Incremental Memory Safety in an Established Software Stack

Lessons learned from Swift

Doug Gregor, Apple / Swift Language Steering Group

TLP:CLEAR

Cybersecurity & Infrastructure Security Agency

National Security Agency United States of America

Department of Justice Federal Bureau of Investigation

Australian Government
Australian Signals Directorate

ASD Australian Signals Directorate
ACSC Australian Cyber Security Centre

Communications Security Establishment
Canadian Centre for Cyber Security

Centre de la sécurité des télécommunications
Centre canadien pour la cybersécurité

National Cyber Security Centre
a part of GCHQ

National Cyber Security Centre
PART OF THE GCSB

certnz

The Case for Memory Safe Roadmaps

How do we get there from here?

C

C++

C#

Go

Java

Python

Rust

Swift

Swift was designed for this

The (initial) ecosystem: apps for Apple platforms

APIs in the Apple Software Development Kit (SDK) written in C & Objective-C

Millions of developers writing in C, C++, Objective-C

Binary stability over many years

Leveraging and improving the existing ecosystem

Swift made all existing C & Objective-C APIs available on day one

Interoperability made choice of language independent from library stack

Incremental adoption let people adopt at their own pace

It worked for Apple platforms

Existing C/Objective-C ecosystem moving toward Swift

Movement at all levels of the software stack

- Apps at the high level
- Binary-compatible replacements of (Objective-)C libraries with Swift
- Firmware

We think these lessons apply to other platforms and ecosystems

Agenda

Memory safety in Swift

Interoperability with the C family of languages

Build interoperability

Memory safety across the language boundary

Memory Safety in Swift

Swift one-pager

General-purpose language that is a joy to write

Approachable language with power tools for expert users

Native compilation & performance

Open-source since 2015

Cross-platform (Apple, Linux, Windows, Android, WebAssembly, Embedded, …)

https://github.com/swiftlang/

Memory safety protects the abstract machine

Programmers will create errors

Memory safety prevents those errors from escalating into security vulnerabilities

Important preconditions must be checked by the language

- Statically if possible
- Dynamically when necessarily

It’s better to halt than corrupt

Dimensions of memory safety

Lifetime safety - use-after-free

Bounds safety - out-of-bounds accesses

Type safety - type confusion

Initialization safety - use of uninitialized data

Thread safety - data races that compromise other safety guarantees

Lifetime safety

Value types

Types for which a copy is completely independent of the original

Example:

```
var names: [String] = [“Ada”, “Barbara”, “Grace”]
var otherNames = names
names.append(“Katherine”) // only modifies names

print(names)  // [“Ada”, “Barbara”, “Grace”, “Katherine”]
print(otherNames) // [“Ada”, “Barbara”, “Grace”]
```

Structs and enums compose value types

Structures aggregate value types into value types:

```
struct Document {
  var title: String
  var authorNames: [String]
}
```

Enums providing a choice between value types are value types:

```
enum DocumentReference {
  case stored(Document)
  case remote(URL)
}
```

Passing by reference

Explicit pass-by-reference with inout parameters:

```
func increment(_ value: inout Int) {
  value += 1
}
```

Call site must provide a reference to mutable data:

```
var x = 1
let y = 2
increment(&x) // okay
increment(&y) // error: ‘y’ is immutable
```

`inout` parameters never alias anything

Swift ensures that an inout parameter uniquely references a value

```
func swap<T>(_ x: inout T, _ y: inout T) { … }

swap(&a, &b) // okay
swap(&a, &a) // error: overlapping accesses to ‘a',
             // but modification requires exclusive access
```

Object-oriented programming in Swift

```
class Person: DatabaseRecord {
  var name: String

  init(name: String) { … }
  override func checkConsistency() throws { … }
}


let otherPerson = Person(name: “Hedy”)
let person = otherPerson
person.name = “Ada”
print(otherPerson.name)  // “Ada”
```

