

# Lecture 07

## Styling and Layout with Flexbox

# Contents

- Style
- Height and Width
- Layout with Flexbox

# Style in React Native

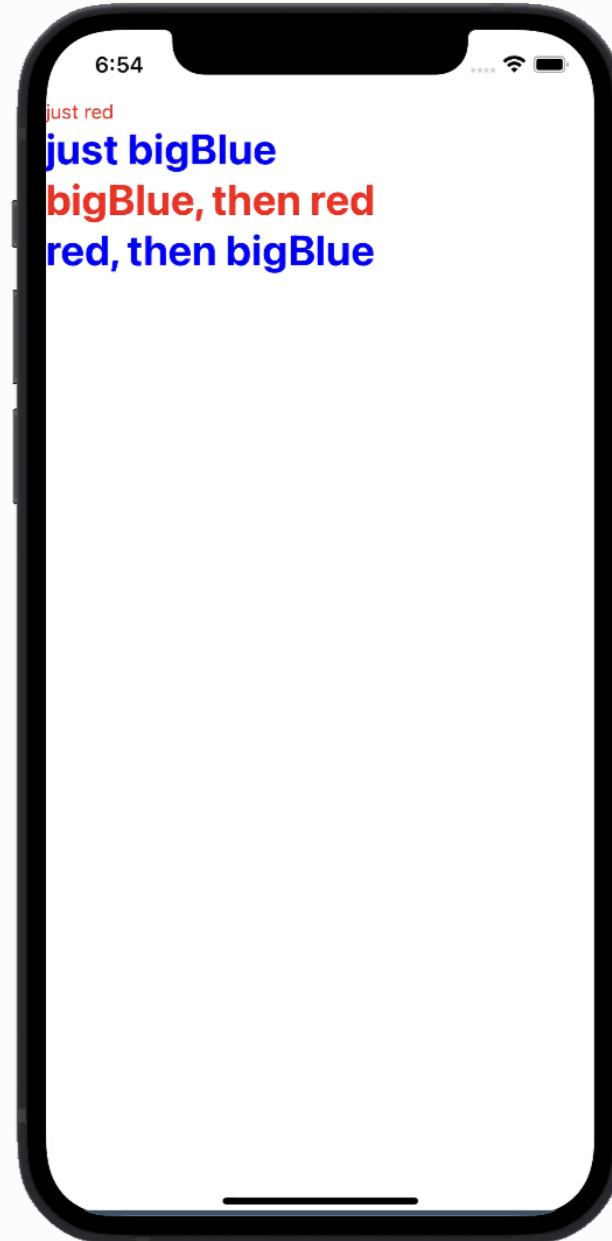
- **Styling with JavaScript:** React Native components accept a style prop, with property names written in camelCase (e.g., `backgroundColor` instead of `background-color`).
- **Defining styles:**
  - Styles can be a simple `JavaScript` object.
  - An array of styles can be used (the last style in the array takes precedence).
  - For complex components, using `StyleSheet.create` helps keep styles organized.

# Style in React Native

- As a component grows in complexity, it is often cleaner to use `StyleSheet.create` to define several styles in one place.
- Benefits:
  - **Performance**: styles are immutable and can be referenced efficiently.
  - **Validation**: validation, suggestion and warnings for invalid properties.
  - **Consistency**
  - **Better Integration**: integrates better with React Native, styles are optimized for the platform.
  - **Minification**: styles are minified, reducing app bundle size.

# Style in React Native

```
import React from 'react';
import { StyleSheet, Text, View } from 'react-native';
export default function LotsOfStyles() {
  return (<View style={styles.container}>
    <Text style={styles.red}>just red</Text>
    <Text style={styles.bigBlue}>just bigBlue</Text>
    <Text style={[styles.bigBlue, styles.red]}>
      bigBlue, then red</Text>
    <Text style={[styles.red, styles.bigBlue]}>
      red, then bigBlue</Text>
  </View>);
}
const styles = StyleSheet.create({
  container: { marginTop: 50 },
  bigBlue: { color: 'blue', fontWeight: 'bold', fontSize: 30 },
  red: { color: 'red' }
});
```



