NOKIA

When C++ wants to meet Java

Designing cppJNI framework

- Michał Łoś (michal.los@nokia.com)

© 2017

Agenda

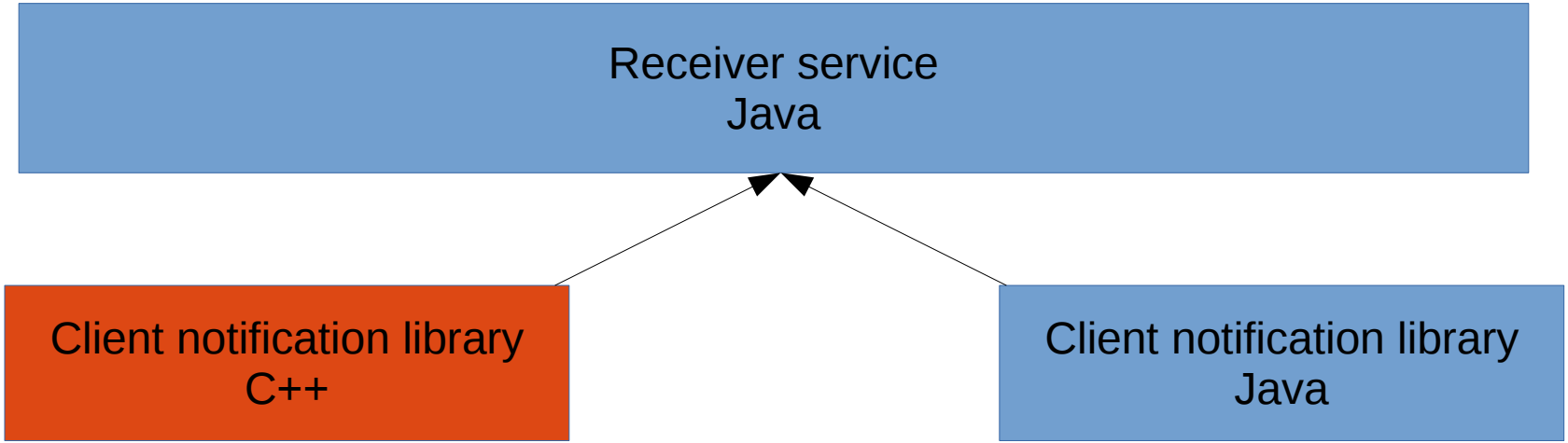
```
//BTW: I hate agendas..
using cppJNIAgenda = mpl::list <
    Motivation,
    InitialDesignDecisions,
    HowDoesThingsWorkInCppJNI,
    Caching,
    CompilationOptimization,
    QandA
>;
```


