KEEP: Native Result Pattern Support

Author: [Your Name/Organization]
Status: Proposal
Type: Feature
Created: 2024
Discussion: [Link to GitHub issue or forum thread]

Summary

This proposal introduces native language support for Result patterns in Kotlin, providing first-class syntax for declaring Result types, pattern matching, and error propagation. The feature addresses the common pattern of using sealed class hierarchies for Result types by reducing boilerplate, improving ergonomics, and enabling Rust-style error propagation.

Motivation

Problem Statement

Kotlin developers frequently use sealed class hierarchies to model Result types for type-safe error handling. This pattern is widely adopted in production codebases (e.g., Arrow, Arrow-kt, and many others) but requires significant boilerplate:

1. **Verbose Type Definitions**: Each Result type requires:
   - A sealed class declaration
   - A nested `Failure` sealed class
   - Multiple error case data classes
   - Repetitive extension functions (`onSuccess`, `onFailure`, `getOrThrow`, `fold`)
   - Manual `isSuccess`/`isFailure` properties
2. **Repetitive Pattern Matching**: `when` expressions require verbose type checks and manual property access:

```
when (val result = fileService.readFile(path)) {
    is ReadFileResult.Success -> {
        val content = result.content  // Manual property access
        // Use content
    }
    is ReadFileResult.Failure.FileNotFound -> {
        println("File not found: ${result.path}")
    }
    is ReadFileResult.Failure.PermissionDenied -> {
        println("Permission denied: ${result.path}")
    }
```

```
    // ... more cases
}
```

3. **No Error Propagation**: Unlike Rust's `?` operator, Kotlin requires manual error handling at each step, leading to verbose code:

```
val fileResult = fileService.readFile(path)
if (fileResult is ReadFileResult.Failure) {
    return ProcessResult.Failure.ReadError(fileResult)
}
val content = fileResult.content

val parseResult = parser.parse(content)
if (parseResult is ParseResult.Failure) {
    return ProcessResult.Failure.ParseError(parseResult)
}
val data = parseResult.data
// ... repeat for each operation
```

4. **Type Information Loss**: The standard library's `Result<T>` type erases specific error type information, forcing developers to use custom sealed hierarchies.

Real-World Impact

Analysis of production codebases using Result patterns shows:

- **~15-20% of total code** in error handling modules
- **~70% boilerplate** in Result type definitions
- **Repetitive patterns** across dozens of Result types per codebase
- **Common use cases**: File I/O, network requests, parsing, validation, database operations

Goals

1. Reduce boilerplate in Result type definitions by ~70%
2. Enable concise error propagation similar to Rust's `?` operator
3. Improve pattern matching ergonomics with automatic destructuring
4. Maintain backward compatibility with existing sealed class Result types
5. Preserve type safety and exhaustive checking

Description

1. Result Type Declaration

Introduce a `result` keyword for declaring Result types with built-in success/error variants:

```
result ReadFileResult {
    success String  // File content
    failure {
        FileNotFound(path: String)
        PermissionDenied(path: String)
        IOError(path: String, reason: String, cause: Throwable? = null)
    }
}
```

Semantics:

- Generates a sealed class hierarchy equivalent to the current manual approach
- `success` declares the success value type
- `failure { ... }` declares failure variants as data classes
- Automatically generates `isSuccess`, `isFailure` properties
- Automatically generates `onSuccess`, `onFailure`, `getOrThrow`, `fold` methods
- Supports exhaustive checking in `when` expressions

Compilation Target:

```
// Generated code (conceptual):
sealed class ReadFileResult {
    data class Success(val value: String) : ReadFileResult()
    sealed class Failure : ReadFileResult() {
        data class FileNotFound(val path: String) : Failure()
        data class PermissionDenied(val path: String) : Failure()
        data class IOError(val path: String, val reason: String, val cause:
Throwable? = null) : Failure()
    }

    val isSuccess: Boolean get() = this is Success
    val isFailure: Boolean get() = this is Failure

    inline fun <R> fold(
        onFailure: (Failure) -> R,
        onSuccess: (String) -> R
    ): R = when (this) {
        is Success -> onSuccess(value)
        is Failure -> onFailure(this)
    }

    // ... other generated methods
}
```

