support non-fixed extents.

How do lists without variable item extent perform layout?

- When extent is not fixed, position cannot be directly computed from a child's index. Instead, position must be measured by laying out all preceding children. Since this would be quite expensive and eliminate the benefit of viewport culling, an optimization is used whereby previously laid out children serve to anchor newly revealed children (i.e., positions are determined relative to previously laid out children).
    - Note that this can lead to inconsistencies if off-screen children change size or are removed; scroll offset corrections are applied to address these errors.
- Layout proceeds by establishing the range of children currently visible in the viewport; that is, the range of children beginning from the child starting at or intersecting the viewport's leading edge to the child ending at or intersecting its trailing edge. Unlike a fixed extent list, this range cannot be computed directly but must be measured by traversing and building children starting from the first known child. Layout is computed as follows:
    - Ensure that there's a first known child (since, as mentioned, layout is performed relative to previously laid out children).
        - If there isn't, create (but do not layout) this initial child with an offset and index of zero. If this fails, the list has zero extent and layout is complete.
    - If the first known child does not precede or start at the viewport's leading edge, build children toward the leading edge until such a child is identified.

        - Leading children must be built as the walk progresses since they must not have existed (else, there would have been a different first known child). As part of building, the item is inserted into the list's child model.
        - If there are no more children to build, the last successfully processed child is positioned at offset zero (i.e., bumped up to the top of the list).
            - If the incoming scroll offset is already zero, then the walk ends; this child satisfies the search criteria.
            - If the scroll offset is non-zero, a scroll offset correction is needed.
                - A non-zero offset implies that a portion of the list precedes the leading edge. Given that there aren't even enough children to reach the leading edge, this cannot be the case.
                - The correction ensures that the incoming scroll offset will be zero when layout is reattempted. Since the earliest child is now positioned at offset zero, the inconsistency is corrected.
        - The newly built child's scroll offset is computed by subtracting its paint extent from the last child's scroll offset.
            - If the resulting offset is negative, a scroll offset correction is needed.
                - A negative offset implies that there is insufficient room for the child. The last child's offset represents the number of pixels available before that child; if this is less than the new child's extent, the list is too small and the incoming scroll offset must be invalid.
                - All preceding children that do not fit (including the one that triggered this case) are built and measured to determine their total extent. The earliest such child is positioned at offset zero.
                - A scroll offset correction is calculated to allow sufficient room for the overflowing children while ensuring that the last processed child appears at the same visual location.
                    - This quantity is the total extent needed minus the last walked item's scroll offset.
        - Position the child at the calculated scroll offset.
    - The first child in the list must now precede or start at the viewport's leading edge. Ensure that it has been laid out (e.g., if the preceding walk wasn't necessary).
    - Advance to find the child starting at or intersecting with the viewport's leading edge (there may have already been several such children in the child list). Then, advance to find the child ending at or intersecting with the viewport's trailing edge.
        - Advance by identifying the next child in the list while incrementing an index counter to detect gaps.

            - If the next child hasn't been built or a gap is detected, build and layout the child at the current index.
                - If no more children can be built, report failure.
            - If the next child hasn't been laid out, lay it out now.
        - While advancing, position each child directly after the preceding child.
        - If advancing fails before the leading edge is reached, remove all but the latest such child (via garbage collection). Maintain this child as it captures the list's total dimensions (i.e., its position plus its paint extent corresponds to the list's scroll extent).
            - Complete layout with zero paint extent, using the last item to compute overall scroll and maximum paint extents.
    - Count the children that wholly precede the viewport's leading edge. Once the trailing child is found, count all children following it. Remove the corresponding children via garbage collection since they are no longer visible.
    - Return the resulting geometry:
        - `SliverGeometry.scrollExtent` : estimated maximum extent (this is correct for fixed extent lists).
        - `SliverGeometry.maxPaintExtent` : estimated maximum extent (this is the most that can be painted).
        - `SliverGeometry.paintExtent` : the portion of reified children that are actually visible (via `RenderSliver.calculatePaintOffset`).
        - `SliverGeometry.hasVisualOverflow` : true if the trailing child extends beyond the viewport's leading edge, or the list precedes the viewport's leading edge (i.e., incoming scroll offset is greater than zero).
    - If the list was fully scrolled, it will not have had an opportunity to lay out children. However, it is still necessary to report underflow to the manager.

What are the building blocks of grid layout?

- `SliverGridGeometry` captures the geometry of an item within a grid. This encompasses a child's scroll offset, cross axis offset, main axis extent, and cross axis extent.
- `SliverGridParentData` extends `SliverMultiBoxAdaptorParentData` to include the child's cross axis offset. This is necessary since multiple children within a grid can

share the same scroll offset while appearing at different cross axis offsets.
- `SliverGridLayout` encapsulates positioning, sizing, and spacing logic. It is consulted during layout to determine the position and size of each grid item. This information is provided by returning the minimum and maximum index for a given scroll offset (via `SliverGridLayout.getMinChildIndexForScrollOffset` and `SliverGridLayout.getMaxChildIndexForScrollOffset`), as well as the grid geometry for a given index (via `SliverGridLayout.getGeometryForChildIndex`).
- `SliverGridDelegate` builds a `SliverGridLayout` subclass on demand (i.e., during grid layout). This allows the calculated layout to adjust to incoming sliver constraints.
- `SliverGridRegularTileLayout` calculates a layout wherein children are equally sized and spaced. As such, all aspects of layout are computed directly (i.e., without measuring adjacent children).
- `SliverGridDelegateWithFixedCrossAxisCount` configures a `SliverGridRegularTileLayout` such that the same number of children appear at a given scroll offset. Children are sized to share the available cross axis extent equally.
- `SliverGridDelegateWithMaxCrossAxisExtent` configures a `SliverGridRegularTileLayout` such that tiles are no larger than the provided maximum cross axis extent. A candidate extent is calculated that divides the available space evenly (i.e., without a remainder) and that is as large as possible.

How do grids perform layout?

- Grids are laid out similarly to fixed extent lists. Once a layout is computed (via `SliverGridDelegate.getLayout`), it is treated as a static description of the grid (i.e., positioning is absolute). As a result, children do not need to be reified to measure extent and position. Layout proceeds as follows:
  - Compute a new `SliverGridLayout` via `SliverGridDelegate`, providing the incoming constraints..
  - Target first and last indices are computed based on the children that would be visible given the scroll offset and the remaining paint extent (via `SliverGridLayout.getMinChildIndexForScrollOffset` and `SliverGridLayout.getMaxChildIndexForScrollOffset`, respectively).
    - Shrink wrapping viewports have infinite extent. In this case, there is no last index.

- Any children that were visible but are now outside of the target index range are garbage collected (via `RenderSliverMultiBoxAdaptor.collectGarbage`). This also cleans up any expired keep alive children.
- If there are no children attached to the grid, insert (but do not lay out) an initial child at the first index.
  - If this child cannot be built, layout is completed with scroll extent and maximum paint extent set to the calculated max scroll offset (via `SliverGridLayout.computeMaxScrollOffset`); all other geometry remains zero.
- All children still attached to the grid fall in the visible index range and there is at least one such child.
- If indices have become visible that precede the first child's index, the corresponding children are built and laid out (via `RenderSliverMultiBoxAdaptor.insertAndLayoutLeadingChild`).
  - These children will need to be built since they could not have been attached to the grid by assumption.
  - If one of these children cannot be built, layout will fail. This is likely a bug.
- Identify the child with the largest index that has been built and laid out so far. This is the trailing child.
  - If there were leading children, this will be the leading child adjacent to the initial child. If not, this is the initial child itself (which is now laid out if necessary).
- Lay out every remaining child until there is no more room (i.e., the target index is reached) or no more children (i.e., a child cannot be built). Update the trailing child as layout progresses.
  - The trailing child serves as the "after" argument when inserting children (via `RenderSliverMultiBoxAdaptor.insertAndLayoutChild`).
  - The children may have already been attached to the grid. If so, the child is laid out without being rebuilt.
  - Layout offsets for both the main and cross axes are assigned according the geometry reported by the `SliverGridLayout` (via `SliverGridLayout.getGeometryForChildIndex`).
- Compute the estimated maximum extent using the first and last index that were actually reified as well as the enclosing leading and trailing scroll offsets.
- Return the resulting geometry:

        - `SliverGeometry.scrollExtent` : estimated maximum extent (this is correct for grids with fixed extent).
        - `SliverGeometry.maxPaintExtent` : estimated maximum extent (this is the most that can be painted).
        - `SliverGeometry.paintExtent` : the visible portion of the range defined by the leading and trailing scroll offsets.
        - `SliverGeometry.hasVisualOverflow` : always true, unfortunately.
    - If the list was fully scrolled, it will not have had an opportunity to lay out children. However, it is still necessary to report underflow to the manager.


Animation

Animation

How are animations scheduled?

- `Window.onBeginFrame` invokes `SchedulerBinding.handleBeginFrame` every frame, which runs all transient callbacks scheduled during the prior frame. Ticker instances utilize transient callbacks (via `SchedulerBinding.scheduleFrameCallback`), and are therefore evaluated at this point. All tickers update their measure of elapsed time using the same frame timestamp, ensuring that tickers tick in unison.
- `AnimationController` utilizes an associated `Ticker` to track the passage of time. When the ticker ticks, the elapsed time is provided to an internal simulation which transforms real time into a decimal value. The simulation (typically `_InterpolationSimulation`) interpolates between `AnimationController.lowerBound` and `AnimationController.upperBound` (if spanning the animation's full range), or `AnimationController.value` and `AnimationController.target` (if traversing from the current value to a new one), applying a curve if available. Listeners are notified once the simulation produces a new value.
- The animation's behavior (playing forward, playing backward, animating toward a target, etc) is a consequence of how this internal simulation is configured (i.e., by reversing the bounds, by altering the duration, by using a `_RepeatingSimulation` or `SpringSimulation`). Typically, the simulation is responsible for mapping from real time to a value representing animation progress.
  - In the general case, an `_InterpolationSimulation` is configured in `AnimationController._animateToInternal`.
  - Next, the controller's ticker is started, which advances the the simulation once per frame. The simulation is advanced using the elapsed time reported by the ticker.
  - Listeners are notified whenever the simulation is queried or reaches an endpoint, potentially changing the animation's status (`AnimationStatus`).
- Composing animations (e.g., via `_AnimatedEvaluation` or `_ChainedEvaluation`) works by proxying the underlying listenable (i.e., by delegating listener operations to the parent animation, which advances as described above).

What is an animation?

- An animation, as represented by `Animation<double>`, traverses from zero to one (and vice versa) over a user-defined interval (this is typically facilitated by an `AnimationController`, a special `Animation<double>` that advances in real time). The resulting value represents the animation's progress (i.e., a timing value) and is often fed into a chain of animatables or descendant animations. These are re-evaluated every time the animation advances and therefore notifies its listeners. Some descendants (e.g., curves) transform the animation's timing value into a new timing value; these affect the animation's rate of change (e.g., easing) but not its duration. Others produce derived values that can be used to update the UI (e.g., colors, shapes, and sizes). Repeatedly updating the UI using these values is the basis of animation.

What are the animation building blocks?

- `Animation<T>` couples `Listenable` with `AnimationStatus` and produces a sequence of values with a beginning and an end. The animation's status is derived from the sequence's directionality and whether values are currently being produced. In particular, the animation can be stopped at the sequence's beginning or end (`AnimationStatus.dimissed`, `AnimationStatus.completed`), or actively producing values in a particular order (`AnimationStatus.forward`, `AnimationStatus.reverse`). `Animation<T>` extends `ValueListenable` which produces a sequence of values but does not track status.
  - `Animation<double>` is the most common specialization of `Animation<T>` (and the only specialization that can be used with `Animatable<T>`); as a convention, `Animation<double>` produces values from zero to one, though it may overshoot this range before completing. These values are typically interpreted as the animation's progress (referred to as timing values, below). How this interval is traversed over time determines the behavior of any downstream animatables.
  - `Animation<double>` may represent other sequences, as well. For instances, an `Animation<double>` might describe a sequence of border radii or line thicknesses.
  - More broadly, `Animation<T>`, where `T` is not a timing value, is devoid of conventional significance. Such animations progress through their values as

described earlier, and are typically driven by a preceding `Animation<double>` (that does represent a timing value).

