Washtenaw Community College
ANN ARBOR, MICHIGAN

CPS251

Android Development

by Scott Shaper

Kotlin Basics

Kotlin gives you powerful tools to store and manage data in your programs. Think of variables as labeled containers that hold different kinds of information - numbers, text, true/false values, and more complex data. Unlike Java, Kotlin was designed with safety and conciseness in mind, helping you avoid common errors like null pointer exceptions while writing less code.

Quick Reference Table

Concept	Description	Common Use
val	Immutable variable (read-only)	For values that shouldn't change
var	Mutable variable (can be changed)	For values that need to be updated
Type?	Nullable type that can hold null values	For optional or potentially missing data
?. operator	Safe call operator	Safely access properties of nullable variables
?: operator	Elvis operator	Provide default values for null cases

Primitive Data Types:

NOTE: Kotlin doesn't have primitive types in the same way as Java does. However, it provides a set of classes that represent primitive data types under the hood. These classes are optimized for performance and behave like primitives, but they are still objects.

When to Use

- Use Int, Long for whole numbers depending on size needed
- Use Double for decimal numbers (Float for memory-constrained cases)
- Use Boolean for true/false conditions
- Use Char for single characters

Type	What It Represents	Example
Byte	8-bit signed integer (-128 to 127)	val smallNumber: Byte = 100
Short	16-bit signed integer (-32,768 to 32,767)	val mediumNumber: Short = 2000
Int	32-bit signed integer (about ±2 billion)	val standardNumber = 1000000
Long	64-bit signed integer (very large range)	val bigNumber = 3000000000L
Float	32-bit floating point number	val decimalNumber = 3.14f
Double	64-bit floating point number	val preciseDecimal = 3.14159265359
Char	16-bit Unicode character	val letter = 'A'
Boolean	true or false value	val isActive = true

Object Data Types:

When to Use

- Use String for text of any length
- Use Array for fixed-size collections
- Use List, Set, Map for flexible collections

Type	What It Represents	Example
String	Text sequence	val name = "Alex"
Array	Fixed-size collection	val numbers = arrayOf(1, 2, 3)
List	Ordered collection	val items = listOf("apple", "banana")
Set	Unique elements collection	val uniqueItems = setOf(1, 2, 3)
Map	Key-value pairs	val ages = mapOf("Alice" to 29)

Variables: Immutable and Mutable

When to Use

- Use val by default (prefer immutability)
- Use var only when a value needs to change

Type	What It Does	When to Use
val	Defines an immutable (read-only) variable	For values that should not change after initialization
var	Defines a mutable variable that can be changed	For values that need to be updated

