🟦 Essentials

▼ 1. SOLID Principles

SOLID is a set of **five design principles** intended to make object-oriented code more understandable, flexible, and maintainable. It's super popular in interviews and system design discussions—even for React (not just backend!).

Here's a quick, simple summary of each letter:

S — Single Responsibility Principle (SRP)

> One class/module/function should have only one reason to change.

- Each piece should do **one job only**.
- Example: Don’t mix fetching data and rendering UI in the same component.

O — Open/Closed Principle (OCP)

> Software entities should be open for extension, but closed for modification.

- You should be able to add new functionality **without changing existing code**.
- Example: Use props, hooks, or inheritance to extend behavior, not rewrite the component/class itself.

L — Liskov Substitution Principle (LSP)

> Objects of a superclass should be replaceable with objects of a subclass without breaking the app.

- If you replace a parent class with a child, things should still work as expected.
- Example: A `Button` and a `FancyButton` (extending `Button`) should both work wherever a `Button` is expected.

I — Interface Segregation Principle (ISP) (Separation)

- **Focus:** How you design your interfaces (or APIs/classes).
- **Key Idea:**

> "No client should be forced to depend on methods it does not use."

- **In practice:**
  - Break up large interfaces into smaller, focused ones.
  - If you're in React, this means **don't create components with huge prop lists** that do too much—break them into smaller, specialized components.
- **Problem ISP Solves:**
  - Prevents unnecessary dependencies, bloated classes/components, and confusion for users of your code.

D — Dependency Inversion Principle (DIP)

- **Focus:** How your code depends on other code (high-level vs. low-level modules).
- **Key Idea:**

> "Depend on abstractions, not concretions."

- **In practice:**
  - Don't make your main components or business logic depend directly on specific utilities, APIs, or implementations.

- Use abstractions (like interfaces, hooks, or context) so you can swap the implementation easily (great for testing or future changes).

- **Problem DIP Solves:**
  - Reduces tight coupling between layers, makes code more modular and testable.

▼ 2. OOP

▼ Four Pillars of OOP

1. Encapsulation

- **Definition:** Wrapping data (state) and methods (behavior) together inside a class or object, and hiding the internal details from outside access.
- **Why it matters:** Protects the internal state and only exposes what's necessary through a public API (methods).
- **Example:**

```
class Counter {
  #count = 0; // private
  increment() { this.#count++; }
  getCount() { return this.#count; }
}
```

---

2. Abstraction

- **Definition:** Hiding complex implementation details and showing only the essential features.
- **Why it matters:** Makes code easier to use and understand.
- **Example:**

```
class Car {
  startEngine() { /* details hidden */ }
```

```
}
// You don’t need to know HOW startEngine works, just that you can call it.
```

---

3. Inheritance

- **Definition:** One class can inherit (reuse) properties and methods from another class.
- **Why it matters:** Promotes code reuse and establishes a parent-child relationship.
- **Example:**

```
class Animal { speak() { console.log('Sound'); } }
class Dog extends Animal { speak() { console.log('Woof!'); } }
```

---

4. Polymorphism

- **Definition:** The ability for different classes to be treated as instances of the same parent class, usually by sharing a common interface/method.
- **Why it matters:** Allows for flexible and interchangeable code.
- **Example:**

```
let animals = [new Dog(), new Cat()];
animals.forEach(animal ⇒ animal.speak()); // Calls the correct meth
od for each
```

▼ Composition vs Inheritance

- **Inheritance** means creating new classes based on existing ones (the new class "is a" type of the old class).
  - *Example:* A `Dog` is an `Animal` (Dog extends Animal).
- **Composition** means building classes using other classes as parts (the new class "has a" relationship).

- *Example:* A `Car` has an `Engine`.

- **Why does it matter?**
  - **Inheritance** can lead to rigid, tightly-coupled code if overused.
  - **Composition** offers more flexibility. You can change or swap parts without changing the parent class.
  - **Rule of thumb:** “Favor composition over inheritance” for code flexibility and reuse.

