Part 2 Test Preparation - Understanding Questions

Closed Book, No Gemini, No References (60%, 45 mins)

How to Answer Short Answer Questions

Question Format: You'll see code snippets and be asked to explain:

- What does this code do?
- Why is this needed?
- What would happen if we remove/change this?

Good Answer Structure:

1. **Define** - What is it?
2. **Purpose** - Why do we use it?
3. **Example** - How does it work?

Keep answers:

- Clear and direct
- 2-4 sentences per part
- Use simple language

Section 1: State Management

mutableStateOf()

What is it?

Creates a value that Compose watches. When the value changes, Compose automatically redraws the UI that uses it.

Why do we need it?

Regular variables don't trigger UI updates. If you change a regular variable, the screen won't update to show the new value.

Example:

```kotlin
var count by remember { mutableStateOf(0) }
```

When `count` changes, any Text showing `count` automatically updates.

Common Question: What happens without mutableStateOf?

Answer: The variable changes but the UI doesn't update. The screen still shows the old value even though the variable has a new value.

`remember { }`

What is it?

Tells Compose to keep a value across recompositions (UI redraws).

Why do we need it?

Compose redraws your UI frequently. Without `remember`, your state resets to its initial value every time.

Example:

```kotlin
var name by remember { mutableStateOf("") }
```

The name survives when Compose redraws the UI.

Common Question: What's the difference between remember and ViewModel?

Answer:

- `remember` - Survives UI redraws but NOT screen rotation
- `ViewModel` - Survives both UI redraws AND screen rotation

`by` keyword

What is it?

A Kotlin property delegate that lets you read and write state values directly.

Why do we use it?

Without `by`, you have to write `.value` every time.

Example:

```kotlin
// Without by
var name = mutableStateOf("")
name.value = "John"  // Must use .value

// With by
var name by mutableStateOf("")
name = "John"  // Direct assignment
```

Common Question: Is `by` required?

Answer: No, but it makes code cleaner and easier to read.

`data class` for UI State

What is it?

A class that groups all screen state into one object.

Why do we use it?

Keeps all related state organized in one place. Makes it easy to update state immutably using `.copy()`.

Example:

```kotlin
data class LoginUiState(
    val username: String = "",
    val password: String = "",
    val isLoading: Boolean = false
)

// Update only username, keep others same
uiState = uiState.copy(username = "john")
```

Common Question: Why use `.copy()` instead of changing directly?

Answer: Creates a new state object instead of modifying the old one. This makes state changes predictable and prevents bugs.

Section 2: ViewModel

What is a ViewModel?

Definition:

A class that holds UI-related data and survives configuration changes like screen rotation.

Why we need it:

When you rotate your phone, Android destroys and recreates the screen. Without ViewModel, all data is lost.

What happens during rotation:

Without ViewModel:

1. Screen destroyed
2. All `remember` state lost
3. Screen recreated
4. Data gone

With ViewModel:

1. Screen destroyed
2. ViewModel survives
3. Screen recreated
4. Reconnects to same ViewModel
5. Data still there

Example:

```kotlin
```

```
class LoginViewModel : ViewModel() {
    var username by mutableStateOf("")
        private set

    fun updateUsername(newName: String) {
        username = newName
    }
}
```

Common Question: How does ViewModel survive rotation?

Answer: Android keeps ViewModel in memory during rotation. When the screen is recreated, it reconnects to the existing ViewModel instead of creating a new one.

---

private set

What is it?

Makes a property read-only from outside the class, but writable inside.

Why we use it:

Enforces one-way data flow. UI can only read state, not change it directly. All changes must go through ViewModel functions.

Example:

kotlin

```
var username by mutableStateOf("")
    private set  // Only ViewModel can change this

fun updateUsername(newName: String) {
    username = newName  // OK - inside ViewModel
}

// In UI:
// viewModel.username = "john"  // ERROR - can't write
// val name = viewModel.username  // OK - can read
```

Common Question: Why restrict who can change state?

Answer: Prevents UI from directly changing state. Forces all changes through ViewModel functions, making data flow predictable and easier to debug.

`viewModel()` function

What is it?

A function that creates or retrieves a ViewModel for a composable.

How it works:

- First time: Creates new ViewModel
- On rotation: Returns existing ViewModel
- When screen closes: Cleans up ViewModel

Example:

```kotlin
@Composable
fun LoginScreen(viewModel: LoginViewModel = viewModel()) {
    // viewModel() ensures same instance across recompositions
}
```

Common Question: What if we just create ViewModel with constructor?

