1. **Types**
    a. **Named types**
    b. **Class, structures, enumerations, protocols, and any user defined named types.**
    c. **Difference between class method and instance method?**
2. **What is a closure? Why use closure instead of function sometime?**
   Self-contained chunk of code that can be passed around and used in your code. You can assign a closure to a variable. Sometimes the codes are long, so you want to use a completion block, where the program tells you when the long code is done, and you can use closures for that. And sometimes it makes the code more succinct. Like in the case of just sorting, you can omit majority of words. For instance, parameter and the return type and arrow, those can be omitted.

3. **What is a trailing closure?**
   If the last parameter to a function is a closure, swift lets you use a special syntax called trailing closure syntax. Here, rather than passing in your closure as a parameter, you pass it directly after the function inside braces.
   So, call the closure by just calling the function name without the parenthesis.

4. **What is a capture and capture list in closure?**
   Capture is a functionality of a closure, where it can use surrounding context to refer to variable and constant. This is called closing over those variable and constants.
   Capture list is a way to stop memory leak.

5. **What are escaping/ non-escaping closures?**

   Escaping: A closure is said to escape a function when the closure is passed as an argument to the function but is called after the function returns.
    i. It can be helpful to think of an escaping closure as code that outlives the function it was passed into. Think of that function as a jail, and the closure as a bandit who's escaping the jail.

   NON-Escaping

6. **What is a property?**
    a. They are part of another object (var flavor
    b. Values that are associated with **Class, Struct or Enum.**
    c. **Stored Properties:** a property that can contain or store value as an instance of class or struct.
    d. **Computed Properties:** a property used to calculate values

e. **Lazy property:** a property whose initial values isn't calculated until the first time it's used. Keyword **lazy.**
    i.

7. **What is enumeration? (enum playground)**
    a. Defines a group of related values and enables you to work with those values in a type-safe way withing your code.
    b. Keyword – enum

8. **What are structures and classes?**

Structures	**Classes**	**Similarities**
	Classes can inherit from another class, like you inherit from UIViewController to create your own view controller subclass	Both structs and classes can define properties to store values, and they can define functions
	type casting enables you to check and interpret the type of a class instance at runtime.	They can define subscripts to provide access to values with subscript syntax
	Classes can be deinitialized, i.e. you can invoke a deinit() function before the class is destroyed to free up resources	They can define initializers to set up their initial state, with init()
stack	Heap	They can be extended with extension (this is important!
Value type: Type whose value is copied when assigned to Const or Var, or when it's passed to function	Reference type: A reference to same exact instance is used.	They can conform to protocols, for example to support Protocol Oriented Programming

Struct keyword	**Class** keyword	They can work with generics to provide flexible and reusable types .

9. **When to use structures and classes?**
    a. Structs are much safer and bug free.it follows stack so thread has its own stack space, so no other stack can access your value type directly. Hence no race conditions, locks, deadlocks, or any related thread synchronization complexity
    b. Struct also has automatically generated memberwise initializer. So, the recommended Var and Const pops up when you type in the struct name.
    c. Use structs if you want value type
        i. Use value type when Comparing instance data with == makes sense
        ii. You want copies to have independent state
        iii. The data will be used in code across multiple threads
    d. Use class if you want reference type
        i. Comparing instance identity with === makes sense
        ii. You want to create shared, mutable state.
        iii. Even though struct and enum don't support inheritance, they are great for protocol-oriented programming. A subclass inherits all the required and unwanted functionalities from the superclass and is a bad programming practice. Better to use a struct with protocol-oriented programming concept which fixes the above-said issue.
        iv. Class does support Inheritance. Class is a reference type and is stored in the heap part of memory which makes a class comparatively slower than a struct in terms of performance. Unlike a class, a struct is created on the stack. So, it is faster to instantiate (and destroy) a struct than a class. Unless struct is a class member in which case it is allocated in heap, along with everything else.
        v. Value types do not need dynamic memory allocation or reference counting, both of which are expensive operations. At the same time methods on value types are dispatched statically. These create a huge advantage in favor of value types in terms of performance.

10. **What are Methods?**
    a. Methods are functions that are associated with a particular type. A function inside a class.
    b. Its uses are:-

        i. To access and modify instance properties
        ii. To provide functionality related to instances need.

11. What is the SELF property?

a. Self is an implicit property which equals to the instance of itself.
b. Its uses are: -
    i. To differentiate between property names and argument in initializers
    ii. When referencing properties in closure express as required by the compiler

12. What is subscript?

a. Shortcuts for accessing the member elements for a collection, list, or sequence.
b. Usage:
    i. To set and retrieve values by index without needing separate methods.

13. What is inheritance?

a. It's a technique to inherit the methods, properties, and other characteristics from another class.
b. When one class inherit from another, the inheriting class in known as **Subclass**, and the class its inheriting from is **Superclass**
c. Subclass can change characteristics of a superclass by using @**override**
    i. the override makes the swift compiler check the superclass for matching declaration.

14. What is initialization?

a. It's a process of preparing enumeration, instance of class or structure for use. The process involves: -
    i. Setting initial value for each stored property
    ii. Performing any other setup or initialization required before new instance is ready for use.
b. Syntax: like method but doesn't have a name. use keyword **init()**. So the parameter name are important to specify which initializers is being called.