© 2017

```
using motivation =
    mpl::eager::head<
        agenda
    >;
```

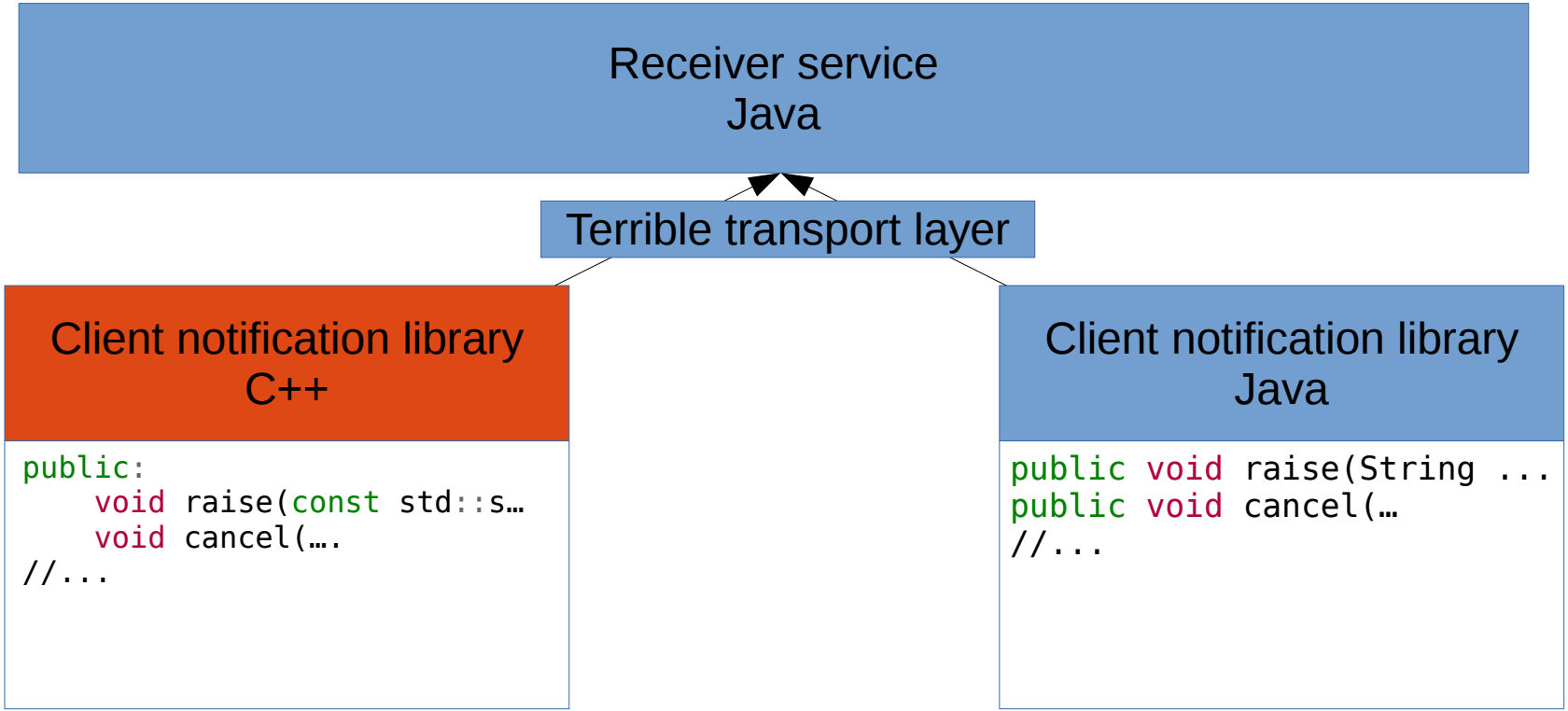

© 2017

Some 15 years ago in some project…



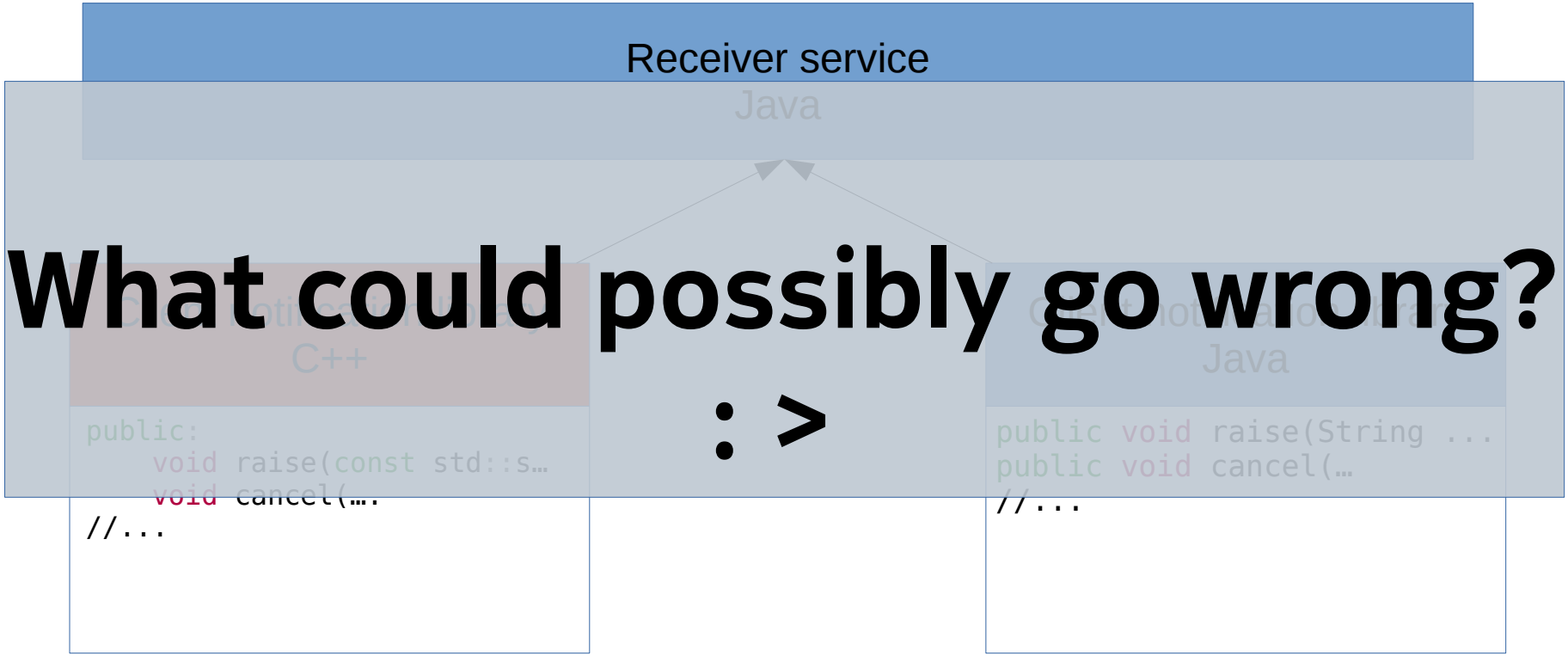
© 2017

Some 15 years ago in some project…



© 2017
NOKIA

Some 15 years ago in some project…

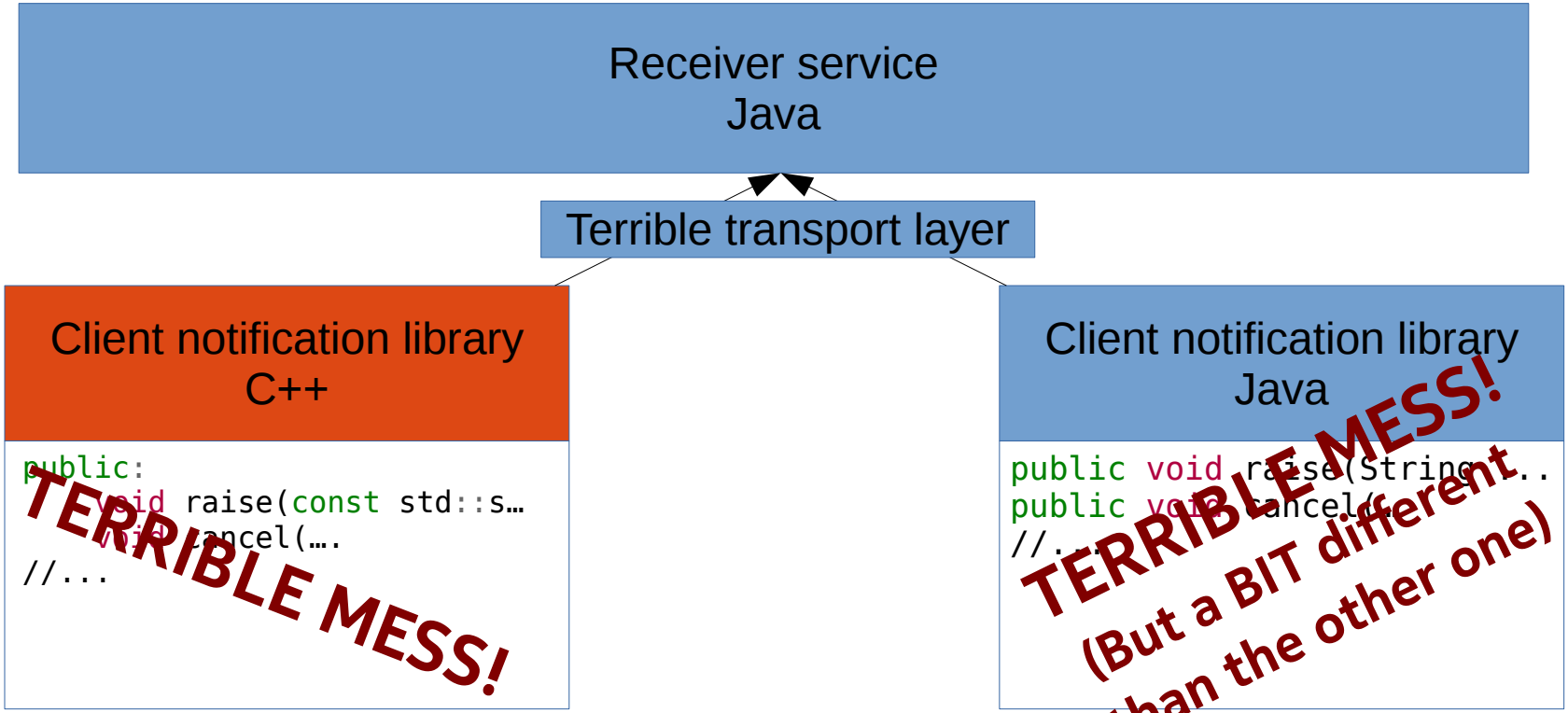


What could possibly go wrong?

