Flutter Cheat Sheet

A comprehensive cheat sheet for Flutter development, covering widgets, layouts, state management, navigation, and more.

Core Widgets

Basic Widgets

`Text`	Displays a string of text. `Text('Hello, Flutter!')`
`Image`	Displays an image from various sources. `Image.asset('assets/my_image.png')`
`Icon`	Displays an icon from the built-in icon set. `Icon(Icons.star)`
`RaisedButton`	A button that elevates when pressed. `RaisedButton(child: Text('Press'), onPressed: () {})`
`FlatButton`	A simple button without elevation. `FlatButton(child: Text('Press'), onPressed: () {})`
`IconButton`	A button with an icon. `IconButton(icon: Icon(Icons.add), onPressed: () {})`

Layout Widgets

`Container`	A widget for applying padding, margin, borders, background color, and more. `Container(padding: EdgeInsets.all(16.0), child: Text('Hello'))`
`Row`	Arranges children in a horizontal line. `Row(children: [Text('A'), Text('B')])`
`Column`	Arranges children in a vertical line. `Column(children: [Text('A'), Text('B')])`
`Stack`	Overlays children on top of each other. `Stack(children: [Container(color: Colors.red), Text('Overlay')])`
`Center`	Centers its child within itself. `Center(child: Text('Centered'))`
`Padding`	Adds space around its child. `Padding(padding: EdgeInsets.all(8.0), child: Text('Padded'))`

State Management

StatelessWidget

A widget that does not require mutable state.

```
class MyStatelessWidget extends
StatelessWidget {
 @override
 Widget build(BuildContext context) {
 return Text('Hello');
 }
}
```

StatefulWidget

A widget that has mutable state which can change during the lifetime of the widget.

```
class MyStatefulWidget extends StatefulWidget
{
 @override
 _MyStatefulWidgetState createState() =>
_MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends
State<MyStatefulWidget> {
 int _counter = 0;

 @override
 Widget build(BuildContext context) {
 return Text('Counter: $_counter');
 }
}
```

setState

A method to trigger UI updates in a StatefulWidget.

```
setState(() {
 _counter++;
});
```

Provider

Overview	A popular package for state management, using inherited widgets and dependency injection.
Usage	`ChangeNotifierProvider( create: (_) => MyModel(), child: Consumer<MyModel>( builder: (context, model, child) => Text(model.data), ), );`

Bloc/Cubit

Overview	Libraries for managing state using reactive programming principles.
Usage	`BlocProvider( create: (_) => MyBloc(), child: BlocBuilder<MyBloc, MyState>( builder: (context, state) => Text(state.data), ), );`

Riverpod

Overview	A reactive caching and data-fetching system. It simplifies accessing network requests with zero boilerplate.

Usage	`final helloWorldProvider = Provider((_) => 'Hello world');`
	
	`Consumer(builder: (context, watch, _) { final value = watch(helloWorldProvider); return Text(value); });`

Navigation & Routing

Basic Navigation

`Navigator.push`	Pushes a new route onto the navigator's stack. `Navigator.push(context, MaterialPageRoute(builder: (context) => NewRoute()));`
`Navigator.pop`	Pops the current route off the navigator's stack. `Navigator.pop(context);`
`MaterialPageRoute`	A route that transitions to the next screen using a platform-specific animation. `MaterialPageRoute(builder: (context) => NewRoute())`

Named Routes

Defining Routes	Define routes in `MaterialApp`. `MaterialApp( routes: { '/newRoute': (context) => NewRoute(), }, );`
Navigating to Named Routes	Navigate using `Navigator.pushNamed`. `Navigator.pushNamed(context, '/newRoute');`

Passing Data to Routes

Pass data via route arguments.

```
Navigator.push(
 context,
 MaterialPageRoute(
 builder: (context) => NewRoute(data: 'Hello'),
 ),
);
```

Returning Data from Routes

Return data using `Navigator.pop` with a value.

```
Navigator.pop(context, 'Returned data');
```

Asynchronous Operations

Future

Definition	Represents a value that will be available at some time in the future.
Usage	`Future<String> fetchData() async { await Future.delayed(Duration(seconds: 2)); return 'Data loaded'; }`
Handling Future	`fetchData().then((data) { print(data); });`

async/await

Definition	Syntactic sugar for working with Futures in a more readable way.
Usage	`Future<void> main() async { String data = await fetchData(); print(data); }`

Stream

Definition	A sequence of asynchronous events.
Usage	`Stream<int> countStream() async* { for (int i = 0; i < 5; i++) { await Future.delayed(Duration(seconds: 1)); yield i; } }`
Listening to Stream	`countStream().listen((number) { print(number); });`

FutureBuilder

Definition	A widget that builds itself based on the latest snapshot of interaction with a Future.
Usage	`FutureBuilder<String>( future: fetchData(), builder: (context, snapshot) { if (snapshot.hasData) { return Text(snapshot.data); } else if (snapshot.hasError) { return Text('Error: ${snapshot.error}'); } else { return CircularProgressIndicator(); } }, );`

StreamBuilder

Definition	A widget that builds itself based on the latest snapshot of interaction with a Stream.
Usage	`StreamBuilder<int>( stream: countStream(), builder: (context, snapshot) { if (snapshot.hasData) { return Text('Number: ${snapshot.data}'); } else if (snapshot.hasError) { return Text('Error: ${snapshot.error}'); } else { return CircularProgressIndicator(); } }, );`