🧠 Conceptual Difference

Feature	React Context 🧬	Global Object 🌐
Definition	Built-in React API for sharing state/data across components	Global variables accessible throughout your JS runtime
Scoped To	React component tree only	Whole JavaScript environment (e.g., browser or Node)
Reactivity	✅ React automatically re-renders consumers when context changes	❌ Changing a global variable doesn't trigger re-renders
Usage	For passing props/data deeply without prop drilling	For cross-file values like constants, config, etc.
Examples	Theme, Auth, Language, Cart State	`window.user = ...`, `globalThis.appConfig = ...`
Safe for SSR	✅ Yes	❌ No (shared across requests)
Modular/Encapsulated	✅ Yes	❌ No (Pollutes global scope)

🔍 Analogy

- **React Context** = 📦 A secure locker inside React Mall. Only components with the key (consumer) can open it.
- **Global Object** = 🏬 A public notice board in town square. Anyone can read/write anything, but changes aren't tracked automatically.

💻 Code Example

✅ Using React Context

```
// ThemeContext.js
import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

```
// useTheme.js
export const useTheme = () => useContext(ThemeContext);
```

```
// App.js
import { ThemeProvider, useTheme } from './ThemeContext';

const ThemeToggler = () => {
  const { theme, setTheme } = useTheme();
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Current: {theme} - Toggle Theme
    </button>
  );
};

export default function App() {
  return (
    <ThemeProvider>
      <ThemeToggler />
    </ThemeProvider>
  );
}
```

❌ Using Global Object (Not Recommended in React)

```
// global.js
window.appTheme = "light";

// anywhere.js
function changeTheme() {
  window.appTheme = "dark";
}

// Problem: no React component will re-render when this changes!
```

🔥 Key Differences Summary

Feature	React Context	Global Object
React-aware updates	✅ Yes	❌ No
Safe in concurrent rendering	✅	❌ Risky
SSR-friendly	✅	❌ (shared between requests)
Modular	✅	❌

Feature	React Context	Global Object
Preferred in React	✅	❌ Only for config/constants

🧪 When to Use What?

Scenario	Use Context	Use Global Object
Theme, Auth, User Info	✅	❌
Environment Config (constants like API_URL)	❌	✅
Persistent app-wide state (Cart, Lang)	✅	❌
Debug/test data (one-off vars)	❌	✅ maybe

🏁 Best Practice

> 🧠 **Use React Context for dynamic shared data inside your React app.**
>
> 🚫 Avoid using `window`, `globalThis`, or module-scoped globals for anything reactive.

🤔 Is Context value accessible *only* to children inside the provider?

✅ Short Answer: Yes — only components that are descendants of the `Provider` can access the context value.

Your `MyProvider` component:

```
const MyProvider = ({ children }) => {
  const [value, setValue] = useState("Hello World");

  return (
    <MyContext.Provider value={{ value, setValue }}>
      {children}
    </MyContext.Provider>
  );
};
```

🔍 Let's Understand with a Diagram:

```
<App>
 ├── ❌ ComponentOutside (CAN'T access context)
 └── <MyProvider>
       ├── ✅ ComponentInside (CAN access context via useContext)
       └── ✅ EvenDeeperComponent (also CAN access)
```

💧 Code Example to Demonstrate

```
// ❌ Outside.js
import { useContext } from 'react';
import { MyContext } from './MyContext';

const OutsideComponent = () => {
  const { value } = useContext(MyContext); // ❌ This will NOT work properly
  return <p>Value: {value}</p>;
};
```

```
// ✅ Inside.js
import { useContext } from 'react';
import { MyContext } from './MyContext';

const InsideComponent = () => {
  const { value } = useContext(MyContext); // ✅ Will work fine
  return <p>Value: {value}</p>;
};
```

```
// App.js
<MyProvider>
  <InsideComponent />  {/* ✅ OK */}
</MyProvider>

<OutsideComponent />    {/* ❌ Will get undefined or default */}
```

🛠️ Why This Happens?

- React Context uses the **React Tree** to determine what can access the value.
- Only components **inside the `<Provider>` subtree** will get the value via `useContext()`.

🧠 Pro Tips

❗ **Wrong**	✅ **Right**
Using `useContext()` **outside** of a Provider	Always place consumer components **inside** the Provider
Relying on context's `defaultValue`	Always wrap your app or sections with Provider
Deeply nested context usage in many files	Use `custom hooks` like `useTheme()` for cleaner access

✅ So Yes, To Summarize:

Question	Answer
Can components outside the Provider access context?	❌ No
Who can access the value?	✅ Only descendants (children, grandchildren...)
What if accessed outside?	Will receive the `defaultValue` (if any), or `undefined`

---

withContextGuard()` HOC

✅ 1. `MyContext.js` – Create Your Context

```
import { createContext, useContext } from "react";

// Create context
export const MyContext = createContext(undefined);

// Create a custom hook with guard
export const useMyContext = () => {
  const context = useContext(MyContext);
  if (context === undefined) {
    throw new Error("❗ useMyContext must be used within a <MyProvider>.");
  }
  return context;
};
```

✅ 2. `MyProvider.js` – Create the Provider

```
import React, { useState } from "react";
import { MyContext } from "./MyContext";

export const MyProvider = ({ children }) => {
  const [value, setValue] = useState("🚀 Hello from Context");

  return (
    <MyContext.Provider value={{ value, setValue }}>
      {children}
    </MyContext.Provider>
  );
};
```

✅ 3. `Component.js` – Consume the Context Safely