2. Result Construction

Provide concise syntax for constructing Result values:

```
// Success case
fun readFile(path: String): ReadFileResult = success(fileContent)

// Failure cases
fun readFile(path: String): ReadFileResult = failure.FileNotFound(path)
fun readFile(path: String): ReadFileResult = failure.IOError(path, "Failed to read",
cause)
```

Semantics:

- `success(value)` constructs a `Success` variant
- `failure.VariantName(...)` constructs a `Failure.VariantName` variant
- Type inference determines the Result type from context

3. Enhanced Pattern Matching

Improve `when` expressions with automatic destructuring and smart casting:

```
when (val result = fileService.readFile(path)) {
    success(content) -> {
        // content is automatically available, smart-cast to String
        println("File content: $content")
    }
    failure.FileNotFound(path) -> {
        // path is automatically destructured
        println("File not found: $path")
    }
    failure.PermissionDenied(path) -> {
        println("Permission denied: $path")
    }
    failure.IOError(path, reason, cause) -> {
        // All properties automatically destructured
        println("IO Error: $reason")
    }
}
```

Semantics:

- `success(pattern)` matches Success and destructures the value
- `failure.VariantName(pattern)` matches Failure variants and destructures properties
- Variables in patterns are automatically smart-cast
- Exhaustive checking ensures all cases are handled

4. Error Propagation Operator (`?`)

Introduce Rust-style error propagation:

```
suspend fun processFile(path: String): ProcessResult {
    val content = fileService.readFile(path)?  // Auto-propagate on failure
    val parsed = parser.parse(content)?  // Auto-propagate on failure
    return success(parsed)
}
```

Semantics:

- `result?` extracts the success value if `result` is `Success`
- If `result` is `Failure`, returns early with the failure
- The return type must be a Result type compatible with the failure
- Works in functions returning Result types

Compilation Target:

```
// result? expands to:
val content = when (val temp = fileService.readFile(path)) {
    is ReadFileResult.Success -> temp.value
    is ReadFileResult.Failure -> return ProcessResult.Failure.from(temp)
}
```

5. Result Builder DSL

Provide a builder DSL for complex error handling chains:

```
result {
    val content = fileService.readFile(path)?
    val parsed = parser.parse(content)?
    val validated = validator.validate(parsed)?
    success(validated)  // Final success value
} catch { failure ->
    // Handle any propagated failures
    ProcessResult.Failure.from(failure)
}
```

Semantics:

- `result { ... }` creates a Result builder context
- The `?` operator propagates failures within the builder

- The last expression must be a `success(...)` or `failure(...)`
- `catch { ... }` handles propagated failures (optional)

6. Result Type Composition

Allow composing Result types from base error types:

```
// Define base error type
result FileError {
    failure {
        FileNotFound(path: String)
        PermissionDenied(path: String)
        IOError(path: String, reason: String, cause: Throwable?)
    }
}

// Compose with success types
result ReadFileResult = Result<String, FileError>
result WriteFileResult = Result<Unit, FileError>
result DeleteFileResult = Result<Unit, FileError>
```

Semantics:

- `Result<SuccessType, ErrorType>` creates a Result type from existing error types
- Enables sharing error hierarchies across multiple Result types
- Maintains type safety and exhaustive checking