# Style in React Native

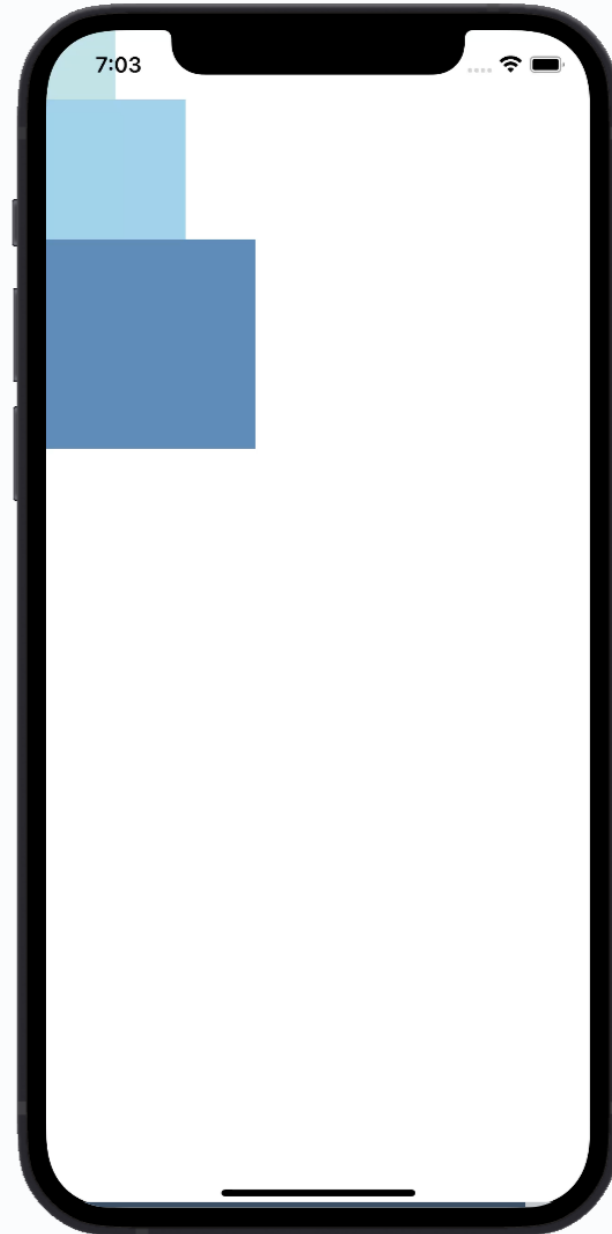
- **Inheriting styles:** A component can accept a style prop and pass it to subcomponents, allowing styles to "cascade" like in CSS.
- **Compatibility issues:** Some differences from web CSS include touch areas not extending beyond the parent view and negative margins not being supported on Android.

# Fixed Dimensions

- The general way to set the dimensions of a component is by adding a fixed `width` and `height` to style.
- All dimensions in React Native are *unitless*, and represent density-independent pixels (**dp**).
- Setting dimensions this way is common for components whose size should always be fixed to a number of points and not calculated based on screen size.

# Fixed Dimensions

```
<View style={{ flex: 1 }}>
  <View style={{
    width: 50,
    height: 50,
    backgroundColor: 'powderblue'
  }} />
  <View style={{
    width: 100,
    height: 100,
    backgroundColor: 'skyblue'
  }} />
  <View style={{
    width: 150,
    height: 150,
    backgroundColor: 'steelblue'
  }} />
</View>
```