▼ Aggregation vs Composition

- **Both are “has-a” relationships.**
- **Aggregation:** The child can exist independently of the parent.
  - *Example:* A `Team` has `Players` (players exist even if the team is deleted).
- **Composition:** The child **cannot** exist independently.
  - *Example:* A `House` has `Rooms` (if the house is destroyed, the rooms go too).
- **Why does it matter?**
  - Helps model real-world relationships accurately.

▼ Interfaces & Abstract Classes

1. Interface

- **What is it?**

  An interface is a list of rules or a contract. It *doesn't* say *how* things should be done, only *what* needs to be done.

- **Analogy:**

  Think of a **remote control interface**:
  - All remotes must have `powerOn()`, `powerOff()`, `volumeUp()`, `volumeDown()`.
  - It doesn’t matter *how* the remote works inside, just that these buttons exist.

- **In code (TypeScript example):**

```
interface RemoteControl {
  powerOn(): void;
  powerOff(): void;
}

class TVRemote implements RemoteControl {
  powerOn() { console.log("TV is on"); }
  powerOff() { console.log("TV is off"); }
}

class ACRemote implements RemoteControl {
  powerOn() { console.log("AC is on"); }
  powerOff() { console.log("AC is off"); }
}
```

  - **Key:** Both `TVRemote` and `ACRemote` must have `powerOn` and `powerOff` methods.

---

2. Abstract Class

- **What is it?**

  An abstract class is a blueprint. It can have:

  - Some *implemented* methods (real code you can use)
  - Some *abstract* methods (no code, must be defined in subclasses)
  - You **cannot** create an object directly from an abstract class.

- **Analogy:**

  Think of an **abstract class** as a partially assembled toy:

  - Some pieces are finished.

    - Some are missing and must be added by whoever assembles the toy.
- **In code (TypeScript/JS style):**

```
abstract class Animal {
  abstract speak(): void; // Must be implemented by subclass

  move() {
    console.log("Animal is moving");
  }
}

class Dog extends Animal {
  speak() { console.log("Woof!"); } // Required by abstract class
}

// let animal = new Animal(); // ❌ Not allowed
let dog = new Dog();
dog.speak(); // "Woof!"
dog.move();  // "Animal is moving"
```

    - **Key:**
        - You can't do `new Animal()`.
        - You can do `new Dog()`, and `Dog` must define `speak()`, but gets `move()` for free.

▼ Access Modifiers (Public/Private/Protected)

- **Public:** Can be accessed from anywhere.
- **Private:** Only accessible inside the class.
- **Protected:** Accessible inside the class and subclasses.

▼ Method Overriding vs Overloading

- **Overriding:**

- Subclass replaces a method from the parent class with its own implementation.
- *Example:*

```
class Animal { speak() { console.log("sound"); } }
class Dog extends Animal { speak() { console.log("woof!"); } }
```

- **Overloading:**
  - Same method name, different parameters (not natively in JS, but possible in TypeScript or other languages).
  - *Example (TypeScript):*

```
function add(a: number, b: number): number;
function add(a: string, b: string): string;
function add(a: any, b: any): any { return a + b;
```

▼ Design Patterns in OOP

- **What are they?**
  - Proven, reusable solutions to common problems in software design (e.g., how to create objects, how to structure code, how to handle events).
- **Common OOP Patterns:**
  - **Singleton:** Only one instance exists.
  - **Factory:** A function/class that creates objects for you.
  - **Observer:** Objects notify others about changes.
  - **Decorator:** Add features to objects dynamically.
  - **Strategy:** Select algorithm/behavior at runtime.
- **Why does it matter?**
  - Makes your code more scalable, modular, and easier to maintain.

▼ Debouncing/Throttling

1. Debouncing

- **Definition:**

  Limits how often a function is called by only running it **after** a certain amount of time has *passed since the last trigger*.

- **Use case:**
  - Search bars (wait until the user stops typing before sending a request)
  - Resizing events
- **Example:**

  If you debounce a function with 500ms, and the user keeps typing, the function won't run until they stop for 500ms.

