Washtenaw Community College
ANN ARBOR, MICHIGAN

CPS251

Android Development

by Scott Shaper

Kotlin Functions

Think of functions in programming like recipes in a cookbook. Just as a recipe is a set of instructions that you can follow to create a dish, a function is a set of instructions that your program can follow to perform a specific task. Functions help you organize your code into reusable pieces, making it easier to write, understand, and maintain.

In this lesson, we'll explore how to create and use functions in Kotlin. We'll learn about different types of functions, how to pass information to them, and how to get results back. Understanding functions is crucial for writing clean, efficient, and reusable code.

Quick Reference Table

Function Type	Description	When to Use
Basic Function	Standard function with parameters and return type	For most programming tasks
Single Expression	Simplified function that returns one expression	For simple calculations or transformations
Local Function	Function defined inside another function	When you need helper functions that are only used in one place
Lambda Expression	Anonymous function passed as an argument	For short, one-time operations
Higher-Order Function	Function that takes or returns another function	For flexible, reusable code patterns

Basic Functions

When to Use

- When you need to perform a specific task multiple times
- When you want to organize your code into logical units
- When you need to reuse code in different parts of your program
- When you want to make your code more readable and maintainable

Component	What It Does	Example
`fun` keyword	Declares a function	`fun greet() { }`
Function name	Identifies the function	`fun calculateTotal() { }`
Parameters	Accepts input values	`fun add(a: Int, b: Int) { }`
Return type	Specifies what the function returns	`fun multiply(): Int { }`
Function body	Contains the code to execute	`{ println("Hello") }`

Practical Examples

Copy Code

```
// ==========================================
// BASIC FUNCTION EXAMPLES
// ==========================================

// Example 1: Function with no parameters
// This function performs a simple task without needing any input
fun greet() {
    println("Hello, World!")
    println("Welcome to Kotlin programming!")
}

// Example 2: Function with parameters
// This function accepts input and uses it in its logic
fun greetPerson(name: String) {
    // 'name' is a parameter - it holds the value passed when calling the function
    println("Hello, $name!")
```

```
    println("Nice to meet you!")

    // You can use the parameter multiple times
    if (name.length > 5) {
        println("That's a long name, $name!")
    }
}

// Example 3: Function with return value
// This function performs a calculation and returns the result
fun calculateTotal(price: Double, quantity: Int): Double {
    // Calculate the total cost
    val subtotal = price * quantity

    // Apply 10% tax
    val tax = subtotal * 0.10

    // Calculate final total
    val total = subtotal + tax

    // Return the calculated total
    return total
}

// Example 4: Function with multiple parameters and complex logic
// This function creates a user profile string with validation
fun createUser(name: String, age: Int, isActive: Boolean): String {
    // Validate the input parameters
    if (name.isEmpty()) {
        return "Error: Name cannot be empty"
    }

    if (age < 0 || age > 150) {
        return "Error: Age must be between 0 and 150"
```

```
    }

    // Create a status message based on age and active status
    val status = when {
        age < 18 -> "Minor"
        age < 65 -> "Adult"
        else -> "Senior"
    }

    val activeStatus = if (isActive) "Active" else "Inactive"

    // Build and return the user profile string
    return "User: $name, Age: $age, Status: $status, Account: $activeStatus"
}

// Example 5: Function that returns different types based on conditions
// This function demonstrates conditional returns
fun getMessage(score: Int): String {
    return when {
        score >= 90 -> "Excellent! You're doing great!"
        score >= 80 -> "Good job! Keep up the good work!"
        score >= 70 -> "Not bad, but there's room for improvement."
        score >= 60 -> "You're passing, but consider studying more."
        else -> "You need to work harder to pass this course."
    }
}

// ==========================================
// CALLING FUNCTIONS EXAMPLES
// ==========================================

println("=== CALLING BASIC FUNCTIONS ===")

// Call the greet function (no parameters needed)
```

```
greet()
// Output:
// Hello, World!
// Welcome to Kotlin programming!

// Call greetPerson with different names
greetPerson("Alice")
// Output:
// Hello, Alice!
// Nice to meet you!
// That's a long name, Alice!

greetPerson("Bob")
// Output:
// Hello, Bob!
// Nice to meet you!

// Call calculateTotal and store the result
val totalCost = calculateTotal(29.99, 3)
println("Total cost for 3 items at $29.99 each: $${String.format("%.2f", totalCost)}")
// Output: Total cost for 3 items at $29.99 each: $98.97

// Call createUser with different parameters
val user1 = createUser("John Doe", 25, true)
val user2 = createUser("Jane Smith", 17, false)
val user3 = createUser("", 30, true)  // Invalid name

println("User 1: $user1")
println("User 2: $user2")
println("User 3: $user3")

// Call getMessage with different scores
println("Score 95: ${getMessage(95)}")
```

```
println("Score 75: ${getMessage(75)}")
println("Score 45: ${getMessage(45)}")
```

