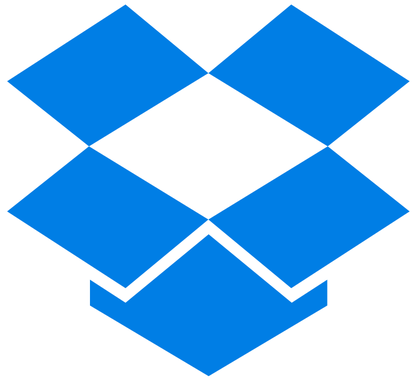
Bridging Languages Cross-Platform

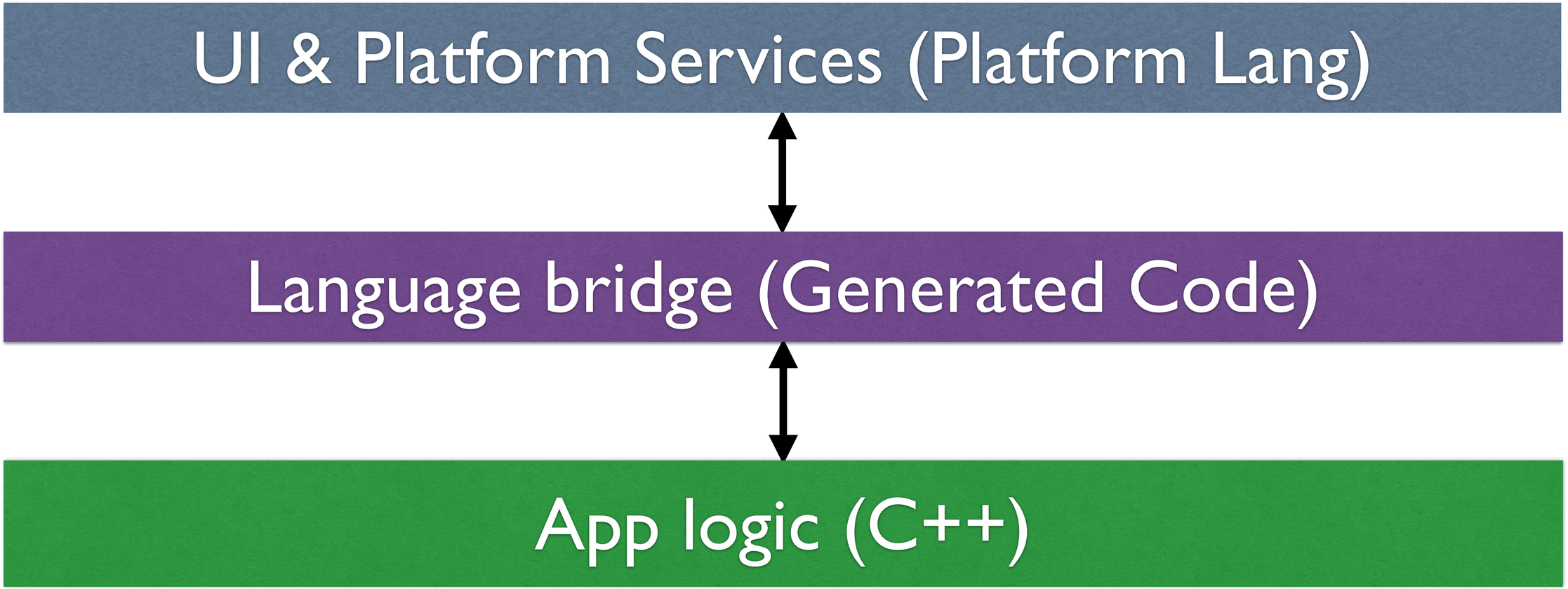
Andrew Twyman & Jacob Potter

Who are we?

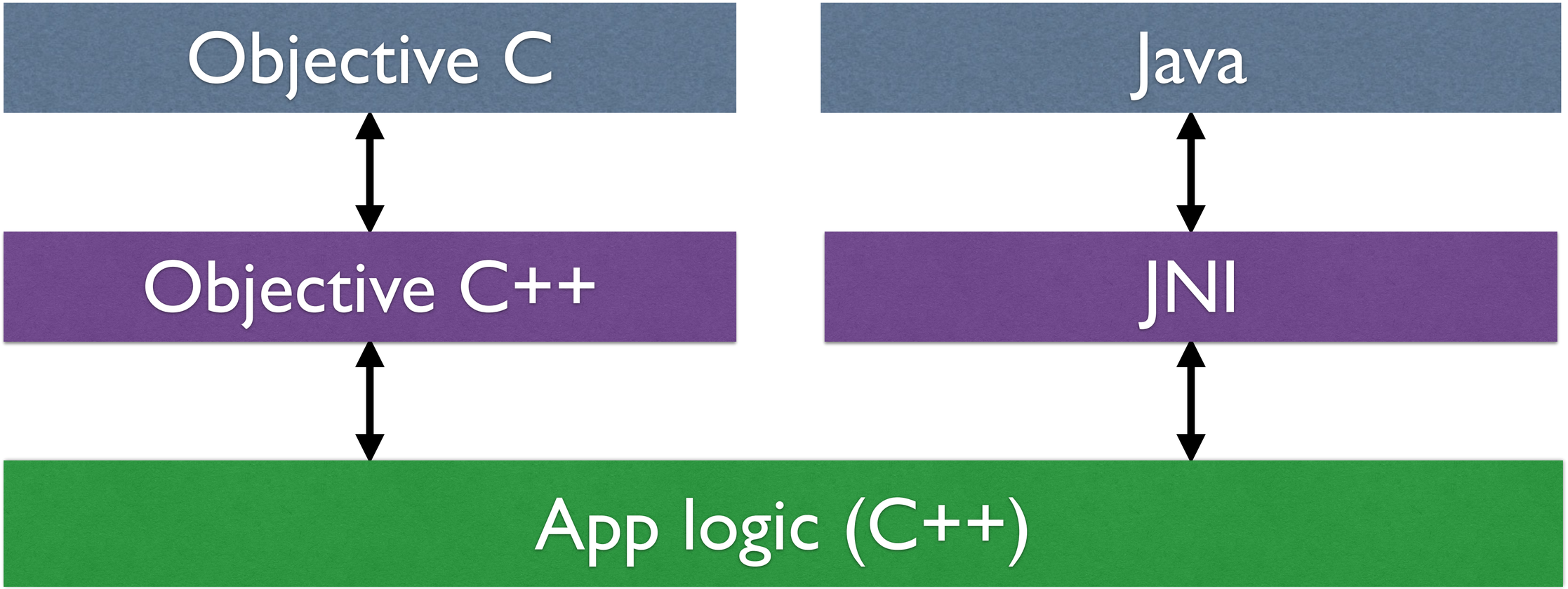
- Cross-platform development at Dropbox
- Mobile focused, but desktop and server are on our radar