- `Animatable<T>` describes an animatable value, mapping an `Animation<double>` (which ranges from zero to one) to a sequence of derived values (via `Animatable<T>` .evaluate, which forwards the animation's value to `Animatable<T>.transform` to produce a new value of type `T`). The animatable may be driven by an animation (i.e., repeatedly evaluated as the animation generates notifications, via `Animatable<T>.animate`). It may also be associated with a parent animatable to create an evaluation chain (i.e., the parent evaluates the animation value, then the child evaluates the parent's value, via `Animatable<T>.chain`). Unless the parent animatable is driven by an animation, however, chaining will not cause the animatable to animate; it only describes a sequence of transformations.
  - `Animatable<T>.evaluate` always maps from a double to a value of type `T` . Conventionally, the provided double represents a timing value (ranging from zero to one), but this is not a requirement.
- Tween is a subclass of `Animatable<T>` that linearly interpolates between beginning and end values of type `T` (via `Tween.lerp`). By default, algebraic linear interpolation is used, though many types implement custom methods (via `T.lerp`) or provide overloaded operators.
  - `TweenSequence` is an animatable that allows an animation to drive a sequence of tweens, associating each with a portion of the animation's duration (via `TweenSequenceItem.weight`).
- Simulation models an object in one-dimensional space with a position (`Simulation.x`), velocity (`Simulation.dx`), and completion status (`isDone`) using logical units. Simulations are queried using a time value, also in logical units; as some simulations may be stateful, queries should generally use increasing values. A `Tolerance` instance specifies epsilon values for time, velocity, and position to determine when the simulation has settled.
- `AnimationController` is an `Animation<double>` subclass introducing explicit control and frame-synchronized timing. When active, the animation controller is driven at approximately 60 Hz. This is facilitated by a corresponding `Ticker` instance, a synchronized timer that triggers at the beginning of each frame; this instance may change over the course of the controller's lifespan (via `AnimationController.resync`). The animation can be run backwards and forwards (potentially without bounds), toward and away from target values (via `AnimationController.animateTo` and

`AnimationController.animateBack`), or cyclically (via `AnimationController.repeat`). Animations can also be driven by a custom `Simulation` (via `AnimationController.animateWith`) or a built-in spring simulation (via `AnimationController.fling`). The controller's value (`AnimationController.value`) is advanced in real time, using a duration (`AnimationController.duration`) to interpolate between starting and ending values (`AnimationController.upperBound`, `AnimationController.lowerBound`), both doubles. An immutable view of the animation is also exposed (`AnimationController.view`).

- Ticker invokes a callback once per frame (via a transient callback scheduled using `SchedulerBinding.scheduleFrameCallback`), passing a duration corresponding to how long the ticker has been ticking. This duration is measured using a timestamp set at the beginning of the frame (`SchedulerBinding.handleBeginFrame`). All tickers advance using the same timestamp and are therefore synchronized. When a ticker is enabled, a transient frame callback is registered via `SchedulerBinding.addFrameCallback`; this schedules a frame via `Window.scheduleFrame`, ensuring that the ticker will begin ticking.
  - Tickers measure a duration from when they first tick. If a ticker is stopped, the duration is reset and progress is lost. Muting a ticker allows time (the duration) to continue advancing while suppressing ticker callbacks. The animation will not progress while muted and will appear to jump ahead when unmuted. A ticker can absorb another ticker so that animation progress is not lost; that is, the new ticker will retain the old ticker's elapsed time.
  - `TickerFuture` exposes the ticker status as a `Future`. When stopped, this future resolves; in all other cases, the future is unresolved. A derivative future, `TickerFuture.orCancel`, extends this interface to throw an exception if the ticker is cancelled.
- `TickerProvider` vends `Ticker` instances. `TickerProviderStateMixin` and `SingleTickerProviderStateMixin` fulfill the `TickerProvider` interface within the context of a `State` object (the latter has less overhead since it only tracks a single ticker). These mixins query an inherited `TickerMode` that can enable and disable all descendent tickers en masse; this allows tickers to be muted and unmuted within a subset of the widget tree efficiently.
- `AnimationLocalListenersMixin` and `AnimationLocalStatusListenersMixin` provide implementations for the two listenable interfaces supported by animations: value listeners (`Animation.addListener`, `Animation.removeListener`), and status

listeners (`Animation.addStatusListener` , `Animation.removeStatusListener`). Both store listeners in a local `ObserverList` and support hooks indicating when a listener is registered and unregistered (`didRegisterListener` and `didUnregisterListener` , respectively). A number of framework subclasses depend on these mixins (e.g., `AnimationController`) since `Animation<T>` doesn't provide a concrete implementation.

- `AnimationLazyListenerMixin` uses the aforementioned hooks to notify the client when there are no more listeners. This allows resources to be released until a listener is once again added (via `AnimationLazyListenerMixin.didStartListening` and `AnimationLazyListenerMixin.didStopListening`).
- `AnimationEagerListenerMixin` ignores these hooks, instead introducing a dispose protocol; resources will be retained through the animation's lifespan and therefore must be disposed before the instance is released.

How are animations curved?

- Curve determines an animation's rate of change by specifying a mapping from input to output timing values (i.e., from [0, 1] to [0, 1], though some curves stretch this interval, e.g., `ElasticInCurve`). `Animation<double>` produces suitable input values that may then be transformed (via `Curve.transform`) into new timing values. Later, these values may be used to drive downstream animatables (or further transformed), effectively altering the animation's perceived rate of change.
  - Geometrically, a curve may be visualized as mapping an input timing value (along the X-axis) to an output timing value (along the Y-axis), with zero corresponding to `AnimationStatus.dismissed` and one corresponding to `AnimationStatus.completed` .
  - Curves cannot alter the overall duration of an animation, but will affect the rate that an animation is advanced during that interval. Additionally, even if they overshoot the unit interval, curves must map zero and one to values that round to zero or one, respectively.
  - There are a number of built-in curve instances:
    - Cubic defines a curve as a cubic function.

    - `ElasticInCurve`, `ElasticOutCurve`, `ElasticInOutCurve` define a spring-like curve that overshoots as it grows, shrinks, or settles, respectively.
    - Interval maps a curve to a subinterval, clamping to 0 or 1 at either end.
    - Threshold is 0 until a threshold is reached, then 1 thereafter.
    - `SawTooth` produces N linear intervals, with no interpolation at edges
    - `FlippedCurve` transforms an input curve, mirroring it both horizontally and vertically.
    - Curves exposes a large number of pre-defined curves.
- `CurvedAnimation` is an `Animation<double>` subclass that applies a curve to a parent animation (via `AnimationWithParentMixin`). As such, `CurvedAnimation` proxies the parent animation, transforming each value before any consumers may read it (via `Curve.transform`). `CurvedAnimation` also allows different curves to be used for forward and reverse directions.
- `CurveTween` is an `Animatable<double>` subclass that is analogous to `CurvedAnimation`. As an animatable, `CurveTween` delegates its transform (via `Animatable<double>.transform`) to the provided curve transform (via `Curve.transform`). Since `CurveTween` doesn't perform interpolation, but instead represents an arbitrary mapping, it isn't actually a tween.
- `AnimationController` includes built-in curve support (via `_InterpolationSimulation`). When the simulation is advanced to transform elapsed wall time into a timing value (by querying `_InterpolationSimulation.x`), if available, a curve is used when computing the new value. As the resulting value is generally interpreted as a timing value, this influences the perceived rate of change of the animation.

How are animations composed?

- `AnimationWithParentMixin` provides support for building animations that delegate to a parent animation. The various listener methods (`AnimationWithParentMixin.addListener`, `AnimationWithParentMixin.addStatusListener`) are forwarded to the parent; all relevant state is also read from the parent. Clients provide a value accessor that constructs a derivative value based on the parent's value.

- Composition is managed via `Animatable.chain` or `Animatable.animate`; `Animation<T>.drive` delegates to the provided animatable.
  - `_AnimatedEvaluation` is an `Animation<T>` that applies an animatable to a parent animation. All listenable methods delegate to the parent animation (via `AnimationWithParentMixin`); thus, the resulting animation is driven by the parent. The value accessor is overwritten so that parent's value may be transformed by the animatable (via `Animatable.evaluate`).
  - `_ChainedEvaluation` is an `Animatable<T>` that combines a parent animatable with a child animatable. In particular, `_ChainedEvaluation.transform` first evaluates the parent animatable (via `Animatable<T>.evaluate`), then passes this value to the child animatable.
- `CompoundAnimation` is an `Animation<T>` subclass that combines two animations. `CompoundAnimation.value` is overwritten to produce a final value using the first and second animation's values (via `CompoundAnimation.first`, `CompoundAnimation.next`). Note that `CompoundAnimation` is driven by two animations (i.e., it ticks when either animation ticks), unlike earlier composition examples that drive an animatable using a single parent animation.

What are the higher level animation building blocks?

- `ProxyAnimation` provides a read-only view of a parent animation that will reflect any changes to the original animation. It does this by proxying the animation listener methods as well as the status and value accessors. Additionally, `ProxyAnimation` supports replacing the parent animation inline; the transition is seamless from the perspective of any listeners.
- `TrainHoppingAnimation` monitors two animations, switching from the first to the second when the second emits the same value as the first (e.g., because it is reversed or moving toward the value more quickly). `TrainHoppingAnimation` utilizes `AnimationEagerListenerMixin` because it relies on the parent animations' notifications to determine when to switch tracks, regardless of whether there are any external listeners.
- `CompoundAnimation` combines two animations, ticking when either animation ticks (this differs from, e.g., `Animation.animate`, which drives an animatable via an

animation). The status is that of the second animation (if it's running), else the first. The values are combined by overriding the `Animation.value` accessor; the constituent animations are referenced as `CompoundAnimation.first` and `CompoundAnimation.next`, respectively. This animation is lazy – it will only listen to the sub-animations when it has listeners, and will avoid generating useless notifications.
    - `CompoundAnimation` is the basis of `MaxAnimation`, `MinAnimation`, and `MeanAnimation`.
- `AlwaysStoppedAnimation` exposes a constant value and never changes status or notifies listeners.
- `ReverseAnimation` plays an animation in reverse, using the appropriate status and direction. That is, if the parent animation is played forward (e.g., via `AnimationController.forward`), the `ReverseAnimation`'s status will be reversed. Moreover, the value reported by `ReverseAnimation` will be the inverse of the parent's value assuming a [0, 1] range (thus, one minus the parent's value). Note that this differs from simply reversing a tween (e.g., tweening from one to zero); though the values would be reversed, the animation status would be unchanged.

What are the highest level animation building blocks?

- `AnimatedWidget` is an abstract stateful widget that rebuilds whenever the provided listenable notifies its clients. When this happens, the associated state instance is marked dirty (via `_AnimatedState.setState`) and rebuilt. `_AnimatedState.build` delegates to the widget's build method, which subclasses must implement; these utilize the listenable (typically an animation) to update the UI.
- `AnimatedBuilder` extends `AnimatedWidget` to accept a build function (`TransitionBuilder`); this builder is invoked whenever the widget rebuilds (via `AnimatedBuilder.build`). This allows clients to utilize the `AnimatedWidget` flow without creating an explicit subclass.

How does implicit animation work?

