1. Components

- **Definition**: The building blocks of a React application, which are independent, reusable pieces of UI.
- **Types**: Functional components (use hooks) and Class components (use lifecycle methods).
- **Props**: Short for properties, used to pass data from parent to child components.
- **State**: Local state management within a component to handle dynamic data.

2. JSX

- **Definition**: JavaScript XML, a syntax extension that allows writing HTML within JavaScript.
- **Usage**: Makes the code more readable and easier to write, but needs to be compiled by tools like Babel.
- **Rules**: Must return a single parent element, self-closing tags for empty elements, and expressions within `{}`.

3. State Management

- **Local State**: Managed within individual components using `useState` or `this.state` in class components.
- **Global State**: Managed across multiple components, commonly using libraries like Redux, Context API, or MobX.
- **Updating State**: Involves using state update functions like `setState` in class components and hooks like `useState` in functional components.

4. Hooks

- **Introduction**: Introduced in React 16.8, allowing functional components to use state and other React features.
- **Common Hooks**:
  - `useState`: For managing local state.
  - `useEffect`: For side effects like data fetching, subscriptions.
  - `useContext`: For accessing context.
  - `useReducer`: For complex state logic.
- **Custom Hooks**: User-defined hooks to reuse stateful logic.

5. Lifecycle Methods

- **Class Components**: Methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
- **Functional Components**: Managed through `useEffect`, which can mimic lifecycle behavior.

6. Props and PropTypes

- **Props**: Mechanism to pass data and event handlers to child components.
- **PropTypes**: Typechecking feature for props, ensuring the correct type of props are passed to components.

7. React Router

- **Purpose**: Enables routing in a React application, allowing for navigation between different components.
- **Components**:
  - `BrowserRouter` or `HashRouter`: Wrapping component for routing.
  - `Route`: Defines a route in the application.
  - `Link` and `NavLink`: For navigation.
- **Hooks**: `useHistory`, `useLocation`, `useParams` for more control over routing in functional components.

8. Context API

- **Purpose**: Allows for state sharing across components without prop drilling.
- **Components**:
  - `React.createContext()`: Creates a context.
  - `Provider`: Wraps components that need access to the context.
  - `Consumer`: Consumes the context value.
- **Hooks**: `useContext` for consuming context in functional components.

9. Event Handling

- **Syntax**: Similar to handling events in plain HTML but uses camelCase.
- **Binding**: Ensuring `this` refers to the correct context in class components, typically using `.bind()` or arrow functions.

10. Refs

- **Purpose**: Provide a way to access DOM nodes or React elements directly.
- **Usage**:
  - `React.createRef()`: Creating refs in class components.
  - `useRef()`: Creating refs in functional components.
- **Forwarding Refs**: Using `React.forwardRef` to pass refs through components.

11. Higher-Order Components (HOCs)

- **Definition**: Functions that take a component and return a new component with additional props or behavior.
- **Usage**: Commonly used for cross-cutting concerns like logging, caching, authentication.

12. Error Handling

- **Error Boundaries**: Special components that catch JavaScript errors anywhere in their child component tree and display a fallback UI.
- **Methods**: `componentDidCatch` and `getDerivedStateFromError` in class components.
- **Functional Component Handling**: Using `ErrorBoundary` components to wrap around parts of the component tree.