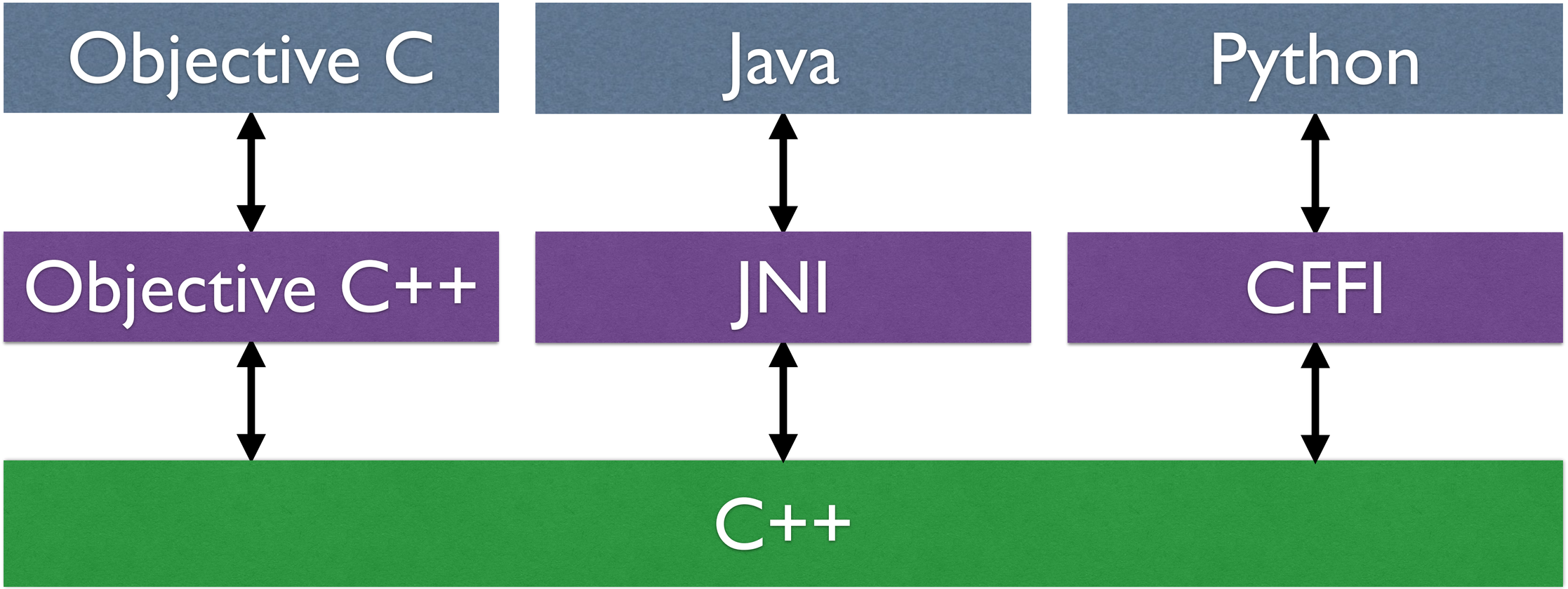
Our Cross-Platform Architecture



Supported Languages



Supported Languages



Today's non-Agenda

- Not going to repeat last year's talk
  - Why and how we created this architecture
  - How we use Djinni to do it
  - Check it out yourself:
    - bit.ly/djinnivideo, bit.ly/djinnitalk
- Not just about Djinni
  - These are techniques you can use anywhere

Today's Agenda

- Quick review: Language bridging with Djinni
- Bridging a new language: Python
  - I think our experience generalizes to other languages
- Deep dives
  - Interesting implementation techniques from all languages

What is Djinni for?

- Say you're following our architecture, on Android & iOS
- You want your UI code to call a common implementation
- Your interface might be something like this

```
struct SomeInfo { /* … */ };

class MyModel {
    void do_stuff(const SomeInfo & info);
};
```

What do you do?

- (2 files) C++: Implement `MyModelImpl` class with one method
- (1 file) Java: Call from your UI code
- (1 file) ObjC: Call from your UI code
- (1 file) Djinni IDL: describes your interface

- With Djinni, you had to write 4 things
  - 3 of them are the custom code you have to write
  - 1 is IDL to define what you want the rest to be

What does Djinni do for you?

- (3 files) `MyModel` abstract class (C++/Java/ObjC)
- (3 files) `SomeInfo` data class (C++/Java/ObjC)
- (4 files) JNI bridging: Java -> C++
- (4 files) Objective-C++ bridging: ObjC -> C++

Djinni's Principles

- Developers interact with natural-looking code in all languages
- Method calls pass control between languages
  - Not serialized RPC
- Callable interfaces can be referenced across languages
- Data data is copied across languages

Djinni's Custom Types

- **Interfaces:** like abstract base classes
  - Implement in any language, own and call from any language
- **Records:** like structs
  - Immutable data, implemented for you, marshaled by copy
  - Contain data (not records) by value (no recursion)
- **Enums:** like scoped enums
  - Same values defined in all languages

Other stuff...

- Primitive types: numbers, strings, bytes
- Containers, Optionals
- Static methods
- Constants
- Extensible and external types
- ...
- See GitHub for more

```
# An enum for specialized use.
wish_difficulty = enum {
    easy;
    medium;
    hard;
}

# A simple data struct.
wish = record {
    difficulty: wish_difficulty;
    request: string;
} deriving (eq, ord)

# A platform service, called by C++
djinni = interface +j +o {
    const max_wishes: i32 = 3;

    grant_wish(my_wish: wish): bool;
    past_wishes(): set<wish>;

    # Factory method
    static rub_lamp(): djinni;
}
```

Sync Progress

- Djinni overview
- **Adding a new language (Python)**
  - Problems presented by bridging to a new language
  - Solutions we used for C++ to Python
- Implementation techniques

Basic Steps of our Approach

- Pick a representation for IDL types
- Pick a bridging technology
- Does it meet the basic requirements?
- Build features on top of those basics

Step 1: Python Types

- Be idiomatic. Generated classes should look like Python.
- Use native Python types the programmer expects.
- Support the expected flexibility (duck typing).
- Support both Python 2, and Python 3.

Python Types: Basics

- Numbers: integer/long, float
  - Looser size constraints than Djinni
- Containers: list, dict, set
- Strings: unicode (2.7) or str (3.x)
- Bytes: str (2.7) or bytes (3.x)
- Optional: determines if `None` is legal

Python Types: Records

- Python class with named fields
- Python only has inclusion by reference
- `__init__` is all that defines the fields

```
class SomeInfo:
    """ Info used by my data model. """

    def __init__(self, user_id, obj_id, color):
        self.user_id = user_id
        self.obj_id = obj_id
        self.color = color
```

Python Types: Enums

- Enum type introduced in Python 3.4
  - Available back-ported to Python 2
- IntEnum also acts like an int, for compatibility

```
@unique
class Color(IntEnum):
    Red = 0
    Green = 1
    Blue = 2
```

Python Types: Interfaces

- Python class expected to have specific methods
- Enforceable abstract base class introduced in Python 3
  - Available back-ported to Python 2

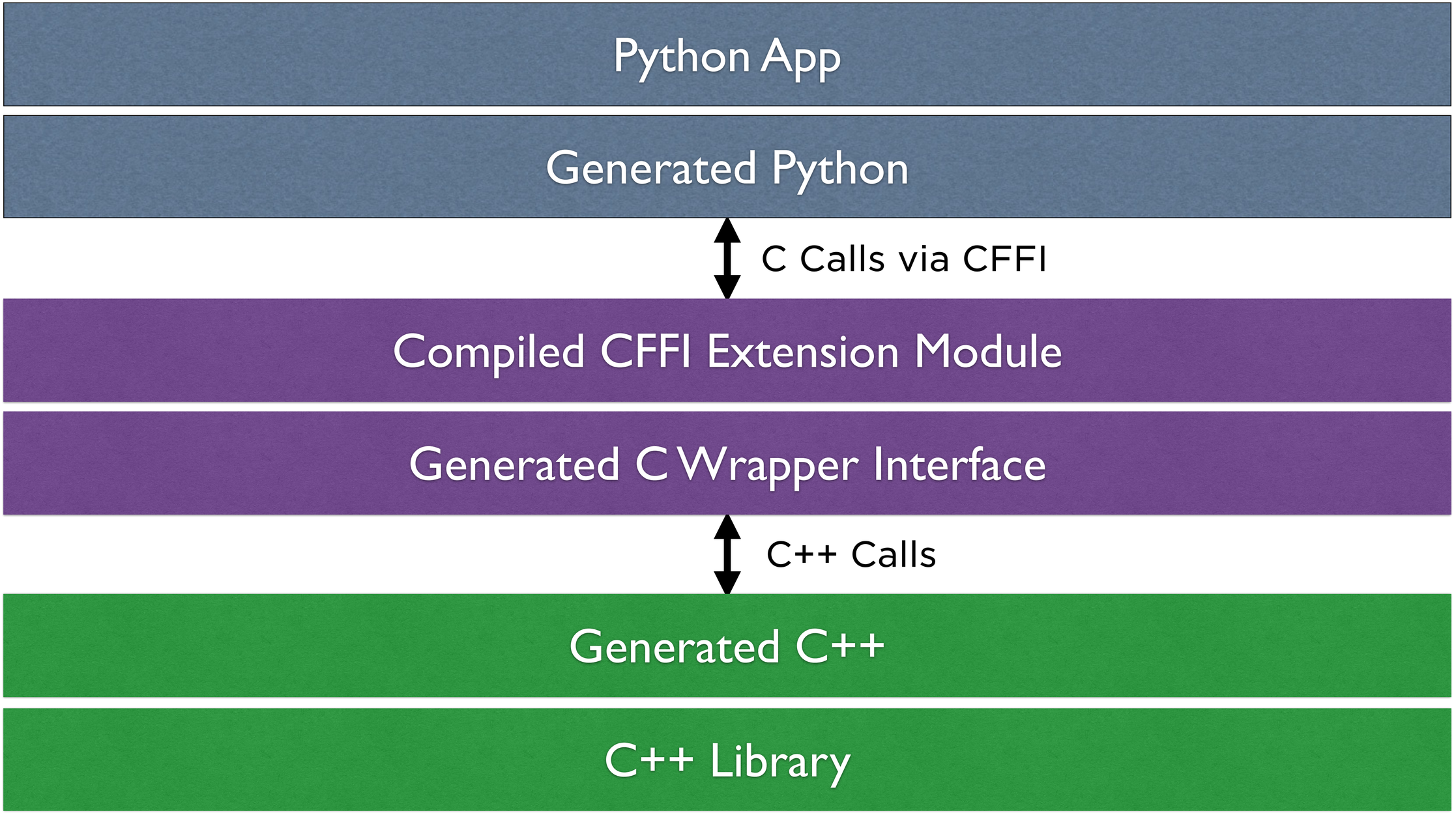
```
class MyModel(with_metaclass(ABCMeta)):
    """ Represents the data model of my app. """

    @abstractmethod
    def do_stuff(self, info):
        """ Does something with the given info. """
        raise NotImplementedError
```

What about duck typing?