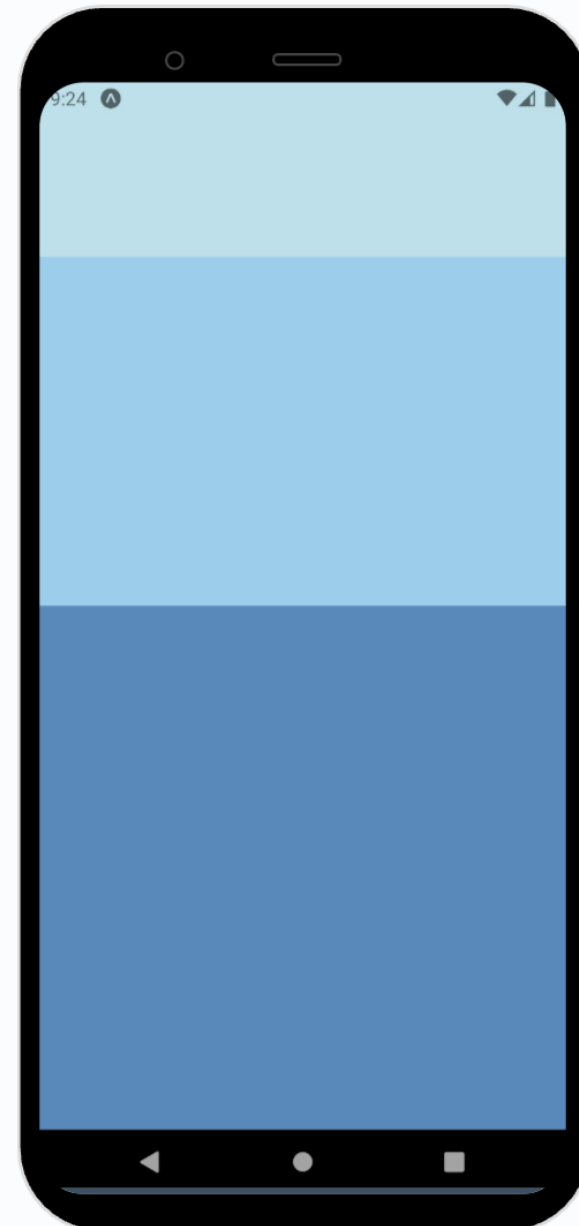


# Flex Dimensions

- Use `flex` in a component's style to make it expand or shrink based on available space.
  - `flex: 1` makes a component fill all available space, sharing it equally with siblings.
  - A larger `flex` value gives a component a greater share of space.
  - A component can only expand if its parent has a defined size (fixed dimensions or `flex`).

# Flex Dimensions

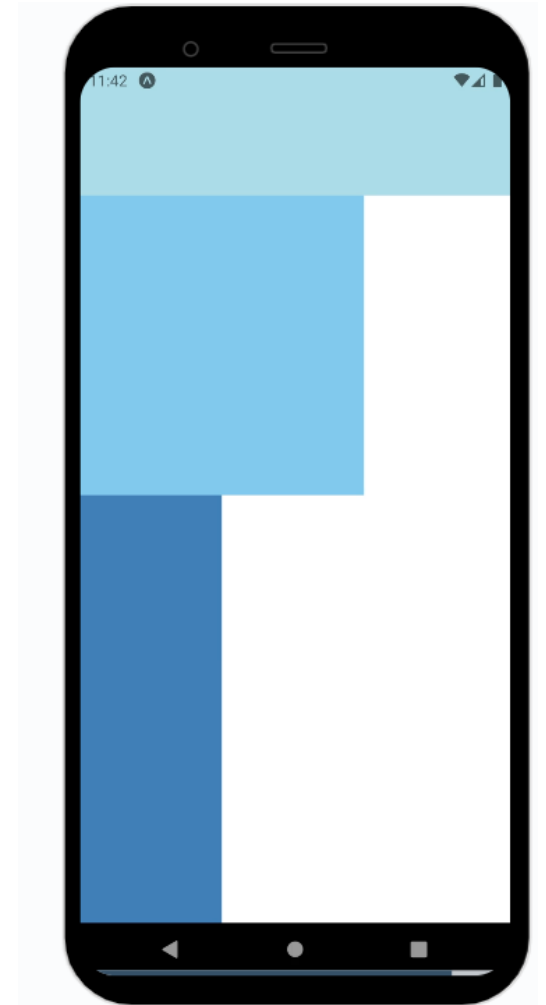
```
export default function FlexDimensionsBasics() {  
  return (  
    // Try removing the `flex: 1` on the parent View.  
    // The parent will not have dimensions, so the  
    // children can't expand.  
    // What if you add `height: 300` instead of `flex: 1`?  
    <View style={{ flex: 1 }}>  
      <View style={{  
        flex: 1, backgroundColor: 'powderblue'  
      }} />  
      <View style={{  
        flex: 2, backgroundColor: 'skyblue'  
      }} />  
      <View style={{  
        flex: 3, backgroundColor: 'steelblue'  
      }} />  
    </View>  
  );  
};
```



# Percentage Dimensions

- Instead of `flex`, you can use **percentage values** to control a component's size, but the parent must have a defined size.

```
<View style={{ height: '100%' }}>
  <View style={{
    height: '15%',
    backgroundColor: 'powderblue'
  }} />
  <View style={{
    width: '66%',
    height: '35%',
    backgroundColor: 'skyblue'
  }} />
  <View style={{
    width: '33%',
    height: '50%',
    backgroundColor: 'steelblue'
  }} />
</View>
```



# Layout with Flexbox

- A component can specify the layout of its children using the Flexbox algorithm. Flexbox is designed to provide a consistent layout on different screen sizes.
- You will normally use a combination of `flexDirection`, `alignItems`, and `justifyContent` to achieve the right layout.