- `ImplicitlyAnimatedWidget` provides support for widgets that animate in response to changes to selected properties; the initial value is not animated. Though descendant widgets are only able to customize the animation's duration and curve, `ImplicitlyAnimatedWidget` are often convenient in that they fully manage the underlying `AnimationController` .
  - Subclasses must use a `State` instance that extends `ImplicitlyAnimatedWidgetState` . Those that should be rebuilt (i.e., marked dirty) whenever the animation ticks extend `AnimatedWidgetBaseState` , instead.
    - `ImplicitlyAnimatedWidgetState.forEachTween` is the engine that drives implicit animation. Subclasses implement this method such that the provided visitor (`TweenVisitor`) is invoked once per implicitly animatable property.
    - The visitor function requires three arguments: the current tween instance (constructed by the superclass but cached locally, e.g., `ExampleState._opacityTween`), the target value (typically read from the widget, e.g., `ExampleState.widget.opacityValue`), and a constructor (`TweenConstructor`) that returns a new tween instance starting at the provided value. The visitor returns an updated tween; this value is typically assigned to the same field associated with the first argument.
    - Tweens are constructed during state initialization (via `ImplicitlyAnimatedWidgetState._constructTweens`) for all implicitly animatable properties with non-null target values (via `ImplicitlyAnimatedWidgetState.forEachTween`). Tweens may also be constructed outside of this context as they transition from null to non-null target values.
    - When the widget is updated (via `ImplicitlyAnimatedWidgetState.didUpdateWidget`), `ImplicitlyAnimatedWidgetState.forEachTween` steps through the subclass's animatable properties to update the tweens' bounds (via `ImplicitlyAnimatedWidgetState._updateTween`). The tween's start is set using the current animation value (to avoid jumping), with the tween's end set to the target value.
    - Last, the animation is played forward if the tween wasn't already animating toward the target value (i.e., the tween's previous endpoint didn't match the target value, via `ImplicitlyAnimatedWidgetState._shouldAnimateTweens`).
  - The subclass is responsible for using the animation (`ImplicitlyAnimatedWidgetState.animation`) and tween directly (i.e., by

evaluating the tween using the animation’s current value).


Assets

Asset Management

How are assets managed?

- `AssetBundle` is a container that provides asynchronous access to application resources (e.g., images, strings, fonts). Resources are associated with a string-based key and can be retrieved as bytes (via `AssetBundle.load`), a string (via `AssetBundle.loadString`), or structured data (via `AssetBundle.loadStructuredData`). A variety of subclasses support different methods for obtaining assets (e.g., `PlatformAssetBundle`, `NetworkAssetBundle`). Some bundles also support caching; if so, keys can be evicted from the bundle's cache (via `AssetBundle.evict`).
- `CachingAssetBundle` caches strings and structured data throughout the application's lifetime (unless explicitly evicted). Binary data is not cached since the higher level methods are built atop `AssetBundle.load`, and the final representation is more efficient to store.
- Every application is associated with a `rootBundle`. This `AssetBundle` contains the resources that were packaged when the application was built (i.e., as specified by `pubspec.yaml`). Though this bundle can be queried directly, `DefaultAssetBundle` provides a layer of indirection so that different bundles can be substituted (e.g., for testing or localization).

How are assets fetched?

- `NetworkAssetBundle` loads resources over the network. It does not implement caching; presumably, this is provided by the network layer. It provides a thin wrapper around dart's `HttpClient`.
- `PlatformAssetBundle` is a `CachingAssetBundle` subclass that fetches resources from a platform-specific application directory via platform messaging (specifically, `Engine` :: `HandleAssetPlatformMessage`).

Images

How are images represented?

- At the lowest level, images are represented as a `Uint8List` (i.e., an opaque list of unsigned bytes). These bytes can be expressed in any number of image formats, and must be decoded to a common representation by a coded.
- `instantiateImageCodec` accepts a list of bytes and returns the appropriate codec from the engine already bound to the provided image. This function accepts an optional width and height; if these do not match the image's intrinsic size, the image is scaled accordingly. If only one dimension is provided, the other dimension remains the intrinsic dimension. `PaintingBinding.instantiateImageCodec` provides a thin wrapper around this function with the intention of eventually supporting additional processing.
- Codec represents the application of a codec on a pre-specified image array. Codecs process both single frames and animated images. Once the `Codec` is retrieved via `instantiateImageCodec`, the decoded `FrameInfo` (which contains the image) may be requested via `Codec.nextFrame`; this may be invoked repeatedly for animations, and will automatically wrap to the first frame. The `Codec` must be disposed when no longer needed (the image data remains valid).
- `DecoderCallback` provides a layer of indirection between image decoding (via the `Codec` returned by `instantiateImageCodec`) and any additional decoding necessary for an image (e.g., resizing). It is primarily used with `ImageProvider` to encapsulate decoding-specific implementation details.
- `FrameInfo` corresponds to a single frame in an animated image (single images are considered one-frame animations). Duration, if application, is exposed via `FrameInfo.duration`. Otherwise, the decoded `Image` may be read as `FrameInfo.image`.
- Image is an opaque handle to decoded image pixels managed by the engine, with a width and a height. The decoded bytes can be obtained via `Image.toByteData` which accepts an `ImageByteFormat` specifying (e.g., `ImageByteFormat.rawRgba`, `ImageByteFormat.png`). However, the raw bytes are often not required as the `Image` handle is sufficient to paint images to the screen.
- `ImageInfo` associates an `Image` with a pixel density (i.e., `ImageInfo.scale`). Scale describes the number of image pixels per one side of a logical pixel (e.g., a scale of

`2.0` implies that each 1x1 logical pixel corresponds to 2x2 image pixels; that is, a 100x100 pixel image would be painted into a 50x50 logical pixel region and therefore have twice the resolution depending on the display).

What are the building blocks for managing image data?

- The image framework must account for a variety of cases that complicate image handling. Some images are obtained asynchronously; others are arranged into image sets so than an optimal variant can be selected at runtime (e.g., for the current resolution). Others correspond to animations which update at regular intervals. Any of these images may be cached to avoid unnecessary loading.
- `ImageStream` provides a consistent handle to a potentially evolving image resource; changes may be due to loading, animation, or explicit mutation. Changes are driven by a single `ImageStreamCompleter`, which notifies the `ImageStream` whenever concrete image data is available or changes (via `ImageInfo`). The `ImageStream` forwards notifications to one or more listeners (i.e., `ImageStreamListener` instances), which may be invoked multiple times as the image loads or mutates. Each `ImageStream` is associated with a key that can be used to determine whether two `ImageStream` instances are backed by the same completer [?].
- `ImageStreamListener` encapsulates a set of callbacks for responding to image events. If the image is being loaded (e.g., via the network), an `ImageChunkListener` is invoked with an `ImageChunkEvent` describing overall progress. If an image has become available, an `ImageListener` is invoked with the final `ImageInfo` (including a flag indicating whether the image was loaded synchronously). Last, if the image has failed to load, an `ImageErrorListener` is invoked.
    - The chunk listener is only called when an image must be loaded (e.g., via `NetworkImage`). It may also be called after the `ImageListener` if the image is an animation (i.e., another frame is being fetched).
    - The `ImageListener` may be invoked multiple times if the associated image is an animation (i.e., once per frame).
    - `ImageStreamListeners` are compared on the basis of the contained callbacks.
- `ImageStreamCompleter` manages image loading for an `ImageStream` from an asynchronous source (typically a `Codec`). A list of `ImageStreamListener` instances are notified whenever image data becomes available (i.e., the completer “completes”),

either in part (via `ImageStreamListener.onImageChunk`) or in whole (via `ImageStreamListener.onImage`). Listeners may be invoked multiple times (e.g., as chunks are loaded or with multiple animation frames). The completer notifies listeners when an image becomes available (via `ImageStreamCompleter.setImage`). Adding listeners after the image has been loaded will trigger synchronous notifications; this is how the `ImageCache` avoids refetching images unnecessarily.
    - The corresponding `Image` must be resolved to an `ImageInfo` (i.e., by incorporating scale); the scale is often provided explicitly.
    - `OneFrameImageStreamCompleter` handles one-frame (i.e., single) images. The corresponding `ImageInfo` is provided as a future; when this future resolves, `OneFrameImageStreamCompleter.setImage` is invoked, notifying listeners.
    - `MultiFrameImageStreamCompleter` handles multi-frame images (e.g., animations or engine frames), completing once per animation frame as long as there are listeners. If the image is only associated with a single frame, that frame is emitted immediately. An optional stream of `ImageChunkEvents` allows loading status to be conveyed to the attached listeners. Note that adding a new listener will attempt to decode the next frame; this is safe, if inefficient, as `Codec.getNextFrame` automatically cycles.
        - The next frame is eagerly decoded by the codec (via `Codec.getNextFrame`). Once available, a non-repeating callback is scheduled to emit the frame after the corresponding duration has lapsed (via `FrameInfo.duration`); the first frame is emitted immediately. If there are additional frames (via `Codec.frameCount`), or the animation cycles (via `Codec.repetitionCount`), this process is repeated. Frames are emitted via `MultiFrameImageStreamCompleter.setImage`, notifying all subscribed listeners.
        - In this way, the next frame is decoded eagerly but only emitted during the first application frame after the duration has lapsed. If at any point there are no listeners, the process is paused; no frames are decoded or emitted until a listener is added.
- A singleton `ImageCache` is created by the `PaintingBinding` during initialization (via `PaintingBinding.createImageCache`). The cache maps keys to `ImageStreamCompleters`, retaining only the most recently used entries. Once a maximum number of entries or bytes is reached, the least recently accessed entries are evicted. Note that any images actively retained by the application (e.g., `Image`,

`ImageInfo`, `ImageStream`, etc.) cannot be invalidated by this cache; the cache is only useful when locating an `ImageStreamCompleter` for a given key. If a completer is found, and the image has already been loaded, the listener is notified with the image synchronously.
    - `ImageCache.putIfAbsent` serves as the main interface to the cache. If a key is found, the corresponding `ImageStreamCompleter` is returned. Otherwise, the completer is built using the provided closure. In both cases, the timestamp is updated.
    - Because images are loaded asynchronously, the cache policy can only be enforced once the image loads. Thus, the cache maintains two maps: `ImageCache._pendingImages` and `ImageCache._cache`. On a cache miss, the newly built completer is added to the pending map and assigned an `ImageStreamListener`; when the listener is notified, the final image size is calculated, the listener removed, and the cache policy applied. The completer is then moved to the cache map.
    - If an image fails to load, it does not contribute to cache size but it does consume an entry. If an image is too large for the cache, the cache is expanded to accommodate the image with some headroom.
- `ImageConfiguration` describes the operating environment so that the best image can be selected from a set of alternatives (i.e., a double resolution image for a retina display); this is the primary input to `ImageProvider`. A configuration can be extracted from the element tree via `createLocalImageConfiguration`.
- `ImageProvider` identifies an image without committing to a specific asset. This allows the best variant to be selected according to the current `ImageConfiguration`. Any images managed via `ImageProvider` are passed through the global `ImageCache`.
    - `ImageProvider.obtainKey` produces a key that uniquely identifies a specific image (including scale) given an `ImageConfiguration` and the provider's settings.
    - `ImageProvider.load` builds an `ImageStreamCompleter` for a given key. The completer begins fetching the image immediately and decodes the resulting bytes via the `DecoderCallback`.
    - `ImageProvider.resolve` wraps both methods to (1) obtain a key (via `ImageProvider.obtainKey`), (2) query the cache using the key, and (3) if no completer is found, create an `ImageStreamCompleter` (via `ImageProvider.load`) and update the cache.

- `precacheImage` provides a convenient wrapper around `ImageProvider` so that a given image can be added to the `ImageCache`. So long as the same key is used for subsequent accesses, the image will be available immediately (provided that it has fully loaded).

How are images provided and painted?

- `ImageProvider` federates access to images, selecting the best image given the current environment (i.e., `ImageConfiguration`). The provider computes a key that uniquely identifies the asset to be loaded; this creates or retrieves an `ImageStreamCompleter` from the cache. Various provider subclasses override `ImageProvider.load` to customize how the completer is configured; most use `SynchronousFuture` to try to provide the image without needing to wait for the next frame. The `ImageStreamCompleter` is constructed with a future resolving to a bound codec (i.e., associated with raw image bytes). These bytes may be obtained in a variety of ways: from the network, from memory, from an `AssetBundle`, etc. The completer accepts an optional stream of `ImageChunkEvents` so that any listeners are notified as the image loads. Once the raw image has been read into memory, an appropriate codec is provided by the engine (via a `DecoderCallback`, which generally delegates to `PaintingBinding.instantiateImageCodec`). This codec is used to decode frames (potentially multiple times for animated images). As frames are decoded, listeners (e.g., an image widget) are notified with the finalized `ImageInfo` (which includes decoded bytes and scale data). These bytes may be painted directly via `paintImage`.