Practical Examples

Copy Code

```
// ==========================================
// VARIABLE DECLARATION EXAMPLES
// ==========================================

// Example 1: Immutable variables (val)
```

```
// These variables cannot be changed after they're created
val immutableVal = "I cannot be changed"  // Immutable
val pi = 3.14159                          // Mathematical constant
val maxRetries = 3                        // Configuration value
val companyName = "TechCorp"              // Company name

// Example 2: Mutable variables (var)
// These variables can be updated as needed
var mutableVar = "I can be changed"       // Mutable
var currentScore = 0                      // Game score that changes
var temperature = 72.5                    // Temperature that fluctuates
var isLoggedIn = false                    // Login status that changes

// Example 3: Changing mutable variables
println("=== MUTABLE VARIABLE CHANGES ===")
println("Initial value: $mutableVar")
mutableVar = "See, I changed!"            // This works - updating the value
println("After change: $mutableVar")

currentScore = 100                        // Update score
println("Score updated to: $currentScore")

temperature = 85.2                        // Update temperature
println("Temperature changed to: $temperature")

isLoggedIn = true                         // Update login status
println("Login status: $isLoggedIn")

// Example 4: Attempting to change immutable variables (will cause errors)
// The following lines would cause compile errors:
// immutableVal = "Trying to change"      // Error: Val cannot be reassigned
// pi = 3.14                              // Error: Val cannot be reassigned
// maxRetries = 5                         // Error: Val cannot be reassigned
```

```
// Example 5: Type annotation examples
// You can explicitly specify the type, or let Kotlin infer it
val explicitInt: Int = 123              // Type annotation
val inferredInt = 123                   // Type inference (Kotlin knows it's Int)
val explicitString: String = "Hello"    // Type annotation
val inferredString = "World"            // Type inference (Kotlin knows it's String)

// Example 6: Real-world variable usage
println("\n=== REAL-WORLD VARIABLE USAGE ===")

// User profile information (immutable - shouldn't change)
val userId = "user_12345"
val dateOfBirth = "1990-05-15"
val email = "user@example.com"

// User session information (mutable - changes during use)
var loginAttempts = 0
var lastLoginTime = "2024-01-01 00:00:00"
var isOnline = false

// Game state variables (mutable - change during gameplay)
var playerHealth = 100
var playerLevel = 1
var experiencePoints = 0
var goldCoins = 50

println("User ID: $userId")
println("Login attempts: $loginAttempts")
println("Player health: $playerHealth")

// Update game state
playerHealth -= 20                  // Player took damage
experiencePoints += 150             // Player gained experience
goldCoins += 25                     // Player found treasure
```

```
println("After updates:")
println("Player health: $playerHealth")
println("Experience points: $experiencePoints")
println("Gold coins: $goldCoins")
```

Nullability

When to Use

- Use non-nullable types by default
- Use nullable types only when a value might legitimately be absent
- Use safe call operator when working with nullable values

Kotlin's type system helps prevent null pointer exceptions by making "nullability" explicit in the type system. By default, regular types cannot hold null:

Feature	What It Does	Example
Non-nullable type	Cannot hold null values	`val name: String = "Alex"`
Nullable type	Can hold null values	`val name: String? = null`
Safe call operator	Safely calls methods on nullable types	`val length = name?.length`
Elvis operator	Provides default value for null cases	`val display = name ?: "Unknown"`
Not-null assertion	Forces treating as non-null (unsafe)	`val length = name!!.length`