- We C++ programmers like our strict type checks.
  - Some Python programmers… not so much.
- We provide tools so that user-defined code can **be** Djinni types.
- But everything will work if user code **acts like** Djinni types.

Step 2: Pick a Bridging Tech

- We use CFFI: “C Foreign Function Interface”
  - Well accepted and supported
  - Supports CPython & PyPy
  - Supports Python 2 and 3
- Generates efficient compiled code
- Has multiple modes: we use “API Mode, Out-of-Line”



CFFI at Build Time

- C++ code and C wrappers linked into a library (libfoo.dylib)
- Djinni feeds CFFI declarations of our C wrapper functions

```
void cw__foo_setmsg(DjinniWrapperFoo * self,
                    const char * msg)
```

- CFFI generates C code to call them from Python
- Compiled into a Python extension (foo_cffi.so)
- Result: Compiled binary optimized for interacting with Python
  - Specific to a version and platform

CFFI at Run Time

- Load the compiled extension like any Python module

```
from foo_cffi import lib
```

- Call C functions from Python

```
lib.cw__foo_setmsg(cself, 'hello')
```

- Our generated Python classes hide this behind a proxy

```
foo.setmsg('hello')
```

Step 3: Basic Bridging

- How do you call Python → C++?
- How do you call C++ → Python?
- How do you pass data?
- How do you reference an object from the other language?

CFFI Calls: Python → C++

- Easy after the build-time preparation

```
lib.cw__foo_setmsg(cself, 'hello')
```

CFFI Calls: C++ → Python

- Trickier since C++ can't speak to CFFI directly
- Python can create a callback for use by C++
- Passed to C++ as function pointer (stored for later use)

```
@ffi.callback('void *(const char *)')
def log(msg):
    print(msg)

lib.cw__foo_addcallback_log(log)
```

- We can do this during Python module load

CFFI Data

- CFFI can understand all C types
  - primitives, arrays, structs, pointers, ...
- We use:
  - primitives: `bool`, integers, floating-point
  - Pointers to opaque structs
- Wrapper structs are declared to CFFI but not defined
- All creation and manipulation of structs happens in C++

Objects: C++ → Python

- Wrap it in an opaque struct
- Pass a pointer to Python
- CFFI holds it as a `c_data` object

Objects: Python → C++

- `ffi.new_handle(obj)` creates a `c_data` from a Python object
- Pass to C++ as `void*` (actually we cast to `PyObjectHandle*`)
- C++ can pass it to a callback
- Recover the Python object with `ffi.from_handle(ptr)`
- Object lifetime is tricky - we'll come back to that

Step 4: Bridging Features

- Everything else can be built on those basics
- Primary required features for Djinni
  - Proxies for Interfaces
  - Marshaling structured data
  - Object ownership across languages
  - Handling exceptions

Proxy Objects

- A proxy is an interface object usable in language A, which stands in for an object in language B
  - Separate proxies for each direction
- Example: `MyModel` is an interface, so `MyModelCppProxy`
  - Is a Python object of Python class `MyModel`
  - Owns a reference to a C++ object of C++ class `MyModel`
  - Forwards all method calls across the bridge to the other object
- Deep dive into proxy management coming later

Marshaling Structured Data

- Marshaling takes multiple steps
  - Construct string
  - Construct record1
  - Put it into a list (and repeat)
  - Put that into record2
- But don’t copy at every step

```
record1 = record {
    s: string;
}

record2 = record {
    list1: list<record1>;
}
```

How do we minimize copies?

- C++ always controls the marshaling
  - C++ understands by-value and move
  - Python allows incremental construction of class fields

- Generate C++ to marshal a whole object either direction
- Call into Python for sub-parts as necessary
- Generator creates helper callbacks for each part

Python → C++

- Python passes reference to whole object
- C++ requests sub-parts as needed
  - One call per field of a record
  - Or iterator-style calls for a containers
- Constructs object all at once
- Move or RVO avoids copies between steps

C++ → Python

- C++ asks Python to create an empty object
- C++ passes sub-parts one at a time
  - Add new class fields dynamically
  - Or add to a list/dict/set
- When done, pass full object to Python
- Everything in Python is by-reference, so no copying

Object Ownership

- No way to directly hold ownership between languages
- C types cannot express ownership transfer
- **Our golden rule:** An object passed across the language boundary comes with ownership
  - (Except for a few special cases for internal helpers)

Implementing Ownership

- Once ownership is passed, it requires an explicit call to delete
- Automated via RAII wherever possible
  - unique_ptr with custom deleters
  - Helper classes and `with` blocks in Python

Explicit Ownership in Python

- Python's GC will try to delete an object we passed to C++
- Need to delay GC until C++ deletes it
- We use an extra global in Python
  - `ffi.new_handle(obj)` creates `c_data` object
  - Passed to C++ as a `void*`
  - Stored in a per-class `c_data_set` to keep it alive
  - Removed when a deleter callback is called

Explicit Ownership in Python

- `handle = fi.new_handle(obj)`
  - creates `c_data` object
- Passed to C++ as a `void*`
  - Python's GC doesn't track this.
  - Will delete the object! (Both `obj` and `handle`)
  - Need to delay GC until C++ deletes
- Store `handle` in a global per-class `c_data_set` to keep it alive
- Remove it in the deleter callback

Exclusive Ownership

- Most objects have unique ownership as they cross the bridge
  - Requires a callback to `delete()`
- C wrapper structs held by Python (all temporary)
  - `DjinniString`, `DjinniBinary`
  - `BoxedI32`, `BoxedF64`, etc.
- Python objects held by C++ (via handle)

Shared Ownership

- Interface objects need to cross the bridge more than once
  - Require ref-counting via calls to `inc_ref()` and `dec_ref()`
- C wrapper struct `DjinniWrapperMyModel`
  - Has its own ref-count for references from Python
  - Holds a `shared_ptr<MyModel>` for references from C++
  - `shared_ptr` might point to a C++ object, or a `PythonProxy`
  - Proxy holds unique ownership of a Python object (handle)

Propagating Exceptions

- Without help, CFFI will ignore exceptions, or crash
- Bridge code must catch, marshal, and re-throw
- C doesn't have exceptions, so we need a way to pass them
- We store the Exception in a per-thread state
  - Like `errno` or `GetLastError()`
  - But generated code will never forget to check

Exception State

- Per-thread variable holding an exception (or not)
  - Contains a Python exception
  - Held in C++ (by handle)
- Only populated while crossing the boundary
  - Checked and cleared after each call
- Symmetric code in both directions

Exception Responsibilities

- Callee
  - try/catch around full implementation
  - In catch block, set exception state and return
- Caller
  - After return, check exception state
  - If found, clear state and throw

Marshaling Exceptions

- Djinni provides simple translation by default
- You can plug in your own custom version
- The model is largely the same as Java
  - I'll describe it in a deep dive later

Python Wrap-up

- The rest can be built on this foundation
  - Hand-written marshaling helpers
  - Global proxy cache
  - Derived comparison/hash operations
  - Static methods, constants, ...
- Python is mostly complete, but still experimental
  - Check it out on the python branch on GitHub
  - https://github.com/dropbox/djinni/tree/python

Sync Progress

- Djinni overview
- Adding a new language (Python)
- Implementation techniques
  - **Proxy caching for identity semantics**
  - Nullability
  - Customized Java exception translation

Proxy objects

- We generate interfaces in each language
- Also generate implementations that perform cross-language calls
  - Java calling C++
    - `native void foo();`
    - `void Java_pkg_Widget_foo(JNIEnv *, jobject) { … }`
  - C++ calling Java

```
weather = record {
    . . .
}

# An object that receives weather reports
weather_listener = interface +j +o {
    weather_report(date: date, forecast: weather);
}

# A service for subscribing to weather reports
weather_service = interface +c {
    add_listener(listener: weather_listener);
    remove_listener(listener: weather_listener);
    # ...
}
```