Answer: `val vm = LoginViewModel()` would create a NEW ViewModel every recomposition. All data would be lost constantly. `viewModel()` ensures you get the same instance.

`viewModelScope.launch { }`

What is it?

A coroutine scope tied to the ViewModel's lifecycle, used to run background operations.

Why we need it:

Database operations are `suspend` functions. You can't call them from regular code. You need a coroutine scope.

How it helps:

- Runs code in background (doesn't freeze UI)
- Auto-cancels when ViewModel destroyed (prevents crashes)

Example:

```kotlin
fun saveNote(text: String) {
    viewModelScope.launch {
        // This runs in background
        dao.insert(Note(text = text))
    }
}
```

Common Question: What happens if we don't use viewModelScope?

Answer: You get a compiler error. Suspend functions can only be called from coroutines or other suspend functions.

Section 3: Room Database

Entity

What is it?

A data class that represents a database table.

Key annotations:

- `@Entity` - This is a table
- `@PrimaryKey` - This is the unique ID
- `autoGenerate = true` - Room assigns IDs automatically

Example:

```kotlin
@Entity(tableName = "notes")
data class Note(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val text: String
)
```

Common Question: What does autoGenerate do?

Answer: Room automatically assigns increasing ID numbers (1, 2, 3...). You don't have to specify IDs when inserting.

DAO (Data Access Object)

What is it?

An interface that defines how to interact with the database.

Key annotations:

- `@Dao` - Marks this as a DAO
- `@Query` - Custom SQL query
- `@Insert` - Add to database
- `@Update` - Modify existing row
- `@Delete` - Remove from database

Example:

```kotlin
@Dao
interface NoteDao {
    @Query("SELECT * FROM notes")
    fun getAllNotes(): Flow<List<Note>>

    @Insert
    suspend fun insert(note: Note)

    @Delete
    suspend fun delete(note: Note)
}
```

Common Question: Why does getAllNotes return Flow but insert is suspend?

Answer:

- `Flow` - For queries that update over time. Automatically emits new data when database changes.
- `suspend` - For one-time operations (insert, delete). Runs in background and completes.

Database class

What is it?

The main access point to your database. Brings Entities and DAOs together.

Key parts:

- `@Database` - Marks this as a database
- `entities = [Note::class]` - Lists all tables
- `version = 1` - Database version number
- Abstract functions for DAOs

Example:

```kotlin
@Database(entities = [Note::class], version = 1)
abstract class NoteDatabase : RoomDatabase() {
    abstract fun noteDao(): NoteDao
}
```

Common Question: Why is it abstract?

Answer: Room generates the actual implementation at compile time. You just define the blueprint.

`OnConflictStrategy.REPLACE`

What is it?

Tells Room what to do when you try to insert a row with an existing ID.

Options:

- `REPLACE` - Delete old row, insert new one
- `IGNORE` - Keep old row, skip insert
- `ABORT` - Throw error

Example:

```kotlin
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insert(note: Note)
```

Common Question: When would you use REPLACE vs IGNORE?

Answer:

- `REPLACE` - When new data should overwrite old (like updating a record)
- `IGNORE` - When you only want to insert if it doesn't exist (like adding initial data once)

Section 4: Coroutines and Flow

`suspend` function

What is it?

A function that can pause and resume without blocking the thread.

Why we need it:

Database and network operations are slow. If they run on the main thread, the app freezes. `suspend` functions run on background threads.

Example:

```kotlin
suspend fun saveData() {
    delay(1000)  // Wait 1 second without blocking
    dao.insert(data)
}
```

Common Question: Where can you call suspend functions?

Answer: Only from:

1. Other suspend functions
2. Coroutine scopes (like `viewModelScope.launch { }`)

Coroutines

What are they?

Lightweight threads for running code in the background without blocking the UI.

Why we need them:

To call suspend functions from regular code.

Example:

```kotlin
// Regular function
fun onClick() {
    viewModelScope.launch {  // Start coroutine
        dao.insert(note)  // Now can call suspend function
    }
}
```

Common Question: What's the difference between coroutine and thread?

Answer: Coroutines are much more efficient. Thousands of coroutines can run on just a few threads. Creating thousands of threads would crash the app.

Flow

What is it?

A stream that emits multiple values over time and automatically notifies observers of changes.