# Layout with Flexbox

- **CAUTION**

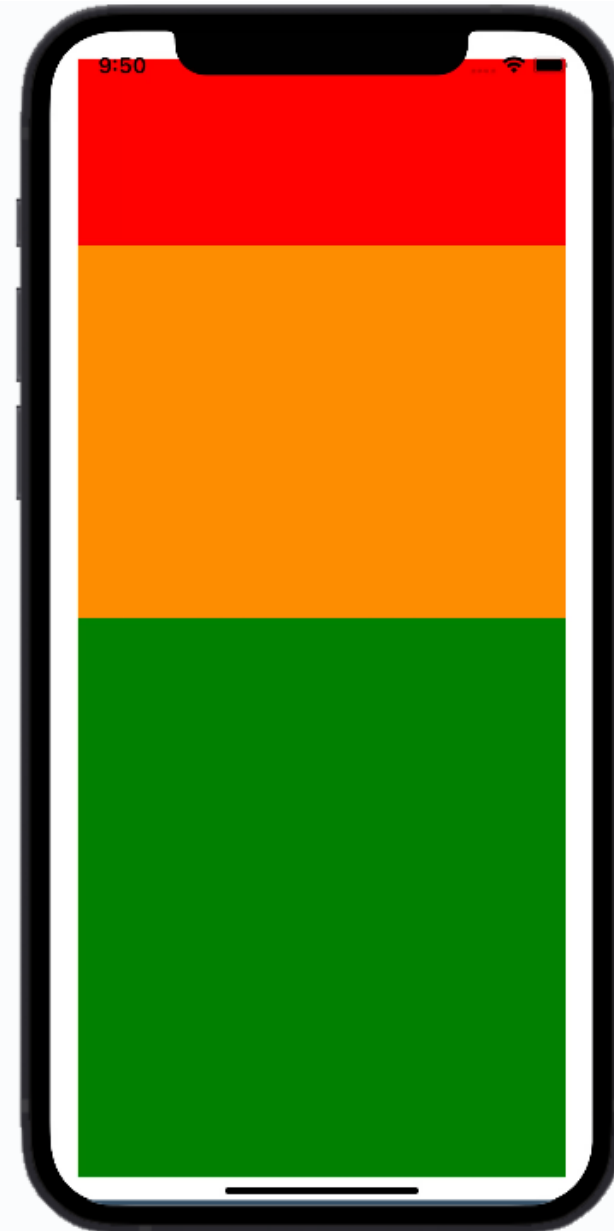
- Flexbox works the same way in React Native as it does in CSS on the web, with a few exceptions.
- The defaults are different, with `flexDirection` defaulting to `column` instead of `row`, `alignContent` defaulting to `flex-start` instead of `stretch`, `flexShrink` defaulting to `0` instead of `1`, the `flex` parameter only supports a single number.

# Flex

- `flex` will define how your items are going to “**fill**” over the available space along your main axis. Space will be divided according to each element's flex property.
- In the following example, the red, orange, and green views are all children in the container view that has `flex: 1` set. The red view uses `flex: 1`, the orange view uses `flex: 2`, and the green view uses `flex: 3`. **1+2+3 = 6**, which means that the red view will get 1/6 of the space, the orange 2/6 of the space, and the green 3/6 of the space.

# Flex

```
export default function Flex() {  
  return (  
    <View style={ [  
      styles.container,  
      { // Try setting 'flexDirection' to 'row'  
        flexDirection: 'column'  
      } ] }>  
      <View style={{flex: 1, backgroundColor: 'red'}} />  
      <View style={{flex: 2, backgroundColor: 'darkorange'}} />  
      <View style={{flex: 3, backgroundColor: 'green'}} />  
    </View>);  
};  
  
const styles = StyleSheet.create({  
  container: {  
    flex: 1,  
    padding: 20  
  }  
});
```



# Flex Direction

- `flexDirection` controls the direction in which the children of a node are laid out. This is also referred to as the main axis. The cross axis is the axis perpendicular to the main axis, or the axis which the wrapping lines are laid out in.
  - `column` (default value): Align children from top to bottom. If wrapping is enabled, then the next line will start to the right of the first item on the top of the container.