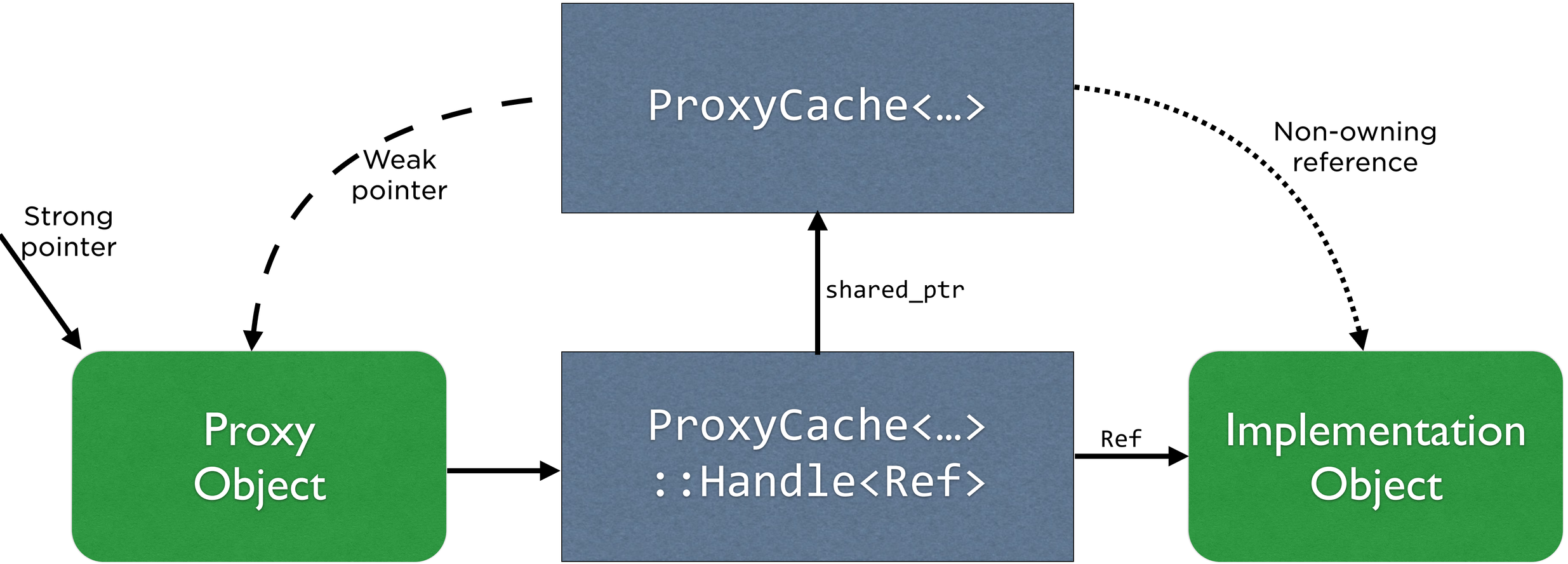
```
class MyWeatherService : public WeatherService
{
public:
    void add_listener(const shared_ptr<WeatherListener> & listener) {
        m_listeners.insert(listener);
    }
    void remove_listener(const shared_ptr<WeatherListener> & listener) {
        m_listeners.erase(listener);
    }

private:
    set<shared_ptr<WeatherListener>> m_listeners;
};
```

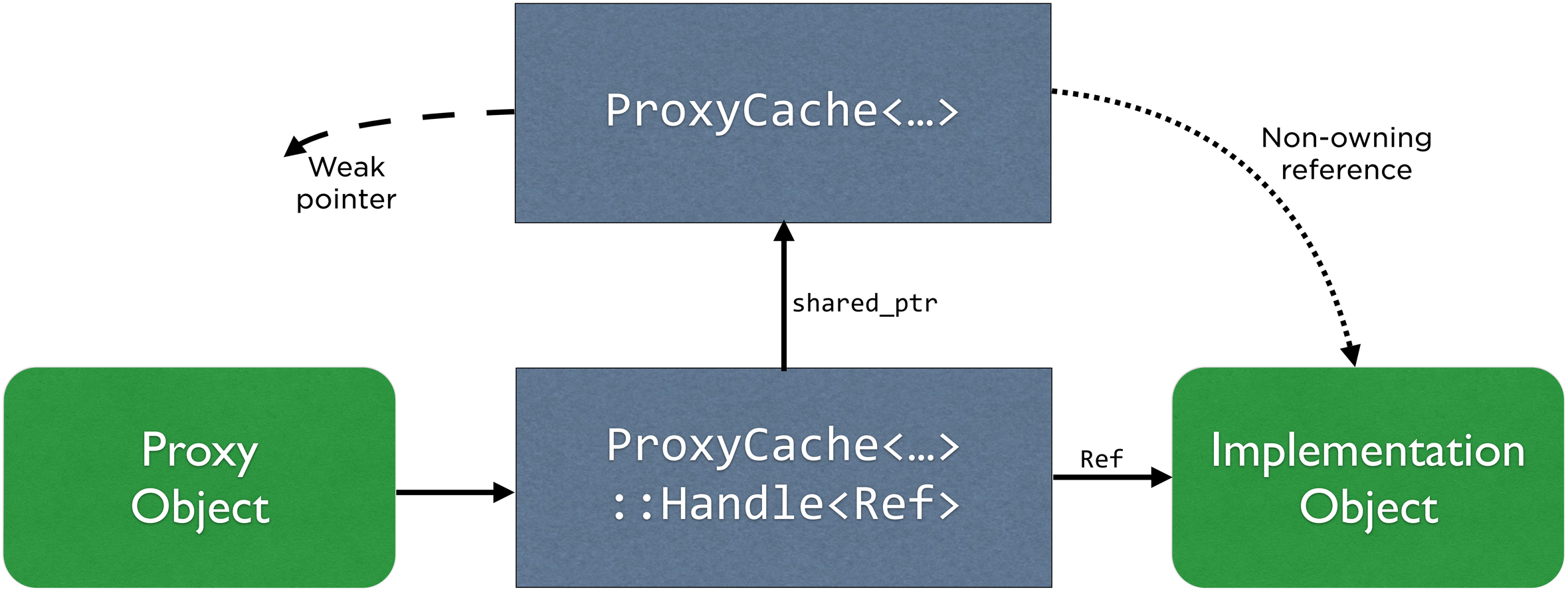
Proxy caching

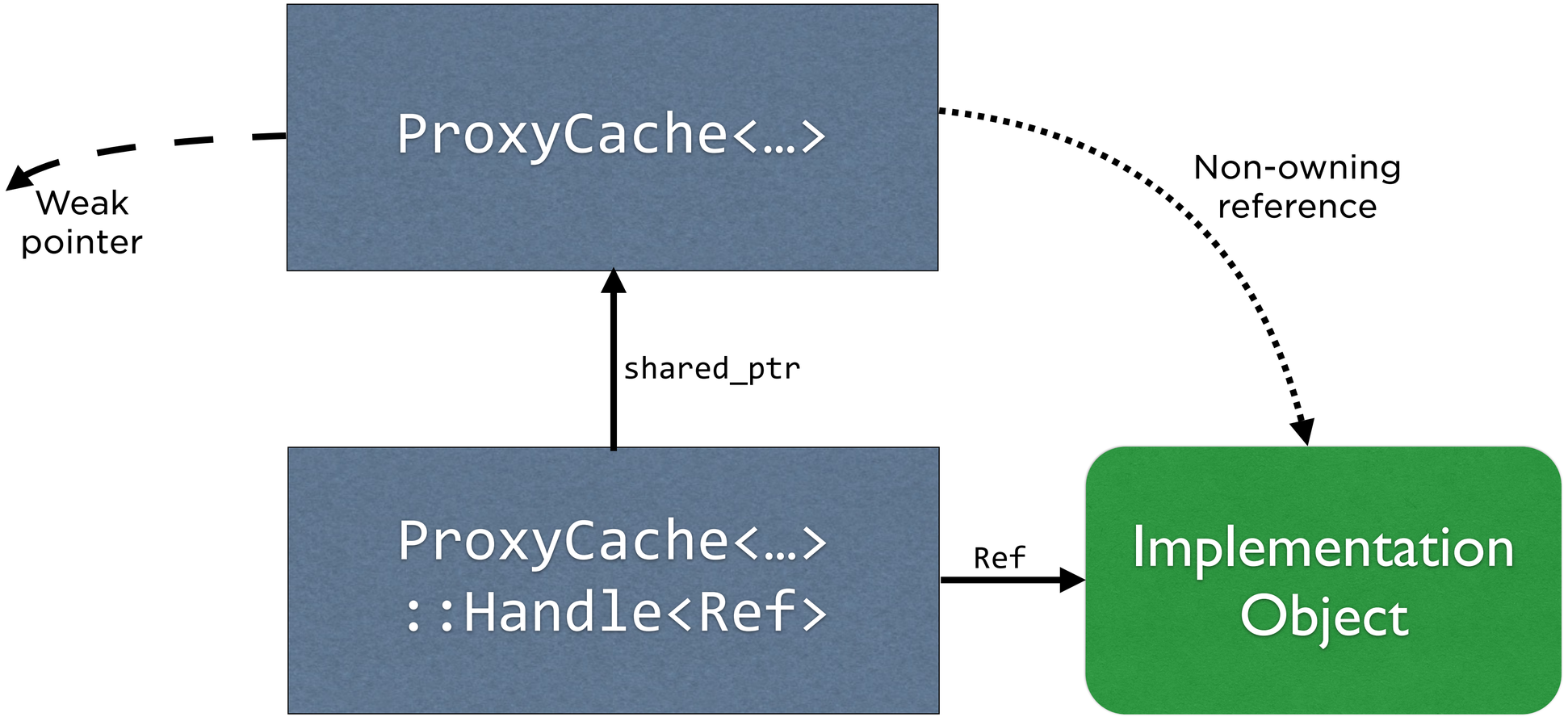
- When we call `set::erase`, we need the *same* listener object, even though we made two different calls through the bridge
- Performance benefits too, but mostly this is for correctness
- Need to enable idioms that developers want to use
- Solution:
  - Keep a reference to proxy objects, and reuse if possible
  - Store map from *non-owning reference* to *weak pointer*