Function Features

When to Use

- Use default parameters when some arguments are optional
- Use varargs when you need to accept any number of arguments
- Use single expression functions for simple calculations
- Use local functions for helper code that's only needed in one place

Feature	What It Does	When to Use It
Default Parameters	Provides default values for parameters	When some parameters are optional
Varargs	Accepts any number of arguments	When you don't know how many arguments you'll need
Single Expression	Simplifies functions that return one expression	For simple calculations
Local Functions	Defines functions inside other functions	When you need helper functions that are only used in one place

Practical Examples

Copy Code

```
// ========================================
// DEFAULT PARAMETERS EXAMPLES
// ========================================
```

```
// Example 1: Function with default parameters
// This makes the function more flexible by providing sensible defaults
fun greetUser(name: String = "Guest", greeting: String = "Hello") {
    println("$greeting, $name!")

    // You can use the parameters in conditional logic
    if (name == "Guest") {
        println("Welcome! Please consider creating an account.")
    } else {
        println("Welcome back, $name!")
    }
}

// Example 2: Function with multiple default parameters
// This function calculates shipping cost with optional parameters
fun calculateShipping(
    weight: Double,
    distance: Double = 100.0,  // Default distance of 100 miles
    isExpress: Boolean = false,  // Default to standard shipping
    insurance: Double = 0.0     // Default to no insurance
): Double {
    // Base shipping cost based on weight
    var baseCost = weight * 0.50

    // Add distance cost
    baseCost += distance * 0.10

    // Add express shipping premium
    if (isExpress) {
        baseCost *= 1.5
        println("Express shipping selected - 50% premium applied")
    }

    // Add insurance cost
```

```
    if (insurance > 0) {
        baseCost += insurance * 0.05
        println("Insurance added: $${String.format("%.2f", insurance * 0.05)}")
    }

    return baseCost
}

// ==========================================
// VARARGS EXAMPLES
// ==========================================

// Example 1: Function that accepts any number of strings
// The 'vararg' keyword allows flexible argument counts
fun printAll(vararg messages: String) {
    println("Received ${messages.size} messages:")

    // Process each message
    for ((index, message) in messages.withIndex()) {
        println("  ${index + 1}. $message")
    }

    // You can also perform operations on all messages
    val totalLength = messages.sumOf { it.length }
    println("Total characters in all messages: $totalLength")
}

// Example 2: Function that processes any number of numbers
// This function can handle different amounts of numeric data
fun analyzeNumbers(vararg numbers: Int): Map {
    if (numbers.isEmpty()) {
        return mapOf("error" to "No numbers provided")
    }
```

```
    val result = mutableMapOf()

    // Calculate various statistics
    result["count"] = numbers.size
    result["sum"] = numbers.sum()
    result["average"] = numbers.average()
    result["min"] = numbers.minOrNull() ?: 0
    result["max"] = numbers.maxOrNull() ?: 0

    // Find even and odd numbers
    val evens = numbers.filter { it % 2 == 0 }
    val odds = numbers.filter { it % 2 != 0 }

    result["evenCount"] = evens.size
    result["oddCount"] = odds.size

    return result
}

// ==========================================
// SINGLE EXPRESSION FUNCTIONS
// ==========================================

// Example 1: Simple mathematical functions
// These functions are so simple they can be written in one line
fun square(x: Int) = x * x
fun cube(x: Int) = x * x * x
fun isEven(x: Int) = x % 2 == 0
fun isPositive(x: Int) = x > 0

// Example 2: String utility functions
// Simple text processing functions
fun capitalizeFirst(str: String) = str.replaceFirstChar { it.uppercase() }
fun reverseString(str: String) = str.reversed()
```

```
fun countVowels(str: String) = str.count { it.lowercase() in "aeiou" }

// Example 3: Type conversion functions