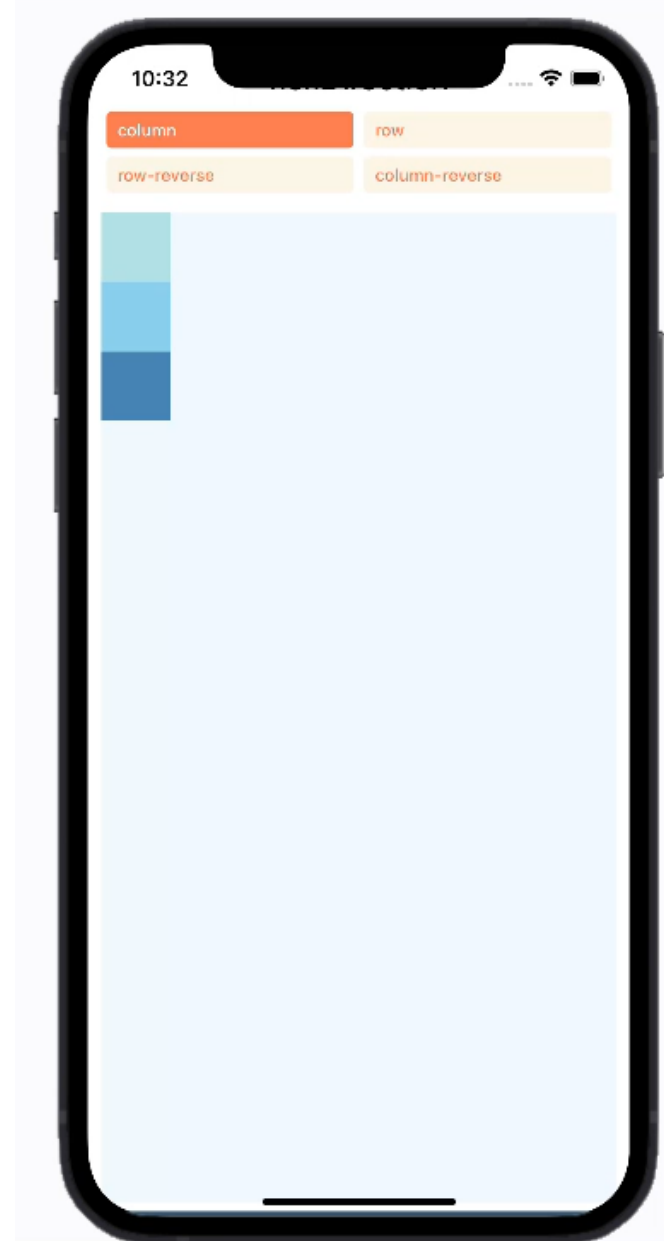


# Flex Direction

- `row`: Align children from left to right. If wrapping is enabled, then the next line will start under the first item on the left of the container.
- `column-reverse`: Align children from bottom to top. If wrapping is enabled, then the next line will start to the right of the first item on the bottom of the container.
- `row-reverse`: Align children from right to left. If wrapping is enabled, then the next line will start under the first item on the right of the container.

# Example of `flexDirection` in React Native

- This React Native program demonstrates the use of `flexDirection` to control the layout direction of child elements inside a container.
- It allows users to interactively change the `flexDirection` and observe how the child elements (colored boxes) rearrange dynamically.



# Example of flexDirection in React Native

- This is the **initial setup** for demonstrating flexDirection in a React Native app. It serves as a **foundation** for later slides where interactive elements and styling will be added.

```
import React, { useState } from 'react';
import { StyleSheet, Text, View } from 'react-native';
const FlexDirectionExample = () => {
  const [flexDirection, setFlexDirection] = useState('column');
  return (
    <View style={styles.container}>
      <Text style={styles.label}>flexDirection: {flexDirection}</Text>
      {/* Remaining parts will be introduced in later slides */}
    </View>
  );
};
```