: >

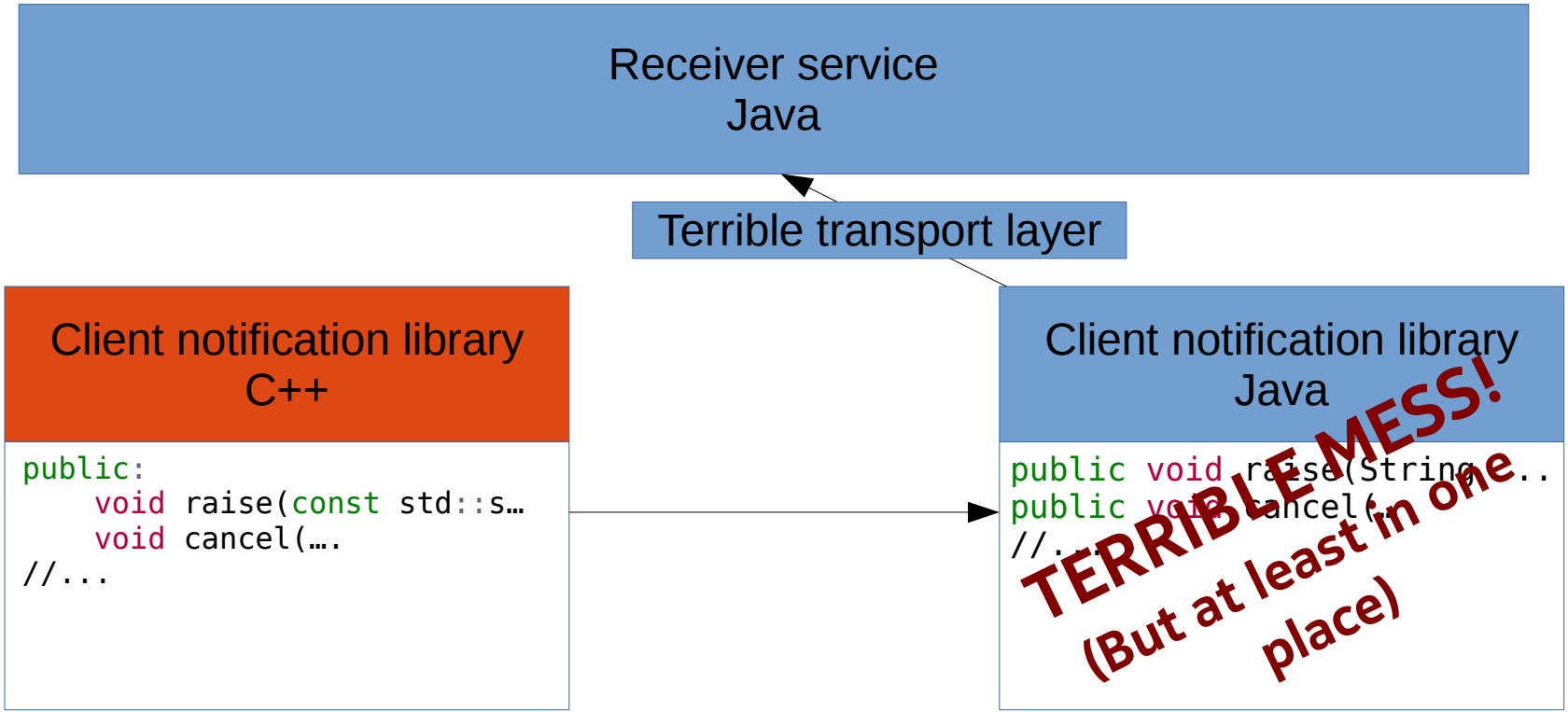
© 2017
NOKIA

Same project recently



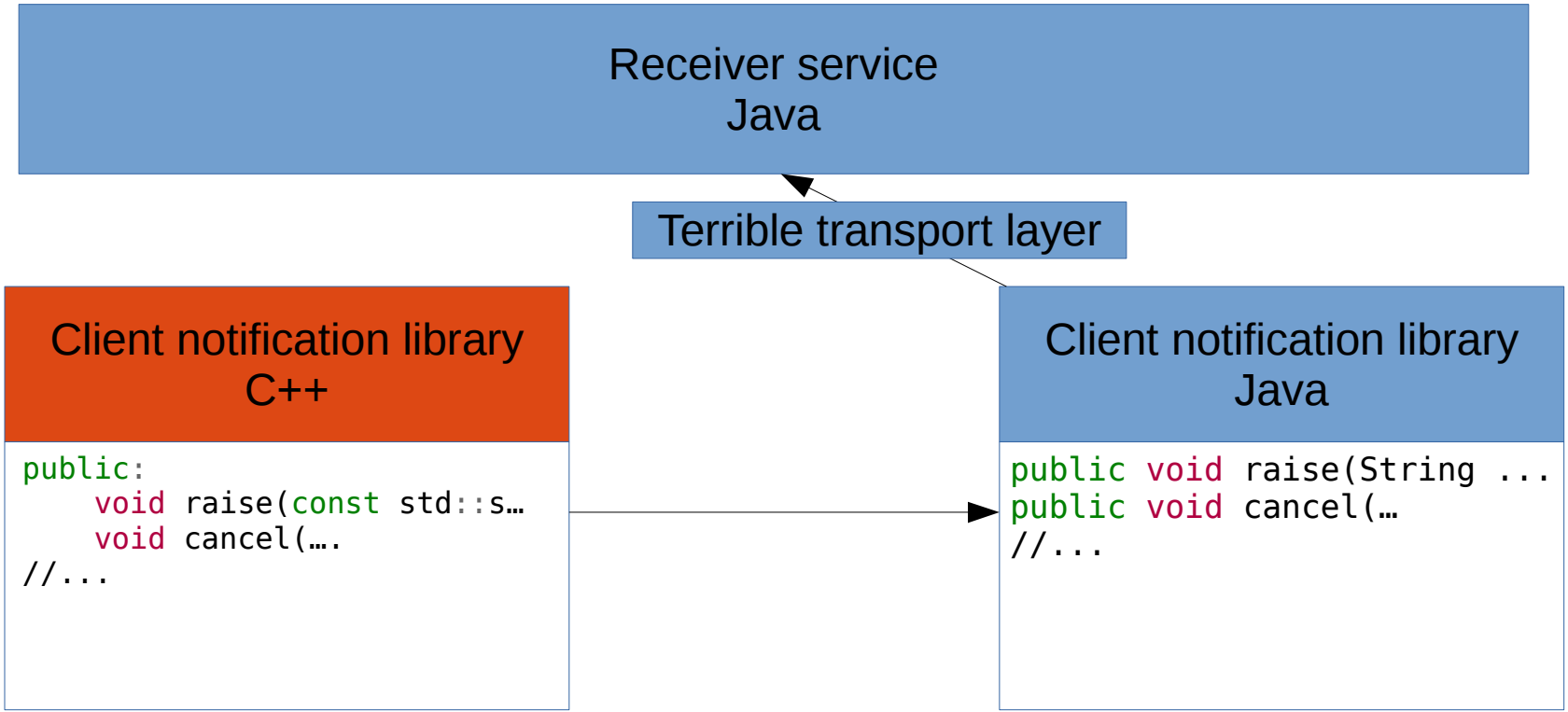
© 2017
NOKIA

Redesign plan



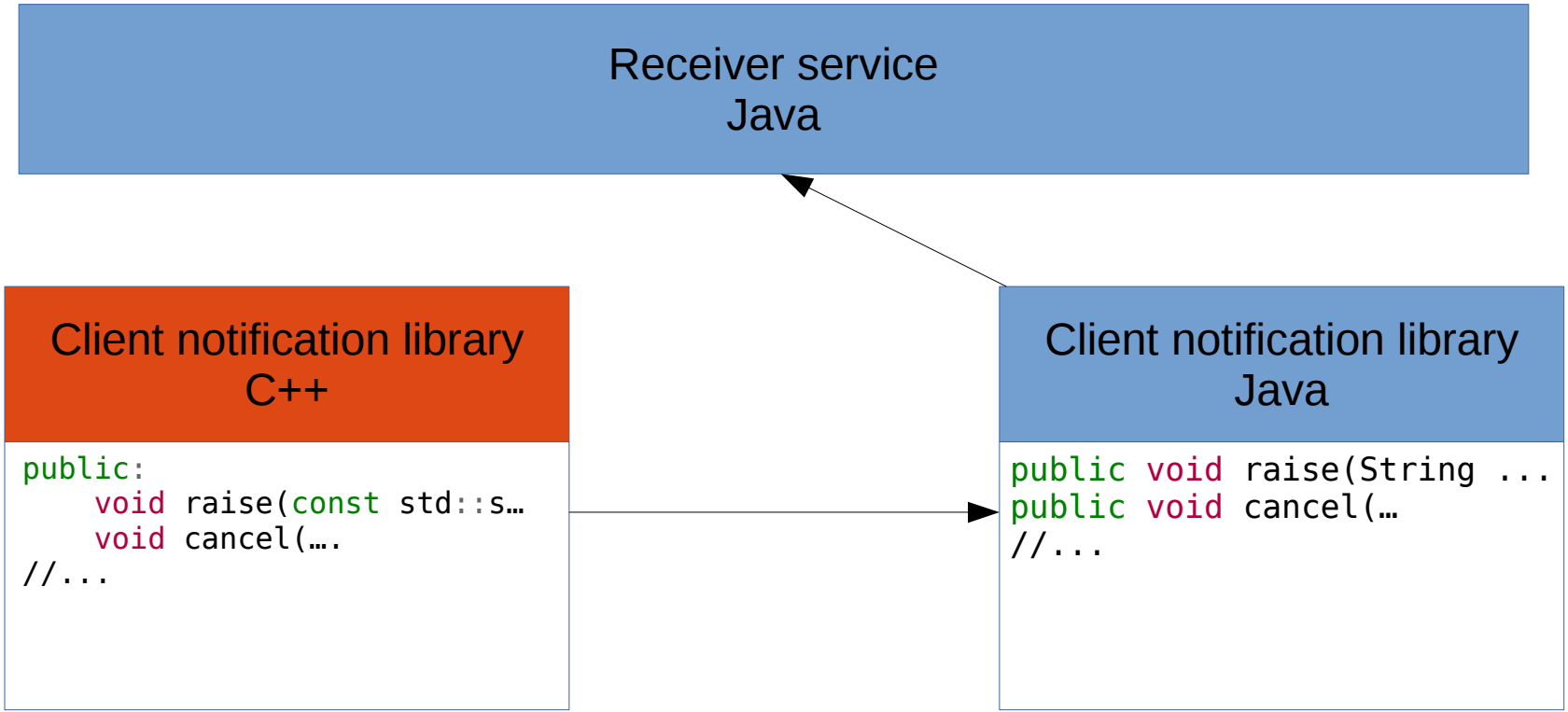
© 2017
NOKIA

Redesign plan



© 2017
NOKIA

Redesign plan



© 2017

How to call Java method from C++ using JNI

```
//given Java code
package some.package;
public class MyClass {
    public void myMethod(String param) {}
    public String myDifferentMethod(String p1, int p2) {
        return ...;
    }
}
```


© 2017

How to call Java method from C++ using JNI

```
//given C++ code
jobject jniObject = ... ; JNIEnv* env = ...;
jstring jstr = env->NewStringUTF("blabla");
jclass clazz = env->FindClass("some/package/MyClass");
jmethodID myMethodID = env->GetMethodID(
    clazz,
    "myMethod", "(Ljava/lang/String;)V");
env->CallVoidMethod(jni_object, myMethodID, jstr);
env->ReleaseStringUTFChars(jstr, nullptr); /*don't forget!*/
```


© 2017

```
using initialDesignDecisions =
    mpl::eager::apply<
        mpl::pipeline<
            mpl::drop<1>, mpl::first_of
        >,
        agenda
    >;
```


© 2017

How to call Java better: Qt for Android

```
QAndroidJniObject jniObject("some/package/MyClass");
QAndroidJniObject jstr =
    QAndroidJniObject::fromString(QString("blabla"));
jniObject.callMethod<jvoid>(
    "myMethod",
    "(Ljava/lang/String;)V", jstr);
QAndroidJniObject result = jniObject.callObjectMethod(
    "myDifferentMethod",
    "(Ljava/lang/String;I)Ljava/lang/String;", jstr, 5);
```


© 2017

How to call Java better: Qt for Android

```
QAndroidJniObject jniObject("some/package/MyClass");
QAndroidJniObject jstr =
    QAndroidJniObject::fromString(QString("blabla"));
jniObject.callMethod<jvoid>(
    "myMethod",
    "(Ljava/lang/String;)V", jstr);
QAndroidJniObject result = jniObject.callObjectMethod(
    "myDifferentMethod",
    "(Ljava/lang/String;I)Ljava/lang/String;", jstr, 5);
```


© 2017

How to call Java better: Qt for Android

```
QAndroidJniObject jniObject("some/package/MyClass");
QAndroidJniObject jstr =
    QAndroidJniObject::fromString(QString("blabla"));
jniObject.callMethod<jvoid>(
    "myMethod",
    "(Ljava/lang/String;)V", jstr);
QAndroidJniObject result = jniObject.callObjectMethod(
    "myDifferentMethod",
    "(Ljava/lang/String;I)Ljava/lang/String;", jstr, 5);
```

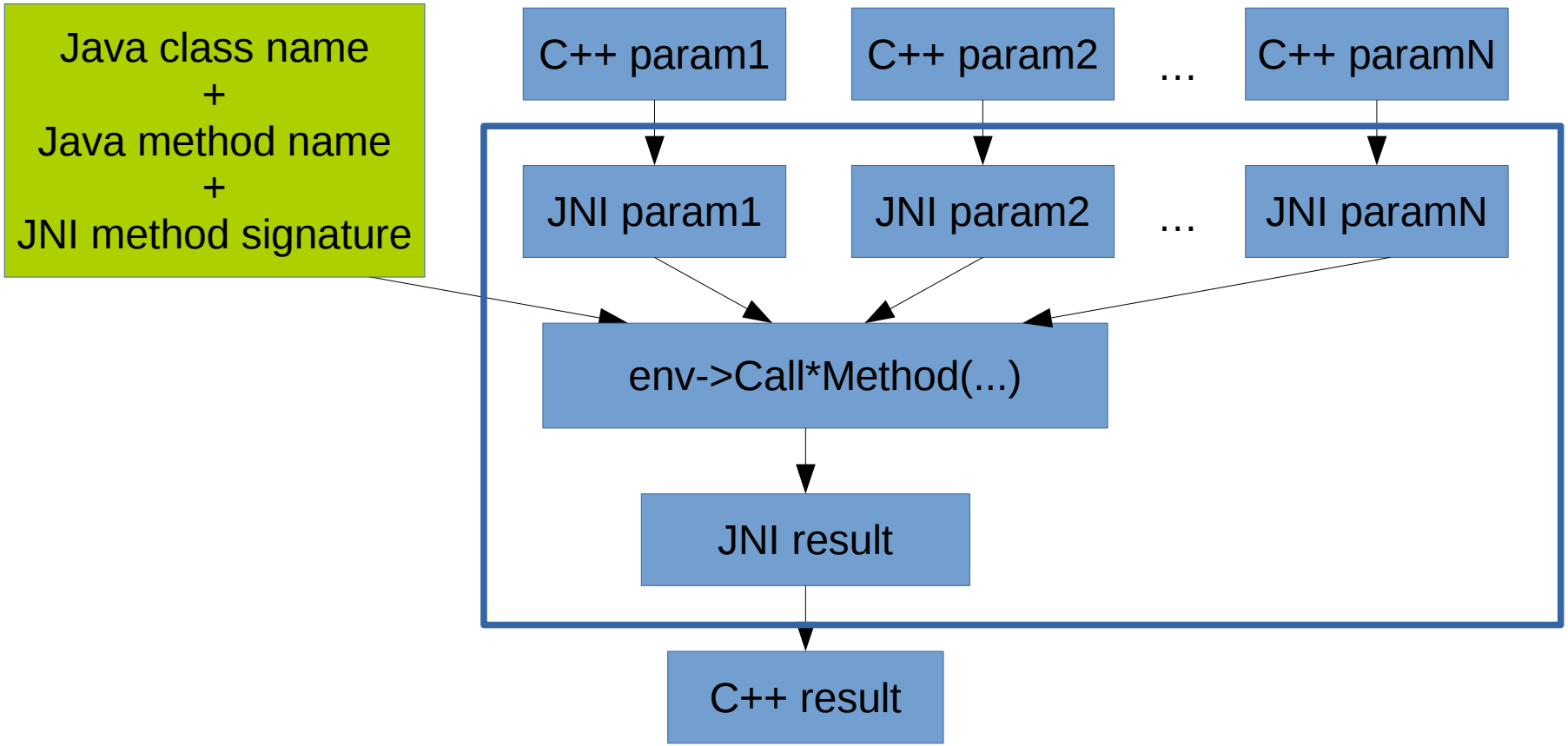

© 2017

How to call Java better: Qt for Android

```
QAndroidJniObject jniObject("some/package/MyClass");
QAndroidJniObject jstr =
    QAndroidJniObject::fromString(QString("blabla"));
jniObject.callMethod<jvoid>(
    "myMethod",
    "(Ljava/lang/String;)V", jstr);
QAndroidJniObject result = jniObject.callObjectMethod(
    "myDifferentMethod",
    "(Ljava/lang/String;I)Ljava/lang/String;", jstr, 5);
```


© 2017

cppJNI: Run-time goals



© 2017
NOKIA

cppJNI: Compile-time goals

- Java class ↔ Java class method
- Java method name ↔ Java method signature
- Generate Java method signature at compile-time
- Do not require external code generator

© 2017

The holy grail!

```
some::package::MyClass obj();

obj.myMethod("blabla");

auto result = obj.myDifferentMethod(str, 5);
```


© 2017

The holy gra... Ok, not so fast.