---

2. Throttling

- **Definition:**

  Limits a function to run **at most once every X milliseconds**, no matter how many times it's triggered.

- **Use case:**
  - Handling scroll events (don't call an API every pixel!)
  - Repeated button presses
- **Example:**

  If you throttle with 1000ms, even if the event fires 100 times in one second, the function only runs once per second.

How to Implement in React Native?

You can use utility libraries like **lodash** or write your own debounce/throttle functions.

Using Lodash (Recommended)

First, install lodash (if you haven't):

```
npm install lodash
```

Debounce Example (Search Input):

```
import React, { useCallback } from 'react';
import { TextInput } from 'react-native';
import debounce from 'lodash/debounce';

const SearchBar = () => {
 // Debounced search function
 const handleSearch = useCallback(
  debounce((text) => {
   // Your API call here
   console.log('Searching for:', text);
  }, 500), []
 );

 return (
  <TextInputplaceholder="Type to search..."
   onChangeText={handleSearch}
  />
 );
};

export default SearchBar;
```

How it works:

- `handleSearch` will only trigger 500ms **after** the user stops typing.

Throttle Example (Button):

```
import React from 'react';
import { Button } from 'react-native';
```

```
import throttle from 'lodash/throttle';

const ThrottledButton = () => {
  // Throttled function
  const handlePress = throttle(() => {
    console.log('Button pressed!');
  }, 2000);

  return <Button title="Press Me" onPress={handlePress} />;
};

export default ThrottledButton;
```

How it works:

- Even if you tap the button many times, it will only run the function once every 2 seconds.

Why Use Debouncing/Throttling?

- **Performance:** Prevents unnecessary API calls or expensive calculations.
- **User experience:** Smooth, responsive UI without lag or “spamming” the backend.

Pro tip:

- Debounce = “wait and do it after the user pauses.”
- Throttle = “limit how often you do it.”

Common Use Cases for Debounce

1. Search-As-You-Type (API Calls)

- Wait until the user stops typing for, say, 400ms before sending an API request.
- Prevents sending a request on every keystroke.

2. Window Resize/Orientation Change

- In a web app: only run a resize handler after the user finishes resizing the window.
- In React Native: after the user finishes rotating the phone, update layout.

3. Auto-Save Form

- Wait for the user to pause typing in a form before auto-saving to the server.

4. Input Validation

- Don’t show validation errors until the user stops typing, so they’re not distracted by messages while editing.

5. Scroll-based UI changes

- Only perform expensive calculations (e.g., showing/hiding a header) after the user has stopped scrolling.

6. Search Filter or Autocomplete Dropdown

- As the user types, wait for them to finish before updating suggestions.

Common Use Cases for Throttling

1. Scroll Events

- When the user scrolls a page or list, the scroll event can fire dozens or hundreds of times per second.
- Throttle heavy tasks (like lazy-loading images or updating analytics) so they only run, for example, once every 200ms.

2. Button Spamming

- Prevent a button (like "Submit", "Like", or "Load More") from firing multiple times rapidly.

- Throttle the click handler so it can only run once per second (or whatever interval you choose).

3. Window Resize Events

- On web, resizing the browser window can trigger a ton of events.
- Throttle the resize handler to recalculate layout only every 300ms, for instance.

4. Drag and Drop / Swiping Gestures

- Limit how often updates happen during a drag or swipe, making animations smoother and reducing CPU usage.

5. Infinite Scroll / Pagination

- When the user scrolls near the bottom, throttle how often you fetch more data to avoid spamming the API.

6. GPS or Sensor Data

- In apps that get high-frequency location or sensor data, throttle updates to only process data at a set interval.

7. Analytics or Logging

- Throttle event tracking (e.g., mouse movement, touch events, view tracking) to avoid flooding your analytics backend.

▼ Currying Vs. Callback

Currying

- **Definition:**

  Transforming a function that takes multiple arguments into a sequence of functions, each with a single argument.