# Example of flexDirection in React Native

- Changing flexDirection with Buttons.

```
<View style={styles.buttons}>
  {['column', 'row', 'row-reverse', 'column-reverse'].map(value => (
    <TouchableOpacity
      key={value}
      onPress={() => setFlexDirection(value)}
      style={[styles.button, flexDirection === value && styles.selected]}>
        <Text style={styles.buttonText}>{value}</Text>
      </TouchableOpacity>
    )
  )
}</View>
```

# Example of `flexDirection` in React Native

- Changing layout with `flexDirection`.

```
<View style={[styles.boxContainer, { flexDirection }]}>  
  <View style={[styles.box, { backgroundColor: 'powderblue' }]} />  
  <View style={[styles.box, { backgroundColor: 'skyblue' }]} />  
  <View style={[styles.box, { backgroundColor: 'steelblue' }]} />  
</View>
```

# Example of flexDirection in React Native

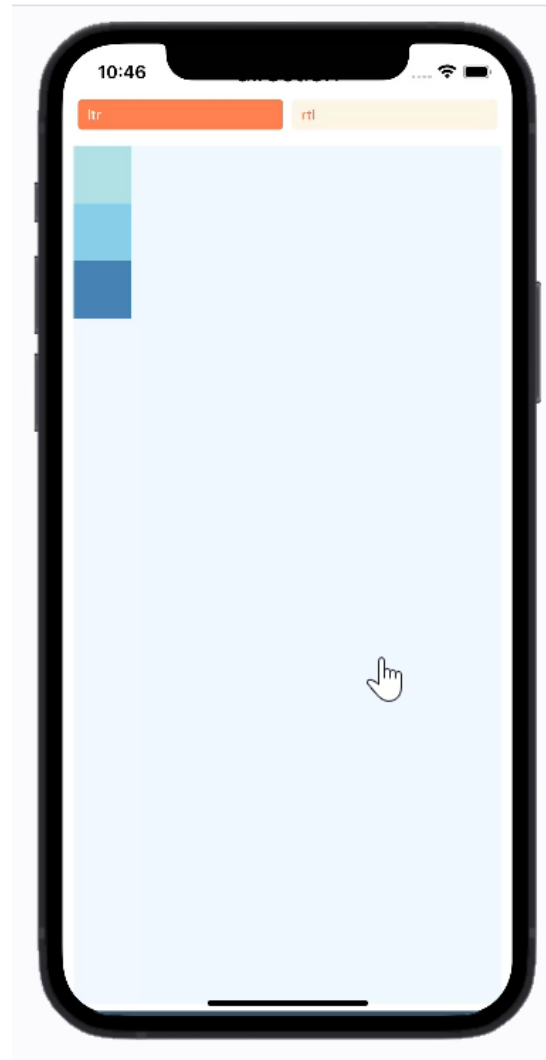
- Styles for flexDirection example app:

```
const styles = StyleSheet.create({
  container: { flex: 1, padding: 10 },
  label: { textAlign: "center", fontSize: 18, marginBottom: 10 },
  buttons: { flexDirection: "row", justifyContent: "center" },
  button: {
    padding: 8,
    margin: 5,
    backgroundColor: "oldlace",
    borderRadius: 5,
  },
  selected: { backgroundColor: "coral" },
  boxContainer: { flex: 1, justifyContent: "center", alignItems: "center" },
  box: { width: 50, height: 50, margin: 5 }
});
```

# Layout Direction

- Layout `direction` specifies the direction of a component's children and text.
- It affects what `start` and `end` refer to. In the default `LTR` mode, `start` means left and `end` means right.
  - `LTR` (default value): Text and children are laid out from left to right. Margin and padding applied to the start of an element are applied on the left side.
  - `RTL`: Text and children are laid out from right to left. Margin and padding applied to the start of an element are applied on the right side.

# Layout Direction App Demo





# Justify Content

- `justifyContent` describes how to align children within the main axis of their container.
  - Example: use this property to center a child horizontally within a container with `flexDirection` set to `row` or vertically within a container with `flexDirection` set to `column`.
- `flex-start` (default value): Align children of a container to the start of the container's main axis.
- `flex-end`: Align children of a container to the end of the container's main axis.

# Justify Content

- `center`: Align children of a container in the center of the container's main axis.
- `space-between`: Evenly space off children across the container's main axis, distributing the remaining space between the children.