... // some other way to get *obj* using JVM environment

```
obj.myMethod("blabla");
auto result = obj.myDifferentMethod(str, 5);
```


© 2017

```
using howThingsWorkInCppJNI =
    mpl::eager::apply<
        mpl::filter<
            mpl::is_same_as<
                HowThingsWorkInCppJNI
        >>,
        agenda>;
```


© 2017

Class ↔ method relation

```
struct MyClass
{
    //..
    ? myMethod( ? )
    { ? }
    //..
};
```

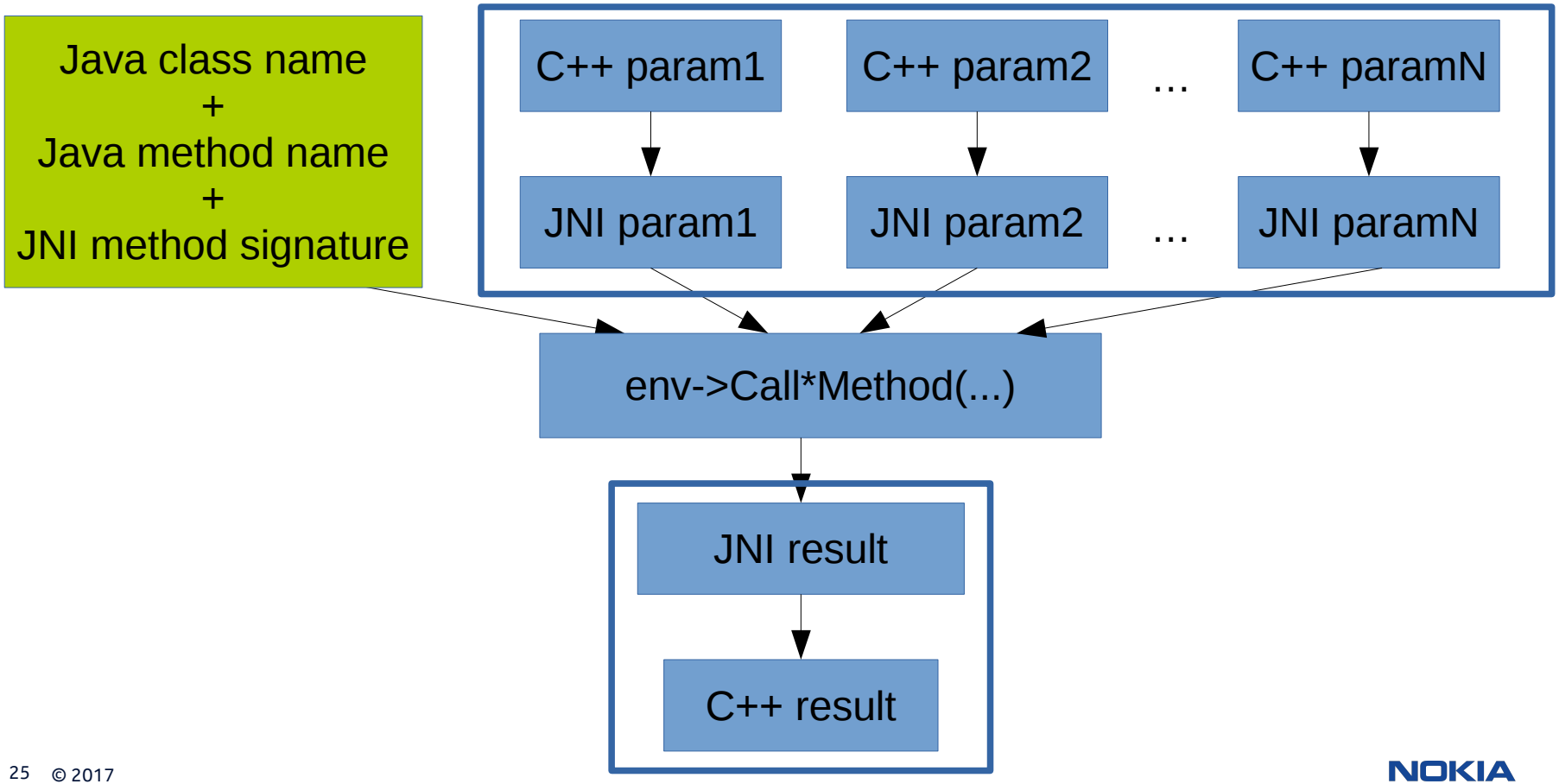

© 2017
NOKIA

Declaring Java method using cppJNI

```
struct MyClass
{
    //..
    void myMethod(const std::string&) const
    { ? }
    //..
};
```

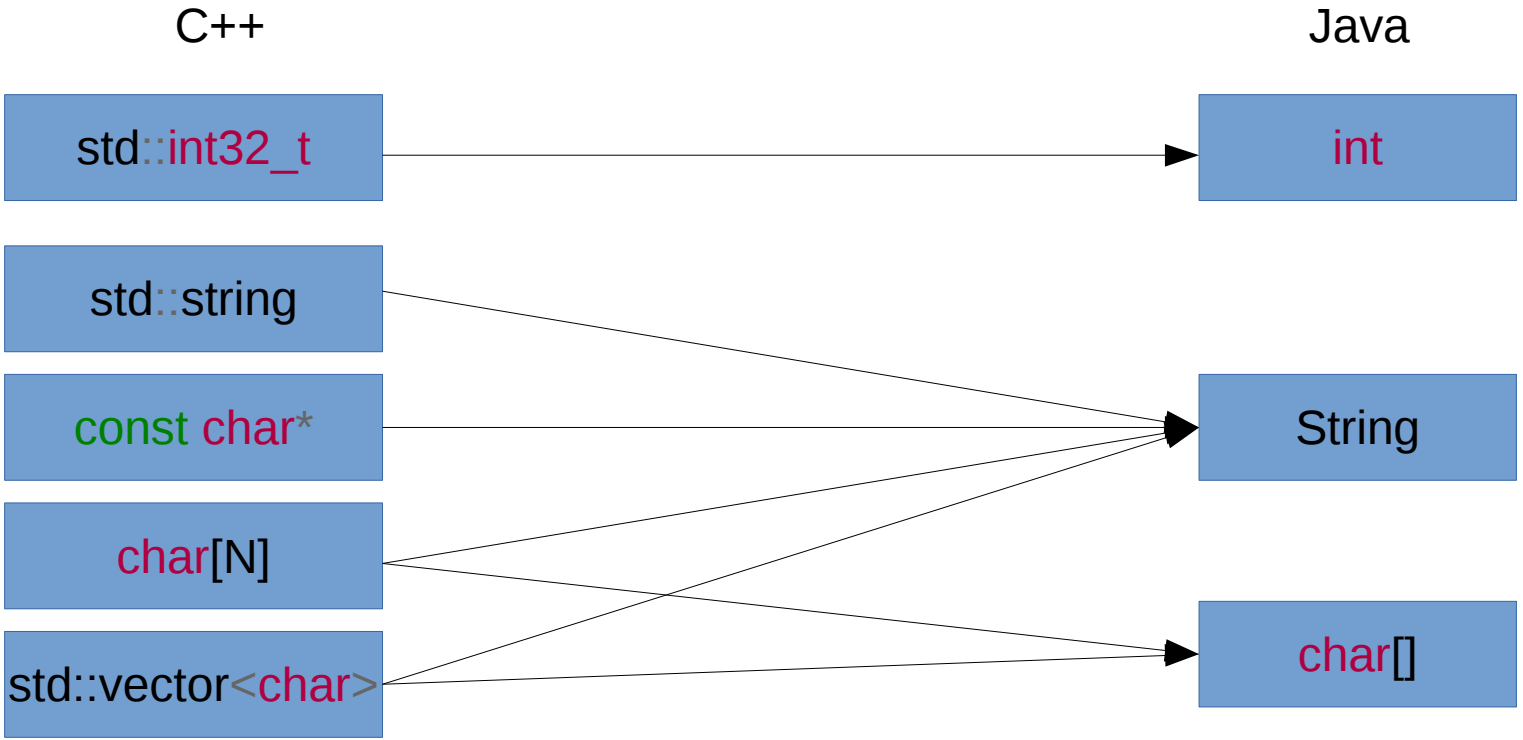

© 2017

Method ↔ signature relation



© 2017
NOKIA

Method ↔ signature relation



+ User defined

© 2017
NOKIA

Declaring Java method using cppJNI

```
struct MyClass
{
    //"myMethod", Void, String
    Void myMethod(String&&) const
    { ? }
    //..
};
```


© 2017

Declaring Java method using cppJNI

```
struct MyClass
{
    auto myMethod(String&& a) const
    {
        //"myMethod", Void, a
    }
};
```


© 2017

Declaring Java method using cppJNI

```
struct MyClass
{
    template<typename... Ts>
    auto myMethod(Ts&&... args) const
    {
        //"myMethod", Void, String
    }
};
```


© 2017
NOKIA

Java ↔ C++ type relation

```
struct String {
    static constexpr char* signature = "Ljava/lang/String";
    template<typename T>
    using isConvertible = ...;
    static constexpr jint
        convertToJni(const std::string& s) const {...}
    static constexpr std::string
        convertFromJni(jstring I) const {...}
};
```