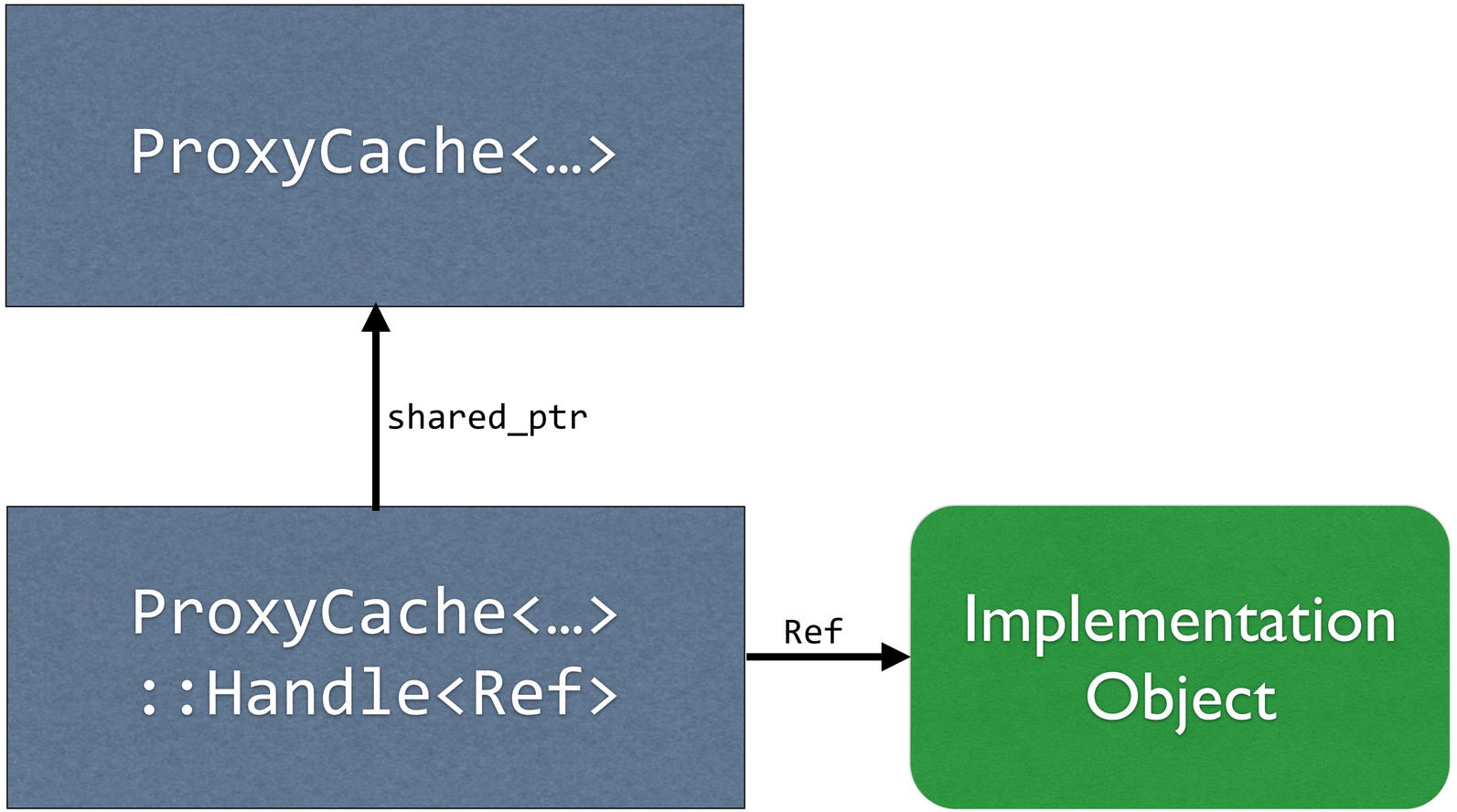
Invariants

- Guarantee: no two wrappers for the same object are simultaneously visible
  - Not LRU
  - Wrappers don’t last forever
  - *Visible*, not *in existence*
    - Weak-pointer expiry determines when we can make a new wrapper