What image providers are available?

- `FileImage` provides images from the file system. As its own key, `FileImage` overrides the equality operator to compare the target file name and scale. A `MultiFrameImageStreamCompleter` is configured with the provided scale, and a `Codec` instantiated using bytes loaded from the file (via `File.readAsBytes`). The completer will only notify listeners when the image is fully loaded.

- `MemoryImage` provides images directly from an immutable array of bytes. As its own key, `MemoryImage` overrides the equality operator to compare scale as well as the actual bytes. A `MultiFrameImageStreamCompleter` is configured with the provided scale, and a `Codec` instantiated using the provided bytes. The completer will only notify listeners when the image is fully loaded.
- `NetworkImage` defines a thin interface to support different means of providing images from the network; it relies on instances of itself for a key.
    - `io.NetworkImage` implements this interface using `Dart`'s standard `HttpClient` to retrieve images. As its own key, `io.NetworkImage` overrides the equality operator to compare the target `URL` and scale. A `MultiFrameImageStreamCompleter` is configured with the provided scale, and a `Codec` instantiated using the consolidated bytes produced by `HttpClient.getUrl`. Unlike the other providers, `io.NetworkImage` will report loading status to its listeners via a stream of `ImageChunkEvents`. This relies on the "Content-Length" header being correctly reported by the remote server.
- `AssetBundleImageProvider` provides images from an `AssetBundle` using `AssetBundleImageKey`. The key is comprised of a specific asset bundle, asset key, and image scale. A `MultiFrameImageStreamCompleter` is configured with the provided scale, and a `Codec` instantiated using bytes loaded from the bundle (via `AssetBundle.load`). The completer will only notify listeners when the image is fully loaded.
    - `ExactAssetImage` is a subclass that allows the bundle, asset, and image scale to be set explicitly, rather than read from an `ImageConfiguration`.
    - `AssetImage` is a subclass that resolves to the most appropriate asset given a set of alternatives and the current runtime environment. Primarily, this subclass selects assets optimized for the device's pixel ratio using a simple naming convention. Assets are organized into logical directories within a given parent. Directories are named "Nx/", where N is corresponds to the image's intended scale; the default asset (with 1:1 scaling) is rooted within the parent itself. The variant that most closely matches the current pixel ratio is selected.
        - The main difference from the superclass is method by which keys are produced; all other functionality (e.g., `AssetImage.load`, `AssetImage.resolve`) is inherited.
        - A `JSON`-encoded asset manifest is produced from the pubspec file during building. This manifest is parsed to locate variants of each asset according to

the scheme described above; from this list, the variant nearest the current pixel ratio is identified. A key is produced using this asset's scale (which may not match the device's pixel ratio), its fully qualified name, and the bundle that was used. The completer is configured by the superclass.
    - The equality operator is overridden such that only the unresolved asset name and bundle are consulted; scale (and the best fitting asset name) are excluded from the comparison.
- `ResizeImage` wraps another `ImageProvider` to support size-aware caching. Ordinarily, images are decoded using their intrinsic dimensions (via `instantiateImageCodec`); consequently, the version of the image stored in the `ImageCache` corresponds to the full size image. This is inefficient for images that are displayed at a different size. `ResizeImage` addresses this by augmenting the underlying key with the requested dimensions; it also applies a `DecoderCallback` that forwards these dimensions via `instantiateImageCodec`.
    - The first time an image is provided, it is loaded using the underlying provider (via `ImageProvider.load`, which doesn't update the cache). The resulting `ImageStreamCompleter` is cached using the `ResizeImage`'s key (i.e., `_SizeAwareCacheKey`).
    - Subsequent accesses will hit the cache, which returns an image with the corresponding dimensions. Usages with different dimensions will result in additional entries being added to the cache.

What are the building blocks for image rendering?

- There are several auxiliary classes allowing image rendering to be customized. `BlendMode` specifies how pixels from source and destination images are combined during compositing (e.g., `BlendMode.multiply`, `BlendMode.overlay`, `BlendMode.difference`). `ColorFilter` specifies a function combining two colors into an output color; this function is applied before any blending. `ImageFilter` provides a handle to an image filter applied during rendering (e.g., gaussian blur, scaling transforms). `FilterQuality` allows the quality/performance of said filter to be broadly customized.
- Canvas exposes the lowest level API for painting images into layers. The principal methods include `Canvas.drawImage`, which paints an image at a particular offset,

`Canvas.drawImageRect`, which copies pixels from a source rectangle to a destination rectangle, `Canvas.drawAtlas`, which does the same for a variety of rectangles using a "sprite atlas," and `Canvas.drawImageNine`, which slices an image into a non-uniform 3x3 grid, scaling the cardinal and center boxes to fill a destination rectangle (the corners are copied directly). Each of these methods accept a `Paint` instance to be used when compositing the image (e.g., allowing a `BlendMode` to be specified); each also calls directly into the engine to perform any actual painting.

- `paintImage` wraps the canvas API to provide an imperative API for painting images in a variety of styles. It adds support for applying a box fit (e.g., `BoxFit.cover` to ensure the image covers the destination) and repeated painting (e.g., `ImageRepeat.repeat` to tile an image to cover the destination), managing layers as necessary.

How are images integrated with the render tree?

- Image encapsulates a variety of widgets, providing a high level interface to the image rendering machinery. This widget configures an `ImageProvider` (selected based on the named constructor, e.g., `Image.network`, `Image.asset`, `Image.memory`) which it resolves to obtain an `ImageStream`. Whenever this stream emits an `ImageInfo` instance, the widget is rebuilt and repainted. Conversely, if the widget is reconfigured, the `ImageProvider` is re-resolved, and the process repeated. From this flow, `Image` extracts the necessary data to fully configure a `RawImage` widget, which manages the actual `RenderImage`
  - If a cache width or cache height are provided, the underlying `ImageProvider` is wrapped in a `ResizeImage` (via `Image._resizeIfNeeded`). This ensures that the image is decoded and cached using the provided dimensions, potentially limiting the amount of memory used.
  - Image adds support for image chrome (e.g., a loading indicator) and semantic annotations.
  - If animations are disabled by `TickerMode`, `Image` pauses rendering of any new animation frames provided by the `ImageStream` for consistency.
  - The `ImageConfiguration` passed to `ImageProvider` is retrieved from the widget environment via `createLocalImageConfiguration`.

- `RawImage` is a `LeafRenderObjectWidget` wrapping a `RenderImage` and all necessary configuration data (e.g., the `ui.Image`, scale, dimensions, blend mode).
- `RenderImage` is a `RenderBox` leaf node that paints a single image; as such, it relies on the widget system to repaint whenever the associated `ImageStream` emits a new frame. Painting is performed by `paintImage` using a destination rectangle sized by layout and positioned at the current offset. Alignment, box fit, and repetition determines how the image fills the available space.
    - There are two types of dimensions considered during layout: the image's intrinsic dimensions (e.g., the number of bytes comprising the image divided by scale) and the requested dimensions (e.g., the value of width and height specified by the caller).
    - During layout, the incoming constraints are applied to the requested dimensions (via `RenderImage._sizeForConstraints`): first, the requested dimensions are clamped to the constraints. Next, the result is adjusted to match the image's intrinsic aspect ratio while remaining as large as possible. If there is no image associated with the render object, the smallest possible size is selected.
    - The intrinsic dimension methods apply the same logic. However, instead of using the incoming constraints, one dimension is fixed (i.e., corresponding to method's parameter) whereas the other is left unconstrained.


Text

Text Rendering

What are the building blocks of a font?

- Each glyph (i.e., character) is laid out on a baseline, with the portion above forming its ascent, and the portion below forming its descent. The glyph's origin precedes the glyph and is anchored to the baseline. A cap line marks the upper boundary of capital letters; the mean line, or median, serves the same purpose for lowercase letters. Cap height and x-height are measured from the baseline to each of these lines, respectively. Ascenders extend above the cap line or the median, depending on capitalization. Descenders extend below the baseline. Tracking denotes letter spacing throughout a unit of text; kerning is similar, but only measured between adjacent glyphs. Height is given as the sum of ascent and descent. Leading denotes additional spacing split evenly above and below the line. Collectively, height and leading comprise the line height.
- Font metrics are measured in logical units with respect to a box called the "em-square." `Internally`, the em-square has fixed dimensions (e.g., 1,000 units per side). Externally, the em-square is scaled to the font size (e.g., 12 pixels per side). This establishes a correspondence between internal/logical units and external/physical units (e.g., `0.012` pixels per unit).
- Line height is determined by leading and the font's height (ascent plus descent). If an explicit font height is provided, ascent and descent are constrained such that (1) their sum equals the desired height and (2) their ratio matches the original ratio. This alters the font's spacing but not the size of its glyphs.
    - Note that explicitly setting the height to the font size is not the same as the default behavior.
        - When height isn't fixed, the internal ascent and descent are used directly (after scaling). These values are font-specific and need not sum to the font size.
        - When height is fixed, ascent and descent have no relation to their internal counterparts (other than sharing a ratio). These values are chosen to sum to the target height (e.g., the font size).
- Text size is further adjusted according to a text scale factor (an accessibility feature). This factor represents the number of logical pixels per font size pixel (e.g., a value of `1.5` would cause fonts to appear 50% larger). This effect is achieved my multiply the font size by the scale factor.

- A given paragraph of text may contain multiple runs. Each run (or text box) is associated with a specific font and may be broken up due to wrapping. Adjacent runs can be aligned in different ways, but are generally positioned to achieve a common baseline. Boxes with different line height can combine to create a larger line height depending on positioning. Overall height is generally measured from the top of the highest box to the bottom of the lowest.

What are the building blocks for describing text?

- `FontStyle` and `FontWeight` characterize the glyphs used during rendering (specifying slant and glyph thickness, respectively).
- `FontFeature` encodes a feature tag, a four-character tag associated with an integer value that customizes font-specific features (i.e. these can be anything from enabling slashed zeros to selecting random glyph variants).
- `TextAlign` describes the horizontal alignment of text. “Left”, “right”, and “center” describe the text’s alignment with respect to its container. “Start” and “end” do the same, but in a directionality-aware manner. “Justify” stretches wrapped text such that it fills the available width.
- `TextBaseline` identifies the horizontal lines used to vertically align glyphs (alphabetic or ideographic).
- `TextDecoration` is a linear decoration (underline, overline, or a strikethrough) that can be applied to text; overlapping decorations merge intelligently. `TextDecorationStyle` alters how decorations are rendered: using a solid, double, dotted, dashed, or wavy line.
- `TextStyle` describes the size, position, and rendering of text in a way that can be transformed for use by the engine. This description includes colors, spacing, the desired font family, the text decoration, and so on.
    - Specifying a fixed height generally alters the font’s default spacing. Ordinarily, the font’s ascent and descent (space above and below the baseline) is calculated without constraint. When a height is specified, both metrics must sum to the desired height while maintaining their original ratio; this is simply different than the default behavior.
- `TextPosition` represents an insertion point in rendered and unrendered text (i.e., a position between letters). This is encoded as an offset indicating the index of the letter immediately after the position, even if that index exceeds string bounds. An affinity is

used to disambiguate the following cases where an offset becomes ambiguous after rendereding:
    - Positions adjacent to automatic line breaks are ambiguous: the insertion point might be the end of the first line or the start of the second (with explicit line breaks, the newline is just another character, so there is no ambiguity).
    - Positions at the interface of right-to-left and left-to-right strings are ambiguous: the renderer will flip half the string, so it's unclear whether the offset corresponds to the pre-render or post-render version (`e.x` ., offset 3 can be "abc| `ABC` " or " `abcCBA` |").
- `TextAffinity` resolves ambiguity when an offset can correspond to multiple positions after rendering (e.g., because a renderer might insert line breaks or flip portions due to directionality). `TextAffinity.upstream` selects the option closer to the start of the string (e.g., the end of the line before a break, "abc| `ABC` ") whereas `TextAffinity.downstream` selects the option closer to the end (e.g., the start of the line after a break, " `abcCBA` |").
- `TextBox` identifies a rectangular region containing text relative to the parent's top left corner. Provides direction-aware accessors (i.e., `TextBox.start` , `TextBox.end`).
- `TextWidthBasis` enumerates approaches for measuring the width of a paragraph. Longest line selects the minimum space needed to contain the longest line (e.g., a chat bubble). Parent selects the width of the container for multi-line text or the actual width for a single line of text (e.g., a paragraph of text).
- `RenderComparison` describes the renderable difference between two inline spans. Spans may have differences that will affect their layout (and painting), their painting, their metadata, etc.