// Simple conversion functions
fun toFahrenheit(celsius: Double) = celsius * 9/5 + 32
fun toCelsius(fahrenheit: Double) = (fahrenheit - 32) * 5/9
fun toMiles(kilometers: Double) = kilometers * 0.621371

// ==========================================
// LOCAL FUNCTIONS EXAMPLES
// ==========================================

// Example 1: Function with local helper function
// This function calculates total cost with tax and applies discounts
fun calculateTotalWithTax(price: Double, quantity: Int): Double {
    // Local function to calculate tax
    fun applyTax(amount: Double): Double {
        val taxRate = 0.08  // 8% tax rate
        return amount * taxRate
    }

    // Local function to apply quantity discount
    fun applyQuantityDiscount(amount: Double, qty: Int): Double {
        return when {
            qty >= 10 -> amount * 0.15  // 15% discount for 10+ items
            qty >= 5 -> amount * 0.10   // 10% discount for 5+ items
            else -> 0.0                 // No discount
        }
    }

    // Calculate subtotal
    val subtotal = price * quantity

    // Apply discount
```

```
    val discount = applyQuantityDiscount(subtotal, quantity)
    val discountedSubtotal = subtotal - discount

    // Apply tax
    val tax = applyTax(discountedSubtotal)

    // Return final total
    return discountedSubtotal + tax
}

// Example 2: Function with local validation functions
// This function processes user registration with multiple validation steps
fun registerUser(username: String, email: String, age: Int): String {
    // Local function to validate username
    fun isValidUsername(name: String): Boolean {
        return name.length >= 3 &&
               name.all { it.isLetterOrDigit() || it == '_' } &&
               name[0].isLetter()
    }

    // Local function to validate email format
    fun isValidEmail(email: String): Boolean {
        return email.contains("@") &&
               email.contains(".") &&
               email.indexOf("@") < email.lastIndexOf(".")
    }

    // Local function to validate age
    fun isValidAge(age: Int): Boolean {
        return age >= 13 && age <= 120
    }

    // Perform all validations
    if (!isValidUsername(username)) {
```

```
        return "Error: Invalid username. Must be 3+ characters, start with letter, only
letters/numbers/underscore allowed."
    }

    if (!isValidEmail(email)) {
        return "Error: Invalid email format. Must contain @ and . in correct positions."
    }

    if (!isValidAge(age)) {
        return "Error: Invalid age. Must be between 13 and 120."
    }

    // If all validations pass, return success message
    return "User '$username' registered successfully with email '$email' (age: $age)"
}

// ========================================
// USING ALL FUNCTION FEATURES TOGETHER
// ========================================

println("=== DEMONSTRATING ALL FUNCTION FEATURES ===")

// Test default parameters
println("\n--- Default Parameters ---")
greetUser()  // Uses all defaults
greetUser("Alice")  // Uses default greeting
greetUser("Bob", "Hi there")  // Overrides all defaults

// Test varargs
println("\n--- Varargs ---")
printAll("Hello", "World", "Kotlin", "Programming")
printAll("Single message")

val stats = analyzeNumbers(10, 25, 30, 15, 40, 5)
```

```
println("Number analysis: $stats")

// Test single expression functions
println("\n--- Single Expression Functions ---")
println("5 squared: ${square(5)}")
println("3 cubed: ${cube(3)}")
println("Is 8 even? ${isEven(8)}")
println("Is -5 positive? ${isPositive(-5)}")
println("'hello' capitalized: ${capitalizeFirst("hello")}")
println("'kotlin' reversed: ${reverseString("kotlin")}")
println("Vowels in 'programming': ${countVowels("programming")}")

// Test local functions
println("\n--- Local Functions ---")
val totalCost = calculateTotalWithTax(25.0, 3)
println("Total cost with tax and discounts: $${String.format("%.2f", totalCost)}")

val registrationResult = registerUser("john_doe", "john@example.com", 25)
println("Registration result: $registrationResult")

val invalidRegistration = registerUser("j", "invalid-email", 5)
println("Invalid registration result: $invalidRegistration")
```