Why we use it:

Automatically updates UI when database changes. No manual refresh needed.

How it's different from regular return:

```kotlin
```

```kotlin
// Regular function - one-time value
fun getNotes(): List<Note>
// If database changes, this doesn't know

// Flow - continuous stream
fun getAllNotes(): Flow<List<Note>>
// Automatically emits new list when database changes
```

Example:

```kotlin
@Query("SELECT * FROM notes")
fun getAllNotes(): Flow<List<Note>>
```

Common Question: When does Flow emit new values?

Answer: Whenever the database changes (insert, update, delete), Flow automatically emits the updated list.

`collectAsState()` and `collectAsStateWithLifecycle()`

What they do:

Convert Flow to Compose State so UI automatically updates.

Difference:

- `collectAsState()` - Always collecting
- `collectAsStateWithLifecycle()` - Stops when app in background (saves battery)

Example:

```kotlin
```

```
val notes by viewModel.allNotes.collectAsStateWithLifecycle(
    initialValue = emptyList()
)

LazyColumn {
    items(notes) { note ->
        Text(note.text)
    }
}
```

Common Question: What is initialValue?

Answer: The value to show while waiting for database to load. Usually empty list or null.

Section 5: Navigation

NavController

What is it?

The object that manages navigation state and the back stack.

What it does:

- Tracks current screen
- Maintains history (back stack)
- Provides navigation methods

Key methods:

```
kotlin

navController.navigate("detail")  // Go to detail screen
navController.popBackStack()      // Go back
```

Common Question: What is the back stack?

Answer: A history of screens you've visited. When you press back, it pops the top screen off the stack and shows the previous one.

Example:

```
Start:        [Home]
Navigate:     [Home, Detail]
Press Back:   [Home]
```

NavHost

What is it?

A container that displays the correct screen based on the current route.

How it works:

Links routes (strings) to composable screens.

Example:

```kotlin
NavHost(
    navController = navController,
    startDestination = "home"
) {
    composable("home") {
        HomeScreen()
    }
    composable("detail") {
        DetailScreen()
    }
}
```

Common Question: What is startDestination?

Answer: The first screen to show when app starts.

Routes

What are they?

Unique string identifiers for each screen.

Simple route:

```kotlin
"home"
"settings"
```

Route with arguments:

```kotlin
"detail/{id}"  // {id} is placeholder
```

Example navigation:

```kotlin
navController.navigate("detail/5")
```

This navigates to detail screen with id = 5

Common Question: How does Navigation know which screen to show?

Answer: It matches the route string against patterns defined in NavHost. When it finds a match, it displays that screen.

Navigation Arguments

What are they?

Data passed between screens through the route.

Complete example:

1. Define route with placeholder:

```kotlin
```

```
composable(
    route = "detail/{bookId}",
    arguments = listOf(
        navArgument("bookId") {
            type = NavType.IntType
        }
    )
) { backStackEntry ->
    val bookId = backStackEntry.arguments?.getInt("bookId")
    DetailScreen(bookId)
}
```

2. Navigate with actual value:

```
kotlin

navController.navigate("detail/5")
```

3. What happens:

- Navigation sees "detail/5"
- Matches pattern "detail/{bookId}"
- Extracts 5 as bookId
- Passes it to DetailScreen

Common Question: Why pass ID instead of entire object?

Answer:

- IDs are stable (don't change)
- Objects might be deleted while navigating
- IDs are simple to pass in URL format
- Can look up object from database using ID

`navArgument` and NavType

What is navArgument?

Defines what type of data the argument is.

Available types:

- `NavType.StringType` - Text
- `NavType.IntType` - Integer
- `NavType.BoolType` - true/false
- `NavType.FloatType` - Decimal

Example:

```kotlin
arguments = listOf(
    navArgument("userId") {
        type = NavType.StringType
    }
)
```

Common Question: Why specify the type?

Answer: Navigation needs to know how to convert the string in the URL to the correct data type.

Section 6: UI Components

Column vs LazyColumn

Column:

- Creates ALL children immediately
- Good for: Small, fixed lists (< 20 items)
- Problem: Memory issues with large lists

LazyColumn:

- Creates ONLY visible items
- Good for: Long lists (100s or 1000s)
- Benefit: Efficient memory use, smooth scrolling

Example:

```kotlin
```

```kotlin
// Column - creates 1000 items
Column {
    items.forEach { item ->
        Text(item)
    }
}

// LazyColumn - creates only ~10 visible items
LazyColumn {
    items(items) { item ->
        Text(item)
    }
}
```

Common Question: How does LazyColumn save memory?