```
import React from "react";
import { useMyContext } from "./MyContext";

const SafeComponent = () => {
  const { value, setValue } = useMyContext();

  return (
    <div>
      <p>🎉 Context Value: {value}</p>
      <button onClick={() => setValue("🔥 Updated!")}>Change</button>
    </div>
  );
};

export default SafeComponent;
```

❌ 4. Try Using Outside Provider (It Throws Error)

```
import SafeComponent from "./SafeComponent";

const App = () => {
  return (
    <>
      {/* ❌ This will throw: useMyContext must be used within a <MyProvider> */}
      <SafeComponent />
    </>
  );
};
```

✅ 5. Correct Usage in `App.js`

```
import { MyProvider } from "./MyProvider";
import SafeComponent from "./SafeComponent";

const App = () => {
  return (
    <MyProvider>
      {/* ✅ Safe usage inside provider */}
      <SafeComponent />
    </MyProvider>
  );
};
```

🧠 Why This Is Useful

Benefit	Description
🛡 Safety	Prevents silent bugs when devs forget the Provider
💧 Dev Experience	Gives a clear error message on misuse
🧩 Scalability	Works great in large-scale apps with many contexts

🧠 Bonus: Use It in Any Context

You can generalize this pattern to any context:

```
// myHooks/useContextGuard.js
export const createSafeContext = () => {
  const Context = createContext(undefined);
  const useSafeContext = () => {
    const ctx = useContext(Context);
    if (ctx === undefined) {
      throw new Error("useSafeContext must be used within its Provider.");
    }
    return ctx;
  };
  return [Context, useSafeContext];
};
```

Then use it like:

```
const [AuthContext, useAuth] = createSafeContext();
```

🌲 Nested Context Providers in React

> 📦 Multiple independent contexts working together

🧠 What Are Nested Contexts?

React lets you **nest multiple context providers**. This is useful when:

- You want to **modularize global state** (e.g., `UserContext`, `ThemeContext`, `CartContext`)
- Different parts of the app need **access to different contexts**
- You want **clean separation of concerns** ✅

🧾 Step-by-Step Example: User + Theme Context

1. **Create Contexts**

```js
// context/UserContext.js
import { createContext } from "react";
export const UserContext = createContext();

// context/ThemeContext.js
import { createContext } from "react";
export const ThemeContext = createContext();
```

---

2. **AppLayout with Nested Providers**

```js
// AppLayout.js
import React, { useState } from "react";
import { Outlet } from "react-router-dom";
import Header from "./components/Header";
import { UserContext } from "./context/UserContext";
import { ThemeContext } from "./context/ThemeContext";

const AppLayout = () => {
  const [user, setUser] = useState({ name: "Darshan", loggedIn: true });
  const [theme, setTheme] = useState("light");

  return (
    <UserContext.Provider value={{ user, setUser }}>
      <ThemeContext.Provider value={{ theme, setTheme }}>
        <div className={`app ${theme}`}>
          <Header />    {/* ✅ has access to both contexts */}
          <Outlet />    {/* ✅ children can also access both */}
        </div>
      </ThemeContext.Provider>
    </UserContext.Provider>
  );
};

export default AppLayout;
```

✅ Now, **all child components** of `AppLayout` can use **both** `UserContext` and `ThemeContext`.

---

3. **Accessing Nested Contexts in a Component**

```js
// components/Dashboard.js
import React, { useContext } from "react";
import { UserContext } from "../context/UserContext";
import { ThemeContext } from "../context/ThemeContext";

const Dashboard = () => {
  const { user } = useContext(UserContext);
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <div>
      <h2>👋 Welcome, {user.name}</h2>
      <p>Current Theme: {theme}</p>
      <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
        🎨 Toggle Theme
      </button>
    </div>
  );
};

export default Dashboard;
```

🧩 Visualization of the Tree

```
<UserContext.Provider>
  <ThemeContext.Provider>
    <App>
      <Header />
      <Dashboard />  // both contexts available here
    </App>
  </ThemeContext.Provider>
</UserContext.Provider>
```

✅ Benefits of Nested Contexts

Benefit	Why it's Useful
🔗 Separation of Concerns	Each context handles one thing
♻️ Reusability	Reuse `ThemeContext` or `UserContext` elsewhere
⚙️ Scalability	Easily extend app with more contexts

🛑 Gotchas / Best Practices

🚫 Don't nest **too deeply** — it becomes hard to manage ✅ Use **custom hooks** for better abstraction (`useUser()`, `useTheme()`) ✅ Group related providers into a single component (see below 👇)

🔄 Bonus: Combined Provider Component

```
// context/GlobalProvider.js
import { UserContext } from "./UserContext";
import { ThemeContext } from "./ThemeContext";

const GlobalProvider = ({ children }) => {
  const [user, setUser] = useState({ name: "Darshan", loggedIn: true });
  const [theme, setTheme] = useState("light");

  return (
    <UserContext.Provider value={{ user, setUser }}>
      <ThemeContext.Provider value={{ theme, setTheme }}>
        {children}
      </ThemeContext.Provider>
    </UserContext.Provider>
  );
};

export default GlobalProvider;
```

And then wrap your app like this:

```
// index.js or main App
<GlobalProvider>
  <AppLayout />
</GlobalProvider>
```

💬 Final Words

Nested context is the **React way** to manage modular and scalable shared state ✨ Use it when:

- Different concerns need separate state (auth vs theme vs cart)
- You want reusable and maintainable architecture 🛠️