© 2017

Java ↔ C++ type relation

```
struct Void {
    static constexpr char* signature = "V";
    template<typename T>
    struct isConvertible
    {
        static_assert(sizeof(T) != 0,
            "Void cannot be a function argument!");
    };
};
```


© 2017

Declaring Java method using cppJNI

```
struct MyClass
{
    MAKE_JAVA_METHOD(myMethod, Void, String);
};
```


© 2017

Declaring Java method using cppJNI

```
//given Java code
package some.package;
public class MyClass {
    public void myMethod(String param) {}
    public String myDifferentMethod(String p1, int p2) {
        return ...;
    }
}
```


© 2017

Dealing with overloading

```
//given Java code
package some.package;
public class MyClass {
    public void myMethod(String param) {}
    public void myMethod(int param) {}
    public String myDifferentMethod(String p1, int p2) {
        return ...;
    }
}
```


© 2017

Dealing with overloading

```
struct MyClass
{
    MAKE_JAVA_METHOD(myMethod, Void, String);
    MAKE_JAVA_METHOD(myMethod, Void, Int);
};
```


© 2017

Dealing with overloading

```
struct MyClass
{
    MAKE_JAVA_METHOD(myMethod,
        Void(String),
        Void(Int)
    );
};
```


© 2017
NOKIA

Dealing with overloading

```
template<typename... Ts>
auto myMethod(Ts&&... args) const
{
    const char* name = "myMethod";
    return cppjni::method_caller<
       /* bunch of meta-info */
    >::call(*this, name, std::forward<Ts>(args)...);
}
```


© 2017

Overload resolution - simplified

```
using overload_resolutor =
    typename mpl::pipeline<
        mpl::filter<
            has_args_count<sizeof...(passed_args_t)>
                ::template f
        >,
        mpl::find_if<
            are_convertible<passed_args_t...>::template f
        >
    >;
```


© 2017
NOKIA

Overload resolution - simplified

```
using found_overload_t =
    typename mpl::apply<
        overload_resolutor,
        all_overloads_t /* mpl::list<Void(String), etc.> */
    >::template continuation<mpl::optional>;
static_assert(
    found_overload_t::has_type::value == true,
    "Couldn't convert all parameters to appropriate
overload!"
);
```


© 2017

Calling Java method

```
using selected_declaration_t =
    typename found_overload_t::type;
constexpr auto converters =
    selected_declaration_t::arguments_t{};
auto&& passedArgs = hana::make_tuple(args...);
auto&& convertedArguments = hana::zip_with(
    [](auto&& argument, auto converter) {
        return converter.convertToJni(argument);
    }, passedArgs, converters
);
```


© 2017

Dealing with overloading

```
//given Java code
package some.package;
public class MyClass {
    public void myMethod(String param) {}
    public void myMethod(int param) {}
    public String myDifferentMethod(String p1, int p2) {
        return ...;
    }
}
```


© 2017

Dealing with overloading - static methods

```
//given Java code
package some.package;
public class MyClass {
    public void myMethod(String param) {}
    public static void myMethod(int param) {}
    public String myDifferentMethod(String p1, int p2) {
        return ...;
    }
}
```


© 2017

Static methods

Java:

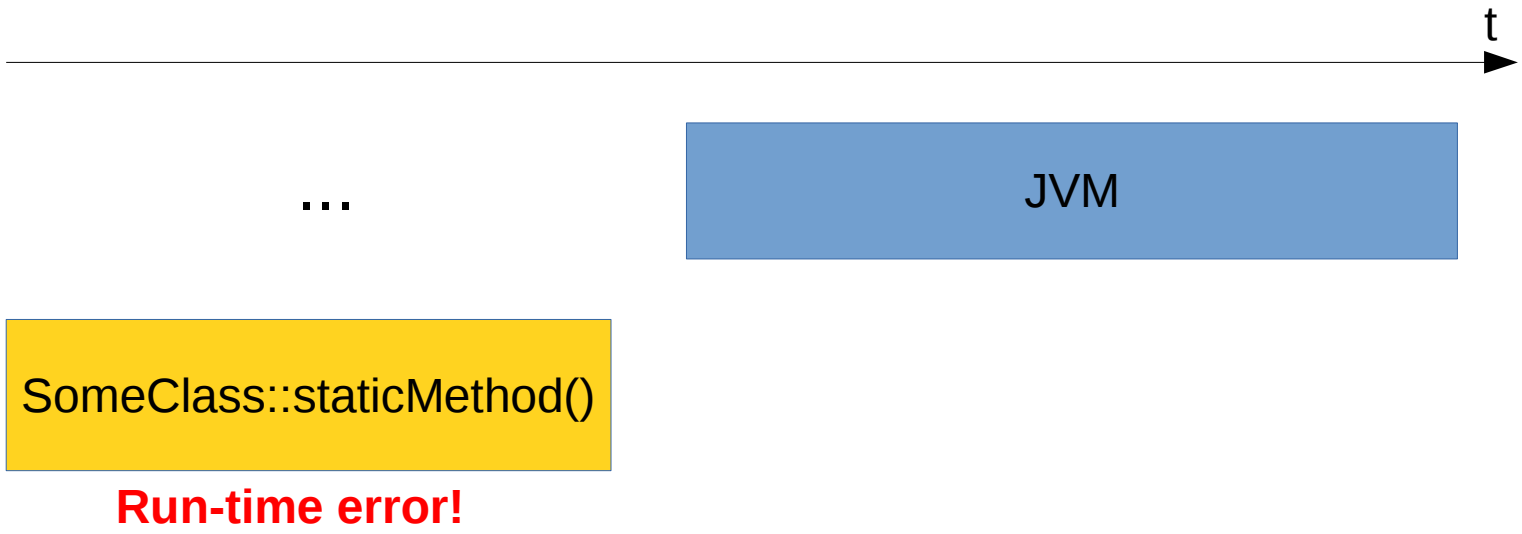
```
SomeClass.staticMethod(); //Pefectly fine
SomeClass x = new SomeClass();
x.staticMethod(); //Warning, but works!
```

C++:

```
SomeClass::staticMethod(); //Perfectly fine
SomeClass x{};
x.staticMethod(); //Warning, but works!
```


© 2017

Static methods



© 2017
NOKIA

Static methods

```
cppjni::VirtualMachine jvm(...);
auto myClass = jvm.loadClass<MyClass>();
myClass.staticMethod();
```


© 2017

Static methods

```
struct MyClass
{
    MAKE_JAVA_METHOD(myMethod,
        Method<Void(String)>,
        StaticMethod<Void(Int)>
    );
};
```

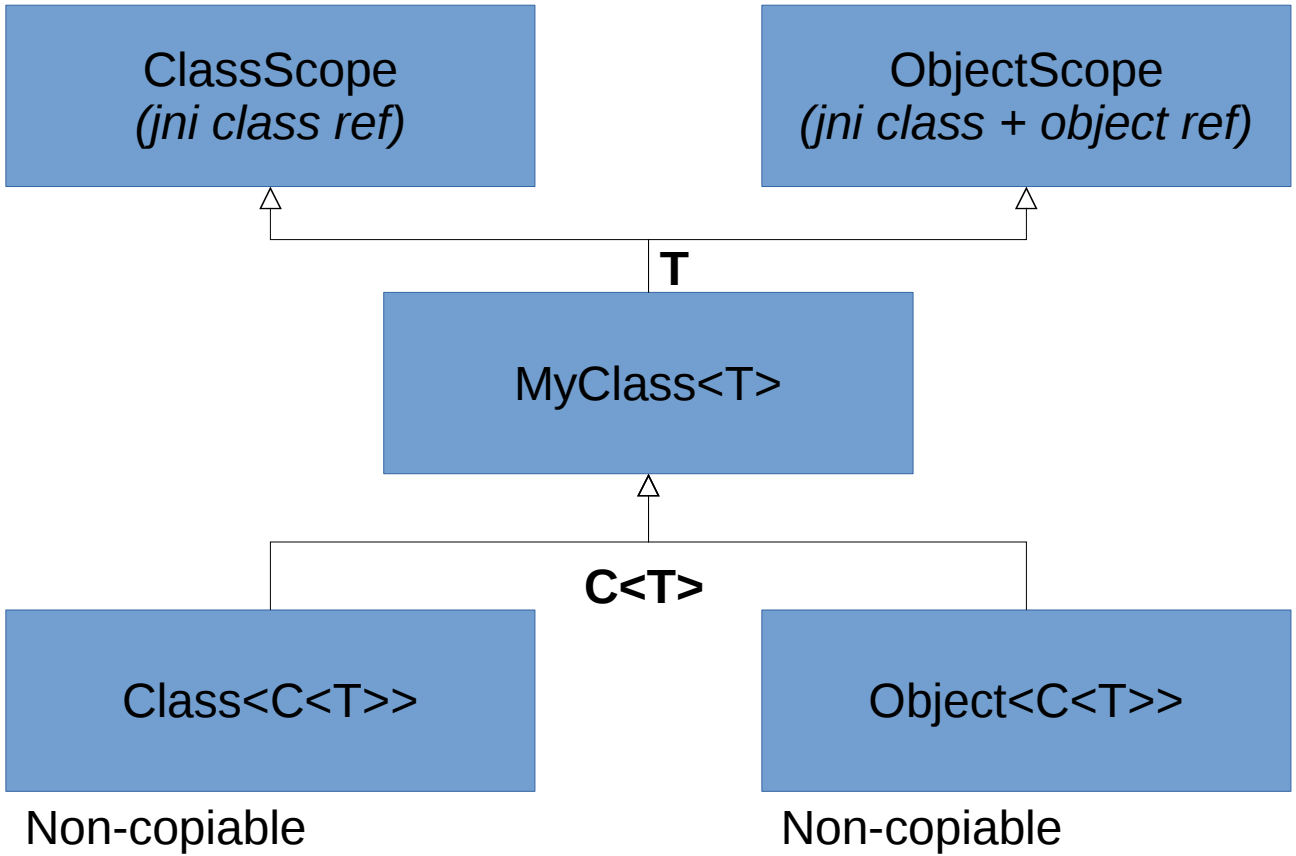

© 2017
NOKIA

Static methods – scope

```
template<typename T> //T will be a "scope"
struct MyClass: T
{
    MAKE_JAVA_METHOD(myMethod,
        Method<Void(String)>,
        StaticMethod<Void(Int)>
    );
};
```


© 2017

Scope



© 2017
NOKIA

Scope

```
cppjni::VirtualMachine jvm(...);
auto myClass
     = jvm.loadClass<MyClass>();
myClass.staticMethod();
```


© 2017

Scope

```
cppjni::VirtualMachine jvm(...);
cppjni::Class<MyClass> myClass
    = jvm.loadClass<MyClass>();
myClass.staticMethod();

auto myClassObject
    = myClass.Construct(...);
```


© 2017

Scope

```
cppjni::VirtualMachine jvm(...);
cppjni::Class<MyClass> myClass
    = jvm.loadClass<MyClass>();
myClass.staticMethod();

cppjni::Object<MyClass> myClassObject
    = myClass.Construct(...);
```


© 2017
NOKIA

```
using caching =
    mpl::eager::apply<
        mpl::pipeline<
            mpl::drop<3>,
            mpl::first_of
        >,
        agenda>;
```


© 2017

Things to cache:

- Java classes
- Java method id’s
- Method signatures?