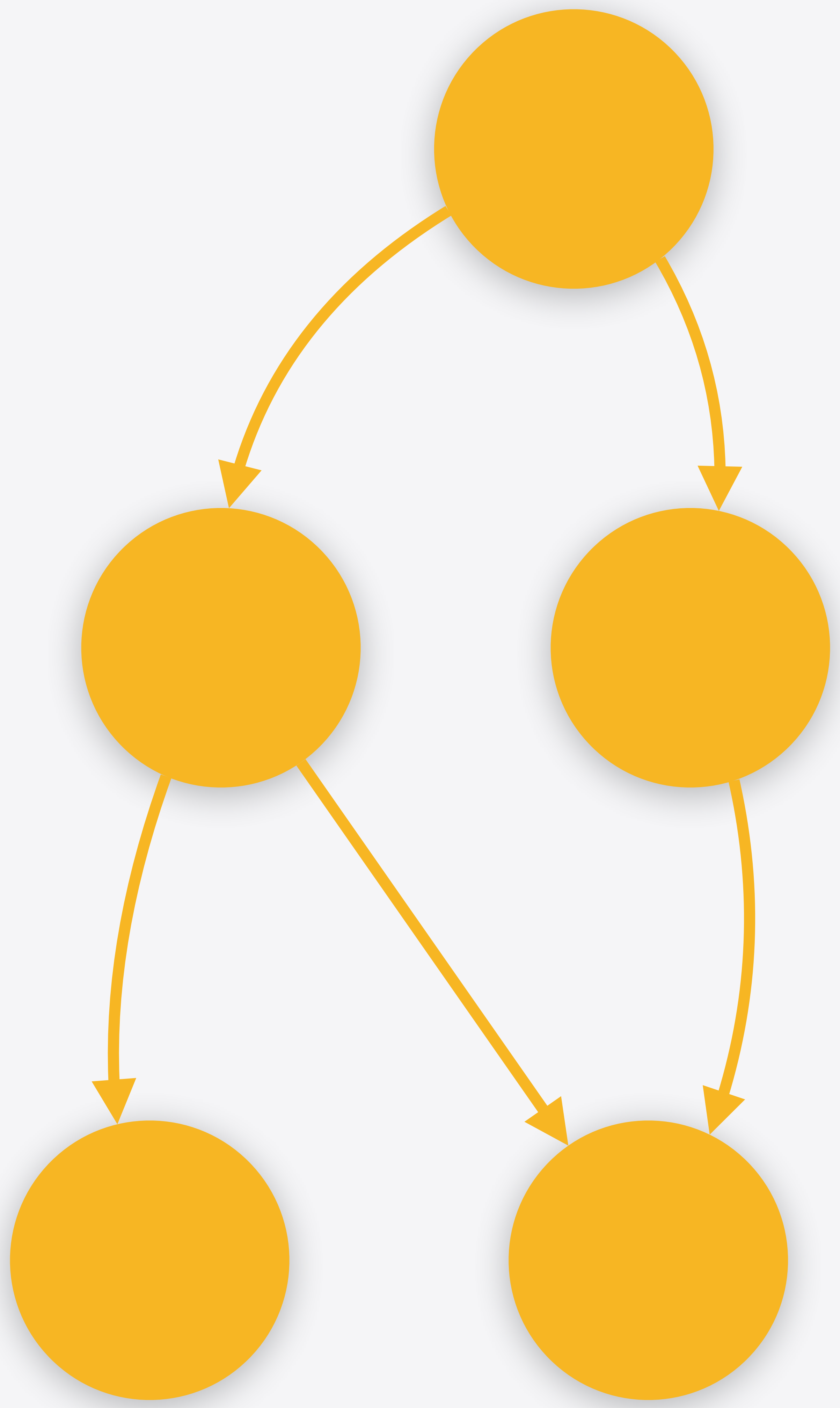
Automatic reference counting

Lifetime safety with very little ceremony

Good implementation tradeoffs vs. traditional GC

- Deterministic
- Locally optimizable
- Small runtime footprint

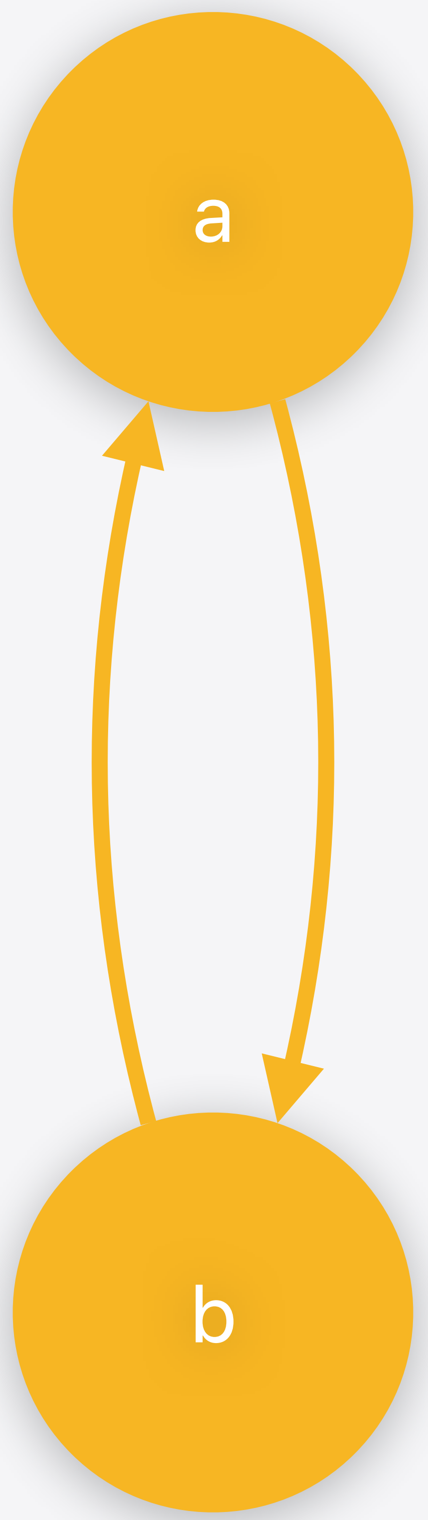
Common in C and C++ libraries



Reference cycles

Classes can be part of cyclic data structures

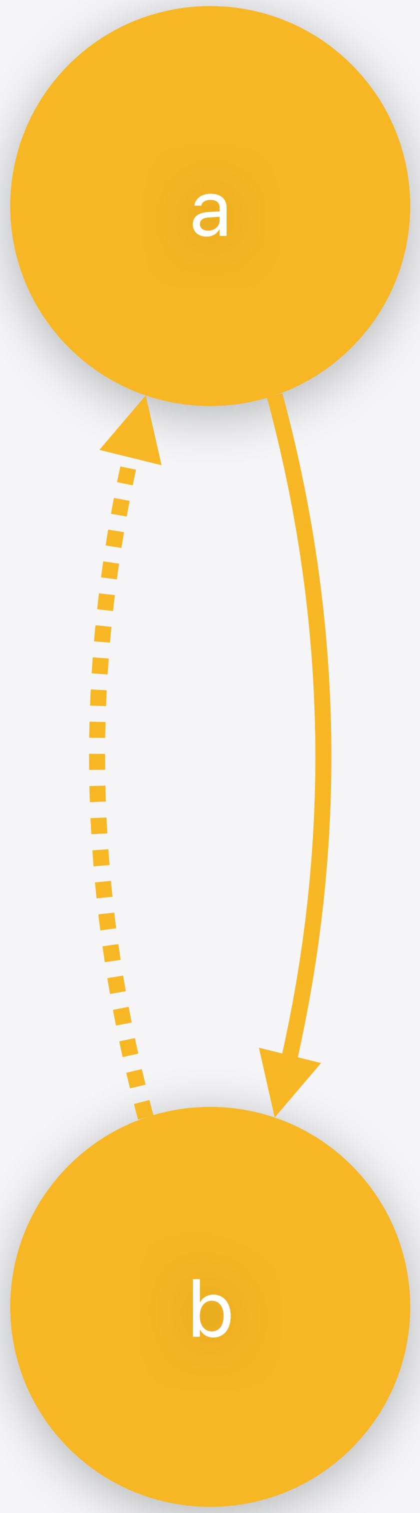
Swift does not provide a cycle collector



Weak references to break cycles

Reference cycles can be broken with weak references

```
weak var delegate: MyDelegate?
if let delegate {
   // delegate is now a strong reference,
   // object won’t go away
   delegate.onStart()
   // …
   delegate.onFinish()
}
```



Bounds safety

Collections with indexing (e.g., Array) bounds-check on access

Integer arithmetic traps on overflow

Type safety

Casts perform a runtime check and return an optional

```
if let subclass = superclassInstance as? Subclass { … }
if let nodeArray = collection as? [Node] { … }
```

Enums use safe access patterns

```
switch documentReference {
  case .stored(let document):
    print(“Document by \(document.author) is local”)

  case .remote(let url):
    print(“Document can be retrieved from \(url)”)
}
```

Initialization safety

A variable must be initialized before use:

```
let count: Int
if let buffer = existingBuffer {
  count = buffer.count
}
return count // error: count used before being initialized
```

Thread safety

Shared mutable state is the root of all… data races

A data race is when

- Two threads access the same data, and
- At least one of the accesses is a write.

