Washtenaw Community College
ANN ARBOR, MICHIGAN

CPS251

Android Development

by Scott Shaper

Introduction to Kotlin

Kotlin is a modern programming language that runs on the Java Virtual Machine (JVM) and has emerged as a powerful alternative to Java for Android development. It's designed to be more concise, safer, and more expressive than Java, while maintaining full interoperability with Java code.

Kotlin at a Glance

Feature	Description	Common Use
Conciseness	Reduces boilerplate code significantly	Writing cleaner, more maintainable code
Null Safety	Type system helps eliminate null pointer exceptions	Building more robust applications
Interoperability	Works seamlessly with existing Java code	Gradual migration of Java projects
Modern Features	Lambda expressions, extension functions, etc.	More expressive and functional programming

The Emergence of Kotlin

Kotlin was introduced in 2011 by JetBrains, the company known for creating IntelliJ IDEA, a popular IDE for Java development. The language was designed to be fully interoperable with Java while addressing some of the common pain points of Java, such as verbosity, null safety issues, and boilerplate code.

Kotlin's Adoption for Android Development

Kotlin's significance in the Android development landscape was solidified in 2017 when Google announced official support for Kotlin on Android during their annual developer conference, Google I/O. This endorsement was a response to the growing popularity of Kotlin among Android developers who appreciated its concise syntax and robust features, which streamlined common programming tasks and reduced the likelihood of bugs.

When to Use Kotlin

- When starting a new Android project (Google's preferred language)
- When maintaining large codebases where code clarity is crucial
- When working with teams familiar with modern programming paradigms
- When you want to reduce potential runtime errors through better type safety

Key Advantages of Kotlin

Feature	What It Does	When to Use It
Conciseness	Reduces boilerplate code, resulting in fewer lines of code	When you want to write more maintainable, readable code
Null Safety	Prevents null pointer exceptions through compile-time checks	When you want to avoid the "billion-dollar mistake" of null references
Extension Functions	Allows adding new functions to existing classes without inheriting from them	When extending third-party libraries without modifying source code
Smart Casts	Automatically casts types after type checks	When working with polymorphic code to avoid explicit casting