Answer: Only creates composables for items visible on screen (plus a few for buffer). As you scroll, it recycles items that move off-screen and reuses them for new items scrolling on.

Modifier

What is it?

Instructions that modify how a composable looks or behaves.

Common modifiers:

```kotlin
Modifier
    .fillMaxWidth()        // Fill horizontal space
    .padding(16.dp)        // Add space around
    .background(Color.Blue) // Background color
    .clickable { }         // Make clickable
    .weight(1f)            // Take proportional space
```

Key point: Order matters!

```kotlin
```

```
// Blue background, THEN padding
Modifier.background(Blue).padding(16.dp)
// Result: Blue background with white space inside

// Padding, THEN blue background
Modifier.padding(16.dp).background(Blue)
// Result: White space around blue background
```

Common Question: Why does order matter?

Answer: Modifiers are applied in sequence. Each modifier wraps the previous result.

Modifier.weight()

What is it?

Used in Row or Column to make a child take proportional space.

How it works:

```kotlin
Row {
    Box(Modifier.weight(1f))  // Takes 1 part
    Box(Modifier.weight(2f))  // Takes 2 parts
}
```

First box gets 1/3 of space, second gets 2/3.

Example:

```kotlin
Row {
    Icon()  // Natural size
    Column(Modifier.weight(1f))  // Takes remaining space
    Icon()  // Natural size
}
```

Common Question: What does weight(1f) do?

Answer: Makes the component take all remaining space after fixed-size components are measured. Pushes other components to the edges.

Scaffold

What is it?

A layout that provides slots for standard screen elements.

Slots:

- `topBar` - Top app bar
- `bottomBar` - Bottom navigation
- `floatingActionButton` - FAB
- `content` - Main content

Example:

```kotlin
Scaffold(
    topBar = { TopAppBar(title = { Text("App") }) }
) { innerPadding ->
    LazyColumn(
        modifier = Modifier.padding(innerPadding)
    ) {
        // Content
    }
}
```

Common Question: Why use innerPadding?

Answer: Without it, content would be drawn behind the top bar and be hidden. innerPadding adds space so content appears below the top bar.

TopAppBar

What is it?

The bar at the top of the screen showing title and actions.

Key parameters:

- `title` - Text to display

- `navigationIcon` - Left icon (usually back button)
- `actions` - Right icons (like settings, menu)

Example:

kotlin

```kotlin
TopAppBar(
    title = { Text("My App") },
    navigationIcon = {
        IconButton(onClick = { navController.popBackStack() }) {
            Icon(Icons.Default.ArrowBack, "Back")
        }
    },
    actions = {
        IconButton(onClick = { /* settings */ }) {
            Icon(Icons.Default.Settings, "Settings")
        }
    }
)
```

Common Question: When should you show a back button?

Answer: Show back button when there's a previous screen to go back to. Hide it on the home screen (starting destination).

Section 7: Overlay Components

DropdownMenu

What is it:

A menu that appears when you tap an icon (usually three dots).

How to control it:

Use Boolean state for open/closed.

Example:

kotlin

```
var expanded by remember { mutableStateOf(false) }

IconButton(onClick = { expanded = true }) {
    Icon(Icons.Default.MoreVert, "Menu")
}

DropdownMenu(
    expanded = expanded,
    onDismissRequest = { expanded = false }
) {
    DropdownMenuItem(
        text = { Text("Settings") },
        onClick = { expanded = false }
    )
}
```

Common Question: What is onDismissRequest?

Answer: Called when user clicks outside the menu or presses back. Should close the menu by setting expanded to false.

AlertDialog

What is it:

A popup that blocks the UI until user responds.

When to use:

Important decisions that need immediate attention (like confirming deletion).

Example:

kotlin

```
var showDialog by remember { mutableStateOf(false) }

if (showDialog) {
  AlertDialog(
    onDismissRequest = { showDialog = false },
    title = { Text("Delete?") },
    text = { Text("Are you sure?") },
    confirmButton = {
      TextButton(onClick = {
        // Delete item
        showDialog = false
      }) {
        Text("Delete")
      }
    },
    dismissButton = {
      TextButton(onClick = { showDialog = false }) {
        Text("Cancel")
      }
    }
  )
}
```

Common Question: Why does AlertDialog block the UI?