Practical Examples

Copy Code

```
// ==========================================
// NULLABILITY EXAMPLES
// ==========================================

// Example 1: Non-nullable vs nullable types
println("=== NULLABILITY COMPARISON ===")

// Non-nullable String - cannot be null
val nonNullableName: String = "John"
// val nonNullableName2: String = null  // This would cause a compile error!

// Nullable String - can be null
val nullableName: String? = null
val nullableName2: String? = "Jane"
val nullableName3: String? = "Bob"

println("Non-nullable name: $nonNullableName")
println("Nullable names: $nullableName, $nullableName2, $nullableName3")

// Example 2: Safe call operator (?.)
// This safely accesses properties without causing crashes
println("\n=== SAFE CALL OPERATOR ===")

val names = listOf(nullableName, nullableName2, nullableName3, "Alice")

for (name in names) {
    // Safe call - won't crash if name is null
    val length = name?.length
    println("Name: '$name', Length: $length")

    // You can also chain safe calls
    val uppercase = name?.uppercase()
    println("  Uppercase: $uppercase")
```

```
}

// Example 3: Elvis operator (?:)
// This provides default values when dealing with null
println("\n=== ELVIS OPERATOR ===")

val userInput1: String? = null
val userInput2: String? = "Hello World"
val userInput3: String? = ""

// Provide default values for null cases
val displayName1 = userInput1 ?: "Guest"
val displayName2 = userInput2 ?: "Guest"
val displayName3 = userInput3 ?: "Guest"  // Empty string is not null, so no default

println("User 1: '$userInput1' -> Display: '$displayName1'")
println("User 2: '$userInput2' -> Display: '$displayName2'")
println("User 3: '$userInput3' -> Display: '$displayName3'")

// Example 4: Safe conditional with if check
// This is another way to handle null values safely
println("\n=== SAFE CONDITIONAL CHECKS ===")

val potentialName: String? = "Alice"

if (potentialName != null) {
    // Within this block, Kotlin knows potentialName is not null
    val nameLength = potentialName.length  // Smart cast - no need for safe call
    val uppercaseName = potentialName.uppercase()
    println("Name: $potentialName, Length: $nameLength, Uppercase: $uppercaseName")
} else {
    println("No name provided")
}
```

```
// Example 5: Safe usage with let function
// This is a clean way to work with nullable values
println("\n=== SAFE USAGE WITH LET ===")

val nullableEmail: String? = "user@example.com"

nullableEmail?.let {
    // This code only runs if nullableEmail is not null
    // 'it' refers to the non-null email value
    println("Processing email: $it")
    println("Email domain: ${it.substringAfter('@')}")
    println("Email length: ${it.length}")
} ?: run {
    // This runs if nullableEmail is null
    println("No email provided")
}

// Example 6: Not-null assertion (!!) - Use with caution!
// This forces Kotlin to treat a nullable value as non-null
println("\n=== NOT-NULL ASSERTION (DANGEROUS) ===")

val definitelyNotNull: String? = "Safe to use"
val mightBeNull: String? = null

try {
    // This is safe because we know the value is not null
    val length1 = definitelyNotNull!!.length
    println("Safe assertion: $length1")

    // This will crash with NullPointerException!
    val length2 = mightBeNull!!.length
    println("This will never print: $length2")
} catch (e: NullPointerException) {
    println("Caught NullPointerException: ${e.message}")
```

```
}

// Example 7: Real-world nullability scenarios
println("\n=== REAL-WORLD NULLABILITY SCENARIOS ===")

// Simulate user input that might be missing
val userAge: String? = "25"
val userPhone: String? = null
val userAddress: String? = ""

// Process user information safely
val age = userAge?.toIntOrNull() ?: 0
val phone = userPhone ?: "No phone provided"
val address = if (userAddress.isNullOrEmpty()) "No address provided" else userAddress

println("User Profile:")
println("  Age: $age")
println("  Phone: $phone")
println("  Address: $address")

// Example 8: Working with collections that might contain null
println("\n=== NULLABLE COLLECTIONS ===")

val mixedList: List = listOf("Alice", null, "Bob", null, "Charlie")

// Filter out null values
val nonNullNames = mixedList.filterNotNull()
println("All names (including nulls): $mixedList")
println("Non-null names only: $nonNullNames")

// Count null vs non-null values
val nullCount = mixedList.count { it == null }
```

```
val nonNullCount = mixedList.count { it != null }
println("Null values: $nullCount, Non-null values: $nonNullCount")
```

Let Function

When to Use

- When you want to perform operations only if a value is not null
- To avoid repeating a variable name multiple times
- To limit the scope of variables

Pattern	What It Does	When to Use
`value?.let { }`	Executes block only if value is not null	For null-safe operations on optional values
`value?.let { } ?: defaultValue`	Executes block or returns default if null	For transforming values with a fallback