What are the building blocks for describing paragraph layout?

- `ParagraphStyle` describes how lines are laid out by `ParagraphBuilder` . Among other things, this class allows a maximum number of lines to be set as well as the text's directionality and ellipses behavior; crucially, it allows the paragraph's strut to be configured.
- `ParagraphConstraints` describe the input to `Paragraph` layout. Its only value is "width," which specifies a maximum width for the paragraph. This maximum is enforced in two ways: (1) if possible, a soft line break (i.e., located between words) is

inserted before the maximum is reached. Otherwise, (2) a hard line break (i.e., located within words) is inserted, instead.
    - If this would result in an empty line (i.e., due to inadequate width), the next glyph is inserted irrespective of the constraint, followed by a hard line break.
    - This width is used when aligning text (via `TextAlign`); any ellipses is ignored for the purposes of alignment. Ellipses length is considered when determining line breaks [?].
- `StrutStyle` defines a "strut" which dictates a line of text's minimum height. Glyphs assume the larger of their own dimensions and the strut's dimensions (with ascent and descent considered separately). Conceptually, paragraph's prepend a zero-width strut character to each line. The strut can be forced, causing line height to be determined solely by the strut.

What are the building blocks for placeholders in content?

- Placeholders reserve rectangular spaces within paragraph layout. These spaces are subsequently painted using arbitrary content (e.g., a widget).
- `PlaceholderAlignment` expresses the vertical alignment of a placeholder relative to the font. Placeholders can have their top, bottom, or baseline aligned to the parent baseline. Placeholders may also be positioned relative to the font's ascenders, descenders, or median.
- `PlaceholderDimensions` describe the size and alignment of a placeholder. If a baseline-relative alignment is used, the type of baseline must be specified (e.g., alphabetic or ideographic). An optional baseline offset indicates the distance from the top of the box to its logical baseline (e.g., this is used to align inline widgets via `RenderBox.getDistanceToBaseline`).

What are the building blocks for inline spans of content?

- `InlineSpan` represents an immutable span of content within a paragraph. A span is associated with a text style which is inherited by child spans. Spans are added to a `ParagraphBuilder` via `InlineSpan.build` (this is handled automatically by

`TextPainter`); any accessibility text scaling is specified at this point. An ancestor span may be queried to locate the descendent span containing a `TextPosition`.

- `TextSpan` extends `InlineSpan` to represent an immutable span of styled text. The provided `TextStyle` is inherited by all children, which may override all or some styles. Text spans contain text as well as any number of inline span children. Children form a text span tree that is traversed in order. Each node is also associated with a string of text (styled using the span's `TextStyle`) which effectively precedes any children. `TextSpans` are interactive and have an associated gesture recognizer that is managed externally (e.g., by `RenderParagraph`).
- `PlaceholderSpan` extends `InlineSpan` to represent a reserved region within a paragraph.
- `WidgetSpan` extends `PlaceholderSpan` to embed a `Widget` within a paragraph. Widgets are constrained to the maximum width of the paragraph. The widget is laid out and painted by `RenderParagraph`; `TextPainter` simply leaves space within the paragraph.
    - `ParagraphBuilder` tracks the number of placeholders added so far; this number is used when building to identify the `PlaceholderDimensions` corresponding to this span. These dimensions are typically computed by `RenderParagraph`, which lays out all widget spans before building the paragraph so that the necessary amount of space is reserved.

What are the building blocks for paragraphs?

- `Paragraph` is a piece of text wherein each glyph is sized and positioned appropriately; `Paragraph.layout` must be invoked to compute these metrics. Paragraph supports efficient resizing and painting (via `Canvas.addParagraph`) and must be built by `ParagraphBuilder`. Each glyph is assigned an integer offset computed before rendering (thus there is no affinity issue). Note that `Paragraph` is a thin wrapper around engine code.
    - After layout, paragraphs can report height, width, longest line width, and intrinsic width. Maximum intrinsic width maximally reduces the paragraph's height. Minimum intrinsic width is the smallest width allowing correct rendering.

    - Paragraphs can report all placeholder locations and dimensions (reported as `TextBox` instances), the `TextPosition` associated with a 2D offset, and word boundaries given an integer offset.
    - Once laid out, `Paragraphs` can provide bounding boxes for a range within the pre-rendered text. Boxes are derived from runs of text, each of which may have a distinct style and therefore height. Thus, boxes can be sized in a variety of ways (via `BoxHeightStyle`, `BoxWidthStyle`). Boxes can tightly enclose only the rendered glyphs (the default), expand to include different portions of the line height, be sized to the maximum height in a given line, etc.
- `ParagraphBuilder` assembles a single `Paragraph` from a sequence of text and placeholders using the provided `ParagraphStyle`. `TextStyles` are pushed and popped, allowing styles to be inherited and overridden, and placeholders boxes (e.g., for embedded widgets) are reserved and tracked. The `Paragraph` itself is built by the engine.

What are the text rendering building blocks?

- `TextOverflow` describes how visual overflow is handled when rendering text via `RenderParagraph`. Options include clipping, fading, adding ellipses, or tolerating overflow.
- `TextPainter` performs the actual work of building a `Paragraph` from an `InlineSpan` tree and painting it to the canvas; the caller must explicitly request layout and painting. The painter incorporates the various text rendering `APIs` to provide a convenient interface for rendering text. If the container's width changes, layout and painting must be repeated.
    - Text layout selects a width that is within the provided minimum and maximum values, but that is as close to the maximum intrinsic width as possible (i.e., consumes the least height). Redundant calls are ignored.
    - `TextPainter` supports a variety of queries; it can provide bounding boxes for a `TextSelection`, the height and offset of the glyph at a given `TextPosition`, neighboring editable offsets, and can convert a 2D offset into a `TextPosition`.
    - A preferred line height can be computed by rendering a test glyph and measuring the resulting height (via `TextPainter.preferredLineHeight`).

- Text is a wrapper widget for `RichText` which obtains the ambient text style via `DefaultTextStyle`.
- `RichText` is a `MultiChildRenderObjectWidget` that configures a single `RenderParagraph` with a provided `InlineSpan`. Its children are obtained by traversing this span to locate `WidgetSpan` instances. Any render objects produced by the associated widgets will be attached to the `RichText`'s `RenderParagraph`; this is how the `RenderParagraph` is able to layout and paint widgets into the paragraphs' placeholders.
- `TextParentData` is the parent data associated with the inline widgets contained in a paragraph of text. It extends `ContainerBoxParentData` to track the text scale applied to the child.
- `RenderParagraph` displays a paragraph of text, optionally containing inline widgets. It delegates to (and augments) an underlying `TextPainter` which builds, lays out, and paints the actual paragraph. Any operations that depend on text layout generally recompute layout blindly; this is acceptable since `TextPainter` ignores redundant calls.

How is text laid out by the render tree?

- `RenderParagraph` adapts a `TextPainter` to the render tree, adding the ability to render widgets (i.e., `WidgetSpan` instances) within the text. Any such widgets are parented to the `RenderParagraph`; a list of all descendent placeholders (`RenderParagraph._placeholderSpans`) is also cached. The `RenderParagraph` proxies the various inputs to the text rendering system, invalidating painting and layout as values change (`RenderComparison` allows this to be done efficiently).
- Layout is performed in three stages: inline children are laid out to determine placeholder dimensions, text is laid out with placeholder dimensions defined, and children are positioned using the final placeholder positions computed by the engine. Finally, the desired `TextOverflow` effect is applied (by clipping, configuring an overflow shader, etc).
- Children (i.e., inline widgets) are laid out in sequence with infinite height and a maximum width matching that of the paragraph. A list of `PlaceholderDimensions` is assembled using the resulting layout. If the associated `InlineSpan` is aligned to the

baseline, the offset to the child's baseline is computed (via `RenderBox.getDistanceToBaseline`); this ensures that the child's baseline is coincident with the text's baseline. Finally, the dimensions are bound to the `TextPainter` (via `TextPainter.setPlaceholderDimensions`).
- Next, the box constraints are converted to `ParagraphConstraints` (i.e., height is ignored). If wrapping or truncation is enabled, the maximum width is retained; otherwise, the width is considered infinite. These constraints are provided to `TextPainter.layout`, which builds and lays out the underlying `Paragraph` (via the engine).
- Finally, children are positioned based on the final text layout. Positions are read from `TextPainter.inlinePlaceholderBoxes`, which exposes an ordered list of `TextBox` instances.

How is text painted by the render tree?

- Since `TextPainter` is stateful, and certain operations (e.g., computing intrinsic dimensions) destroy this state, the text must be relaid out before painting.
- Any applicable clip is applied to the canvas; if the `TextOverflow` setting requires a shader, the canvas is configured accordingly.
- The `TextPainter` paints the text (via `Canvas.drawParagraph`). Empty spaces will appear in the text wherever placeholders were included.
- Finally, each child is rendered at the offset corresponding to the associated placeholder in the text. If applicable, the text scale factor is applied to the child during painting (e.g., causing it to appear larger).

How is text made interactive?

- All `InlineSpan` subclasses can be associated with a gesture recognizer (`InlineSpan.recognizer`). This instance's lifecycle, however, must be maintained by client code (i.e., the `RenderParagraph`). Additionally, events must be propagated to each `InlineSpan` since spans do not support bubbling themselves.

    - The `RenderParagraph` overrides `RenderObject.handleEvent` to attach new pointers (i.e., `PointerDownEvent`) directly to the span that was tapped.
    - The affected span is identified by first laying out the text, mapping the event's offset to a `TextPosition`, then finally locating the span that contains this position (via `TextPainter.getPositionForOffset` and `InlineSpan.getSpanForPosition`, respectively).
- The `RenderParagraph` is also responsible for hit testing any render objects included via inline widgets. These are hit tested using a similar approach as `RenderBox` that additionally takes into account any text scaling.
- The `RenderParagraph` is always included in the hit test result.

How are text's intrinsic dimensions calculated?

- Intrinsic dimensions cannot be calculated if placeholders are to be aligned relative to the text's baseline; this would require a full layout pass according to the `RenderBox` layout protocol.
- Intrinsics are dependant on two things: inline widgets (children) and text layout. Children must be measured to establish the size of any placeholders in the text. Next, the text is laid out using the resulting placeholder dimensions so its that intrinsic dimensions can be retrieved (`Paragraph.layout` is required to obtain the intrinsic dimensions from the engine).
- The minimum and maximum intrinsic width of text is computed without constraint on width. The minimum value corresponds to the width of the first word and the maximum value corresponds to the full string laid out on a single line.
- The minimum and maximum intrinsic height of text is computed using the provided width (text layout is width-in-height-out). Moreover, the maximum and minimum values are equivalent: the text's height is solely determined by its width. Thus, both values match the height of the text when rendered within the given width.
- Intrinsic dimensions for all children are queried via the `RenderBox` protocol. The input dimension (e.g., height) is provided to obtain the opposite intrinsic dimension (e.g., minimum intrinsic width); this value is used to obtain the remaining intrinsic dimension (e.g., minimum intrinsic height). These values are then wrapped in a `PlaceholderDimension` instance which is aligned appropriately (via `RenderParagraph._placeholderSpans`).

    - Variants obtaining minimum and maximum intrinsic dimensions are equivalent other than the render box methods they invoke.
    - Both the intrinsic width and height children must be computed since the placeholder boxes affect both dimensions of text layout. However, when measuring maximum intrinsic width, height can be ignored since text is laid out in a single line.
- The intrinsic sizes of all children are provided to the text painter prior to layout (via `TextPainter.setPlaceHolderDimensions`). This ensures that the resulting intrinsics accurately reflect any inline widgets. After layout, intrinsics can be read from the `Paragraph` directly. Note that this is destructive in that it wipes out earlier dimensions associated with the `TextPainter`.