# Justify Content

- `space-around`: Evenly space off children across the container's main axis, distributing the remaining space around the children.
  - Compared to `space-between`, using `space-around` will result in space being distributed to the beginning of the first child and end of the last child.
- `space-evenly` Evenly distribute children within the alignment container along the main axis.
  - The spacing between each pair of adjacent items, the main-start edge and the first item, and the main-end edge and the last item, are all exactly the same.

# Align Items

- `alignItems` describes how to align children along the cross axis of their container. It is similar to `justifyContent` but used for cross axis instead of main axis.
  - `stretch` (default value): Stretch children of a container to match the `height` of the container's cross axis.
  - `flex-start`: Align children of a container to the start of the container's cross axis.
  - `flex-end`: Align children of a container to the end of the container's cross axis.
  - `center`: Align children of a container in the center of the container's cross axis.
  - `baseline`: Align children of a container along a common baseline. Individual children can be set to be the reference baseline for their parents.

# Align Self

- `alignSelf` has the same options and effect as `alignItems` but instead of affecting the children within a container, you can apply this property to a single child to change its alignment within its parent.
- `alignSelf` overrides any option set by the parent with `alignItems`.

# Align Content

- `alignContent` defines the distribution of lines along the cross-axis. This only has effect when items are wrapped to multiple lines using `flexWrap`.
  - `flex-start` (default value): Align wrapped lines to the start of the container's cross axis.
  - `flex-end`: Align wrapped lines to the end of the container's cross axis.
  - `stretch`: Stretch wrapped lines to match the height of the container's cross axis.

# Align Content

- `center`: Align wrapped lines in the center of the container's cross axis.
- `space-between`: Evenly space wrapped lines across the container's cross axis, distributing the remaining space between the lines.
- `space-around`: Evenly space wrapped lines across the container's cross axis, distributing the remaining space around the lines. Each end of the container has a half-sized space compared to the space between items.
- `space-evenly`: Evenly space wrapped lines across the container's cross axis, distributing the remaining space around the lines. Each space is the same size.

# Flex Wrap

- The `flexWrap` property is set on containers and it controls what happens when children overflow the size of the container along the main axis.
  - By default, children are forced into a single line (which can shrink elements). If wrapping is allowed, items are wrapped into multiple lines along the main axis if needed.
- When wrapping lines, `alignContent` can be used to specify how the lines are placed in the container.