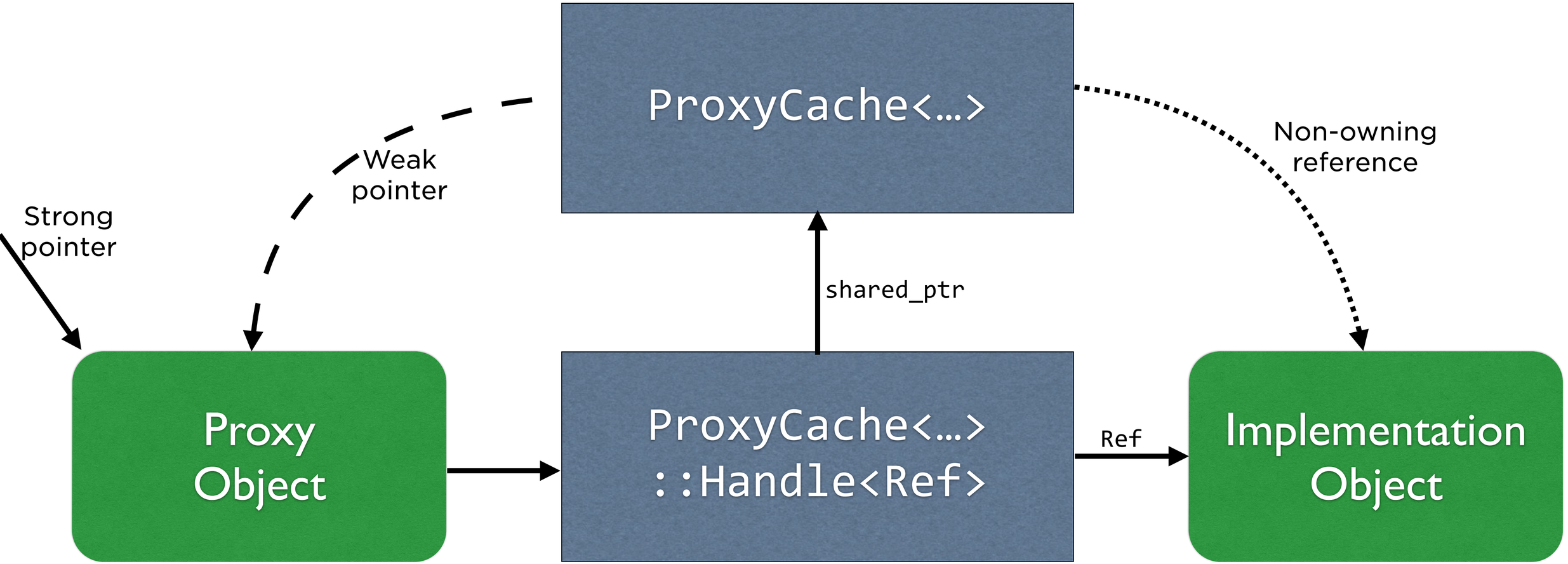


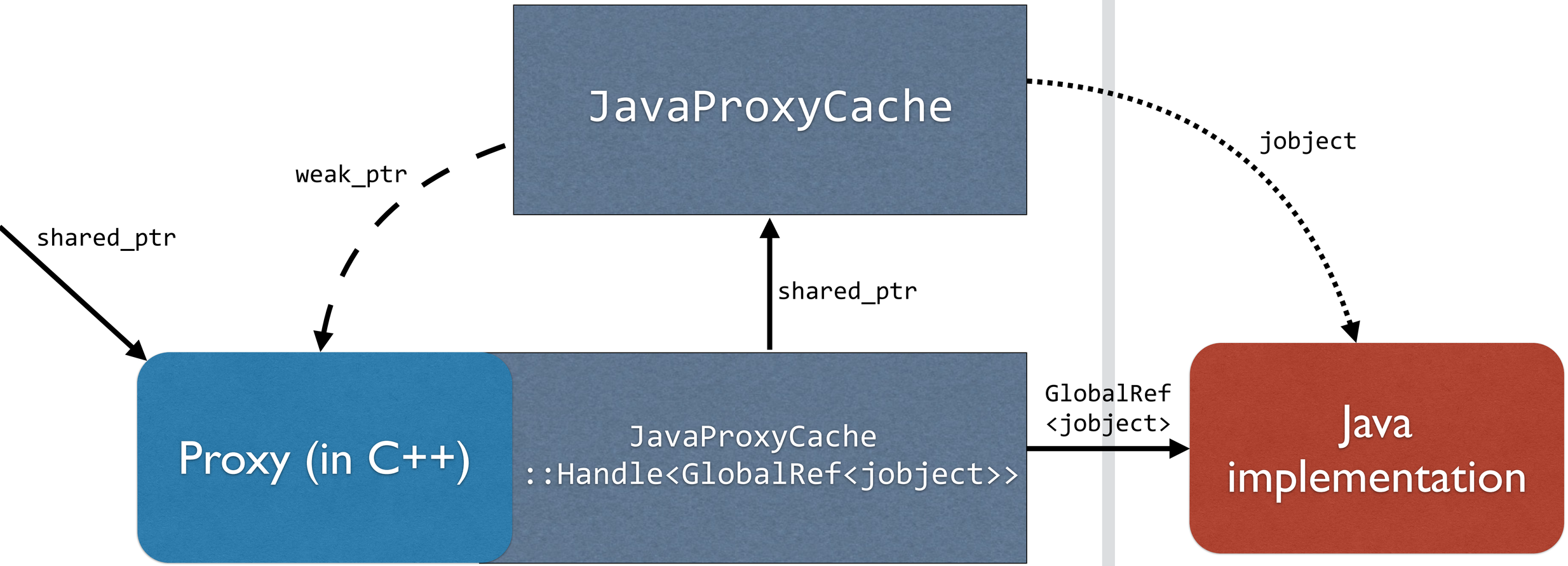


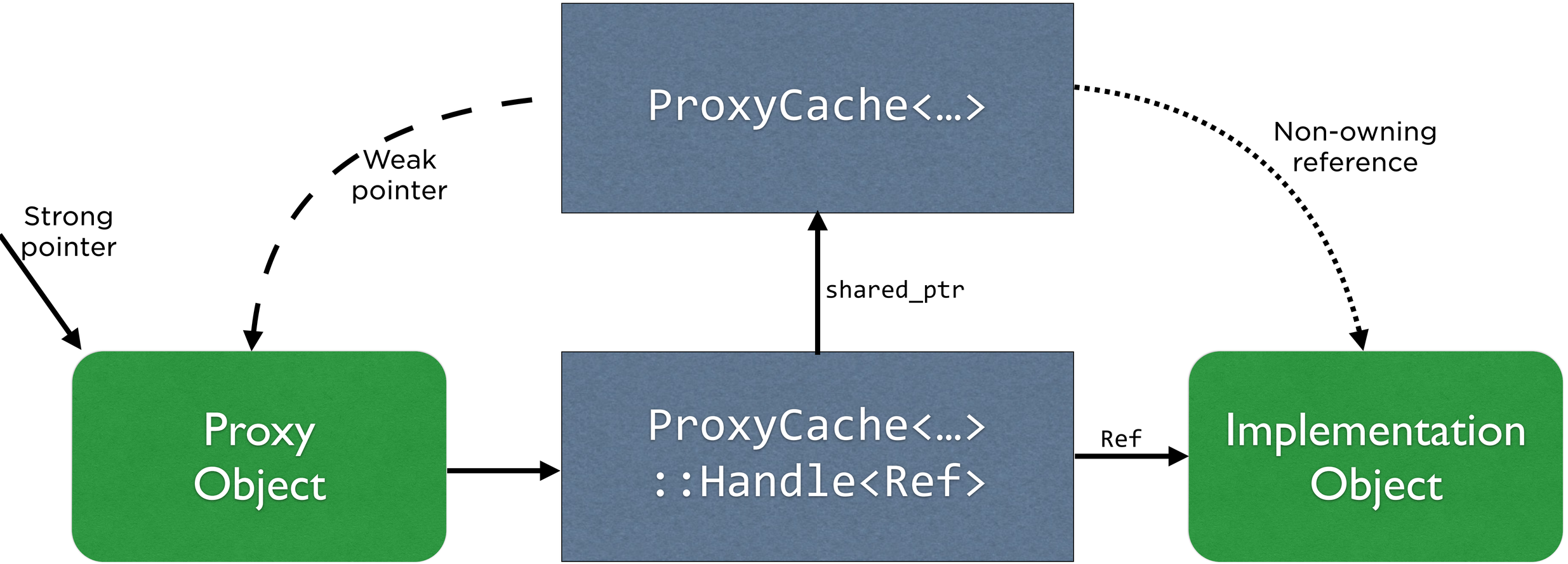
ProxyCache<…>

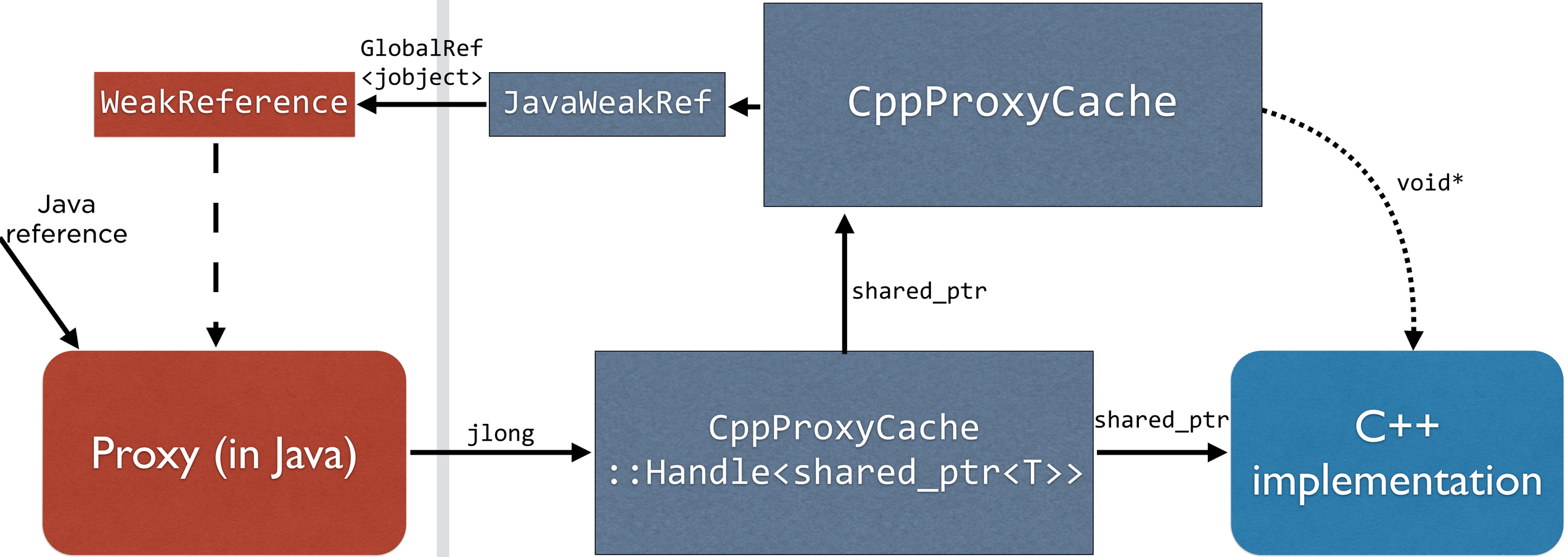
Implementation Object

ProxyCache<…>









Gotchas

- `ProxyCache` is a global singleton
- *But* it has to be held by `shared_ptr`! Why?
- How many kinds of weak reference does Java have?

