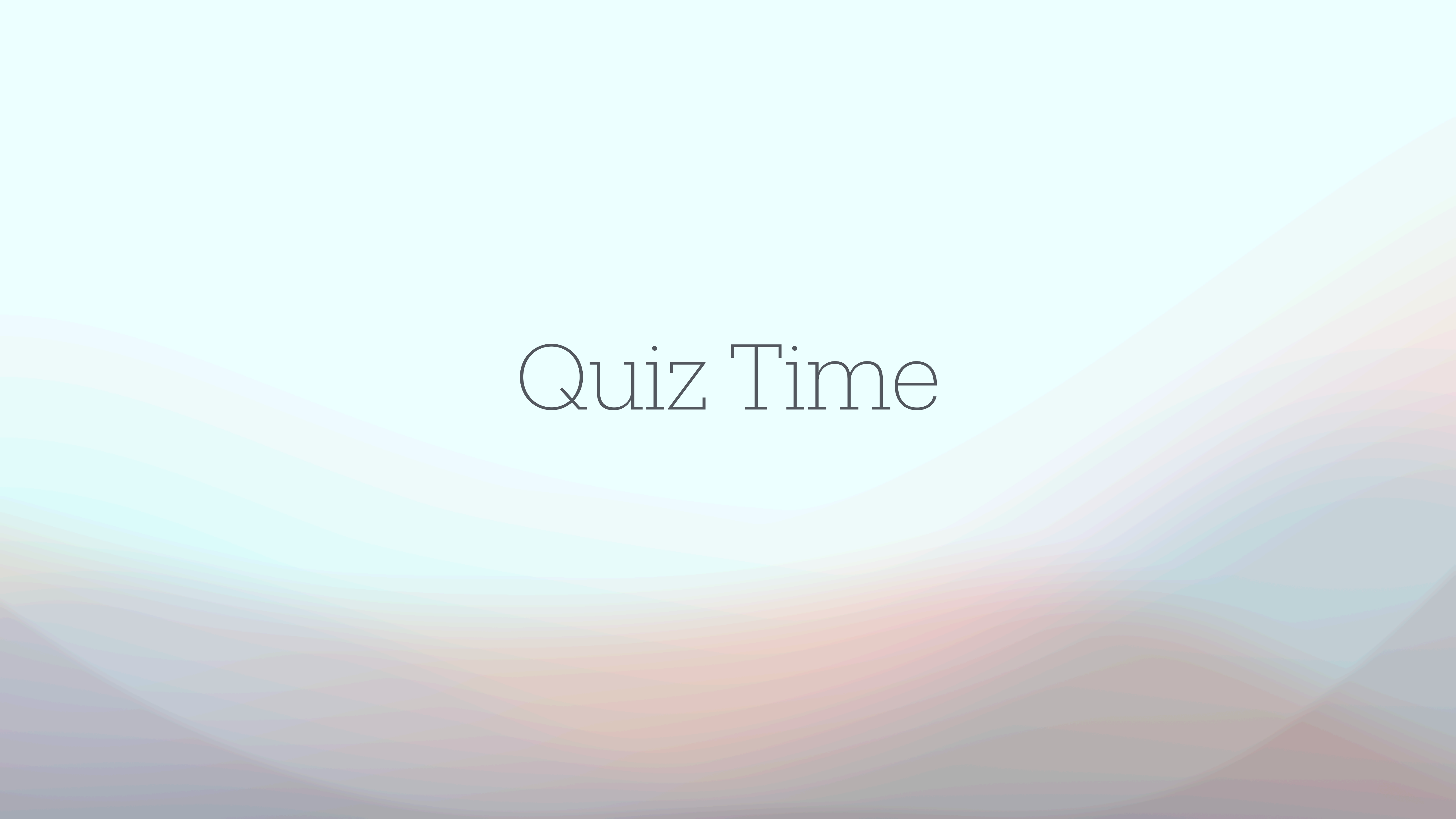


Layouts & More



Quiz Time

Quiz A

```
fun main() {  
    var name: String? = "John"  
    val length: Int? = name?.length  
    name = null  
    println("Name: $name")  
    println("Length: ${length ?:  
"unknown"}")  
}
```

Quiz B

```
fun main() {  
    val a: String? = "Hi"  
    val b: String? = null  
    println(a?.length ?: b?.length ?: 0)  
}
```

Quiz A

Write a function called **calculateSum** that takes a list of integers and returns the sum of all even numbers in the list.

Example:

```
val numbers = listOf(1, 2, 3, 4, 5, 6)
println(calculateSum(numbers))
```

// Should output: 12 (2 + 4 + 6)

Quiz B

Write a function called **countOccurrences** that takes a list of strings and returns a map where the key is the string and the value is how many times that string appears in the list.

