DUMPSARENA

App Development with Swift Certified User Exam

Apple App-Development-with-Swift-Certified-User

Version Demo

Total Demo Questions: 6

Total Premium Questions: 60

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

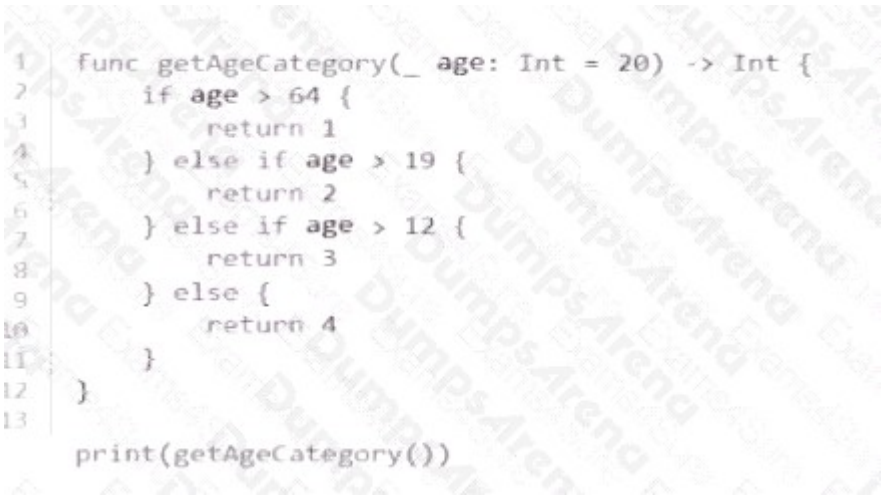
sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Swift Fundamentals	30
Topic 2, User Interface	25
Topic 3, Data	1
Topic 4, App Design	3
Topic 5, Mix Questions	1
Total	60

QUESTION NO: 1 - (SIMULATION)

Review the code snippet.



```
func getAgeCategory(_ age: Int = 20) -> Int {
    if age > 64 {
        return 1
    } else if age > 19 {
        return 2
    } else if age > 12 {
        return 3
    } else {
        return 4
    }
}

print(getAgeCategory())
```

What value does the code output?

Answer the question by typing in the box.

ANSWER: See the explanation for the answer

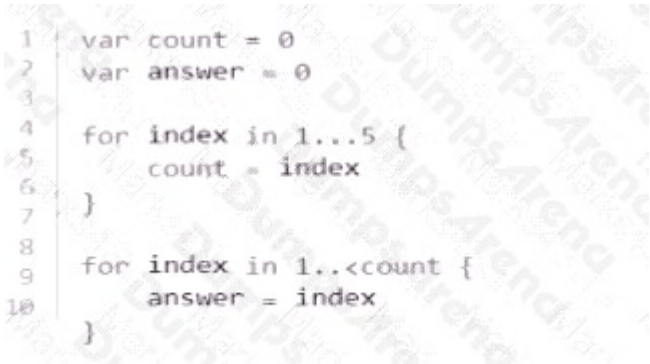
Explanation:

The output is `2`.

`getAgeCategory()` is called without an argument, so its default parameter value is used: `age = 20`. The first condition, `age > 64`, is false. The next condition, `age > 19`, is true, so the function returns `2`.

QUESTION NO: 2 - (SIMULATION)

Review the code snippet.



```
var count = 0
var answer = 0

for index in 1...5 {
    count = index
}

for index in 1..<count {
    answer = index
}
```

What is the value of answer after you run the code?

ANSWER: See the explanation for the answer

Explanation:

Answer: `answer` is `4`.

The first loop uses the closed range `1...5`, so it assigns `count` each value from 1 through 5. When it finishes, `count == 5`.