Use Cases

Use Case 1: File I/O Operations

Before (Current approach):

```
sealed class ReadFileResult {
    data class Success(val content: String) : ReadFileResult()
    sealed class Failure : ReadFileResult() {
        data class FileNotFound(val path: String) : Failure()
        data class PermissionDenied(val path: String) : Failure()
        data class IOError(val path: String, val reason: String, val cause:
Throwable?) : Failure()
    }
}

fun readFile(path: String): ReadFileResult {
    try {
        val file = File(path)
```

```
        if (!file.exists()) {
            return ReadFileResult.Failure.FileNotFound(path)
        }
        if (!file.canRead()) {
            return ReadFileResult.Failure.PermissionDenied(path)
        }
        return ReadFileResult.Success(file.readText())
    } catch (e: IOException) {
        return ReadFileResult.Failure.IOError(path, e.message ?: "Unknown", e)
    }
}

// Usage:
when (val result = readFile("data.txt")) {
    is ReadFileResult.Success -> {
        val content = result.content
        // Use content
    }
    is ReadFileResult.Failure.FileNotFound -> {
        // Handle not found
    }
    is ReadFileResult.Failure.PermissionDenied -> {
        // Handle permission denied
    }
    is ReadFileResult.Failure.IOError -> {
        // Handle IO error
    }
}
```

After (With proposed feature):

```
result ReadFileResult {
    success String
    failure {
        FileNotFound(path: String)
        PermissionDenied(path: String)
        IOError(path: String, reason: String, cause: Throwable? = null)
    }
}

fun readFile(path: String): ReadFileResult {
    result {
        val file = File(path)
        if (!file.exists()) {
            return failure.FileNotFound(path)
        }
        if (!file.canRead()) {
            return failure.PermissionDenied(path)
```

```
        }
        success(file.readText())
    } catch (e: IOException) {
        failure.IOError(path, e.message ?: "Unknown", e)
    }
}

// Usage:
when (val result = readFile("data.txt")) {
    success(content) -> {
        // Use content directly
    }
    failure.FileNotFound(path) -> {
        // Handle not found
    }
    failure.PermissionDenied(path) -> {
        // Handle permission denied
    }
    failure.IOError(path, reason, cause) -> {
        // Handle IO error
    }
}
```

Benefits:

- ~70% reduction in boilerplate
- More concise and readable code
- Automatic destructuring in pattern matching

Use Case 2: Error Propagation Chain

Before:

```
fun processFile(path: String): ProcessResult {
    val readResult = readFile(path)
    if (readResult is ReadFileResult.Failure) {
        return ProcessResult.Failure.ReadError(readResult)
    }
    val content = readResult.content

    val parseResult = parseJson(content)
    if (parseResult is ParseResult.Failure) {
        return ProcessResult.Failure.ParseError(parseResult)
    }
    val data = parseResult.data

    val validateResult = validate(data)
    if (validateResult is ValidateResult.Failure) {
```

```
        return ProcessResult.Failure.ValidationError(validateResult)
    }

    return ProcessResult.Success(validateResult.validData)
}
```

After:

```
fun processFile(path: String): ProcessResult {
    val content = readFile(path)?
    val data = parseJson(content)?
    val validData = validate(data)?
    return success(validData)
}
```

Benefits:

- 75% reduction in code
- Clearer error propagation flow
- Less error-prone (no manual error mapping)

Use Case 3: Complex Error Handling

Before:

```
val result = readFile("config.json")
    .fold(
        onFailure = { failure ->
            when (failure) {
                is ReadFileResult.Failure.FileNotFound ->
                    ProcessResult.Failure.ConfigNotFound(failure.path)
                is ReadFileResult.Failure.PermissionDenied ->
                    ProcessResult.Failure.ConfigAccessDenied(failure.path)
                is ReadFileResult.Failure.IOError ->
                    ProcessResult.Failure.ConfigReadError(failure.reason)
            }
        },
        onSuccess = { content ->
            parseJson(content)
                .fold(
                    onFailure = { parseFailure ->
                        ProcessResult.Failure.ConfigParseError(parseFailure.reason)
                    },
                    onSuccess = { config ->
                        ProcessResult.Success(config)
                    }
```

```
                )
            }
        )
```

After:

```
val result = result {
    val content = readFile("config.json")?
    val config = parseJson(content)?
    success(config)
} catch { failure ->
    ProcessResult.Failure.from(failure)
}
```

Benefits:

- 80% reduction in code
- Linear flow instead of nested callbacks
- Easier to read and maintain

Language Comparisons

This section compares the proposed Kotlin Result pattern with similar features in other modern programming languages.