Strategies for avoiding data races

- Make it immutable
- Don’t share
- Ensure exclude ownership of writes

Value types are great for concurrency

Every copy of a value type is completely independent of its original

Value types can be freely shared in a concurrent system

Actors protect their shared mutable state

```
actor BankAccount {
  var balance: Double

  func withdraw(dollars: Double) throws { … }
}
```

Access from outside of the actor must go through its implicit queue:

```
try await account.withdraw(dollars: 17.0)
```

Language + actor runtime guarantees no concurrent access to actor state

(Just) Rewrite It In ________

Technical hurdles to (Just) Rewrite It In ______

The second-system effects adds risk and delays

Need to keep shipping the old version

Social hurdles to (Just) Write It In ______

Team members not involved in the rewrite will feel left behind

Some people will be worried or ambivalent

Challenges with the rewrite will be attributed to the new language

Avoid silos

Be incremental

Get that first line of memory-safe code into your project *now*

Try to write new code in the new language

Involve the whole team

Targeted component rewrites are okay

Language Interoperability

Embedding a C(++) compiler

Swift provides built-in support for interoperability with C

Import C headers directly into Swift

Export C headers from Swift

POSIX in C

```
int dup(int);
int dup2(int, int);

int pipe(int [2]);

ssize_t read(int, void *, size_t);
```

POSIX in Swift

```
func dup(Int32) -> Int32
func dup2(Int32, Int32) -> Int32

func pipe(UnsafeMutablePointer<Int32>) -> Int32

func read(Int32, UnsafeMutableRawPointer?, Int) -> Int
```

CoreGraphics in C

```c
typedef struct CGColorSpace *CGColorSpaceRef;

CGColorSpaceRef CGColorSpaceRetain(CGColorSpaceRef space);

void CGColorSpaceRelease(CGColorSpaceRef space);

CGColorSpaceRef CGColorSpaceCreateWithName(CFStringRef name);

CFStringRef CGColorSpaceCopyName(CGColorSpaceRef space);

bool CGColorSpaceSupportsOutput(CGColorSpaceRef space);
```

CoreGraphics in Swift

```
class CGColorSpace {

  init?(name: CFString)

  var name: CFString? { get }

  var supportsOutput: Bool { get }
}
```

Interoperability with C++

C++ has richer abstractions than C

Automatically map C++ conventions into corresponding Swift:

- C++ containers imported as Swift collections
- “Move-only” types are noncopyable types in Swift
- const methods are non-mutating in Swift

Build interoperability

Swift started with a single build system

Able to add a single Swift source file to an existing project

Near-zero configuration to get started

Easily manage what APIs cross the language boundary

Package managers are great!

Language-specific package managers can get you up-and-running fast

- `git clone <repository>`
- `swift build / run / test`

Ability to pull in C libraries from the system

```
.systemLibrary(name: "CGLib", pkgConfig: "gio-unix-2.0",
  providers: [
   .brew(["glib", "glib-networking", "gobject-introspection"]),
   .apt(["libglib2.0-dev", "glib-networking",
         "gobject-introspection", “libgirepository1.0-dev”])
  ])
```