Practical Examples

Copy Code

```
// ==========================================
// LET FUNCTION EXAMPLES
// ==========================================

// Example 1: Basic usage with non-null value
// The let function executes the block and returns the result
println("=== BASIC LET USAGE ===")
val name = "Alice"
val result = name.let {
    println("Processing name: $it")
    it.length  // Returns the length of 'name'
}
println("Result: $result")
```

```
// Example 2: Null-safe usage with let
// This is the most common use case for let
println("\n=== NULL-SAFE LET USAGE ===")
val nullableName: String? = "Bob"
val nameLength = nullableName?.let {
    println("Name is: $it")
    it.length  // This block only executes if nullableName is not null
} ?: 0
println("Name length: $nameLength")

// Example 3: Let with null value
// When the value is null, the let block is skipped
println("\n=== LET WITH NULL VALUE ===")
val emptyName: String? = null
val length = emptyName?.let {
    it.length  // This block is skipped if emptyName is null
} ?: 0
println("Length of null name: $length")

// Example 4: Let for data transformation
// Let is great for transforming data safely
println("\n=== LET FOR DATA TRANSFORMATION ===")
val userInput: String? = "  hello world  "

val processedInput = userInput?.let {
    it.trim()           // Remove whitespace
        .lowercase()    // Convert to lowercase
        .replace(" ", "_") // Replace spaces with underscores
} ?: "default_value"

println("Original input: '$userInput'")
println("Processed input: '$processedInput'")
```

```
// Example 5: Let with multiple operations
// You can perform complex operations within the let block
println("\n=== LET WITH MULTIPLE OPERATIONS ===")
val email: String? = "user@example.com"

val userInfo = email?.let {
    val username = it.substringBefore('@')
    val domain = it.substringAfter('@')
    val isValid = it.contains('@') && it.contains('.')

    mapOf(
        "email" to it,
        "username" to username,
        "domain" to domain,
        "isValid" to isValid
    )
} ?: mapOf("error" to "No email provided")

println("User info: $userInfo")

// Example 6: Let for object initialization
// Let can be used to safely initialize objects
println("\n=== LET FOR OBJECT INITIALIZATION ===")
val configString: String? = "name=John,age=25,city=New York"

val config = configString?.let {
    val pairs = it.split(",")
    val configMap = mutableMapOf()

    for (pair in pairs) {
        val (key, value) = pair.split("=")
        configMap[key.trim()] = value.trim()
    }
```

```
    configMap
} ?: emptyMap()

println("Configuration: $config")

// Example 7: Let with chaining
// You can chain let calls for complex operations
println("\n=== LET WITH CHAINING ===")
val numberString: String? = "42"

val result2 = numberString?.let { str ->
    str.toIntOrNull()?.let { num ->
        if (num > 0) {
            num * 2
        } else {
            num * -1
        }
    }
} ?: 0

println("Original string: '$numberString'")
println("Processed result: $result2")

// Example 8: Real-world let usage scenarios
println("\n=== REAL-WORLD LET SCENARIOS ===")

// Scenario 1: Processing user input
val userInput2: String? = "  123.45  "
val processedNumber = userInput2?.let {
    it.trim().toDoubleOrNull()
}?.let { num ->
    if (num > 0) num else 0.0
} ?: 0.0
```

```
println("User input: '$userInput2'")
println("Processed number: $processedNumber")

// Scenario 2: Database query result
val queryResult: String? = "user_id=123,name=Alice,email=alice@example.com"
val user = queryResult?.let {
    val parts = it.split(",")
    val userMap = parts.associate { part ->
        val (key, value) = part.split("=")
        key to value
    }
    userMap
} ?: emptyMap()

println("Query result: '$queryResult'")
println("Parsed user: $user")

// Scenario 3: API response processing
val apiResponse: String? = """{"status": "success", "data": {"name": "Bob", "age": 30}}"""
val userName = apiResponse?.let {
    // In a real app, you'd use a JSON parser
    if (it.contains("\"name\"")) {
        val nameStart = it.indexOf("\"name\": \"") + 9
        val nameEnd = it.indexOf("\"", nameStart)
        it.substring(nameStart, nameEnd)
    } else {
        null
    }
} ?: "Unknown User"

println("API response: '$apiResponse'")
println("Extracted name: $userName")
```

Type Casting

When to Use

- When working with heterogeneous collections
- When dealing with inheritance hierarchies
- When interacting with less type-safe Java code

Operator	What It Does	When to Use
`is`	Checks if an object is of a specific type	Before performing operations specific to a type
`as`	Unsafe cast (throws exception if fails)	When you're certain of the type
`as?`	Safe cast (returns null if fails)	When the cast might fail