The second loop uses the half-open range `1..<count`. With `count` equal to 5, its values are `1`, `2`, `3`, and `4`. The final assignment is therefore:

```
answer = 4
```

QUESTION NO: 3

Review the SwiftUI view.

```
struct CounterView: View {
@State private var count = 0

var body: some View {
Button("Add") {
count += 1
}
Text("Count: \(count)")
}
}
```

Why is `count` declared with `@State`?

A. It stores mutable data owned by the view and updates the interface when that data changes.

B. It creates a binding to a value owned by a parent view.

C. It reads the count from the current device environment.

D. It prevents the button action from changing the count.

ANSWER: A

Explanation:

`@State` stores mutable data owned by a view. When the button action changes `count`, SwiftUI preserves the state value and updates dependent parts of the interface, including the `Text` view.

A plain stored property is not the appropriate mechanism for user-driven state that changes during the view's lifetime. `@Binding` would be used when another view owns the source of truth, while `@Environment` reads values supplied by the view environment.

QUESTION NO: 4

Given the function, which two function calls are valid? (Choose 2.)

A. let sum = rightSum(100, num2: 50, and: 20)

B. let sum = rightSum(100, by: 30, and: 20)

C. let sum = rightSum(num1: 100, by: 50, and: 50)

D. let sum = rightSum(100, by: 50)

E. let sum = rightSum(num1: 100, num2: 50)

ANSWER: B D

Explanation:

The valid calls are `let sum = rightSum(100, by: 30, and: 20)` and `let sum = rightSum(100, by: 50)`. The function's first parameter is declared with _, which suppresses its argument label. Consequently, the first integer is

supplied without a label in each valid call. The second parameter is declared with the external argument label `by`, so each call supplies its second value using `by:`. When a call includes the third argument, it uses that parameter's required external label, `and:`, as shown in `rightSum(100, by: 30, and: 20)`.

The call `rightSum(100, by: 50)` is also valid because the third parameter has the default value `25`. Swift permits callers to omit an argument that has a default parameter value; the function then uses the declared default during execution. Therefore, this call evaluates the sum using `100`, `50`, and the default third value of `25`. Swift's function syntax distinguishes parameter names used inside a function from argument labels used at the call site, and an underscore explicitly removes an argument label. See Apple's Swift Functions documentation and the section on function arguments and default parameter values.

QUESTION NO: 5

Review the SwiftUI code.

```
let tags = ["New", "Sale", "Sale"] List(tags, id: \.self) { tag in Text(tag) }
```

Which statement is true about using `id: \.self` in this list?

A. It is unsuitable because the duplicate Sale values do not have unique identities.

B. It fails because String cannot be used as an identifier.

C. It automatically uses each element's array index as its identifier.

D. It requires every String value to conform to Identifiable.

ANSWER: A

Explanation:

Using `id: \.self` uses each string value as that row's identity. Although `String` is hashable and can be used this way, the two `"Sale"` values have the same identity. List elements should have unique, stable identifiers so that SwiftUI can reliably distinguish rows when rendering and updating the interface.

The code does not require every element to conform to `Identifiable` because an explicit identifier key path is supplied. However, replacing the duplicate values with unique model identifiers would be more suitable for data in which repeated tag names are valid.

QUESTION NO: 6

Review the function.

```
func submit(email: String?) {
guard let email = email else {
return
}
print(email)
}
```

Which two statements about the `guard` statement are true? (Choose 2.)

A. The unwrapped `email` constant is available after the `guard` statement.

B. The `else` block must transfer control out of the function when binding fails.

C. The unwrapped `email` constant is available only inside the `else` block.

D. The `guard` statement force unwraps `email`.

E. The function prints `nil` when the argument is absent.

ANSWER: A B

Explanation:

When optional binding succeeds, `email` is unwrapped as a nonoptional `String` and remains available for the rest of the function body. Therefore, the `print(email)` statement can use it without optional handling.

A `guard` statement must leave its enclosing scope when its condition fails. In this function, `return` satisfies that requirement. The `else` block does not continue to the following `print` statement when the argument is `nil`.