Package managers create silos

A C(++) code base is not using your language-specific package manager

Nobody wants to rewrite their build system to adopt your language

Embracing CMake

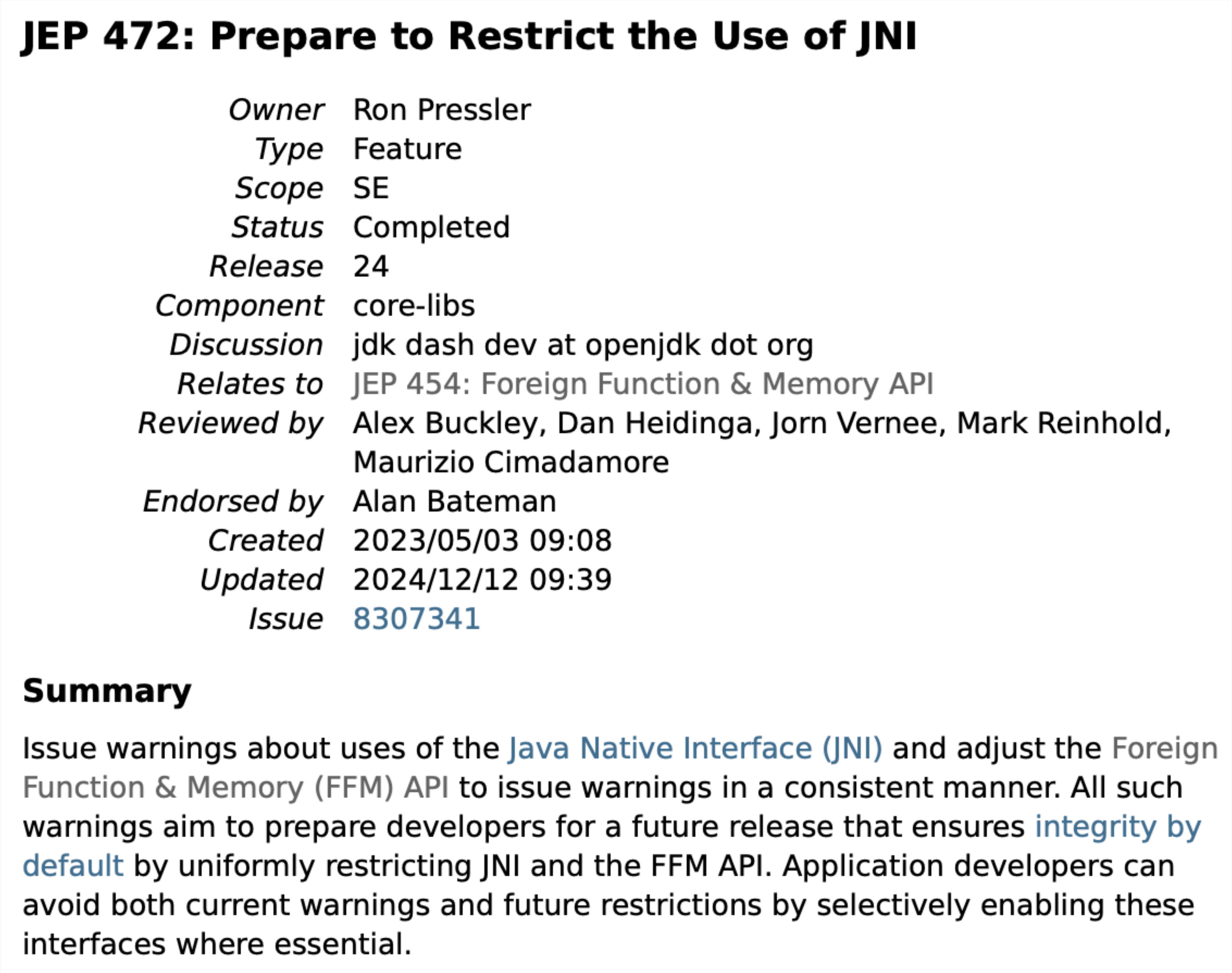
Augmented CMake with support for Swift

```
cmake_minimum_required(VERSION 3.26)

project(hello LANGUAGES CXX Swift)

add_executable(hello
  MyLib.cpp
  Hello.swift)

target_compile_options(hello PUBLIC
  “$<$<COMPILE_LANGUAGE:Swift>:-cxx-interoperability-mode=default>")
```

https://github.com/apple/swift-cmake-examples

Memory-safe interoperability

C(++) does not have cooties



JEP 472: Prepare to Restrict the Use of JNI

Owner	Ron Pressler
Type	Feature
Scope	SE
Status	Completed
Release	24
Component	core-libs
Discussion	jdk dash dev at openjdk dot org
Relates to	JEP 454: Foreign Function & Memory API
Reviewed by	Alex Buckley, Dan Heidinga, Jorn Vernee, Mark Reinhold, Maurizio Cimadamore
Endorsed by	Alan Bateman
Created	2023/05/03 09:08
Updated	2024/12/12 09:39
Issue	8307341

Summary

Issue warnings about uses of the Java Native Interface (JNI) and adjust the Foreign Function & Memory (FFM) API to issue warnings in a consistent manner. All such warnings aim to prepare developers for a future release that ensures integrity by default by uniformly restricting JNI and the FFM API. Application developers can avoid both current warnings and future restrictions by selectively enabling these interfaces where essential.