- **Purpose:**

To create *specialized* functions by “pre-setting” some arguments.

- **Pattern:**

```
const add = a => b => a + b;
const addFive = add(5); // returns a function that adds 5
addFive(2); // 7
```

- **In React:**

  Used to “remember” a value for later (event handlers, HOCs, etc).

Callback Functions

- **Definition:**

  A function passed as an *argument* to another function, which is then called (“called back”) at a later time.

- **Purpose:**

  To let code “call you back” when something happens (like after an API call, button press, timeout, etc).

- **Pattern:**

```
function fetchData(url, callback) {
  // ...fetching data
  callback(response);
}

fetchData('/api', (data) => {
  console.log('Got data:', data);
});
```

- **In React:**

  Event handlers like `onPress`, `onChange`, etc. are callback functions passed to components.

▼ Memoization

What is Memoization?

- **Definition:**

  Memoization is an optimization technique where you store (or “cache”) the results of expensive function calls and **return the cached result** when the same inputs occur again.

- **Why?**

  - **Performance:**

    Reduces lag, improves speed, and prevents wasted renders/calculations—especially on lower-end devices or large lists.

  - **Battery Life:**

    Less computation means less battery drain on mobile.

When Should You Use Memoization?

- When you have **heavy calculations** or **expensive rendering**.
- When passing **callbacks** or **props** to child components that don’t need to change unless specific data changes.
- For **large lists** (e.g., FlatList item rendering).

1️⃣ useMemo Example

Scenario: Expensive Calculation in a List

Suppose you have a large list and need to compute something expensive (like filtering or sorting) **every render**.

```
import React, { useMemo, useState } from 'react';
import { View, Text, Button } from 'react-native';

const numbers = Array.from({ length: 10000 }, (_, i) => i);

const ExpensiveList = () => {
  const [multiplier, setMultiplier] = useState(1);
```

```
  // Expensive computation: only recompute if multiplier changes!
  const multipliedNumbers = useMemo(() => {
    console.log('Expensive calculation running...');
    return numbers.map(num => num * multiplier);
  }, [multiplier]);

  return (
    <View>
      <Button title="Increase Multiplier" onPress={() => setMultiplier(multi
plier + 1)} />
      <Text>First 5 numbers: {multipliedNumbers.slice(0, 5).join(', ')}</Te
xt>
    </View>
  );
};
```

Without `useMemo`**,** the calculation would run **on every render**, even if nothing relevant changed.

With `useMemo`**,** it only runs when `multiplier` changes.

2️⃣ useCallback Example

Scenario: Passing Stable Callbacks to Children

You have a parent component rendering a child with a callback prop.

If the callback is recreated every render, the child may **unnecessarily re-render** (bad for performance), we want to save the reference (WILL EXPLAIN BELOW)

```
import React, { useCallback, useState } from 'react';
import { View, Button, Text } from 'react-native';

const ChildButton = React.memo(({ onAction }) => {
```

```
  console.log('ChildButton rendered');
  return <Button title="Do Something" onPress={onAction} />;
});

const Parent = () => {
  const [count, setCount] = useState(0);

  // Memoize the callback so its reference doesn't change unless count
changes
  const handleAction = useCallback(() => {
    alert('Count is: ' + count);
  }, [count]);

  return (
    <View>
      <ChildButton onAction={handleAction} />
      <Button title="Increase" onPress={() => setCount(count + 1)} />
      <Text>Count: {count}</Text>
    </View>
  );
};
```

- **Without** `useCallback`, `handleAction` would be a new function every render, so `ChildButton` would always re-render.
- **With** `useCallback`, `ChildButton` only re-renders when the relevant data (`count`) changes.