15. What is optional chaining?

a. Is a process for querying and calling properties, methods, and subscripts on a optional that might currently be nil.
    i. If the optional has value, the call will succeed, if it has nil return nil.
b. Optional chaining vs forced unwrapping.
    i. Optional chain will return a value (if it exist) or nil, while forced unwrapping must have a value or else it crashes.

        ii. Optional chain fails gracefully, while forced unwrapping crashes.

16. What is error handling?

- a. Error handling is a process of responding to and recovering from error conditions in your program. Swift provides support for various error handling techniques.
    - i. You can **<u>propagate</u>** the error from a function to the code that calls that function.
        1. Only throwing functions can propagate errors, else you must handle the error inside the function.
    - ii. You can use **<u>do-catch</u>** statement to handle the error.
    - iii. You can use an **<u>optional value</u>** to handle the error
    - iv. You can assert that the error will not occur.

17. What is concurrency?

- a. Concurrency means multiple computations running at the same time.
- b. Why use? Because it runs multiple operations at the same time, which suspends operations that are waiting for an external system and makes it easier to write this code in a memory-safe way.
    - i. Asynchronous code
        1. Asynchronous code is a code that can be stopped and resumed later. The code executes one at a time.
    - ii. Parallel code:
        1. Parallel code is a code that can run multiple pieces of code at the same time.
- c. Task:
    - i. You can use tasks to break up your program into isolated, concurrent pieces. Task are isolated from each other, which is what make it safe from them to run at the same time.
- d. Actors:
    - i. Actors let you safely share information between concurrent code. Actors allows only one Task to access their mutable state at a time.

18. What is typecasting?

- a. Type casting is way to check the type of an instance, or to treat that instance as a different superclass or subclass form somewhere else in its own class hierarchy.
    - i. Implemented using **<u>is</u>** and **<u>as</u>** operators
        1. as for upcasting
        2. is for type checking
        3. as! For force downcasting

4. as? For optional downcasting

19. what are Extensions?
    a. The ability to add new functionality to existing types.
        i. Keyword **extension**
        ii. Access using class object var. func()
    b. **Why?**
        i. if the code is large, it's easier to read when its split
        ii. adds more functionality to already existing class, structure, enumeration or protocol type.
        iii. Add computed instance properties and computed type properties
        iv. Define instance methods and type methods
        v. Define subscripts
        vi. Define and use new nested types
        vii. Make an existing type conform to a protocol (looks like subclassing a protocol and extension)
20. What is protocol?
    a. A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or functionality. It can be adopted by class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of protocol is said to conform to that protocol. Can be extended.
    b. You could say protocols makes the rules and requirements or provides the building block. And class, structures or enumeration can take in these rules and requirement and provide their own functionality to it which is called adopting. And if they are successful in satisfying the rules of a protocol that is called conforming to that protocol.
    c. If two classes conform to a protocol, the class can infer to each other and use their properties and methods without subclassing.
    d. Protocol is super helpful specially in collaborative coding where each has their own job. Each team member just needs a rule (protocol) to follow. Let their code conform to protocol where needed.
    e. Protocol syntax:
        i. Use protocol keyword with the name being capitalized
        ii. Protocol is used to define methods and functions but don't implement them.
    f. Protocol requirements:

i. Property requirements.
    1. Requires just name and type in property. So doesn't matter if its property or computed.
    2. Definition of variable should be same between protocol and the object its being adopted and conformed to
    3. Must be gettable or gettable and settable
        a. If it's both gettable and settable, it can't be fulfilled by a constant stored property or read-only computed property.

ii. Methods requirements
    1. Type methods should be prefixed with @static when defined in protocol.
    2. Methods in protocols cannot have bodies

iii. Mutating Method Requirements.
    1. @mutating keyword
        a. Must match in enum and structure but not class.

iv. Initializer Requirements
    1. An initializer protocol on a conforming class can be designated initializers or convenience initializers but either way use the keyword @required.

g. Protocols as types
    i. Thought it doesn't have a functionality on its own, we can implement protocols as a type. And like most types, it can be:-
    ii. Can be used as a parameter type or return type in a function, initializer, or method.
    iii. Can be used as a type of a constant, variable, or property.
    iv. Can be types of items in an array, dictionary, or other container.

h. Delegation
    i. Delegation is a design pattern that enables a class or structure to hand off (or delegate) some of its responsibilities to an instance of another type.
        1. Delegation can be used to respond to an action or retrieve data from an external source without needing to know its underlying type of that source.

i. Adding protocol conformance with an extension
    i. You can extend existing type to adopt and conform to new protocol, even if you don't have access to the source code. For the existing type. Extension can add new properties, method, and subscripts to an existing

type. Existing instances of a type automatically adopt and conform to a protocol when conformance is added to the instances type in an extension.

j. Uses for delegation: -
    i. Delegation, and the delegation pattern, is a lightweight approach to hand-off tasks and interactions from one class to another.
    ii. You only need a protocol to communicate requirements between classes. This greatly reduces coupling between classes.
    iii. And it separates the responsibilities of the class that generates interactions from the class that responds to these interactions.
    iv. Delegation is more lightweight than subclassing, because you don’t have to inherit a complete class or struct
    v.