Practical Examples

Copy Code

```
// ==========================================
// KOTLIN VS JAVA COMPARISON EXAMPLES
// ==========================================

// Example 1: Data Classes - The Power of Conciseness
// This demonstrates how Kotlin reduces boilerplate code dramatically

// Java equivalent would require:
// - Private fields
// - Constructor
// - Getters and setters
// - equals() and hashCode()
// - toString()
```

```
// - copy() method
// - Approximately 50+ lines of code!

// Kotlin data class - just 1 line!
data class Person(val name: String, val age: Int)

// Using the data class
val person1 = Person("Alice", 25)
val person2 = Person("Bob", 30)
val person3 = Person("Alice", 25)

println("=== DATA CLASS EXAMPLE ===")
println("Person 1: $person1")
println("Person 2: $person2")
println("Person 3: $person3")

// Automatic equals and hashCode
println("Person 1 equals Person 3: ${person1 == person3}")  // true
println("Person 1 equals Person 2: ${person1 == person2}")  // false

// Automatic copy method
val person4 = person1.copy(age = 26)
println("Person 4 (copy with age change): $person4")

// Automatic component functions for destructuring
val (name, age) = person1
println("Destructured: name=$name, age=$age")

// Example 2: Null Safety - Preventing Runtime Crashes
// This shows how Kotlin prevents null pointer exceptions at compile time
println("\n=== NULL SAFETY EXAMPLE ===")

// In Java, this would compile but crash at runtime:
// String name = null;
```

```
// int length = name.length(); // NullPointerException!

// In Kotlin, this won't even compile:
// val name: String = null  // Compile error!

// Safe nullable types
val nullableName: String? = null
val safeName: String = "John"

// Safe calls - won't crash
val length1 = nullableName?.length
val length2 = safeName.length

println("Nullable name length: $length1")  // null
println("Safe name length: $length2")      // 4

// Elvis operator for default values
val displayName = nullableName ?: "Unknown"
println("Display name: $displayName")      // "Unknown"

// Safe calls with chaining
val email: String? = "user@example.com"
val domain = email?.substringAfter('@')?.substringBefore('.')
println("Email domain: $domain")           // "user"

// Example 3: Extension Functions - Adding Methods to Existing Classes
// This demonstrates how Kotlin can extend classes you don't own
println("\n=== EXTENSION FUNCTIONS EXAMPLE ===")

// Add a new method to String class
fun String.addExclamation(): String {
    return "$this!"
}
```

```
// Add a method to check if string is palindrome
fun String.isPalindrome(): Boolean {
    val cleaned = this.lowercase().filter { it.isLetterOrDigit() }
    return cleaned == cleaned.reversed()
}

// Add a method to format phone numbers
fun String.formatPhoneNumber(): String {
    val digits = this.filter { it.isDigit() }
    return when {
        digits.length == 10 -> "(${digits.substring(0, 3)}) ${digits.substring(3,
6)}-${digits.substring(6)}"
        digits.length == 11 && digits.startsWith("1") -> "1 (${digits.substring(1, 4)})
${digits.substring(4, 7)}-${digits.substring(7)}"
        else -> this
    }
}

// Use the extension functions
val greeting = "Hello"
val palindrome = "A man a plan a canal Panama"
val phoneNumber = "5551234567"

println("Greeting with exclamation: ${greeting.addExclamation()}")
println("Is '$palindrome' a palindrome? ${palindrome.isPalindrome()}")
println("Formatted phone: ${phoneNumber.formatPhoneNumber()}")

// Example 4: Smart Casts - Automatic Type Casting
// This shows how Kotlin eliminates the need for explicit casting
println("\n=== SMART CASTS EXAMPLE ===")

// Function that works with different types
fun processValue(value: Any): String {
    return when (value) {
```

```
        is String -> {
            // Kotlin automatically knows 'value' is String here
            "String: '${value.uppercase()}' (length: ${value.length})"
        }
        is Int -> {
            // Kotlin automatically knows 'value' is Int here
            "Integer: $value (doubled: ${value * 2})"
        }
        is List<*> -> {
            // Kotlin automatically knows 'value' is List here
            "List with ${value.size} items: $value"
        }
        is Boolean -> {
            // Kotlin automatically knows 'value' is Boolean here
            "Boolean: $value (opposite: ${!value})"
        }
        else -> "Unknown type: ${value::class.simpleName}"
    }
}

// Test smart casts with different types
val testValues = listOf("Hello", 42, listOf(1, 2, 3), true, 3.14)

for (value in testValues) {
    println("Processing: ${processValue(value)}")
}

// Example 5: Lambda Expressions and Higher-Order Functions
// This demonstrates Kotlin's functional programming capabilities
println("\n=== LAMBDA EXPRESSIONS EXAMPLE ===")

// Lambda expressions
val add = { x: Int, y: Int -> x + y }
val multiply = { x: Int, y: Int -> x * y }
```

```
val isEven = { x: Int -> x % 2 == 0 }

// Higher-order function that takes a function as parameter
fun performOperation(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
    return operation(x, y)
}

// Use the higher-order function with different operations
val sum = performOperation(10, 5, add)
val product = performOperation(10, 5, multiply)

println("10 + 5 = $sum")
println("10 * 5 = $product")

// Lambda with collections
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val evenNumbers = numbers.filter(isEven)
val doubledNumbers = numbers.map { it * 2 }
val sumOfEvens = evenNumbers.sum()

println("Original numbers: $numbers")
println("Even numbers: $evenNumbers")
println("Doubled numbers: $doubledNumbers")
println("Sum of evens: $sumOfEvens")

// Example 6: String Templates and Interpolation
// This shows Kotlin's powerful string handling
println("\n=== STRING TEMPLATES EXAMPLE ===")

val userName = "Alice"
val userAge = 25
val userScore = 95.5

// Simple string interpolation
```

```
val simpleMessage = "Hello, $userName! You are $userAge years old."
println(simpleMessage)

// Complex expressions in strings
val detailedMessage = """
User Profile:
  Name: ${userName.uppercase()}
  Age: $userAge
  Score: ${String.format("%.1f", userScore)}%
  Status: ${if (userScore >= 90) "Excellent" else "Good"}
  Next birthday: ${userAge + 1}
""".trimIndent()

println(detailedMessage)

// Example 7: When Expression - Powerful Switch Statement
// This demonstrates Kotlin's enhanced switch-like functionality
println("\n=== WHEN EXPRESSION EXAMPLE ===")

fun getDayDescription(day: String): String {
    return when (day.lowercase()) {
        "monday" -> "Start of the work week"
        "tuesday" -> "Second day of the work week"
        "wednesday" -> "Hump day - middle of the week"
        "thursday" -> "Almost Friday"
        "friday" -> "TGIF - Thank Goodness It's Friday!"
        "saturday", "sunday" -> "Weekend - time to relax!"
        else -> "Unknown day"
    }
}

fun getGradeDescription(score: Int): String {
    return when {
        score >= 90 -> "A - Excellent"
```

```
        score >= 80 -> "B - Good"
        score >= 70 -> "C - Satisfactory"
        score >= 60 -> "D - Needs Improvement"
        else -> "F - Failed"
    }
}

// Test the when expressions
val testDays = listOf("Monday", "Wednesday", "Saturday", "Invalid")
val testScores = listOf(95, 87, 72, 55, 100)

println("Day Descriptions:")
for (day in testDays) {
    println("  $day: ${getDayDescription(day)}")
}

println("\nGrade Descriptions:")
for (score in testScores) {
    println("  Score $score: ${getGradeDescription(score)}")
}

// Example 8: Real-World Kotlin Usage Scenarios
println("\n=== REAL-WORLD KOTLIN SCENARIOS ===")

// Scenario 1: User Management System
data class User(
    val id: String,
    val name: String,
    val email: String,
    val isActive: Boolean = true,
    val createdAt: String = "2024-01-01"
)

// Extension function for user validation
```

```
fun User.isValidEmail(): Boolean {
    return email.contains("@") && email.contains(".") && email.indexOf("@") <
email.lastIndexOf(".")
}

// Extension function for user display
fun User.getDisplayInfo(): String {
    return """
    User ID: $id
    Name: $name
    Email: $email
    Status: ${if (isActive) "Active" else "Inactive"}
    Created: $createdAt
    """.trimIndent()
}

// Create and use users
val users = listOf(
    User("user1", "Alice Johnson", "alice@example.com"),
    User("user2", "Bob Smith", "bob@example.com", isActive = false),
    User("user3", "Charlie Brown", "invalid-email")
)

println("User Management System:")
for (user in users) {
    println("\n${user.getDisplayInfo()}")
    println("Valid email: ${user.isValidEmail()}")
}

// Scenario 2: Product Catalog with Smart Features
data class Product(
    val id: String,
    val name: String,
    val price: Double,
```

```
    val category: String,
    val inStock: Boolean = true
)

// Extension function for product formatting
fun Product.getFormattedPrice(): String {
    return "$${String.format("%.2f", price)}"
}

// Extension function for stock status
fun Product.getStockStatus(): String {
    return if (inStock) "In Stock" else "Out of Stock"
}

// Extension function for discount calculation
fun Product.calculateDiscountedPrice(discountPercent: Double): Double {
    return price * (1 - discountPercent / 100)
}

val products = listOf(
    Product("prod1", "Laptop", 999.99, "Electronics"),
    Product("prod2", "Book", 29.99, "Books"),
    Product("prod3", "Phone", 699.99, "Electronics", inStock = false)
)

println("\nProduct Catalog:")
for (product in products) {
    val discountedPrice = product.calculateDiscountedPrice(10.0)
    println("""
    ${product.name}
      Price: ${product.getFormattedPrice()}
      Category: ${product.category}
      Stock: ${product.getStockStatus()}
      With 10% discount: $${String.format("%.2f", discountedPrice)}
```

```
    """.trimIndent())
}
```