3️⃣ React.memo Example

Scenario: Prevent Unnecessary Re-Renders in Child Components

Let's say you have a heavy child component that doesn't need to update unless its **props** change:

```
import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';

const HeavyChild = React.memo(({ value }) => {
  console.log('HeavyChild rendered');
  // Simulate heavy rendering
  for (let i = 0; i < 100000000; i++) {}
  return <Text>Value: {value}</Text>;
});

const Parent = () => {
  const [value, setValue] = useState(0);
  const [other, setOther] = useState(0);

  return (
    <View>
      <HeavyChild value={value} />
      <Button title="Update Value" onPress={() => setValue(value + 1)} />
      <Button title="Change Something Else" onPress={() => setOther(oth
er + 1)} />
      <Text>Other: {other}</Text>
    </View>
  );
};
```

- **Without** `React.memo`, `HeavyChild` would re-render every time the parent re-renders—even if `value` didn't change.
- **With** `React.memo`, `HeavyChild` **only** re-renders when its `value` prop changes.

4️⃣ useCallback vs useMemo

useCallback

- **Purpose:** Memoizes a **function** (callback) so that it keeps the same reference between renders unless dependencies change.

- **Returns:** The **function itself**.
- **Use case:** When you want to pass a stable callback to a child component (to prevent unnecessary re-renders).

Example:

```
const handleClick = useCallback(() => {
  console.log('Clicked!');
}, []);
```

---

useMemo

- **Purpose:** Memoizes the **result of a calculation** (any value) so it is only recomputed when dependencies change.
- **Returns:** The **computed value** (could be a number, object, array, JSX, etc.).
- **Use case:** When you want to avoid recomputing an expensive calculation unless inputs change.

Example:

```
const doubled = useMemo(() => value * 2, [value]);
```

5️⃣ What is a “Reference” in JavaScript?

- When you create a function (`const fn = () => {}`), JavaScript assigns it a unique “**reference**” (identity in memory).
- If you re-create a function (even with the same code), it’s a **different reference** every time.

```
const fn1 = () => {};
const fn2 = () => {};
```

```
console.log(fn1 === fn2); // false, because they are different reference
s
```

Why Does Reference Matter in React?

- When you pass a **function as a prop** to a child component (like `<ChildButton onAction={handleAction} />`),

  React **checks if the prop's reference changed** to decide if it should re-render the child.

- If the function reference **changes every render**, even if nothing else changed,

  the child component will **always re-render** (even if using `React.memo`).

useCallback Keeps the Reference Stable

- `useCallback` **remembers** your function between renders **unless its dependencies change**.
- If dependencies *don't* change, it gives you the **same function reference** as last time.

Example:

```
const handleAction = useCallback(() => {
  alert('Count is: ' + count);
}, [count]);
```

- If `count` **does not** change, React will keep reusing the same `handleAction` reference.
- If `count` **does** change, it makes a new function (new reference) because the alert should show the new count.

What if I Don't Use useCallback?

```
const handleAction = () => {
  alert('Count is: ' + count);
};
```

- This function is **created fresh every render**, so its reference is **always different**.
- Even if the code looks the same, React sees it as a new prop every time, so `ChildButton` re-renders every time.

Summary Analogy

- Imagine giving someone a new phone number every day (function reference changes)—they'll keep "updating" their contact (child re-renders).
- If you keep the same phone number unless something changes (useCallback), they won't update it unless needed.

▼ Hooks

🟦 What are Hooks?

- **Hooks** are special functions that let you "hook into" React features (like state, lifecycle, context) in **function components**.
- Introduced in React **16.8**—before that, state and lifecycle were only in class components.
- **Why use them?**
  - Make code **simpler** and **more reusable**.
  - Avoid big, complicated class components.
  - Compose logic easily with **custom hooks**.

🟦 Most Common Built-in Hooks

1️⃣ useState

- **What it does:**

  Lets you add and update **state** (data that changes) in function components.

- **Why use it:**

  Track and update things like input values, toggles, counters, etc.

- **How to use:**

```
import React, { useState } from 'react-native'; // or 'react'

function Counter() {
  const [count, setCount] = useState(0); // 0 is the initial state
  return (
    <View>
      <Text>Count: {count}</Text>
      <Button title="Add" onPress={() => setCount(count + 1)} />
    </View>
  );
}
```

  - `count`: Current value
  - `setCount`: Function to update it

2️⃣ useEffect

- **What it does:**

  Lets you run **side effects** (code that isn't just rendering, like fetching data, timers, subscriptions) in your function components.