Safe language interoperability

Establish safety conventions at language boundaries

Evolve C and C++ toward expressing more safety conventions

Bounds safety in C

C functions often carry pointer-bounds information in separate parameters:

```
double average(
  const double *numbers,
  ptrdiff_t count
);
```

Memory-safe language can only express this unsafely:

```
func average(
  _ numbers: UnsafePointer<Double>, _ count: Int
) -> Double
```

Bounds safety in C

C functions often carry pointer-bounds information in separate parameters:

```
double average(
  const double * __counted_by(count) numbers,
  ptrdiff_t count
);
```

Memory-safe language can only express this unsafely:

```
func average(
  _ numbers: UnsafePointer<Double>, _ count: Int
) -> Double
```

https://clang.llvm.org/docs/BoundsSafety.html

Bounds safety in C

C functions often carry pointer-bounds information in separate parameters:

```
double average(
  const double * __counted_by(count) numbers,
  ptrdiff_t count
);
```

Memory-safe language can honor the bounds convention:

```
func average(
  _ numbers: UnsafeBufferPointer<Double>
) -> Double
```

https://clang.llvm.org/docs/BoundsSafety.html

Automatic reference counting is lifetime safety

GNOME’s `GVariant` type uses reference counting:

```
typedef struct _GVariant GVariant;


GVariant *g_variant_ref(GVariant *value);
void g_variant_unref(GVariant *value);

GVariant *g_variant_new_double(gdouble value);
gdouble g_variant_get_double(GVariant *value);
```

Automatic reference counting is lifetime safety

GNOME's `GVariant` type uses reference counting:

```
typedef struct _GVariant SWIFT_SHARED_REFERENCE(
    g_variant_ref, g_variant_unref)  GVariant;

GVariant *g_variant_ref(GVariant *value);
void g_variant_unref(GVariant *value);

GVariant *g_variant_new_double(gdouble value);
gdouble g_variant_get_double(GVariant *value);
```

Automatic reference counting is lifetime safety

GNOME's `GVariant` type uses reference counting:

```
typedef struct _GVariant SWIFT_SHARED_REFERENCE(
    g_variant_ref, g_variant_unref)  GVariant;

GVariant *g_variant_ref(GVariant *value);
void g_variant_unref(GVariant *value);

GVariant * _Nonnull g_variant_new_double(gdouble value)
  SWIFT_RETURNS_RETAINED;
gdouble g_variant_get_double(GVariant *value);
```

Conventions for reference counting

Documenting reference-counting conventions makes them safe in Swift

- `SWIFT_SHARED_REFERENCE`(retain-func, release-func)
- `SWIFT_RETURNS_RETAINED` / `_UNRETAINED` for return conventions

```
let variant = g_variant_new_double(3.14159)
if g_variant_classify(variant) == G_VARIANT_CLASS_DOUBLE {
  print(g_variant_get_double(variant))
}
```

Conventions for reference counting

Documenting reference-counting conventions makes them safe in Swift

- `SWIFT_SHARED_REFERENCE`(retain-func, release-func)
- `SWIFT_RETURNS_RETAINED` / `_UNRETAINED` for return conventions

Additional annotations provide ergonomic improvements

```
let variant = GVariant(double: 3.14159)
if variant.classify() == .double {
  print(variant.double)
}
```

Additional lifetime safety in C and C++

Clang provides additional annotations regarding lifetime:

- Attribute `noescape` says a pointer parameter doesn’t escape
- Attribute `lifetime_bound(param)` ties the lifetime of a return to a parameter

Static analysis in C and C++ can help check these annotations

Lifetime + bounds safety in C

C functions often carry pointer-bounds information in separate parameters:

```
double average(
  const double * __counted_by(count)
                 __attribute__((noescape)) numbers,
  ptrdiff_t count
);
```

Memory-safe language can only express this unsafely:

```
func average(_ numbers: Span<Double>) -> Double
```

Wha tif you can’t modify the C headers?

API notes describe conventions of C APIs

```
Tags:
- Name: _GVariant
  SwiftImportAs: reference
  SwiftRetainOp: g_variant_ref
  SwiftReleaseOp: g_variant_unref
Functions:
- Name: g_variant_new_double
  SwiftName: "GVariant.init(double:)"
  SwiftReturnOwnership: retained
  ResultType: "GVariant * _Nonnull"
- Name: g_variant_classify
  SwiftName: "GVariant.classify(self:)"
```

Safe interoperability requires coordination

C and C++

Codify conventions in C(++) source code

- Nullability
- Lifetime
- Bounds

C(++) code must benefit

Provide tooling to help with adoption

Memory-safe language

Prioritize C(++) interoperability

- Language-level
- Build system

Honor C(++) conventions

Incrementally moving memory safety forward

Build for adoption

Avoid creating silos

Work across language boundaries to improve safety

Swift resources

swift.org

Swift room here at FOSDEM ’25

- Embedded Swift
- Server-side Swift

Java room here at FOSDEM ’25

- Foreign Function and Memory APIs and Swift/Java interoperability