How is text directionality handled by the framework?

- Unlike other frameworks, `Flutter` does not have a default text direction (`TextDirection`). Throughout the lower levels of the framework, directionality must be specified. At the widget level, an ambient directionality is introduced by the `Directionality` widget. Other widgets use the ambient directionality when interacting with lower level aspects of the framework.
- Often, a visual and a directional variant of a widget is provided (`EdgeInsets` vs. `EdgeInsetsDirectional`). The former exposes top, left, right, and bottom properties; the latter exposes top, start, end, and bottom properties.
- Painting is a notable exception; the canvas (Canvas) is always visually oriented, with the X and Y axes running from left-to-right and top-to-bottom, respectively.

How does the engine layout text?

- Minikin measures and lays out the text.
    - Minikin uses `ICU` to split text into lines.
    - Minikin uses `HarfBuzz` to retrieve glyphs from a font.
- Skia paints the text and text decorations on a canvas.

Text Input

How are key events sent from the keyboard?

- `SystemChannels.keyEvent` exposes a messaging channel that receives raw key data whenever the platform produces keyboard events.
- `RawKeyboard` subscribes to this channel and forwards incoming messages as `RawKeyEvent` instances (which encapsulate `RawKeyEventData`). Physical and logical interpretations of the event are exposed via `RawKeyEvent.physicalKey` and `RawKeyEvent.logicalKey`, respectively. The character produced is available as `RawKeyEvent.character` but only for `RawKeyDownEvent` events. This field accounts for modifier keys / past keystrokes producing null for invalid combinations or a dart string, otherwise.
- The physical key identifies the actual position of the key that was struck, expressed as the equivalent key on a standard `QWERTY` keyboard. The logical key ignores position, taking into account any mappings or layout changes to produce the actual key the user intended.
- Subclasses of `RawKeyEventData` interpret platform-specific data to categorize the keystroke in a portable way (`RawKeyEventDataAndroid`, `RawKeyEventDataMacOs`)

What is an `IME`?

- `IME` stands for “input method editor,” which corresponds to any sort of on-screen text editing interface, such as the software keyboard. There can only be one active `IME` at a time.

How does `Flutter` interact with `IMEs`?

- `SystemChannels.textInput` exposes a method channel that implements a transactional interface for interacting with an `IME`. Operations are scoped to a given

transaction (client), which is implicit once created. Outbound methods support configuring the `IME`, showing/hiding UI, and update editing state (including selections); inbound methods handle `IME` actions and editing changes. Convenient wrappers for this protocol make much of this seamless.

What are the building blocks for interacting with an `IME`?

- `TextInput.attach` federates access to the `IME`, setting the current client (transaction) that can interact with the keyboard.
- `TextInputClient` is an interface to receive information from the `IME`. Once attached, clients are notified via method invocation when actions are invoked, the editing value is updated, or the cursor is moved.
- `TextInputConnection` is returned by `TextInput.attach` and allows the `IME` to be altered. In particular, the editing state can be changed, the `IME` shown, and the connection closed. Once closed, if no other client attaches within the current animation frame, the `IME` will also be hidden.
- `TextInputConfiguration` encapsulates configuration data sent to the `IME` when a client attaches. This includes the desired input type (e.g., “datetime”, “`emailAddress`”, “phone”) for which to optimize the `IME`, whether to enable autocorrect, whether to obscure input, the default action, capitalization mode (`TextCapitalization`), and more.
- `TextInputAction` enumerates the set of special actions supported on all platforms (e.g., “`emergencyCall`”, “done”, “next”). Actions may only be used on platforms that support them. Actions have no intrinsic meaning; developers determine how to respond to actions themselves.
- `TextEditingValue` represents the current text, selection, and composing state (range being edited) for a run of text.
- `RawFloatingCursorPoint` represents the position of the “floating cursor” on `iOS`, a special cursor that appears when the user force presses the keyboard. Its position is reported via the client, including state changes (`RawFloatingCursorDragState`).

Text Editing

What data structures support editable text?

- `TextRange` represents a range using [start, end) character indices: start is the index of the first character and end is the index after the last character. If both are -1, the range is collapsed (empty) and outside of the text (invalid). If both are equal, the range is collapsed but potentially within the text (i.e., an insertion point). If start <= end, the range is said to be normal.
- `TextSelection` expands on range to represent a selection of text. A range is specified as a [`baseOffset` , `extentOffset`) using character indices. The lesser will always become the base offset, with the greater becoming the extent offset (i.e., the range is normalized). If both offsets are the same, the selection is collapsed, representing an insertion point; selections have a concept of directionality (`TextSelection.isDirectional`) which may be left ambiguous until the selection is uncollapsed. Both offsets are resolved to positions using a provided affinity. This ensures that the selection is unambiguous before and after rendering (e.g., due to automatic line breaks).
- `TextSelectionPoint` pairs an offset with the text direction at that offset; this is helpful for determining how to render a selection handle at a given position.
- `TextEditingValue` captures the current editing state. It exposes the full text in the editor (`TextEditingValue.text`), a range in that text that is still being composed (`TextEditingValue.composing`), and any selection present in the UI (`TextEditingValue.selection`). Note that an affinity isn't necessary for the composing range since it indexes unrendered text.

How are editing and selection overlays built?

- `TextSelectionDelegate` supports reading and writing the selection (via `TextSelectionDelegate.textEditingValue`), configures any associated selection actions (e.g., `TextSelectionDelegate.canCopy`), and provides helpers to manage selection UI (e.g., `TextSelectionDelegate.bringIntoView` ,

`TextSelectionDelegate.hideToolbar`). This delegate is utilized primarily by `TextSelectionControls` to implement the toolbar and selection handles.

- `ToolbarOptions` is a helper bundling all options that determine toolbar behavior within an `EditableText` – that is, how the overridden `TextSelectionDelegate` methods behave.
- `TextSelectionControls` is an abstract class that builds and manages selection-related UI including the toolbar and selection handles. This class also implements toolbar behaviors (e.g., `TextSelectionControls.handleCopy`) and eligibility checks (e.g., `TextSelectionControls.canCopy`), deferring to the delegate where appropriate (e.g., `TextSelectionDelegate.bringIntoView` to scroll the selection into view). These checks are mainly used by `TextSelectionControls`' build methods (e.g., `TextSelectionControls.buildHandle`, `TextSelectionControls.buildToolbar`), which construct the actual UI. Concrete implementations are provided for `Cupertino` and `Material` (`_CupertinoTextSelectionControls` and `_MaterialTextSelectionControls`, respectively), producing idiomatic UI for the corresponding platform. The build process is initiated by `TextSelectionOverlay`.
- `TextSelectionOverlay` is the visual engine underpinning selection UI. It integrates `TextSelectionControls` and `TextSelectionDelegate` to build and configure the text selection handles and toolbar, and `TextEditingValue` to track the current editing state; the editing state may be updated at any point (via `TextSelectionOverlay.update`). Updates are made build-safe by scheduling a post-frame callback if in the midst of a persistent frame callback (building, layout, etc; this avoids infinite recursion in the build method).
  - The UI is inserted into the enclosing `Overlay` and hidden and shown as needed (via `TextSelectionOverlay.hide`, `TextSelectionOverlay.showToolbar`, etc).
  - The toolbar and selection handles are positioned using leader/follower layers (via `CompositedTransformLeader` and `CompositedTransformFollower`). A `LayerLink` instance for each type of UI is anchored to a region within the editable text so that the two layers are identically transformed (e.g., to efficiently scroll together). When this happens, `TextSelectionOverlay.updateForScroll` marks the overlay as needing to be rebuilt so that the UI can adjust to its new position.
  - The toolbar is built directly (via `TextSelectionControls.buildToolbar`), whereas each selection handle corresponds to a `_TextSelectionHandleOverlay` widget. These widgets invoke a handler when the selection range changes to update the

`TextEditingValue` (via `TextSelectionOverlay._handleSelectionHandleChanged`).

- `TextSelectionGestureDetector` is a stateful widget that recognizes a sequence of selection-related gestures (e.g., a tap followed by a double tap), unlike a typical detector which recognizes just one. The text field (e.g., `TextField`) incorporates the gesture detector when building the corresponding UI.
  - `_TextSelectionGestureDetectorState` coordinates the text editing gesture detectors, multiplexing them as described above. A map of recognizer factories is assembled and assigned callbacks (via `GestureRecognizerFactoryWithHandlers`) given the widget's configuration. These are passed to a `RawGestureDetector` widget which constructs the recognizers as needed.
  - `_TransparentTapGestureRecognizer` is a `TapGestureRecognizer` capable of recognizing while ceding to other recognizers in the arena. Thus, the same tap may be handled by multiple recognizers. This is particularly useful since selection handles tend to overlap editable text; a single tap in the overlap region is generally processed by the selection handle, whereas a double tap is processed by the editable text.
  - `TextSelectionGestureDetectorBuilderDelegate` provides a hook for customizing the interaction model (typically implemented by the text field, e.g., `_CupertinoTextFieldState`, `_TextFieldState`). The delegate also exposes the `GlobalKey` associated with the underlying `EditableTextState`.
  - `TextSelectionGestureDetectorBuilder` configures a `TextSelectionGestureDetector` with sensible defaults for text editing. The delegate is used to obtain a reference to the editable text and to customize portions of the interaction model.
  - Platform-specific text fields extend `TextSelectionGestureDetectorBuilder` to provide idiomatic interaction models (e.g., `_TextFieldSelectionGestureDetectorBuilder`).

How can editable behavior be customized?

- `TextInputFormatter` provides a hook to transform text just before `EditableText.onChange` is invoked (i.e., when a change is committed – not as the user types). Blocklisting, allowlisting, and length-limiting formatters are available (

`BlacklistingTextInputFormatter`, `WhitelistingTextInputFormatter`, and `LengthLimitingTextInputFormatter`, respectively).

- `TextEditingController` provides a bidirectional interface for interacting with an `EditableText` or subclass thereof; as a `ValueNotifier`, the controller will notify whenever state changes, including as the user types. The text (`TextEditingController.text`), selection (`TextEditingController.selection`), and underlying `TextEditingValue` (`TextEditingController.value`) can be read and written, even in response to notifications. The controller may also be used to produce a `TextSpan`, an immutable span of styled text that can be painted to a layer.

How is editable text implemented?

- `EditableText` is the fundamental text input widget, integrating the other editable building blocks (e.g., `TextSelectionControls`, `TextSelectionOverlay`, etc.) with keyboard interaction (via `TextInput`), scrolling (via `Scrollable`), and text rendering to implement a basic input field. `EditableText` also supports basic gestures (tapping, long pressing, force pressing) for cursor and selection management and `IME` interaction. A variety of properties allow editing behavior and text appearance to be customized, though the actual work is performed by `EditableTextState`. When `EditableText` receives focus but is not fully visible, it will be scrolled into view (via `RenderObject.showOnScreen`).
  - The resulting text is styled and structured (via `TextStyle` and `StrutStyle`), aligned (via `TextAlign`), and localized (via `TextDirection` and `Locale`). `EditableText` also supports a text scale factor.
  - `EditableText` layout behavior is dependant on the maximum and minimum number of lines (`EditableText.maxLines`, `EditableText.minLines`) and whether expansion is enabled (`EditableText.expands`).
    - If maximum lines is one (the default), the field will scroll horizontally on one line.
    - If maximum lines is null, the field will be laid out for the minimum number of lines, and grow vertically.
    - If maximum lines is greater than one, the field will be laid out for the minimum number of lines, and grow vertically until the maximum number of lines is

reached.
        - If a multiline field reaches its maximum height, it will scroll vertically.
        - If a field is expanding, it is sized to the incoming constraints.
    - `EditableText` follows a simple editing flow to allow the application to react to text changes and handle keyboard actions (via `EditableTextState._finalizeEditing`).
        - `EditableText.onChanged` is invoked as the field's contents are changed (i.e., as characters are explicitly typed).
        - `EditableText.onEditingComplete` (by default) submits changes, clearing the controller's composing bit, and relinquishes focus. If a non-completion action was selected (e.g., "next"), focus is retained to allow the submit handler to manage focus itself. A custom handler can be provided to alter the latter behavior.
        - `EditableText.onSubmitted` is invoked last, when the user has indicated that editing is complete (e.g., by hitting "done").
- `EditableTextState` applies the configuration described by `EditableText` to implement a text field; it also manages the flow of information with the platform `TextInput` service. Additionally, the state object exposes a simplified, top-level interface for interacting with editable text. The editing value can be updated (via `EditableTextState.updateEditingValue`), the toolbar toggled (via `EditableTextState.toggleToolbar`), the `IME` displayed (via `EditableTextState.requestKeyboard`), editing actions performed (via `EditableTextState.performAction`), text scrolled into view (via `EditableText.bringIntoView`) and prepared for rendering (via `EditableText.buildTextSpan`). In this respect, `EditableTextState` is the glue binding many of the editing components together.
    - Is a: `TextInputClient`, `TextSelectionDelegate`
        - Breakdown how it is updated by the client / notifies the client of changes to keep things in sync
        - `updateEditingValue` is invoked by the client when user types on keyboard (same for `performAction` / `floatingCursor`).
    - `EditableTextState` participates in the keep alive protocol (via `AutomaticKeepAliveClientMixin`) to ensure that it isn't prematurely destroyed, losing editing state (e.g., when scrolled out of view).