Answer: Forces user to make a decision before continuing. Prevents accidental actions by ensuring user confirms important operations.

Snackbar

What is it:

A brief message that appears at the bottom of the screen and auto-dismisses.

Special features:

- Can have action button (like "Undo")
- Manages queue of multiple messages
- Auto-dismisses after timeout

Why special state?

Problem with Boolean: If user clicks "Save" 5 times quickly:

- Boolean can only be true/false
- Shows only one snackbar
- Other 4 are lost

Solution: SnackbarHostState:

- Maintains queue of messages
- Shows them one after another
- Each message has its own data (text, action, duration)

Example:

```kotlin
val snackbarHostState = remember { SnackbarHostState() }
val scope = rememberCoroutineScope()

Scaffold(
    snackbarHost = { SnackbarHost(snackbarHostState) }
) {
    Button(onClick = {
        scope.launch {
            snackbarHostState.showSnackbar(
                message = "Item saved",
                actionLabel = "Undo"
            )
        }
    }) {
        Text("Save")
    }
}
```

Common Question: Why do we need rememberCoroutineScope?

Answer: `showSnackbar()` is a suspend function (needs to wait for animation). Can't call it directly from onClick. Need coroutine scope to launch it.

ModalBottomSheet

What is it:

A sheet that slides up from the bottom of the screen.

Features:

- Dimmed background
- Can be dismissed by swiping down
- Good for showing options or forms

Why special state?

Bottom sheet has complex state:

- Hidden (off screen)
- Half-expanded (partially visible)
- Fully expanded (fully visible)
- Animating between states
- Current pixel offset

Boolean can't track all this.

Example:

kotlin

```kotlin
var showSheet by remember { mutableStateOf(false) }
val sheetState = rememberModalBottomSheetState()
val scope = rememberCoroutineScope()

if (showSheet) {
    ModalBottomSheet(
        onDismissRequest = { showSheet = false },
        sheetState = sheetState
    ) {
        Column {
            Text("Option 1", Modifier.clickable {
                scope.launch {
                    sheetState.hide()
                    showSheet = false
                }
            })
        }
    }
}
```

Common Question: Why do we need both showSheet and sheetState?

Answer:

- `showSheet` - Controls whether sheet exists or not
- `sheetState` - Controls animation and position when it exists

Section 8: Advanced Patterns

Repository Pattern

What is it:

A layer between ViewModel and DAO that provides clean API for data access.

Why use it:

- ViewModel doesn't need to know about DAO
- Easy to swap data sources (database → network)
- Can add caching in one place
- Easier to test

Example:

kotlin

```kotlin
class BookRepository(private val bookDao: BookDao) {
    val allBooks: Flow<List<Book>> = bookDao.getAllBooks()

    suspend fun insert(book: Book) {
        bookDao.insert(book)
    }
}

// ViewModel uses Repository
class ViewModel(private val repository: BookRepository) {
    val books = repository.allBooks
}
```

Common Question: Isn't Repository just extra code?

Answer: For simple apps, yes. But it makes larger apps easier to maintain. You can change data source without

touching ViewModel.

StateFlow vs Flow

Flow:

- Stream of values
- Starts emitting when collected
- No current value before collection

StateFlow:

- Special type of Flow
- Always has a current value
- Can read value immediately without collecting

Converting Flow to StateFlow:

```kotlin
val flowFromDao: Flow<List<Book>> = dao.getAllBooks()

val stateFlow: StateFlow<List<Book>> = flowFromDao.stateIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed(5000),
    initialValue = emptyList()
)
```

Common Question: When to use StateFlow?

Answer: When you need immediate access to current value without collecting. Good for ViewModels that multiple screens might read from.

Unidirectional Data Flow

What is it:

Data flows in one direction: State → UI → Event → State

The pattern:

1. ViewModel holds STATE
2. UI READS state and displays it
3. User interacts (EVENT)
4. UI calls ViewModel function
5. ViewModel UPDATES state
6. Back to step 2 (UI reads new state)

Example:

```kotlin
// ViewModel - holds state
var username by mutableStateOf("")
    private set

fun updateUsername(new: String) {
    username = new  // Only way to update
}

// UI - reads and sends events
TextField(
    value = viewModel.username,  // Reads
    onValueChange = { viewModel.updateUsername(it) }  // Sends event
)
```

Common Question: Why not let UI change state directly?