Rust: Result<T, E> and the ? Operator

Rust has native support for Result types with exhaustive pattern matching and error propagation.

Rust Example:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn read_file(path: &str) -> Result<String, FileError> {
    match std::fs::read_to_string(path) {
        Ok(content) => Ok(content),
        Err(e) => Err(FileError::IOError(e.to_string())),
    }
}

fn process_file(path: &str) -> Result<Data, ProcessError> {
    let content = read_file(path)?;  // ? operator propagates errors
```

```
    let data = parse_json(&content)?;
    Ok(data)
}
```

Kotlin Equivalent (Proposed):

```
result ReadFileResult {
    success String
    failure {
        IOError(reason: String)
    }
}

fun processFile(path: String): ProcessResult {
    val content = readFile(path)?  // ? operator propagates errors
    val data = parseJson(content)?
    return success(data)
}
```

Key Similarities:

- Both use sealed/enum types for type-safe error handling
- Both support error propagation with `?` operator
- Both enforce exhaustive pattern matching
- Both avoid exceptions for control flow

Key Differences:

- Rust uses enums, Kotlin uses sealed classes (more object-oriented)
- Kotlin's `result` keyword reduces boilerplate compared to Rust's manual enum definitions
- Kotlin supports multiple failure variants in a single Result type (via sealed Failure class)
- Rust's `?` operator works with any `Result` type, Kotlin's would be type-aware

Zig: Error Unions

Zig uses error unions (`!`) for error handling, where functions can return either a value or an error.

Zig Example:

```
const FileError = error{
    FileNotFound,
    PermissionDenied,
    IOError,
};
```

```
fn readFile(path: []const u8) ![]const u8 {
    const file = try std.fs.cwd().openFile(path, .{});
    defer file.close();
    return try file.reader().readAllAlloc(allocator, max_size);
}

fn processFile(path: []const u8) !Data {
    const content = try readFile(path);  // try propagates errors
    return try parseJson(content);
}
```

Kotlin Equivalent (Proposed):

```
result ReadFileResult {
    success String
    failure {
        FileNotFound
        PermissionDenied
        IOError(reason: String)
    }
}

fun processFile(path: String): ProcessResult {
    val content = readFile(path)?  // ? propagates errors
    return success(parseJson(content)?)
}
```

Key Similarities:

- Both use explicit error types (not exceptions)
- Both support error propagation (`try` in Zig, `?` in proposed Kotlin)
- Both require explicit error handling

Key Differences:

- Zig's error unions are simpler (just error names), Kotlin supports data in errors
- Zig's `try` keyword is more explicit, Kotlin's `?` is more concise
- Kotlin's sealed classes allow richer error information
- Zig errors are compile-time checked, Kotlin would be too

Swift: Result<Success, Failure>

Swift has a `Result` type in the standard library, though without native language syntax.

Swift Example:

```
enum Result<Success, Failure: Error> {
    case success(Success)
    case failure(Failure)
}

func readFile(path: String) -> Result<String, FileError> {
    do {
        let content = try String(contentsOfFile: path)
        return .success(content)
    } catch {
        return .failure(.ioError(error.localizedDescription))
    }
}

func processFile(path: String) -> Result<Data, ProcessError> {
    let contentResult = readFile(path)
    switch contentResult {
    case .success(let content):
        return parseJson(content)
    case .failure(let error):
        return .failure(.readError(error))
    }
}
```

Kotlin Equivalent (Proposed):