Practical Examples

Copy Code

```
// ========================================
// TYPE CASTING EXAMPLES
// ========================================

// Example 1: Type checking with 'is' operator
// This is the safest way to check types
println("=== TYPE CHECKING WITH 'IS' ===")
val value: Any = "Hello World"

if (value is String) {
    // Smart cast: value is treated as String inside this block
    println("Value is a String: '$value'")
    println("Length: ${value.length}")
    println("Uppercase: ${value.uppercase()}")
```

```
    // You can also use string-specific methods
    val words = value.split(" ")
    println("Number of words: ${words.size}")
} else {
    println("Value is not a String")
}

// Example 2: Type checking with different types
// You can check for multiple types
println("\n=== CHECKING MULTIPLE TYPES ===")
val mixedList: List = listOf("Hello", 42, 3.14, true, 'A')

for (item in mixedList) {
    when (item) {
        is String -> {
            println("String: '$item' (length: ${item.length})")
        }
        is Int -> {
            println("Integer: $item (doubled: ${item * 2})")
        }
        is Double -> {
            println("Double: $item (rounded: ${item.toInt()})")
        }
        is Boolean -> {
            println("Boolean: $item (opposite: ${!item})")
        }
        is Char -> {
            println("Character: '$item' (code: ${item.code})")
        }
        else -> {
            println("Unknown type: ${item::class.simpleName}")
        }
    }
```

```
}

// Example 3: Unsafe cast with 'as' operator
// This throws an exception if the cast fails
println("\n=== UNSAFE CAST WITH 'AS' ===")

val obj: Any = "I'm a string"
try {
    val str: String = obj as String  // Works because obj is actually a String
    println("Successfully cast to String: '$str'")

    // This would throw ClassCastException
    val willCrash: Int = obj as Int
    println("This will never print: $willCrash")
} catch (e: ClassCastException) {
    println("Cast failed: ${e.message}")
}

// Example 4: Safe cast with 'as?' operator
// This returns null if the cast fails instead of throwing an exception
println("\n=== SAFE CAST WITH 'AS?' ===")

val anyValue: Any = 123
val safeString: String? = anyValue as? String  // Will be null since anyValue is not a String
val safeInt: Int? = anyValue as? Int           // Will be 123

println("Safe cast to String: $safeString")
println("Safe cast to Int: $safeInt")

// Example 5: Combining safe cast with Elvis operator
// This provides a default value when the cast fails
println("\n=== SAFE CAST WITH ELVIS OPERATOR ===")

val mixedData: List = listOf("Hello", 42, 3.14, "World", 100)
```

```
for (item in mixedData) {
    val stringValue = (item as? String) ?: "Not a string"
    val intValue = (item as? Int) ?: 0
    val doubleValue = (item as? Double) ?: 0.0

    println("Item: $item")
    println("  As String: '$stringValue'")
    println("  As Int: $intValue")
    println("  As Double: $doubleValue")
    println()
}

// Example 6: Type casting in inheritance scenarios
// This is common when working with class hierarchies
println("\n=== TYPE CASTING WITH INHERITANCE ===")

// Simulate a class hierarchy
open class Animal(val name: String)
class Dog(name: String, val breed: String) : Animal(name)
class Cat(name: String, val color: String) : Animal(name)

val animals: List = listOf(
    Dog("Buddy", "Golden Retriever"),
    Cat("Whiskers", "Orange"),
    Dog("Max", "Labrador"),
    Cat("Shadow", "Black")
)

for (animal in animals) {
    when (animal) {
        is Dog -> {
            println("Dog: ${animal.name} (Breed: ${animal.breed})")
            // You can access Dog-specific properties
```

```
        }
        is Cat -> {
            println("Cat: ${animal.name} (Color: ${animal.color})")
            // You can access Cat-specific properties
        }
        else -> {
            println("Unknown animal: ${animal.name}")
        }
    }
}

// Example 7: Real-world type casting scenarios
println("\n=== REAL-WORLD TYPE CASTING SCENARIOS ===")

// Scenario 1: Processing API responses
val apiResponse: Any = mapOf(
    "status" to "success",
    "data" to mapOf(
        "users" to listOf(
            mapOf("id" to 1, "name" to "Alice"),
            mapOf("id" to 2, "name" to "Bob")
        )
    )
)

// Safely extract user data
val users = (apiResponse as? Map<*, *>)?.let { response ->
    (response["data"] as? Map<*, *>)?.let { data ->
        (data["users"] as? List<*>)?.let { userList ->
            userList.mapNotNull { user ->
                (user as? Map<*, *>)?.let { userMap ->
                    val id = userMap["id"] as? Int
                    val name = userMap["name"] as? String
                    if (id != null && name != null) {
```

```
                    "User $id: $name"
                } else null
            }
        }
    }
} ?: emptyList()

println("API Response: $apiResponse")
println("Extracted users: $users")

// Scenario 2: Database result processing
val dbResult: Any = listOf(
    mapOf("id" to 1, "name" to "Product A", "price" to 29.99),
    mapOf("id" to 2, "name" to "Product B", "price" to 49.99),
    mapOf("id" to 3, "name" to "Product C", "price" to 19.99)
)

val products = (dbResult as? List<*>)?.mapNotNull { row ->
    (row as? Map<*, *>)?.let { product ->
        val id = product["id"] as? Int
        val name = product["name"] as? String
        val price = product["price"] as? Double

        if (id != null && name != null && price != null) {
            "ID: $id, Name: $name, Price: $${String.format("%.2f", price)}"
        } else null
    }
} ?: emptyList()

println("Database Result: $dbResult")
println("Processed products: $products")
```