Answer: Makes data flow predictable. All changes go through ViewModel, making it easy to track and debug. Prevents bugs from UI components interfering with each other.

Section 9: Lifecycle Concepts

Configuration Changes

What are they:

Events that cause Android to recreate the Activity.

Examples:

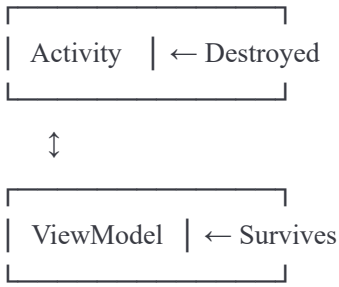
- Screen rotation
- Language change
- Dark mode toggle

- Font size change

What happens:

1. Activity destroyed
2. All local variables lost
3. Activity recreated from scratch

How ViewModel helps:



Common Question: Why does Android do this?

Answer: Screen size changes might need different layouts. Recreating ensures UI adapts properly. Language changes need text to update. Easiest way is to rebuild everything.

Lifecycle-aware Collection

What is it:

Stopping and starting data collection based on app state.

Why we need it:

If app is in background, still collecting data wastes battery and resources.

collectAsStateWithLifecycle vs collectAsState:

```kotlin
// collectAsState - always collecting
val data by flow.collectAsState()

// collectAsStateWithLifecycle - stops when in background
val data by flow.collectAsStateWithLifecycle()
```

What lifecycle means:

- **Active** - App visible, user interacting → collect data
- **Background** - App not visible → stop collecting
- **Foreground again** - App visible again → resume collecting

Common Question: When should you use lifecycle-aware collection?

Answer: Always, in production apps. Saves battery and prevents unnecessary work. Only use basic collectAsState for simple demos or testing.

Section 10: Common Question Patterns

Trace Data Flow Questions

Example: "Trace what happens when user clicks Save button"

How to answer:

1. User action triggers onClick
2. onClick calls ViewModel function
3. ViewModel launches coroutine
4. Coroutine calls DAO insert
5. Database updates
6. Flow emits new data
7. UI collects new data
8. Compose recomposes with new data

"Why do we need X?" Questions

Template answer:

1. **Problem without it:** What breaks or is difficult?
2. **Solution it provides:** How does it fix the problem?
3. **Example:** Brief scenario showing the benefit

Example: "Why do we need ViewModel?" Answer: "Without ViewModel, rotating the phone loses all data because the screen is destroyed. ViewModel survives rotation by being stored outside the screen lifecycle. This means users don't lose their work when rotating the device."

"What would happen if..." Questions

Example: "What would happen if you don't use viewModelScope?"

How to answer:

1. **Immediate effect:** Compiler error / Runtime crash / Wrong behavior
2. **Why:** The technical reason
3. **Correct approach:** What you should do instead

Answer: "You'd get a compiler error because suspend functions can only be called from coroutines or other suspend functions. The DAO insert function is suspend, so it needs to run in a coroutine. You must use viewModelScope.launch { } to create a coroutine scope."

"Explain the difference between..." Questions

Template: Create a comparison table in your mind:

Aspect	Option A	Option B
Purpose	...	...
Use case	...	...
Key difference	...	...

Example: "Difference between Column and LazyColumn?"

Answer: "Column creates all items immediately and is good for small fixed lists. LazyColumn only creates visible items and is good for long lists. The key difference is performance - Column uses lots of memory with many items, while LazyColumn stays efficient regardless of list size."

Final Tips for Part 2

Do's:

✅ Read the question carefully - answer what's asked
✅ Use simple, clear language
✅ Give brief code examples if helpful
✅ Explain the "why" not just the "what"
✅ Keep answers 2-4 sentences per part

Don'ts:

❌ Don't ramble or write paragraphs
❌ Don't copy code without explaining it
❌ Don't use vague terms like "it helps" without saying how
❌ Don't skip the purpose/reason
❌ Don't assume the grader knows what you mean

Time Management:

- 45 minutes for 6 questions
- ~7 minutes per question
- ~2 minutes per part (most questions have 2-3 parts)
- Leave 5 minutes to review

If You're Stuck:

1. Write what you DO know
2. Give an example even if you can't define perfectly
3. Explain in your own words
4. Move on and come back if time permits

Good luck! 🎯 Remember: Clear, simple explanations are better than perfect technical jargon.