Example:

```
fun main() { val fruits = listOf("apple", "banana",
    "apple", "orange") val result =
    countOccurrences(fruits)
```

// Should output map with: apple->2, banana->1, orange->1 }

Introduction to screen densities

- Android devices come with varying screen sizes and pixel densities
- Same pixel dimensions can appear different across devices
- Need for consistent UI elements across different screen densities
- **Physical pixels:** Actual pixels on the screen
- **Density-independent pixels (dp):** Abstract unit that scales with screen density
- **Scale-independent pixels (sp):** Similar to dp, but also scales with user's font size preference

Density-independent Pixels (dp)

- Virtual pixel unit that maintains consistent size across different screen densities
- 1 dp = 1 physical pixel on a 160 dpi screen (baseline density)
- Automatically scales based on the actual screen density

$\text{dp} = (\text{width in pixels} * 160) / \text{screen density}$

$\text{px} = \text{dp} * (\text{dpi} / 160)$

A. Low-density screen (120 dpi): Physical pixels = $(48 \times 120) / 160 = 36$ pixels

B. Medium-density screen (160 dpi - baseline): Physical pixels = $(48 \times 160) / 160 = 48$ pixels

C. High-density screen (240 dpi): Physical pixels = $(48 \times 240) / 160 = 72$ pixels

When to Use dp

- Layout dimensions (width, height)
- Margins and padding
- Button sizes
- Image dimensions
- Any non-text UI element

Scale-independent Pixels (sp)

What is sp ?

- Similar to dp, but also scales with user's font size preferences
- Default scale is 1sp = 1dp
- Changes when user modifies system font size

When to Use sp

- Text sizes ONLY
- Ensures accessibility for users who need larger text
- Respects system-wide font size settings

Table 1. Configuration qualifiers for different pixel densities.

Density qualifier	Description
ldpi	Resources for low-density (<i>ldpi</i>) screens (~120 dpi).
mdpi	Resources for medium-density (<i>mdpi</i>) screens (~160 dpi). This is the baseline density.
hdpi	Resources for high-density (<i>hdpi</i>) screens (~240 dpi).
xhdpi	Resources for extra-high-density (<i>xhdpi</i>) screens (~320 dpi).
xxhdpi	Resources for extra-extra-high-density (<i>xxhdpi</i>) screens (~480 dpi).
xxxhdpi	Resources for extra-extra-extra-high-density (<i>xxxhdpi</i>) uses (~640 dpi).
nodpi	Resources for all densities. These are density-independent resources. The system doesn't scale resources tagged with this qualifier, regardless of the current screen's density.
tvdpi	Resources for screens somewhere between mdpi and hdpi; approximately ~213 dpi. This isn't considered a "primary" density group. It is mostly intended for televisions, and most apps don't need it—providing mdpi and hdpi resources is sufficient for most apps, and the system scales them as appropriate. If you find it necessary to provide tvdpi resources, size them at a factor of 1.33 * mdpi. For example, a 100x100 pixel image for mdpi screens is 133x133 pixels for tvdpi.

For more info <https://developer.android.com/training/multiscreen/screendensities>

Padding & Margins

- **Padding:** Space inside the View
- **Margin:** Space outside the View
- Use dp for all spacing measurements
- Consider both LTR and RTL layouts
- Material Design recommends multiples of 8dp

Visibility

- `android:visibility="visible"` // Default state, view is visible
- `android:visibility="invisible"` // View is hidden but takes up space
- `android:visibility="gone"` // View is hidden and doesn't take space

Linear Layout

- A ViewGroup that arranges children in a single direction
- Can be either horizontal or vertical
- Elements are placed one after another
- Most basic and commonly used layout in Android
- `android:orientation="vertical"` // Arranges items vertically
- `android:orientation="horizontal"` // Arranges items horizontally



Figure 1. A `LinearLayout` with three horizontally oriented children.

Layout Weight

- Distributes space among children
- Uses ratio-based division
- `android:layout_weight="1"` // Takes remaining space
- `android:layout_width="0dp"` // Required for horizontal weight
- `android:layout_height="0dp"` // Required for vertical weight

For more information visit <https://developer.android.com/develop/ui/views/layout/linear>

Linear Layout - Performance tips

- Avoid deep nesting of LinearLayouts
- Use ConstraintLayout for complex layouts
- Keep layout hierarchy shallow
- Use layout weights sparingly

Relative Layout

- Positions views relative to parent or other views
- More flexible than LinearLayout
- Views can overlap if needed
- Each child can specify multiple relationships

Relative layout - Key attributes

- `android:layout_alignParentTop="true"` // Align to parent top
- `android:layout_alignParentBottom="true"` // Align to parent bottom
- `android:layout_alignParentStart="true"` // Align to parent start
- `android:layout_alignParentEnd="true"` // Align to parent end
- `android:layout_centerInParent="true"` // Center in parent
- `android:layout_centerHorizontal="true"` // Center horizontally
- `android:layout_centerVertical="true"` // Center vertically

For more information visit <https://developer.android.com/develop/ui/views/layout/relative>

Constraint Layout

- A powerful **ViewGroup** that allows you to create **complex** and **responsive** layouts with **flat view hierarchies**.
- **Introduced in Android 7.0 (API 24)** to replace nested LinearLayouts and RelativeLayouts.
- Improves **performance** by reducing the number of views in the hierarchy.
- Similar to RelativeLayout but more powerful
- A restriction or limitation on the properties of a View that the layout attempts to respect

Constraint Layout

Basic constraints

- `app:layout_constraintTop_toTopOf="parent"`
- `app:layout_constraintStart_toStartOf="parent"`
- `app:layout_constraintEnd_toEndOf="parent"`
- `app:layout_constraintBottom_toBottomOf="parent"`
- `app:layout_constraintTop_toBottomOf="@id/other_view"`
- `app:layout_constraintStart_toEndOf="@id/other_view"`
- `app:layout_constraintBaseline_toBaselineOf="@id/other_view"`

Thank you!