 © 2017

NOKIA

Caching

```
using class_cache = std::unordered_map<
    std::string, //class path
    cppjni::GlobalReference<jobject>
>;
using method_id_cache = std::unordered_map<
    std::string, //class + method + signature
    jint
>;
```


© 2017

Caching – can we do it better?

What we know at compile time:

- Classes that will be used (string)
- Their method names (string)
- Signatures (string)

What if we could use these?

© 2017

Caching – compile time strings

```
template<char... Chars>
struct string_type
{
    static constexpr char data[] =
        {Chars..., '\0'};
};
```


© 2017

Caching – compile time strings

```
using my_string =
    string_type<'a', 'b', 'c'>;
using my_other_string =
    string_type<'a', 'b', 'c'>;
static_assert(std::is_same<
        my_string,
        my_other_string
    >::value, "");
```


© 2017

Caching – compile time strings

```
using my_string =
    MAKE_STRING_TYPE("abc");
using my_other_string =
    MAKE_STRING_TYPE("abc");
static_assert(std::is_same<
        my_string,
        my_other_string
    >::value, "");
```


© 2017

Caching – compile time strings

```
struct String {
    static constexpr char* signature = "Ljava/lang/String";
    template<typename T>
    using isConvertible = ...;
    static constexpr jint
        convertToJni(const std::string& s) const {...}
    static constexpr std::string
        convertFromJni(jint I) const {...}
};
```


© 2017

Caching – compile time strings

```
struct String {
    using signature_t = MAKE_STRING_TYPE("Ljava/lang/String");
    template<typename T>
    using isConvertible = ...;
    static constexpr jint
        convertToJni(const std::string& s) const {...}
    static constexpr std::string
        convertFromJni(jint I) const {...}
};
```


© 2017

Caching – compile time strings

```
template<typename T>
struct MyClass: T
{
    MAKE_JAVA_METHOD(myMethod,
        Method<Void(String)>,
        StaticMethod<Void(Int)>
    );
};
```


© 2017

Caching – compile time strings

```
template<typename T>
struct MyClass: T
{
    using class_path_t =
        MAKE_STRING_TYPE("some/package/MyClass");
    MAKE_JAVA_METHOD(myMethod,
        Method<Void(String)>,
        StaticMethod<Void(Int)>
    );
};
```


© 2017

Caching – compile time strings

```
template<typename... Ts>
auto myMethod(Ts&&... args) const
{
    const char* method_name = "myMethod";
    return cppjni::method_caller<
       /* ... */
    >::call(*this, method_name, std::forward<Ts>(args)...);
}
```


© 2017

Caching – compile time strings

```
template<typename... Ts>
auto myMethod(Ts&&... args) const
{
    using method_name_t = MAKE_STRING_TYPE("myMethod");
    return cppjni::method_caller<
       method_name_t,
       /* ... */
    >::call(*this, std::forward<Ts>(args)...);
}
```


© 2017

Caching – compile time strings

```
template<typename class_path_t, typename method_name_t,
    typename method_signature_t>
jmethodId getMethodId(jclass* jniClass)
{
    static jmethodId methodId = [jniClass](){
        /*... method_name_t::data ... */
    } ();
    return methodId;
}
```


© 2017

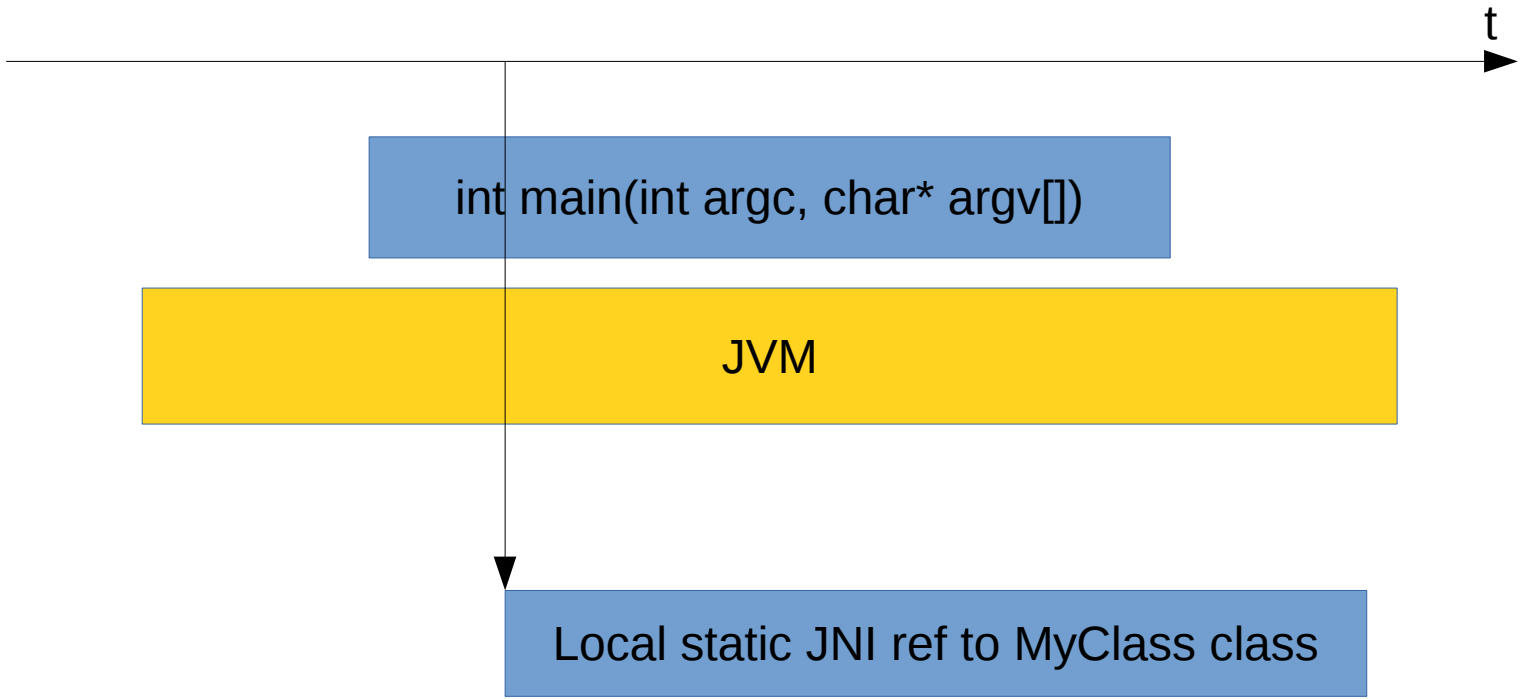
Caching – compile time strings

```
template<typename ClassPath>
const cppjni::GlobalReference<jclass>& getClass()
{
    static GlobalReference<jclass> classRef = [](){
        /*...*/
    } ();
    return classRef;
}
```