Advanced Functions

When to Use

- Use lambda expressions for short, one-time operations
- Use higher-order functions when you need flexible, reusable code patterns
- Use function references when you need to pass functions as arguments

Type	What It Does	When to Use It
Lambda Expression	Anonymous function passed as an argument	For short, one-time operations
Higher-Order Function	Takes or returns another function	For flexible, reusable code patterns
Function Reference	References a function without calling it	When passing functions as arguments

Practical Examples

Copy Code

```
// ==========================================
// LAMBDA EXPRESSIONS EXAMPLES
// ==========================================

// Example 1: Basic lambda expressions
// Lambdas are anonymous functions that can be stored in variables
val add = { x: Int, y: Int -> x + y }
val multiply = { x: Int, y: Int -> x * y }
val isEven = { x: Int -> x % 2 == 0 }
val getLength = { str: String -> str.length }

// Example 2: Lambda with different parameter types
// Lambdas can work with any data types
val formatPrice = { price: Double -> "$${String.format("%.2f", price)}" }
val capitalize = { str: String -> str.uppercase() }
val isAdult = { age: Int -> age >= 18 }

// Example 3: Lambda with complex logic
// Lambdas can contain multiple statements
val processUser = { name: String, age: Int ->
```

```
    val status = if (age >= 18) "Adult" else "Minor"
    val greeting = if (age >= 18) "Welcome" else "Hello young one"
    "$greeting, $name! You are a $status."
}

// ==========================================
// HIGHER-ORDER FUNCTIONS EXAMPLES
// ==========================================

// Example 1: Function that takes another function as parameter
// This function can perform any operation on two numbers
fun performOperation(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
    println("Performing operation on $x and $y")
    val result = operation(x, y)
    println("Result: $result")
    return result
}

// Example 2: Function that returns a function
// This function creates different greeting functions
fun createGreeter(greeting: String): (String) -> String {
    return { name -> "$greeting, $name!" }
}

// Example 3: Function that processes a list with a custom operation
// This function applies any transformation to list elements
fun processList(
    items: List,
    processor: (String) -> String,
    filter: (String) -> Boolean = { true }  // Default filter accepts everything
): List {
    return items
        .filter(filter)          // Apply the filter function
        .map(processor)          // Apply the processor function
```

```
}

// Example 4: Function that combines multiple operations
// This function can chain different operations together
fun  chainOperations(
    initial: T,
    operations: List<(T) -> T>
): T {
    var result = initial
    println("Starting with: $result")

    for ((index, operation) in operations.withIndex()) {
        result = operation(result)
        println("After operation ${index + 1}: $result")
    }

    return result
}

// ==========================================
// FUNCTION REFERENCES EXAMPLES
// ==========================================

// Example 1: Basic function references
// These functions can be referenced and passed around
fun inchesToFeet(inches: Double): Double {
    return inches * 0.0833333
}

fun inchesToYards(inches: Double): Double {
    return inches * 0.0277778
}

fun inchesToMeters(inches: Double): Double {
```

```
    return inches * 0.0254
}

// Example 2: Function that uses function references
// This function can convert measurements using any conversion function
fun convertMeasurement(value: Double, converter: (Double) -> Double): Double {
    val result = converter(value)
    println("Converted $value inches to ${String.format("%.4f", result)} units")
    return result
}

// Example 3: Function that creates conversion chains