- **Why use it:**

  Replace lifecycle methods from class components (componentDidMount, componentDidUpdate, componentWillUnmount).

- **How to use:**

```
import React, { useEffect } from 'react-native';

function MyComponent() {
  useEffect(() => {
    // Code here runs after component mounts or updates
    fetchData();

    return () => {
      // Cleanup code here runs when component unmounts or before eff
ect runs again
      cleanupSomething();
    };
  }, []); // [] = run only once (on mount)
}
```

- The **dependency array** (`[]`): Controls when the effect runs.
  - Empty: only once (on mount)
  - With variables: runs when those change

3️⃣ useContext

- **What it does:**

  Lets you **share values** (like theme, user, language) across the component tree without passing props at every level.

- **Why use it:**

  Avoid “prop drilling.” Makes global/shared data easy.

- **How to use:**

```
// 1. Create a context
const ThemeContext = React.createContext('light');

// 2. Use a provider high in your tree
<ThemeContext.Provider value="dark">
```

```
  <App />
</ThemeContext.Provider>

// 3. Access context in any child
function ThemedText() {
  const theme = React.useContext(ThemeContext);
  return <Text style={{ color: theme === 'dark' ? 'white' : 'black' }}>Hell
o</Text>;
}
```

- If you use context, any child can get the value, no matter how deep.

4️⃣ useRef

- **What it does:**

  Lets you keep a **mutable value** (doesn't trigger re-render when changed).

  Also gives you a way to **access a DOM node or component instance**.

- **Why use it:**

  For storing values that don't cause re-renders (e.g., timers, previous values) or accessing components/inputs directly.

- **How to use:**

```
import React, { useRef } from 'react-native';

function Timer() {
  const intervalRef = useRef(null);

  useEffect(() => {
    intervalRef.current = setInterval(() => {
      // do something
    }, 1000);
    return () => clearInterval(intervalRef.current);
  }, []);
```

```
}
```

- `useRef` keeps the same `.current` object between renders.
- In web React: used to directly access DOM nodes (e.g., focus an input).

5️⃣ useMemo & useCallback

- **useMemo**: Memoizes the **result** of a calculation.
- **useCallback**: Memoizes a **function**.
- See previous explanations for details!

6️⃣ useReducer

- **What it does:**

  Lets you manage **complex state logic** (with multiple values, or logic based on action types), it's like **useState** but this one handle more complex states.

  - `useReducer` gives you `[state, dispatch]`, like Redux but local.

- **Why use it:**

  Good for forms, lists, toggles, or anything with multiple related state changes.

- **How to use:**

```
import React, { useReducer } from 'react-native';

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
```

```
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <View>
      <Text>Count: {state.count}</Text>
      <Button title="+" onPress={() => dispatch({ type: 'increment' })} />
      <Button title="-" onPress={() => dispatch({ type: 'decrement' })} />
    </View>
  );
}
```

7️⃣ Custom Hooks

- **What it does:**

  Lets you extract and reuse **logic that uses hooks** across components.

- **Why use it:**

  Avoid code duplication, make code more organized and testable.

- **How to use:**

```
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(c => c + 1);
  const reset = () => setCount(initialValue);
  return { count, increment, reset };
}

// Use your custom hook in any component
```

```
function MyCounterComponent() {
  const { count, increment, reset } = useCounter(5);
  return (
    <View>
      <Text>{count}</Text>
      <Button title="Add" onPress={increment} />
      <Button title="Reset" onPress={reset} />
    </View>
  );
}
```

- Custom hooks **always** start with `use`.
- You can use other hooks inside your custom hook.

🟦 Why are Hooks Important?

- Make function components as powerful as class components—but **simpler** and **more readable**.
- Enable logic reuse without “wrapper hell” (HOCs, render props).
- Are the **standard** way to write new React and React Native code!