        - When text is specified programmatically (via `EditableTextState.textEditingValue`, `EditableTextState.updateEditingValue`), the underlying `TextInput` service must be notified so that platform state remains in sync (applying any text formatters beforehand). `EditableTextState._didChangeTextEditingValue`
- `RenderEditable`

How are platform-specific text fields implemented?

- `TextField`
- `TextFieldState`
- `CupertinoTextField`

How is the toolbar rendered?

- The toolbar UI is built by `TextSelectionControls.buildToolbar` using the line height, a bounding rectangle for the input (in global, logical coordinates), an anchor position and, if necessary, a tuple of `TextSelectionPoints`.
- `EditableText` triggers on gesture, overlay does the work

How are selection handles rendered?

- Selection handles are visual handles rendered just before and just after a selection. Handles need not be symmetric; `TextSelectionHandleType` characterizes which variation of the handle is to be rendered (left, right, or collapsed).
- Each handle is built by `TextSelectionControls.buildHandle` which requires a type and a line height.
- The handle's size is computed by `TextSelectionControls.getHandleSize`, typically using the associated render editable's line height (

`RenderEditable.preferredLineHeight`), which is derived from the text painter's line height (`TextPainter.preferredLineHeight`). The painter calculates this height through direct measurement.

- The handle's anchor point is computed by `TextSelectionControls.getHandleAnchor` using the type of handle being rendered and the associated render editable's line height, as above.
- `EditableText` triggers on select / cursor position, overlay does the work
- `EditableText` uses the handle's size and anchor to ensure that selection handles are fully visible on screen (via `RenderObject.showOnScreen`).

How does the editable retain state in response to platform lifecycle events?

What is the best way to manage input via forms?

`IME` (input method editor)?


User Interface

Containers

What are the container building blocks?

- `Flex` is the base class for `Row` and `Column`. It implements the flex layout protocol in an axis-agnostic manner.
- `Row` is identical to `Flex` with a default axis of `Axis.horizontal`.
- `Column` is identical to `Flex` with a default axis of `Axis.vertical`.
- `Flexible` is the base class for `Expanded`. It is a parent data widget that alters its child's flex value. Its default fit is `FlexFit.loose`, which causes its child to be laid out with loose constraints
- `Expanded` is identical to `Flexible` with a default fit of `FlexFit.tight`. Consequently, it passes tight constraints to its children, requiring them to fill all available space.

How are flex-based containers laid out?

- All flexible containers follow the same protocol.
  - Layout children without flex factors with unbounded main constraints and the incoming cross constraints (if stretching, cross constraints are tight).
  - Apportion remaining space among flex children using flex factors.
    - Main axis size = `myFlex` * (`freeSpace` / `totalFlex`)
  - Layout each child as above, with the resulting size as the main axis constraint. Use tight constraints for `FlexFit.tight`; else, use loose.
  - The cross extent is the max of all child cross extents.
  - If using `MainAxisSize.max`, the main extent is the incoming max constraint. Else, the main extent is the sum of all child extents in that dimension (subject to constraints).
  - Children are positioned according to `MainAxisAlignment` and `CrossAxisAlignment`.

How are containers laid out?

- In short, containers size to their child plus any padding; in so doing, they respect any additional constraints provided directly or via a width or height. Decorations may be painted over this entire region. Next, a margin is added around the resulting box and, if specified, a transformation applied to the entire container.
    - If there is no child and no explicit size, the container shrinks in unbounded environments and expands in bounded ones.
- The container widget delegates to a number of sub-widgets based on its configuration. Each behavior is layered atop all previous layers (thus, child refers to the accumulation of widgets). If a width or height is provided, these are transformed into extra constraints.
    - If there is no child and no explicit size:
        - Shrink when the incoming constraints are unbounded (via `LimitedBox`); else, expand (via `ConstrainedBox`).
    - If there is an alignment:
        - Align the child within the parent (via `Align`).
    - If there is padding or the decoration has padding...
        - Apply the total padding to the child (via `Padding`).
    - If there is a decoration:
        - Wrap the child in the decoration (via `DecoratedBox`).
    - If there is a foreground decoration:
        - Wrap the child in the foreground decoration (via `DecoratedBox`, using `DecorationPosition.foreground`).
    - If there are extra constraints:
        - Apply the extra constraints to the incoming constraints (via `ConstrainedBox`).
    - If there is a margin...
        - Apply the margin to the child (via `Padding`).
    - If there is a transform...
        - Transform the child accordingly (via `Transform`).

Decoration

What are decorations?

- A decoration is a high level description of graphics to be painted onto a canvas, generally corresponding to a box but may also describe other shapes, too. In addition to being paintable, decorations support interaction and interpolation.

What are the common components of a decoration?

- `DecorationImage` describes an image (as obtained via `ImageProvider`) to be inscribed within a decoration, accepting many of the same arguments as `paintImage`. The alignment, repetition, and box fit determine how the image is laid out within the decoration and, if enabled, horizontal reflection will be applied for right-to-left locales.
    - `DecorationImagePainter` (obtained via `DecorationImage.createPainter`) performs the actual painting; this is a thin wrapper around `paintImage` that resolves the `ImageProvider` and applies any clipping and horizontal reflection.
- `BoxShadow` is a `Shadow` subclass that additionally describes spread distance (i.e., the amount of dilation to apply to the casting element's mask before computing the shadow). Shadows are typically arranged into a list to support a single decoration casting multiple shadows.
- `BorderRadiusGeometry` describes the border radii of a particular box (via `BorderRadius` or `BorderRadiusDirectional` depending on text direction sensitivity). `BorderRadiusGeometry` is composed of four immutable `Radius` instances.
- `BorderSide` describes a single side of a border; the precise interpretation is determined by the enclosing `ShapeBorder` subclass. Each side has a color, a style (via `BorderStyle`), and a width. A width of `0.0` will enable hairline rendering; that is, the border will be 1 physical pixel wide (`BorderStyle.none` is necessary to prevent the border from rendering). When hairline rendering is utilized, pixels may appear darker if they are painted multiple times by the given path. Border sides may be merged provided that they share a common style and color. Doing so produces a new `BorderSide` having a width equal to the sum of its constituents.

What are the components of a shape decoration?

- `ShapeBorder` is the base class of all shape outlines, including those used by box decorations; in essence, it describes a single shape with edges of defined width (typically via `BorderSide`). Shape borders can be interpolated and combined (via the addition operator or `ShapeBorder.add`). Additionally, borders may be scaled (affecting properties like border width and radii) and painted directly to a canvas (via `ShapeBorder.paint`); painting may be adjusted based on text direction. Paths describing the shape's inner and outer edges may also be queried (via `ShapeBorder.getInnerPath` and `ShapeBorder.getOuterPath`).

What are the components of a box decoration?

- `BoxBorder` is a subclass of `ShapeBorder` that is further specialized by `Border` and `BorderDirectional` (the latter adding text direction sensitivity). These instances describe a set of four borders corresponding to the cardinal directions; their precise arrangement is left undefined until rendering. Borders may be combined (via `Border.merge`) provided that all associated sides share a style and color. If so, the corresponding widths are added together.
  - Borders must be made concrete by providing a rectangle and, optionally, a `BoxShape`. The provided rectangle determines how the borders are actually rendered; uniform borders are more efficient ot paint.
- `BoxShape` describes how a box decoration (or border) is to be rendered into its bounds. If rectangular, painting is coincident with the bounds. If circular, the box is painted as a uniform circle with diameter matching the smaller of the bounding dimensions.

What are the decoration building blocks?

- Decoration describes an adaptive collection of graphical effects that may be applied to an arbitrary rectangle (e.g., box). Decorations optionally specify a padding (via `Decoration.padding`) to ensure that any additional painting within a box (e.g., from a

child widget; note that decorations do not perform clipping) does not overlap with the decoration's own painting. Additionally, certain decorations can be marked as complex (via `Decoration.isComplex`) to enable caching.
    - Decorations support hit testing (via `Decoration.hitTest`). A size is provided so that the decoration may be scaled to a particular box. The given offset describes a position within this box relative to its top-left corner. An optional `TextDirection` supports containers that are sensitive to this parameter.
    - Decorations support linear interpolation (via `Decoration.lerp`, `Decoration.lerpFrom`, and `Decoration.lerpTo`). The "t" parameter represents a position on a timeline with 0 corresponding to 0% (i.e., the pre-state) and 1 corresponding to 100% (i.e., the post-state); note that values outside of this range are possible. If the source or destination value is null (indicating that a true interpolation isn't possible), a default interpolation should be computed that reasonably approximates a true interpolation.
- `BoxDecoration` is a `Decoration` subclass that describes the appearance of a graphical box. Boxes are composed of a number of elements, including a border, a drop shadow, and a background. The background is itself comprised of color, gradient, and image layers. While typically rectangular, boxes may be given rounded corners or even a circular shape (via `BoxDecoration.shape`). `BoxDecorations` provide a `BoxPainter` subclass capable of rendering the described box given different `ImageConfigurations`.
- `ShapeDecoration` is analogous to `BoxDecoration` but supports rendering into any shape (via `ShapeBorder`). Rendering occurs in layers: first a fill color is painted, then a gradient, and finally an image. Next, the `ShapeBorder` is painted (clipping the previous layers); the border also serves as the casting element for all associated shadows. `ShapeDecoration` also uses a `BoxPainter` subclass for rendering.
    - Shape decorations may be obtained from box decorations (via `ShapeDecoration.fromBoxDecoration`) since the latter is derived from the former. In general, box decorations are more efficient since they do not need to represent arbitrary shapes; however, shapes support a wider arrange of interpolation (e.g., rectangle to circle).
- `DecoratedBox` incorporates a decoration into the widget hierarchy. Decorations can be painted in the foreground or background via `DecorationPosition` (i.e., in front of or behind the child, respectively). Generally, `Container` is used to incorporate a `DecoratedBox` into the UI.

- `NotchedShape` describes the difference of two shapes (i.e., a guest shape is subtracted from a host shape). A path describing this shape is obtained by specifying two bounding rectangles (i.e., the host and the guest) sharing a coordinate space. The `AutomaticNotchedShape` subclass uses these bounds to determine the concrete dimensions of `ShapeBorder` instances before computing their difference.

How are decorations painted?

- `BoxPainter` provides a base class for instances capable of rendering a `Decoration` to a canvas given an `ImageConfiguration`. The configuration specifies the final size, scale, and locale to be used when rendering; this information allows an otherwise abstract decoration to be made concrete. Since decorations may rely on asynchronous image providers, `BoxPainter.onChanged` notifies client code when the associated resources have changed (i.e, so that painting may be repeated).

Themes

How do themes work?

How are colors managed?