Neat, right!? : }

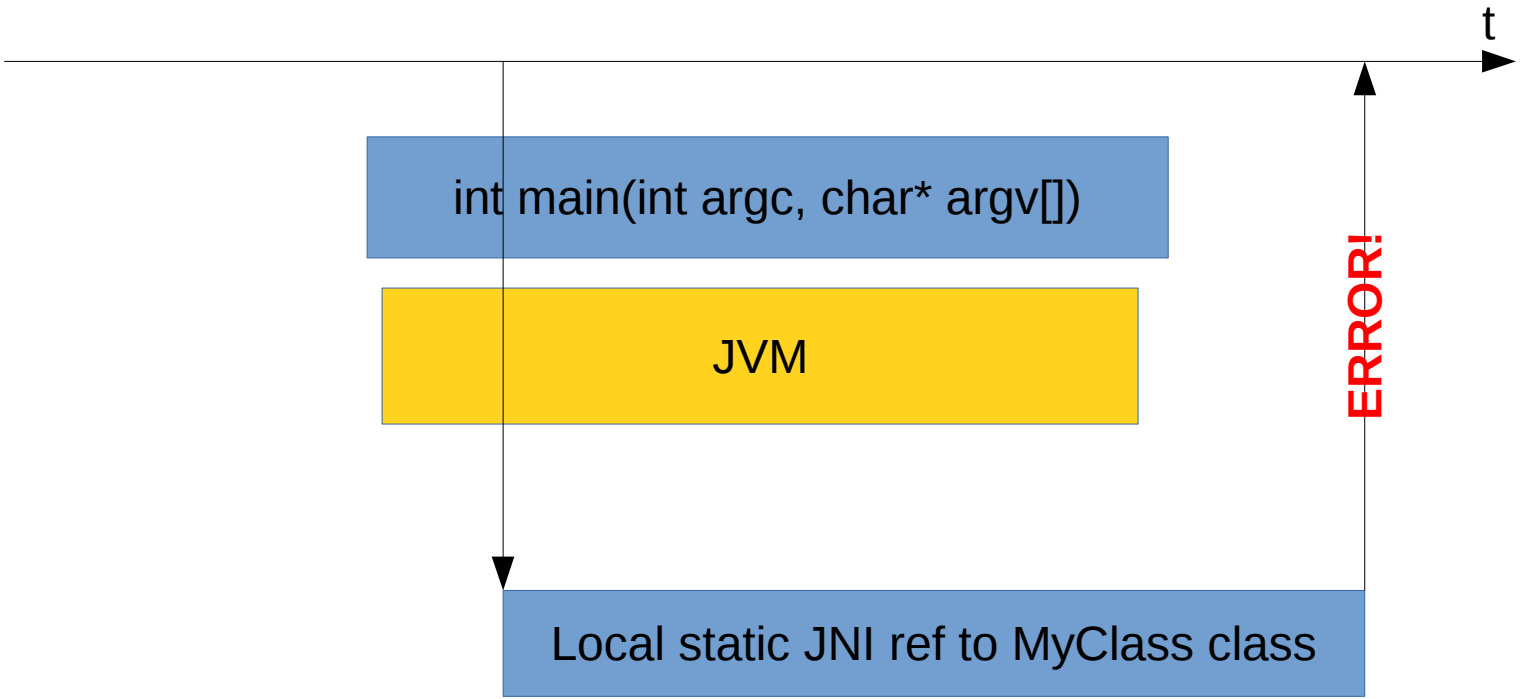
© 2017

Caching – problem with lifetime



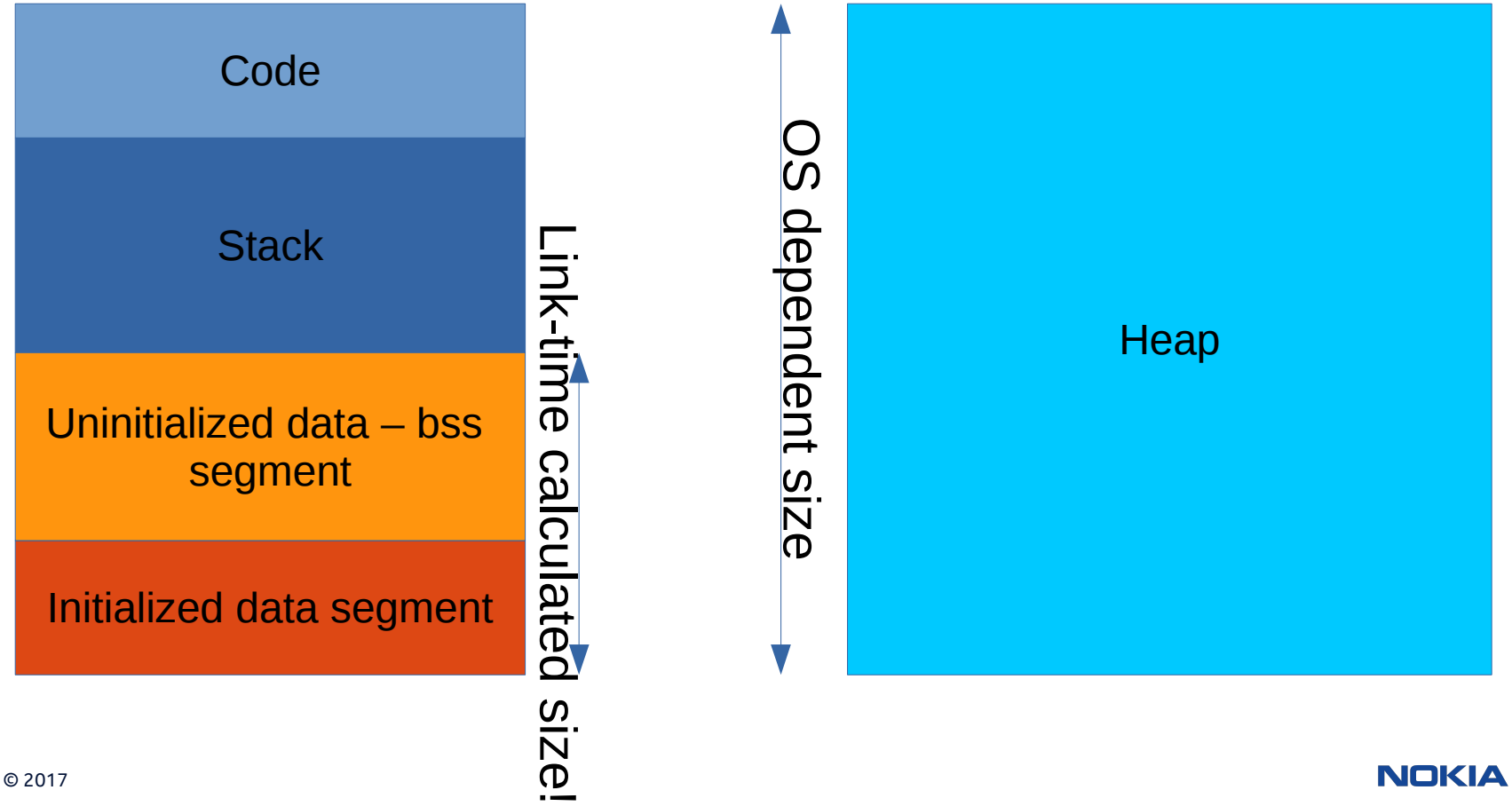
© 2017
NOKIA

Caching – problem with lifetime



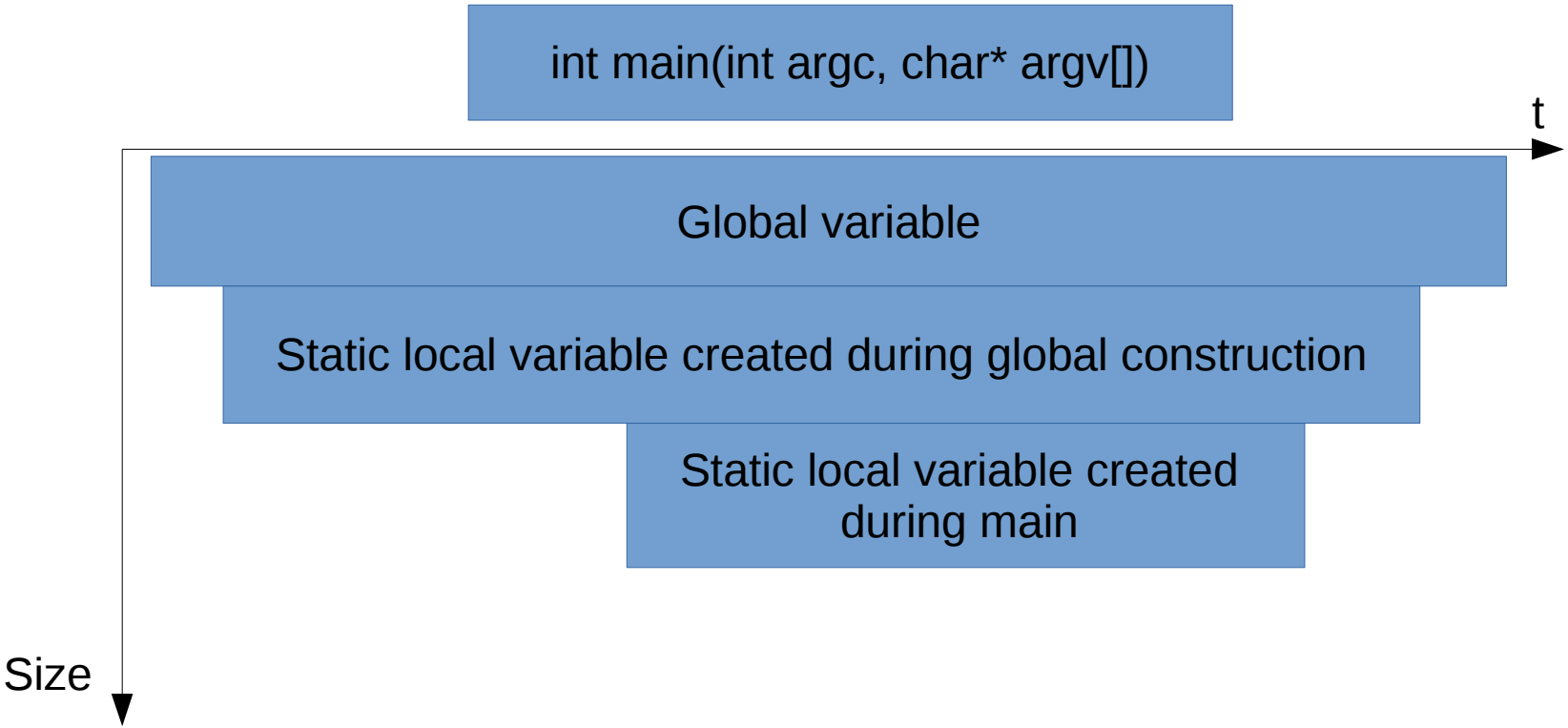
© 2017
NOKIA

Process memory segments



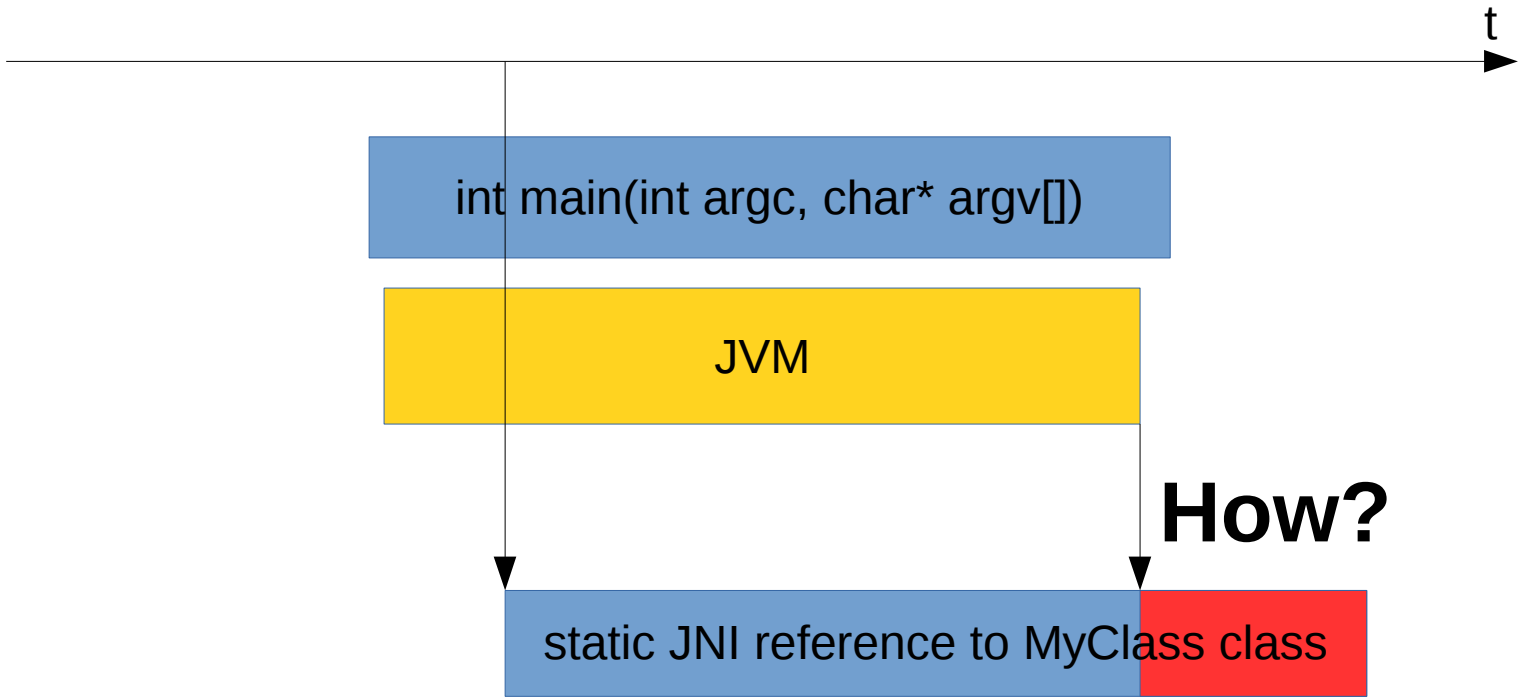
© 2017
NOKIA

Process memory segments



© 2017
NOKIA

Caching – problem with lifetime



© 2017
NOKIA

Caching – StaticMap

```
template<typename base_type_t, typename tag_t = void>
class StaticMap
{
    struct Node
    { /* ... */ };
    //...
private:
    static Node* head;
};
```


© 2017
NOKIA

Caching – StaticMap

```
template<typename base_type_t, typename tag_t = void>
class StaticMap
{
    struct Node
    {