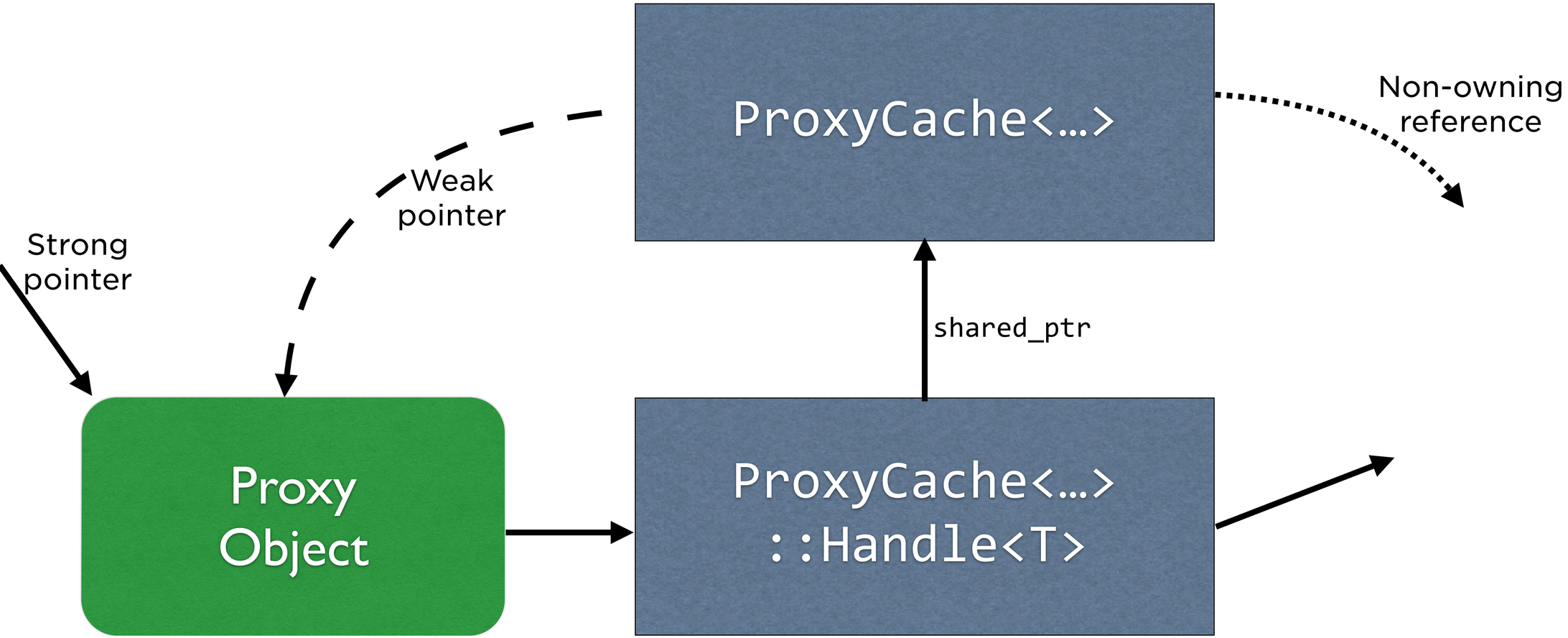
	Java -> C++	C++ -> Java	ObjC -> C++	C++ -> ObjC
UnowningImplPtr	jobject	void *	__unsafe_unretained id	void *
OwningImplPtr	jobject	shared_ptr<void>	__strong id	shared_ptr<void>
OwningProxyPtr	shared_ptr<void>	jobject	shared_ptr<void>	__strong id
WeakProxyPtr	weak_ptr<void>	JavaWeakRef	weak_ptr<void>	__weak id
UnowningImplPtrHash	JavaIdentityHash	std::hash	UnretainedIdHash	std::hash
UnowningImplPtrEqual	JavaIdentityEquals	std::equal_to	std::equal_to	std::equal_to

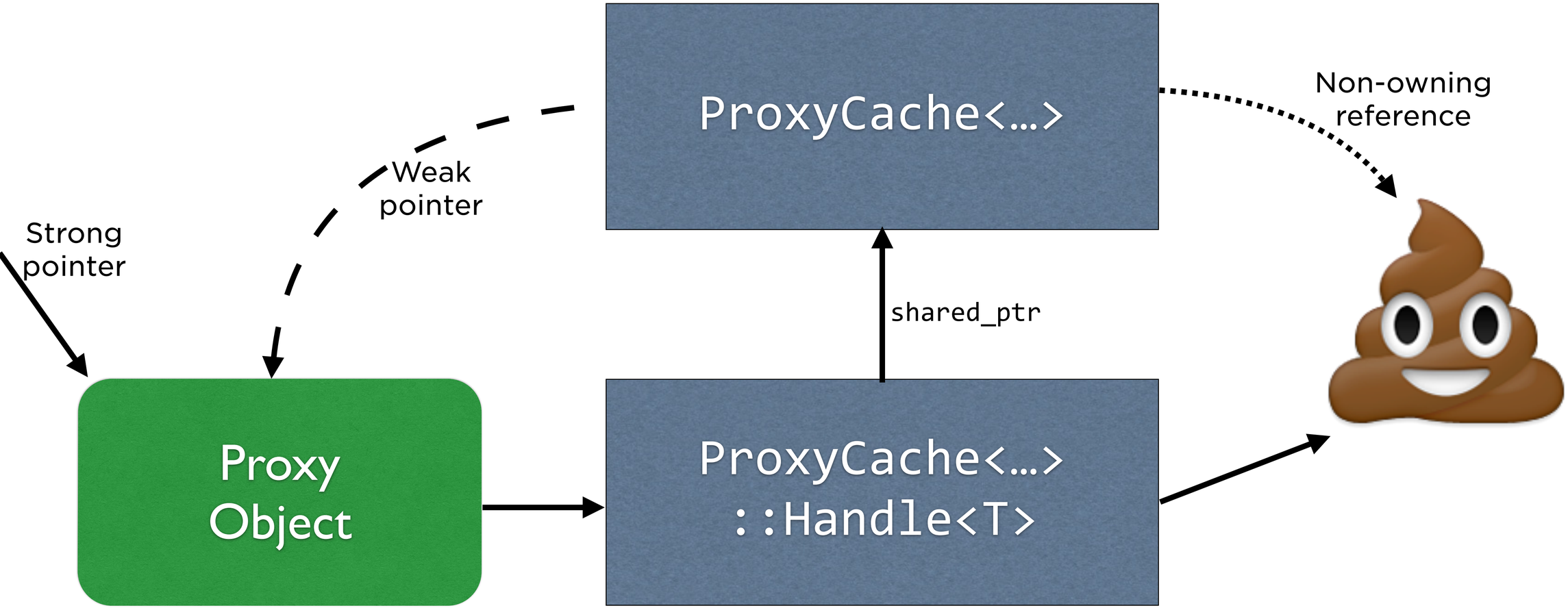
Sync Progress

- Djinni overview
- Adding a new language (Python)
- Implementation techniques
  - Proxy caching for identity semantics
  - **Nullability**
  - Customized Java exception translation

Nullability

- A year ago, we didn’t handle null at all





Nullability

- A year ago, we didn’t handle null at all
- Changed so null was allowed on interfaces
- `optional<interface_type>` becomes `optional<shared_ptr<…>>` ?!
- ObjC and Java both have nullability annotations
  - `@Nonnull`, `@CheckForNull` (et al), `_Nonnull`, `_Nullable`
  - We generated these for data objects and interfaces

Intermediate state

- `Foo`: `@Nonnull Foo`, `_Nonnull Foo*`, `shared_ptr<Foo>`
- `optional<Foo>`: `@CheckForNull Foo`, `_Nullable Foo*`, `shared_ptr<Foo>`
- Generated code includes explicit null checks: ObjC and Java annotations are advisory only