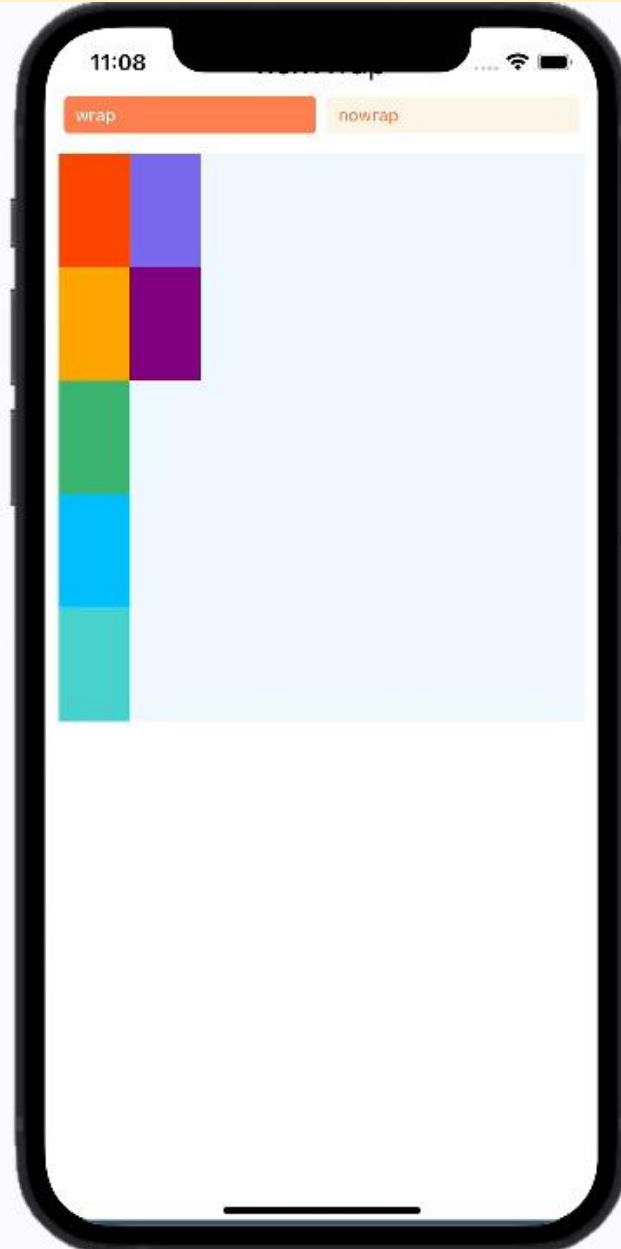


# Flex Wrap

```
<View style={[styles.box, {backgroundColor: 'orangered'}]} />
<View style={[styles.box, {backgroundColor: 'orange'}]} />
<View style={[styles.box, {backgroundColor: 'mediumseagreen'}]}
/>

<View style={[styles.box, {backgroundColor: 'deepskyblue'}]} />
<View style={[styles.box, {backgroundColor:
'mediumturquoise'}]} />
  <View style={[styles.box, {backgroundColor:
'mediumslateblue'}]} />
    <View style={[styles.box, {backgroundColor: 'purple'}]} />
  </PreviewLayout>
);
};

const PreviewLayout = ({
  label,
  children,
  values,
  selectedValue,
  setSelectedValue,
}) => (
  <View style={{padding: 10, flex: 1}}>
    <Text style={styles.label}>{label}</Text>
    <View style={styles.row}>
      {values.map(value => (
        <TouchableOpacity
          key={value}
          onPress={() => setSelectedValue(value)}
          style={[styles.button, selectedValue === value &&
styles.selected]}>
          <Text
```



# Flex Basis and Grow

- **flexBasis**

- Defines the default size of an item along the main axis.
- Behaves like `width` if `flexDirection: row`, or `height` if `flexDirection: column`.
- The item's size before `flexGrow` and `flexShrink` adjustments.

- **flexGrow**

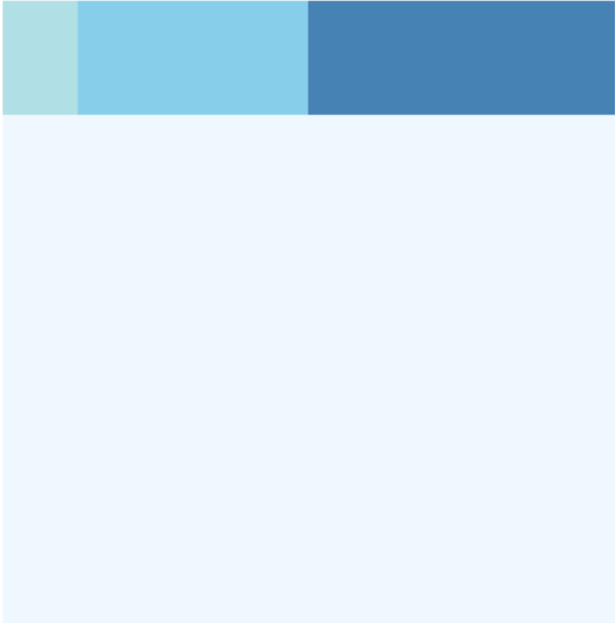
- Determines how much remaining space a child should take in the container.
- Accepts a value  $\geq 0$  (default is 0).
- The space is distributed proportionally based on `flexGrow` values.

# flexShrink

- Defines how a child shrinks when content overflows the container.
- Accepts a value  $\geq 0$  (default is 0, but 1 in web).
- Works with `flexGrow` to allow elements to expand and shrink dynamically.

# Flex Basis, Grow, and Shrink

Box	Box	Box
flexBasis <u>auto</u>	flexBasis <u>100</u>	flexBasis <u>200</u>
flexShrink <u>1</u>	flexShrink <u>0</u>	flexShrink <u>1</u>
flexGrow <u>0</u>	flexGrow <u>1</u>	flexGrow <u>0</u>



# Row Gap, Column Gap and Gap

- `rowGap` sets the size of the gap (gutter) between an element's rows.
- `columnGap` sets the size of the gap (gutter) between an element's columns.
- `gap` sets the size of the gap (gutter) between rows and columns. It is a shorthand for `rowGap` and `columnGap`.
  - You can use `flexWrap` and `alignContent` along with `gap` to add consistent spacing between items.

# Row Gap, Column Gap and Gap

