1.)What is the difference between Inheritance and Composition ?  Which paradigm is preferred in React?  Write a brief note about it in your own words.

|  |  |
| --- | --- |
| **Inheritance** | **Composition** |
| In inheritance, we define the class which we are inheriting(super class) and most importantly it cannot be changed at runtime. | Whereas in composition we only define a type which we want to use and which can hold its different implementation also it can change at runtime. Hence, Composition is much more flexible than Inheritance. |
| Here we can only extend one class, in other words more than one class can’t be extended as java do not support multiple inheritance. | Whereas composition allows to use functionality from different class. |
| In inheritance we need parent class in order to test child class. | Composition allows to test the implementation of the classes we are using independent of parent or child class. |
| Inheritance cannot extend final class. | Whereas composition allows code reuse even from final classes. |

**Some common paradigm :**

Paradigms are **not meant to be mutually exclusive**; a single program can feature multiple paradigms!

* **Imperative**: Programming with an explicit sequence of commands that update state.
* **Declarative**: Programming by specifying the result you want, not how to get it.
* **Structured**: Programming with clean, goto-free, nested control structures.
* **Procedural**: Imperative programming with procedure calls.
* **Functional** (Applicative): Programming with function calls that avoid any global state.
* **Function-Level** (Combinator): Programming with no variables at all.
* **Object-Oriented**: Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
  + **Class-based**: Objects get state and behaviour based on membership in a class.
  + **Prototype-based**: Objects get behaviour from a prototype object.
* **Event-Driven**: Programming with emitters and listeners of asynchronous actions.
* **Flow-Driven**: Programming processes communicating with each other over predefined channels.
* **Logic** (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.
* **Constraint**: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
* **Aspect-Oriented**: Programming cross-cutting concerns applied transparently.
* **Reflective**: Programming by manipulating the program elements themselves.
* **Array**: Programming with powerful array operators that usually make loops unnecessary.

**Some Major Paradigms**

### **1.Imperative Programming**

Control flow in **imperative programming** is explicit: commands show howthe computation takes place, step by step. Each step affects the global **state** of the computation.

### **2.Structured Programming**

**Structured programming** is a kind of imperative programming where control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

### **3.Object Oriented Programming**

OOP is based on the sending of **messages** to objects. Objects respond to messages by performing operations, generally called **methods**. Messages can have arguments.

### **4.Declarative Programming**

Control flow in **declarative programming** is implicit: the programmer states only what the result should look like, **not** how to obtain it.

### **5.Functional Programming**

In functional programming, control flow is expressed by combining function calls, rather than by assigning values to variables.

### **6.Logic and Constraint Programming**

**Logic programming** and **constraint programming** are two paradigms in which programs are built by setting up relations that specify **facts** and inference **rules**, and asking whether or not something is true (i.e. specifying a **goal**.) Unification and backtracking to find solutions (i.e.. satisfy goals) takes place automatically.

## Languages and Paradigms

One of the characteristics of a language is its support for particular programming paradigms. For example, Smalltalk has direct support for programming in the object-oriented way, so it might be called an object-oriented language. OCaml, Lisp, Scheme, and JavaScript programs tend to make heavy use of passing functions around so they are called “functional languages” despite having variables and many imperative constructs.

There are two very important observations here:

* Very few languages implement a paradigm 100%. When they do, they are **pure**. It is incredibly rare to have a “pure OOP” language or a “pure functional” language. A lot of languages have a few escapes; for example in OCaml, you will program with functions 90% or more of the time, but if you need state, you can get it. Another example: very few languages implement [OOP the way Alan Kay envisioned it](http://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented).
* A lot of languages will facilitate programming in one or more paradigms. In Scala you can do imperative, object-oriented, and functional programming quite easily. If a language is purposely designed to allow programming in many paradigms is called a **multi-paradigm language**. If a language only accidentally supports multiple paradigms, we don’t have a special word for that.

**2). In what cases do we need to 'lift the state up'?**

 Lifting up the State: As we know, every component in React has its own state. Because of this sometimes data can be redundant and inconsistent. So, by Lifting up the state we make the state of the parent component as a single source of truth and pass the data of the parent in its children.

Time to use Lift up the State: If the data in “parent and children components” or in “cousin components” is Not in Sync.

Example 1: If we have 2 components in our App. A -> B where, A is parent of B. keeping the same data in both Component A and B might cause inconsistency of data.

Example 2: If we have 3 components in our App.

**3).What is a handler function . When do we use it?**

The handler function, by default, when executed is passed the event object (that was created when the event/action you are interested in happened) as an argument. Defining the event as a parameter of your handler function is optional but, sometimes (most times), it is useful for the handler function to know about the event that happened.

All JavaScript event handlers use an anonymous callback function , this isn’t a React -specific thing : JS uses callbacks to allow you to do one thing (pick up a click event for example) then do another thing (fire a handler function) . React follows the exact same pattern , as it’s how the language API’s work.

The event handler takes a function , not the value of a function . When the event occurs,

the function gets called.