- The common `Color` type is generally used throughout the framework. These may be organized into swatches with a single primary `aRGB` value and a mapping from arbitrary keys to `Color` instances. Material provides a specialization called `MaterialColor` which uses an index value as key and limits the map to ten entries (50, 100, 200, ... 900), with larger indices being associated with darker shades. These are further organized into a standard set of colors and swatches within the `Colors` class.

Tables

How is table layout described?

- `TableColumnWidth` describes the width of a single column in a `RenderTable`. Implementations can produce a flex factor for the column (via `TableColumnWidth.flex`, which may iterate over every cell) as well as a maximum and minimum intrinsic width (via `TableColumnWidth.maxIntrinsicWidth` and `TableColumnWidth.minIntrinsicWidth`, which also have access to the incoming maximum width constraint). Intrinsic dimensions are expensive to compute since they typically visit the entire subtree for each cell in the column. Subclasses implement a subset of these methods to provide different mechanisms for sizing columns.
  - `FixedColumnWidth` produces tight intrinsic dimensions, returning the provided constant without additional computation.
  - `FractionColumnWidth` applies a fraction to the incoming maximum width constraint to produce tight intrinsic dimensions. If the incoming constraint is unbounded, the resulting width will be zero.
  - `MaxColumnWidth` and `MinColumnWidth` encapsulate two `TableColumnWidth` instances, returning the greater or lesser value produced by each method, respectively.
  - `FlexColumnWidth` returns the specified flex value which corresponds to the portion of free space to be utilized by the column (i.e., free space is distributed according to the ratio of the column's flex factor to the total flex factor). The intrinsic width is set to zero so that the column does not consume any inflexible space.
  - `IntrinsicColumnWidth` is the most expensive strategy, sizing the column according to the contained cells' intrinsic widths. A flex factor allows columns to expand even further by incorporating a portion of any unclaimed space. The minimum and maximum intrinsic widths are defined as the maximum value reported by all contained render boxes (via `RenderBox.getMinIntrinsicWidth` and `RenderBox.getMaxIntrinsicWidth` with unbounded height).
- `TableCellVerticalAlignment` specifies how a cell is positioned within a row. Top and bottom ensure that the corresponding side of the cell and row are coincident, middle vertically centers the cell, baseline aligns cells such that all baselines are coincident

(cells lacking a baseline are top aligned), and fill sizes cells to the height of the cell (if all cells fill, the row will have zero height).

How is table appearance described?

- `TableBorder` describes the appearance of borders around and within a table. Similar to `Border`, `TableBorder` exposes `BorderSide` instances for each of the cardinal directions (`TableBorder.top`, `TableBorder.bottom`, etc). In addition, `TableBorder` describes the seams between rows (`TableBorder.horizontalInside`) and columns (`TableBorder.verticalInside`). Borders are painted via `TableBorder.paint` using row and column offsets determined by layout (e.g., the number of pixels from the bounding rectangle's top and left edges for horizontal and vertical borders; there will be one entry for each interior seam).
- `RenderTable` accepts a list of decorations to be applied to each row in order. These decorations span the full extent of each row, unlike any cell-based decorations (which would be limited to the dimensions of the cell; cells may be offset within the row due to `TableCellVerticalAlignment` and).

How are tables rendered?

- `TableCellParentData` extends `BoxParentData` to include the cell's vertical alignment (via `TableCellVerticalAlignment`) as well as the most recent zero-indexed row and column numbers (via `TableCellParentData.y` and `TableCellParentData.x`, respectively). The cell's coordinates are set during `RenderTable` layout whereas the vertical alignment is set by `TableCell`, a `ParentDataWidget` subclass.
- `RenderTable` is a render box implementing table layout and painting. Columns may be associated with a sizing strategy via a mapping from index to `TableColumnWidth` (`columnWidths`); a default strategy (`defaultColumnWidth`) and default vertical alignment (`defaultVerticalAlignment`) round out layout. The table is painted with a border (`TableBorder`) and each row's full extent may be decorated (`rowDecorations`, via `Decoration`). `RenderBox` children are passed as a list of rows; internally, children

are stored in row-major order using a single list. The number of columns and rows can be inferred from the child list, or specifically set. If these values are subsequently altered, children that no longer fit in the table will be dropped.

How does a table manage its children?

- Children are stored in row-major order using a single list. Tables accept a flat list of children (via `RenderTable.setFlatChildren`), using a column count to divide cells into rows. New children are adopted (via `RenderBox.adoptChild`) and missing children are dropped (via `RenderBox.dropChild`); children that are moved are neither adopted nor dropped. Children may also be added using a list of rows (via `RenderTable.setChildren`); this clears all children before adding each row incrementally (via `RenderTable.addRow`). Note that this may unnecessarily drop children, unlike `RenderTable.setFlatChildren`.
- Children are visited in row-major order. That is, the first row is iterated in order, then the second row, and so on. This is the order used when painting; hit testing uses the opposite order (i.e., starting from the last item in the last row).

How are columns widths calculated?

- A collection of `TableColumnWidth` instances describe how each column consumes space in the table. During layout, these instances are used to produce concrete widths given the incoming constraints (via `RenderTable._computeColumnWidths`).
  - Intrinsic widths and flex factors are computed for each column by locating the appropriate `TableColumnWidth` and passing the maximum width constraint as well as all contained cells.
    - The column's width is initially defined as its maximum intrinsic width (flex factor only increases this width). Later, column widths may be reduced to satisfy incoming constraints.
    - Table width is therefore computed by summing the maximum intrinsic width of all columns.

        - Flex factors are summed for all flexible columns; maximum intrinsic widths are summed for all inflexible columns. These values are used to identify and distribute free space.
    - If there are flexible columns and room for expansion given the incoming constraints, free space is divided between all such columns. That is, if the table width (i.e., total maximum intrinsic width) is less than the incoming maximum width (or, if unbounded, the minimum width), there is room for flexible columns to expand.
        - Remaining space is defined as the relevant width constraint minus the maximum intrinsic widths of all inflexible columns.
        - This space is distributed in proportion to the ratio of the column's flex factor to the sum of all flex factors.
        - If this would expand the column, the delta is computed and applied both to the column's width and the table's width.
    - If there were no flexible columns, ensure that the table is at least as wide as the minimum width constraint.
        - The difference between the table width (i.e., total maximum intrinsic width) and the minimum width is evenly distributed between all columns.
    - Ensure that the table does not exceed the maximum width constraint.
        - Columns may be sized using an arbitrary combination of intrinsic widths and flexible space. Some columns also specify a minimum intrinsic width. As a result, it's not possible to resize a table using flex alone. An iterative approach is necessary to resize columns to respect the maximum width constraint without violating their other layout characteristics. The amount by which the table exceeds the maximum width constraint is the deficit.
        - Flexible columns are repeatedly shrunk until they've all reached their minimum intrinsic widths (i.e., no flexible columns remain) or the deficit has been eliminated.
            - The deficit is divided according to each column's flex factor (in relation to the total flex factor, which may change as noted below).
            - If this amount would shrink the column below its minimum width, the column is clamped to this width and the deficit reduced by the corresponding delta. The column is no longer considered flexible (reducing the total flex factor for subsequent calculations). Otherwise, the deficit is reduced by the full amount.
            - This process is iterative because some columns cannot be shrunk by the full amount.

        - Any remaining deficit must be addressed using inflexible columns (all flexible space has been consumed). Columns are considered “available” if they haven’t reached their minimum width. Available columns are repeatedly shrunk until the deficit is eliminated or there are no more available columns.
            - The deficit is divided evenly between available columns.
            - If this amount would shrink the column below its minimum width, the column is clamped to this width and the deficit reduced by the corresponding delta (this reduces the number of available columns). Otherwise, the deficit is reduced by the full amount.
            - This process is iterative because some columns cannot be shrunk by the full amount.

What are a table’s intrinsic dimensions?

- The table’s intrinsic widths are calculated as the sum of each column’s largest intrinsic width (using maximum or minimum dimensions with no height constraint, via `TableColumnWidth`).
- The table’s minimum and maximum intrinsic heights are equivalent, representing the sum of the largest intrinsic height found in each row (using the calculated column width as input). That is, each row is as tall as its tallest child, with the table’s total height corresponding to the sum of all such heights.
    - Concrete column widths are computed (via `RenderTable._computeColumnWidths`) using the width argument as a tight constraint.
    - Next, the largest maximum intrinsic height for each row is calculated (via `RenderBox.getMaxIntrinsicHeight`) using the calculated column width. The maximum row heights are summed to produce the table’s intrinsic height.

How does a table layout its children?

- If the table has zero columns or rows, it’s as small as possible given the incoming constraints.

- First, concrete columns widths are calculated. These widths are incrementally summed to produce a list of x-coordinates describing the left edge of each column (`RenderTable._columnLefts`). The copy of this list used by layout is flipped for right-to-left locales. The overall table width is defined as the sum of all columns widths (e.g., the last column x-coordinate plus the last column's width).
- Next, a list of a y-coordinates describing the top edge of each row is calculated incrementally (`RenderTable._rowTops`). Child layout proceeds as this list is calculated (i.e., row-by-row).
    - The list of row tops (`RenderTable._rowTops`) is cleared and seeded with an initial y-coordinate of zero (i.e., layout starts from the origin along the y-axis). The current row height is zeroed as are before- and after-baseline distances. These values track the maximum dimensions produced as cells within the row are laid out. The before-baseline distance is the maximum distance from a child's top to its baseline; the after-baseline distance is the maximum distance from a child's baseline to its bottom.
    - Layout pass: iterate over all non-null children within the row, updating parent data (i.e., x- and y-coordinates within the table) and performing layout based on the child's vertical alignment (read from parent data and set by `TableCell` , a `ParentDataWidget` subclass).
        - Children with top, middle, or bottom alignment are laid out with unbounded height and a tight width constraint corresponding to the column's width.
        - Children with baseline alignment are also laid out with unbounded height and a tight width constraint.
            - Children with a baseline (via `RenderBox.getDistanceToBaseline`) update the baseline distances to be at least as large as the child's values.
            - Children without a baseline update the row's height to be at least as large as the child's height. These children are positioned at the column's left edge and the row's top edge (this is the only position set during the first pass).
        - Children with fill alignment are an exception; these are laid out during the second pass, once row height is known.
    - If a baseline is produced during the first pass, row height is updated to be at least as large as the total baseline distance (i.e., the sum of before- and after-baseline distances).
        - The table's baseline distance is defined as the first row's before-baseline distance.

    - Positioning pass: iterate over all non-null children within the row, positioning them based on vertical alignment.
        - Children with top, middle, and bottom alignment are positioned at the column's left edge and the row's top, middle, or bottom edges, respectively.
        - Children with baseline alignment and an actual baseline are positioned such that all baselines align (i.e., each child's baseline is coincident with the maximum before baseline distance). Those without baselines have already been positioned.
        - Children with fill alignment are now laid out with tight constraints matching the row's height and the column's width; children are positioned at the column's left edge and the row's top edge.
    - Proceed to the next row by calculating the next row's top using the row height (and adding it to `RenderTable._rowTops`).
- The table's width and height (i.e., size) is defined as the sum of columns widths and row heights, respectively.

How does a table paint its children?

- If the table has zero columns or rows, its border (if defined) is painted into a zero-height rectangle matching the table's width.
- Each non-null decoration (`RenderTable._rowDecorations`) is painted via `Decoration.createBoxPainter`. Decorations are positioned using the incoming offset and the list of row tops (`RenderTable._rowTops`).
- Each non-null child is painted at the position calculated during layout, adjusted by the incoming offset.
- Finally, the table's border is painted using the list of row and column edges (these lists are filtered such that only interior edges are passed to `TableBorder.paint`).
    - The border will be sized to match the total width consumed by columns and total height consumed by rows.
    - The painted height may fall short of the render object's actual height (i.e., if the total row height is less than the minimum height constraint). In this case, there will be empty space below the table.
    - Table layout always satisfies the minimum width constraint, so there will never be empty horizontal space.

Material


Business Logic

Navigation

How does navigation work?

How does navigation integrate with gestures?

When and where are routes rendered?

How does local history work?

- `ModalRoute`

How do overlays work?

State Management

Async Programming

Testing