```
result ReadFileResult {
    success String
    failure {
        IOError(reason: String)
    }
}

fun processFile(path: String): ProcessResult {
    val content = readFile(path)?  // More concise than Swift
    return success(parseJson(content)?)
}
```

Key Similarities:

- Both use Result types for error handling
- Both support pattern matching

Key Differences:

- Swift requires manual error propagation, Kotlin's `?` operator is more concise

- Kotlin's `result` keyword reduces boilerplate
- Swift's Result is a generic type, Kotlin's would be domain-specific types

Go: Explicit Error Returns

Go uses explicit error returns as a language convention.

Go Example:

```
func readFile(path string) (string, error) {
    content, err := os.ReadFile(path)
    if err != nil {
        return "", fmt.Errorf("failed to read file: %w", err)
    }
    return string(content), nil
}

func processFile(path string) (Data, error) {
    content, err := readFile(path)
    if err != nil {
        return nil, fmt.Errorf("read error: %w", err)
    }
    data, err := parseJson(content)
    if err != nil {
        return nil, fmt.Errorf("parse error: %w", err)
    }
    return data, nil
}
```

Kotlin Equivalent (Proposed):

```
fun processFile(path: String): ProcessResult {
    val content = readFile(path)?  // Much more concise
    return success(parseJson(content)?)
}
```

Key Similarities:

- Both use explicit error handling (no exceptions)
- Both return errors as part of the return type

Key Differences:

- Go requires manual error checking at each step, Kotlin's `?` operator is automatic
- Go uses tuple returns `(value, error)`, Kotlin uses sealed classes (type-safe)

- Kotlin's pattern matching is more powerful than Go's `if err != nil` checks
- Kotlin supports multiple error types, Go uses a single `error` interface

Summary of Language Comparisons

Language	Error Type	Propagation	Pattern Matching	Boilerplate
Rust	`Result<T, E>` enum	`?` operator	`match` (exhaustive)	Medium
Zig	Error unions `!T`	`try` keyword	`switch` (exhaustive)	Low
Swift	`Result<Success, Failure>`	Manual	`switch` (exhaustive)	High
Go	`(T, error)` tuple	Manual	`if err != nil`	High
Kotlin (Current)	Sealed classes	Manual	`when` (exhaustive)	Very High
Kotlin (Proposed)	`result` keyword	`?` operator	`when` (exhaustive)	Low

Advantages of Proposed Kotlin Approach:

1. **Lower boilerplate** than Rust (no manual enum definitions)
2. **Richer error types** than Zig (can carry data in errors)
3. **Better propagation** than Swift (native `?` operator)
4. **Type safety** over Go (sealed classes vs tuples)
5. **Familiar syntax** for Kotlin developers (builds on sealed classes)

Alternatives Considered

Alternative 1: Library-Based Solution

Approach: Create a library (e.g., Arrow) that provides Result types and utilities.

Rejected Because:

- Still requires boilerplate for type definitions
- No language-level support for error propagation
- Cannot provide native syntax improvements
- Adds external dependency

Alternative 2: Extend Standard Library Result

Approach: Enhance `kotlin.Result<T>` to support typed errors.

Rejected Because:

- Would break existing `Result<T>` API
- Cannot provide domain-specific error types
- Limited to single error type per Result

Alternative 3: Compiler Plugin

Approach: Implement as a compiler plugin without language changes.

Rejected Because:

- Limited IDE support
- Cannot introduce new keywords
- Harder to maintain and debug
- Not a first-class language feature

Alternative 4: Kotlin Multiplatform Result Types

Approach: Provide Result types as part of Kotlin Multiplatform.

Rejected Because:

- Doesn't address the boilerplate problem
- No language-level syntax improvements
- Still requires manual error propagation

Compatibility

Backward Compatibility

- **Existing sealed class Result types**: Continue to work unchanged
- **Extension functions**: New Result types are compatible with existing extension functions
- **Standard library Result**: Unchanged, can coexist with new Result types
- **Gradual migration**: Can adopt new syntax incrementally

Interoperability

- New Result types compile to standard sealed classes
- Can be used from Java (as sealed classes)
- Compatible with existing Kotlin features (coroutines, inline functions, etc.)

Breaking Changes

None. This is a purely additive feature.

Implementation Details

Compiler Changes

1. **Parser**: Add `result` keyword and syntax parsing
2. **Type System**: Add Result type representation
3. **IR Generation**: Transform Result syntax to sealed class IR
4. **Error Propagation**: Implement `?` operator transformation

5. **Pattern Matching**: Enhance `when` expression handling for Result types

IDE Support

1. **Syntax Highlighting**: Highlight `result`, `success`, `failure` keywords
2. **Code Completion**: Autocomplete Result variants and methods
3. **Refactoring**: Support refactoring between Result types
4. **Inspections**: Warn on non-exhaustive pattern matching
5. **Debugging**: Enhanced debugging experience for Result types

Standard Library

No changes required. The feature generates standard Kotlin code.

Migration Tools

1. **Automatic Migration**: IDE action to convert sealed class Result types to `result` syntax
2. **Inspection**: Suggest using `result` syntax for existing sealed Result types

Open Questions

1. **Naming**: Should we use `result` keyword or `Result` (capitalized)?
    - **Proposal**: Use `result` (lowercase) to match other Kotlin keywords
2. **Error Propagation Scope**: Should `?` operator work in all contexts or only Result-returning functions?
    - **Proposal**: Only in functions returning Result types (type-safe)
3. **Result Builder**: Should `result { }` be a scope function or a special construct?
    - **Proposal**: Special construct for better error handling
4. **Multiple Success Types**: Should Result types support multiple success variants?
    - **Proposal**: No, keep single success type (can use sealed classes for complex cases)
5. **Async Propagation**: Should we support `await?` for coroutines?
    - **Proposal**: Future enhancement, not in initial implementation

References

Language Documentation

- Rust Result Type
- Rust Error Propagation (`?` operator)
- Zig Error Handling
- Swift Result Type

- Go Error Handling

Kotlin Resources

- Arrow-kt Result
- Kotlin Sealed Classes
- Kotlin Pattern Matching
- KEEP Process

Related Proposals and Discussions

- Kotlin YouTrack: Result Type
- Kotlin Slack: #language-proposals

Appendix: Generated Code Example

For the Result type:

```
result ReadFileResult {
    success String
    failure {
        FileNotFound(path: String)
        PermissionDenied(path: String)
        IOError(path: String, reason: String, cause: Throwable? = null)
    }
}
```

The compiler generates (conceptually):

```
sealed class ReadFileResult {
    data class Success(val value: String) : ReadFileResult() {
        // Destructuring support
        operator fun component1() = value
    }

    sealed class Failure : ReadFileResult() {
        data class FileNotFound(val path: String) : Failure() {
            operator fun component1() = path
        }

        data class PermissionDenied(val path: String) : Failure() {
            operator fun component1() = path
        }

        data class IOError(
            val path: String,
```

```
            val reason: String,
            val cause: Throwable? = null
        ) : Failure() {
            operator fun component1() = path
            operator fun component2() = reason
            operator fun component3() = cause
        }
    }

    val isSuccess: Boolean
        get() = this is Success

    val isFailure: Boolean
        get() = this is Failure

    inline fun <R> fold(
        onFailure: (Failure) -> R,
        onSuccess: (String) -> R
    ): R = when (this) {
        is Success -> onSuccess(value)
        is Failure -> onFailure(this)
    }

    inline fun onSuccess(action: (String) -> Unit): ReadFileResult {
        if (this is Success) action(value)
        return this
    }

    inline fun onFailure(action: (Failure) -> Unit): ReadFileResult {
        if (this is Failure) action(this)
        return this
    }

    fun getOrThrow(): String = when (this) {
        is Success -> value
        is Failure -> throw IllegalStateException("Result is failure: $this")
    }

    fun getOrNull(): String? = (this as? Success)?.value
}
```

Revision History

- **2024-XX-XX**: Initial proposal