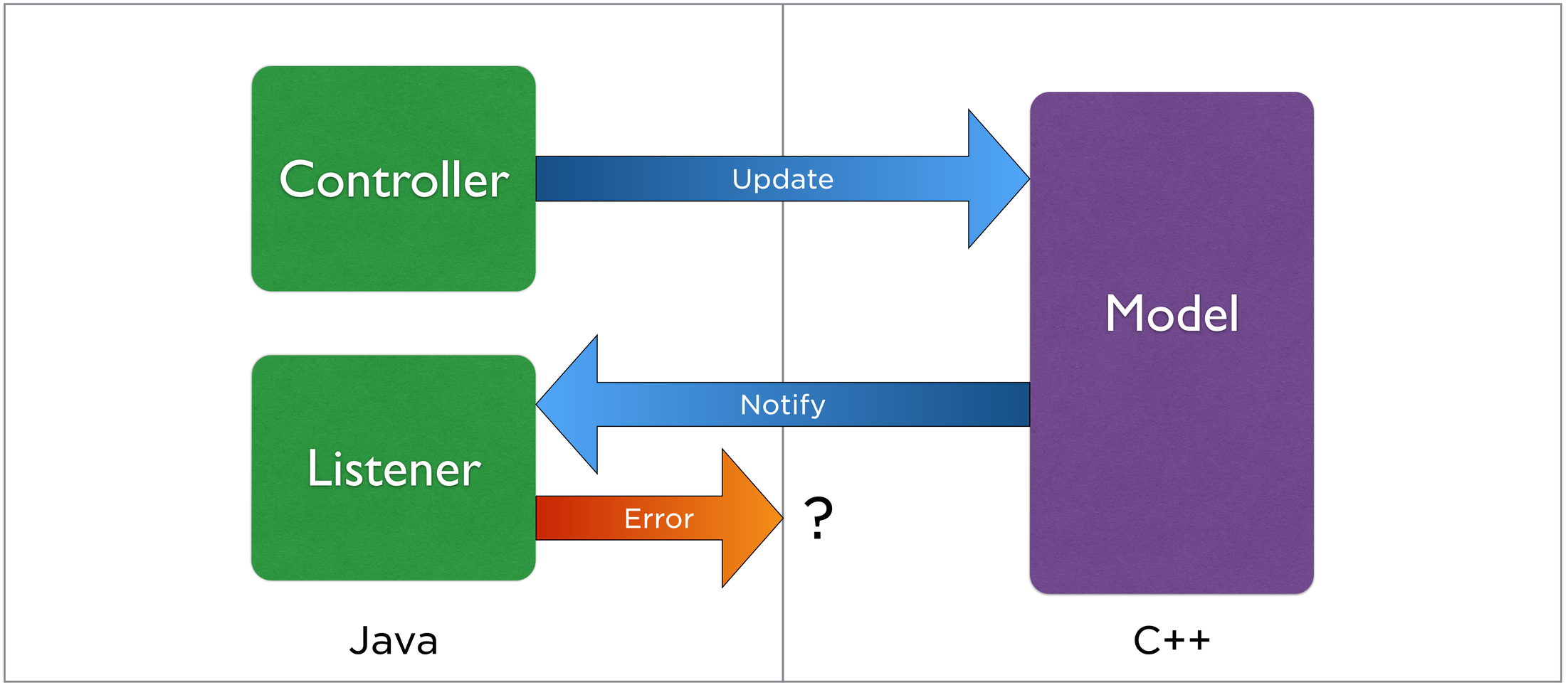
Handling null today

- `Foo` is non-nullable, `optional<Foo>` is nullable
- Can hook into nn (generating `nn_shared_ptr<T>`)
- `nn_shared_ptr<T>`
  - A pointer that has *already* been *explicitly* checked for null
  - (GSL's `not_null` inserts implicit checks)
- For more, see my talk tomorrow, 2:30pm

Sync Progress

- Djinni overview
- Adding a new language (Python)
- Implementation techniques
  - Proxy caching for identity semantics
  - Nullability
  - **Customized Java exception translation**

Exception Translation



JNI Exception Model

- Each thread can have a “pending” exception
- Java side is done for us
  - Catch and Set before return from Java
  - Check and throw after return to Java
- Our generated code must do the other side
  - Check and throw before after return from Java
  - Catch and set before return to Java

Djinni's Default Translation

- After return from Java (Java → C++)
  - Get and clear the pending Java exception
  - Throw `djinni::jni_exception` holding it
- Before return to Java (C++ → Java)
  - Catch `jni_exception`
    - Extract the Java exception
    - Set it as pending again and return
  - For any other C++ exception, create a `RuntimeException`

What if that's not good enough?

- This is free, but simplistic
- What if you have your own custom exception types?
- What if you want stack traces from C++?

Pluggable Functions

- You can replace exception translation with your own version
- Helper functions defined with `__attribute__((weak))`
- Linker will prefer your definition, if you provide it

- For example:
  - You have a `my_security_exception` you use in C++
  - Should translate to/from `SecurityException` in Java

Translating Java → C++

```
void jniThrowCppFromJavaException(JNIEnv * env,
                                  jthrowable java_exception);
```

- Called when a Java exception makes it to C++
- Should always throw some C++ exception

```
void jniThrowCppFromJavaException(JNIEnv * env, jthrowable java_exception) {
    const SecurityExceptionClassInfo & ci =
        djinni::JniClass<SecurityExceptionClassInfo>::get();

    if (env->IsInstanceOf(java_exception, ci.clazz.get())) {
        LocalRef<jstring> jmessage {
            env, (jstring)env->CallObjectMethod(java_exception, ci.getMessage) };

        throw my_security_exception(jniUTF8FromString(env, jmessage.get()));
    }

    throw jni_exception { env, java_exception };
}
```

```
struct SecurityExceptionClassInfo {
    const GlobalRef<jclass> clazz =
        djinni::jniFindClass("java/lang/SecurityException");

    const jmethodID ctor =
        djinni::jniGetMethodID(clazz.get(),
                               "<init>",
                               "(Ljava/lang/String;)V");

    const jmethodID getMessage =
        djinni::jniGetMethodID(clazz.get(),
                               "getMessage",
                               "()Ljava/lang/String;");
};
```

Translating C++ → Java

```
void jniSetPendingFromCurrent(JNIEnv * env, const char * ctx);
```

- Called when a C++ exception makes it to Java
- Called inside a catch block
- Should always set the pending exception and return

```
void jniSetPendingFromCurrent(JNIEnv * env, const char * ctx) noexcept {
    try {
        throw;
    } catch (const my_security_exception & e) {
        const SecurityExceptionClassInfo & ci =
            djinni::JniClass<SecurityExceptionClassInfo>::get();

        LocalRef<jstring> jmessage { env, jniStringFromUTF8(env, e.what()) };

        LocalRef<jthrowable> jexception { env,
            (jthrowable)env->NewObject(ci.clazz.get(), ci.ctor, jmessage.get()) };

        env->Throw(jexception);
        return;
    } catch (const std::exception & e) {
        jniDefaultSetPendingFromCurrent(env, ctx);
    }
}
```

Add Stack Traces

- Can be built using the same two helpers
- Sample code is too big for slides
- I'll give the key points
- I assume Java is always the root (last to catch)
  - It needs to see the full stack trace

Enabling Factors

- Assume you have a way to generate stack traces in C++
- Two things we need to leverage here from Java `Throwable`
  - They can have a cause (another `Throwable`)
  - You can override the stack trace (`StackTraceElement[]`)

Define some helpers

- A C++ a new exception type containing:
  - C++ stack trace
  - `jni_exception` (cause)
- A Java “builder” type for building exceptions
  - Set message
  - Set cause
  - Add stack trace elements (1 at a time)

Java → C++

- Store original Java exception inside your C++ exception
- Fill in the C++ backtrace
- Throw and let it propagate as normal

C++ → Java

- Use your builder to create a cause chain
  - New Java exception (with default stack trace)
  - … caused by pseudo-C++ exception (custom stack trace)
  - … caused by original Java exception
- Set the newly-built exception as pending and return

Other new stuff this year

- New types: f32, date
- External types
- Example extensions
  - Thread creation service
  - JNI Exception customization
- Trackable dependencies
- JetBrains plug-in for .djinni files

Community Links

- Djinni: github.com/dropbox/djinni
- Talk from CppCon 2014:bit.ly/djinnitalk, bit.ly/djinnivideo
- Today's Talk: bit.ly/djinnitalk2, bit.ly/djinnivideo2 (soon)
- Slack Community: mobilecpp.herokuapp.com
- Developer Tutorials: mobilecpptutorials.com
- Contact us: atwyman@dropbox.com, j4cbo@dropbox.com
- Also we're hiring! dropbox.com/jobs