Kotlin Playground

There is a Kotlin Playground where you can write small kotlin scripts to practice and experiment with the language. This is great for learning, but be aware that it's not suited for Android Studio development.

Compilation to Bytecode

Both Kotlin and Java are statically typed languages that compile to the same bytecode format, which runs on the Java Virtual Machine (JVM). This means that when you write Android applications in Kotlin or Java, the source code is compiled into JVM bytecode, which is then executed by the Android Runtime (ART) on Android devices.

The ability to compile to the same bytecode allows Kotlin and Java to be used interchangeably or together within the same project. Android developers can gradually migrate Java codebases to Kotlin, integrating Kotlin files into existing Java applications without needing to rewrite the entire application. This seamless integration is facilitated by the Kotlin plugin for Android Studio, which includes tools for converting Java code to Kotlin automatically.

Tips for Success

- Start with small, isolated Kotlin files in an existing Java project
- Use the "Convert Java File to Kotlin File" feature in Android Studio
- Leverage Kotlin's extension functions to enhance existing Java APIs
- Take advantage of null safety to write more robust code

Common Mistakes to Avoid

- Writing Kotlin code in a Java style (not leveraging Kotlin's features)
- Overusing nullable types when they aren't necessary

- Ignoring compiler warnings about platform types (Java types in Kotlin)
- Converting complex Java files to Kotlin without understanding the generated code

Best Practices

- Write idiomatic Kotlin, not Java-style code in Kotlin syntax
- Use data classes for model/POJO classes to reduce boilerplate
- Leverage scope functions (let, apply, with, run, also) for cleaner code
- Prefer immutability (val over var) when possible
- Use expression bodies for simple functions

Impact on Android Development

Kotlin's introduction into the Android ecosystem has led to significant changes in how Android applications are developed. Its adoption has grown rapidly, with many developers and companies switching to Kotlin for its expressiveness, safety features, and the productivity benefits it offers over Java. Google further endorsed Kotlin by announcing it as the preferred language for Android app development at Google I/O 2019.

In conclusion, Kotlin provides an effective alternative to Java for Android development, addressing many of the shortcomings of Java while enhancing code maintainability and developer productivity. Its compatibility with Java and the JVM ensures that developers can leverage the robust ecosystem of Java while benefiting from the improvements that Kotlin offers.