String Formatting

When to Use

- When precise control over number formatting is needed
- When working with tabular data presentation
- When creating complex string patterns

Format	What It Does	Example
`%s`	String placeholder	`String.format("Hello, %s", name)`
`%d`	Integer placeholder	`String.format("Count: %d", count)`
`%.2f`	Float with 2 decimal places	`String.format("Price: $%.2f", price)`
`${}`	String template (Kotlin preferred)	`"Hello, ${name.capitalize()}"`

Practical Examples

Copy Code

```
// ========================================
// STRING FORMATTING EXAMPLES
// ========================================

// Example 1: Simple string templates (Kotlin's preferred way)
// This is the most readable and flexible approach
println("=== SIMPLE STRING TEMPLATES ===")
val name = "John"
val age = 25
val simple = "Hello, $name! You are $age years old."
println(simple)
```

```
// Example 2: Using expressions in templates
// You can put any valid Kotlin expression inside ${}
println("\n=== EXPRESSIONS IN STRING TEMPLATES ===")
val score = 85
val message = "${name.uppercase()} scored $score points, which is ${if (score >= 80)
"excellent" else "good"}!"
println(message)

// Example 3: Complex expressions in templates
// You can perform calculations and method calls
println("\n=== COMPLEX EXPRESSIONS IN TEMPLATES ===")
val price = 29.99
val quantity = 3
val total = price * quantity
val discount = if (quantity >= 5) 0.10 else 0.0
val finalTotal = total * (1 - discount)

val receipt = """
Receipt for $name:
  Item price: $${String.format("%.2f", price)}
  Quantity: $quantity
  Subtotal: $${String.format("%.2f", total)}
  Discount: ${(discount * 100).toInt()}%
  Final total: $${String.format("%.2f", finalTotal)}
""".trimIndent()

println(receipt)

// Example 4: String.format for precise formatting
// Use this when you need specific formatting control
println("\n=== STRING.FORMAT EXAMPLES ===")
val productName = "Laptop"
val productPrice = 999.99
```

```
val productQuantity = 2

val formatted = String.format("Product: %s, Price: $%.2f, Quantity: %d",
                              productName, productPrice, productQuantity)
println(formatted)

// Example 5: Multiple values in one format string
// This is useful for creating structured output
println("\n=== MULTIPLE VALUES IN FORMAT STRING ===")
val customerName = "Alice Smith"
val orderNumber = "ORD-2024-001"
val orderDate = "2024-01-15"
val orderTotal = 149.99

val orderSummary = String.format("""
Order Summary:
  Customer: %s
  Order #: %s
  Date: %s
  Total: $%.2f
""".trimIndent(), customerName, orderNumber, orderDate, orderTotal)

println(orderSummary)

// Example 6: Number formatting with different precision
// Control decimal places and number formatting
println("\n=== NUMBER FORMATTING ===")
val pi = 3.14159265359
val largeNumber = 1234567.89
val percentage = 0.1234

println("Pi to 2 decimal places: ${String.format("%.2f", pi)}")
println("Pi to 4 decimal places: ${String.format("%.4f", pi)}")