// This function can apply multiple conversions in sequence
fun createConversionChain(
    startValue: Double,
    conversions: List<(Double) -> Double>
): List {
    val results = mutableListOf()
    var currentValue = startValue

    for (conversion in conversions) {
        currentValue = conversion(currentValue)
        results.add(currentValue)
    }

    return results
}

// ==========================================
// USING ALL ADVANCED FUNCTION FEATURES
// ==========================================

println("=== DEMONSTRATING ADVANCED FUNCTION FEATURES ===")
```

```
// Test lambda expressions
println("\n--- Lambda Expressions ---")
println("5 + 3 = ${add(5, 3)}")
println("4 * 6 = ${multiply(4, 6)}")
println("Is 7 even? ${isEven(7)}")
println("Length of 'Kotlin': ${getLength("Kotlin")}")
println("Formatted price: ${formatPrice(29.99)}")
println("Capitalized: ${capitalize("hello world")}")
println("User processing: ${processUser("Alice", 25)}")

// Test higher-order functions
println("\n--- Higher-Order Functions ---")
val sum = performOperation(10, 5, add)
val product = performOperation(10, 5, multiply)

// Create custom operations
val power = { x: Int, y: Int ->
    var result = 1
    repeat(y) { result *= x }
    result
}
val powerResult = performOperation(2, 8, power)

// Test function that returns functions
val formalGreeter = createGreeter("Good day")
val casualGreeter = createGreeter("Hey")
println("Formal: ${formalGreeter("Mr. Smith")}")
println("Casual: ${casualGreeter("John")}")

// Test list processing
val names = listOf("alice", "bob", "charlie", "diana", "eve")
val processedNames = processList(
    items = names,
    processor = { it.uppercase() },
```

```
    filter = { it.length > 3 }
)
println("Processed names: $processedNames")

// Test operation chaining
val numbers = listOf(
    { x: Int -> x + 10 },
    { x: Int -> x * 2 },
    { x: Int -> x - 5 }
)
val finalResult = chainOperations(5, numbers)
println("Final result after chaining: $finalResult")

// Test function references
println("\n--- Function References ---")
val feet = convertMeasurement(12.0, ::inchesToFeet)
val yards = convertMeasurement(36.0, ::inchesToYards)
val meters = convertMeasurement(39.37, ::inchesToMeters)

// Test conversion chains
val conversionChain = createConversionChain(12.0, listOf(
    ::inchesToFeet,
    ::inchesToYards,
    ::inchesToMeters
))
println("Conversion chain results: $conversionChain")

// Test with custom lambda instead of function reference
val customConversion = { inches: Double -> inches * 0.1 }  // Convert to custom units
val customResult = convertMeasurement(10.0, customConversion)
```

Tips for Success

- Give your functions clear, descriptive names that indicate what they do

- Keep functions focused on a single task (single responsibility principle)
- Use default parameters to make functions more flexible
- Consider using single expression functions for simple calculations
- Use meaningful parameter names to make your code more readable
- Add comments to explain complex functions or unusual behavior

Common Mistakes to Avoid

- Creating functions that are too long or do too many things
- Using vague or misleading function names
- Forgetting to specify return types for complex functions
- Not handling all possible cases in functions with multiple paths
- Creating functions that have side effects without documenting them
- Using global variables when parameters would be more appropriate

Best Practices

- Follow the single responsibility principle - each function should do one thing well
- Use meaningful parameter names that indicate their purpose
- Consider using default parameters to make functions more flexible
- Use type inference when the types are obvious, but be explicit when they're not
- Add documentation comments to explain what your function does
- Test your functions with different inputs to ensure they work correctly