println("Large number with commas: ${String.format("%,.2f", largeNumber)}")
```

```
println("Percentage: ${String.format("%.2f%%", percentage * 100)}")

// Example 7: Alignment and spacing
// Control how text is positioned
println("\n=== TEXT ALIGNMENT AND SPACING ===")
val items = listOf("Apple", "Banana", "Cherry", "Date")
val prices = listOf(1.99, 0.99, 3.99, 2.49)

println("Product List:")
for (i in items.indices) {
    val formattedLine = String.format("%-10s $%6.2f", items[i], prices[i])
    println(formattedLine)
}

// Example 8: Real-world formatting scenarios
println("\n=== REAL-WORLD FORMATTING SCENARIOS ===")

// Scenario 1: Financial report
val companyName = "TechCorp Inc."
val revenue = 1250000.50
val expenses = 875000.25
val profit = revenue - expenses
val profitMargin = (profit / revenue) * 100

val financialReport = String.format("""
FINANCIAL REPORT - %s
==========================================
Revenue:     $%,.2f
Expenses:    $%,.2f
Profit:      $%,.2f
Margin:      %.2f%%
""".trimIndent(), companyName, revenue, expenses, profit, profitMargin)

println(financialReport)
```

```
// Scenario 2: User profile display
val userName = "john_doe"
val userEmail = "john.doe@example.com"
val userAge = 28
val userScore = 95.5
val isActive = true

val userProfile = """
USER PROFILE
============
Username:    $userName
Email:       $userEmail
Age:         $userAge
Score:       ${String.format("%.1f", userScore)}%
Status:      ${if (isActive) "Active" else "Inactive"}
""".trimIndent()

println(userProfile)

// Scenario 3: Table formatting
val students = listOf(
    Triple("Alice", 95, "A"),
    Triple("Bob", 87, "B"),
    Triple("Charlie", 92, "A"),
    Triple("Diana", 78, "C")
)

println("STUDENT GRADES")
println("==============")
println(String.format("%-10s %-8s %-6s", "Name", "Score", "Grade"))
println("---------- -------- ------")

for ((name, score, grade) in students) {
```

```
    val formattedRow = String.format("%-10s %-8d %-6s", name, score, grade)
    println(formattedRow)
}
```

Tips for Success

- Always prefer `val` over `var` unless you need mutability
- Use string templates instead of concatenation or complex formatting when possible
- Consider whether a value should really be nullable before using `Type?`
- Take advantage of smart casts after type checking with `is`
- Use the Elvis operator for clean default value handling

Common Mistakes to Avoid

- Using `!!` operator without being certain the value isn't null
- Making everything nullable "just in case" (nullability is a design decision)
- Forgetting that properties of nullable types need safe calls
- Using unsafe casts (`as`) when the type isn't guaranteed
- Not initializing `lateinit` variables before use

Best Practices

- Design your code to minimize the need for nullable types
- Use `let` with safe call for clean null handling
- Always handle the null case when using safe casts
- Use type inference when types are obvious
- Chain safe calls (`obj?.prop1?.prop2`) instead of nested null checks