        Node(base_type_t& elem, Node* head) //...
        Node* next;
        base_type_t& element;
    };
```

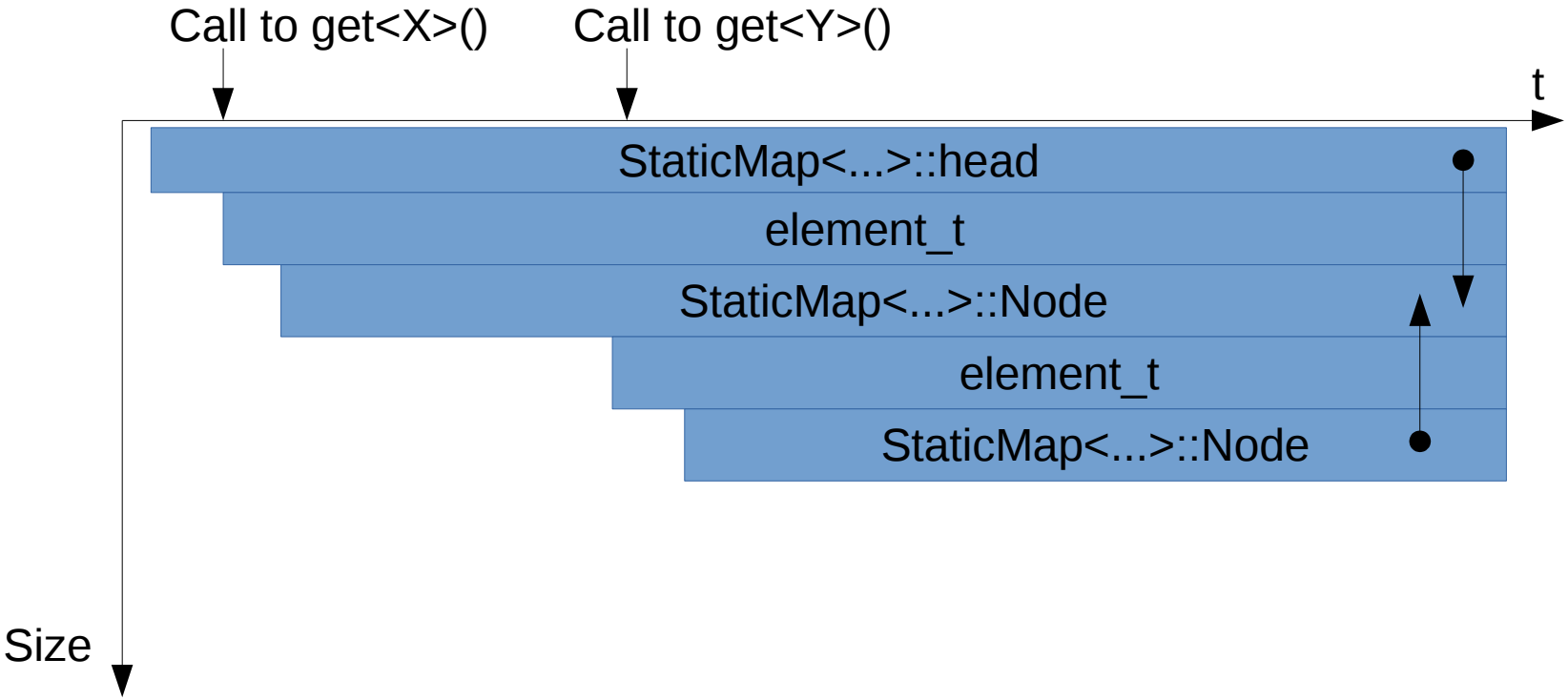

© 2017

Caching – StaticMap

```
template<typename key_type_t>
typename key_type_t::value_type& get()
{
    using element_t = typename key_type_t::value_type;
    static element_t element = key_type_t{}();
    static Node node{element, head};
    return element;
}
```

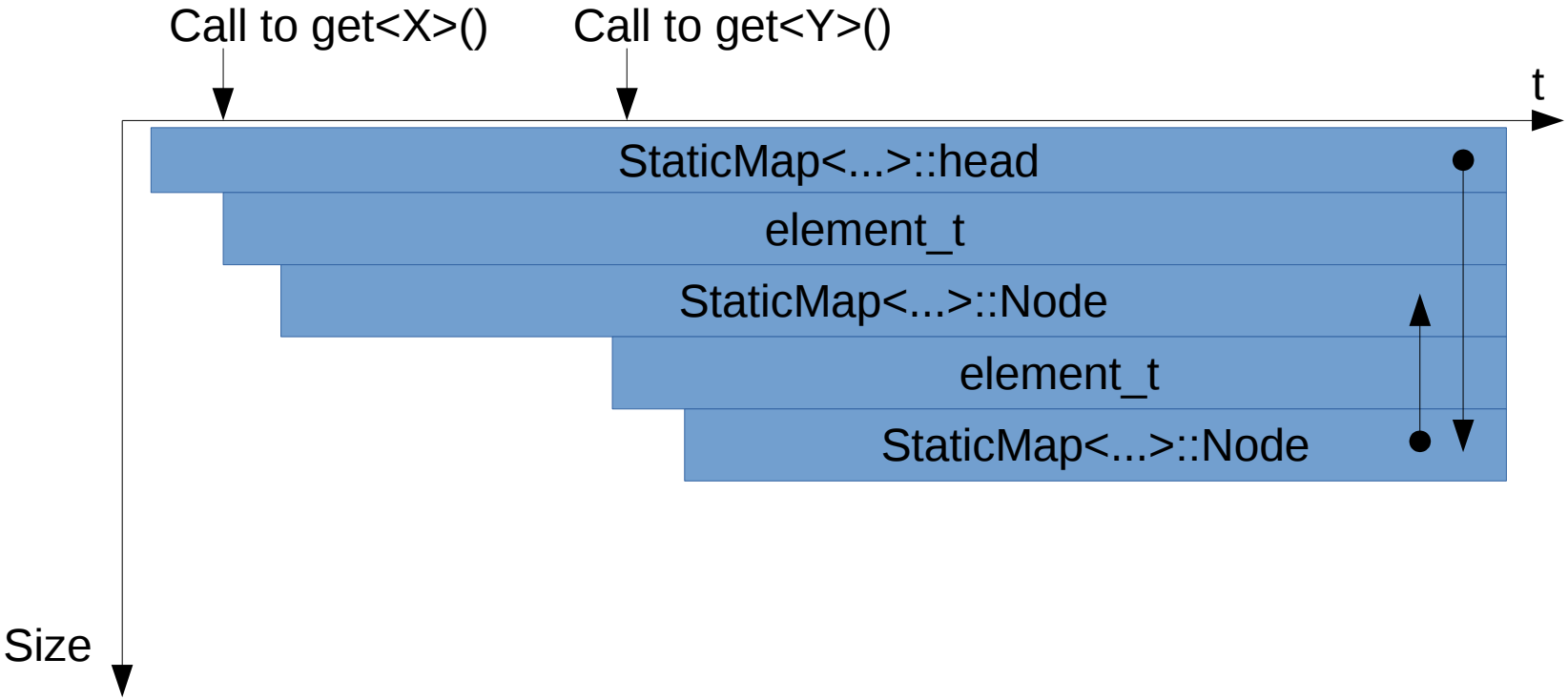

© 2017

Caching – StaticMap



© 2017
NOKIA

Caching – StaticMap



© 2017
NOKIA

Caching – StaticMap

```
StaticMap<GlobalReference<jclass>> map;
auto classRef =
    *map.get<ClassLoader<class_path_t>>();

//in VirtualMachine destructor:
for(auto& classRef: map)
    classRef.release();
```

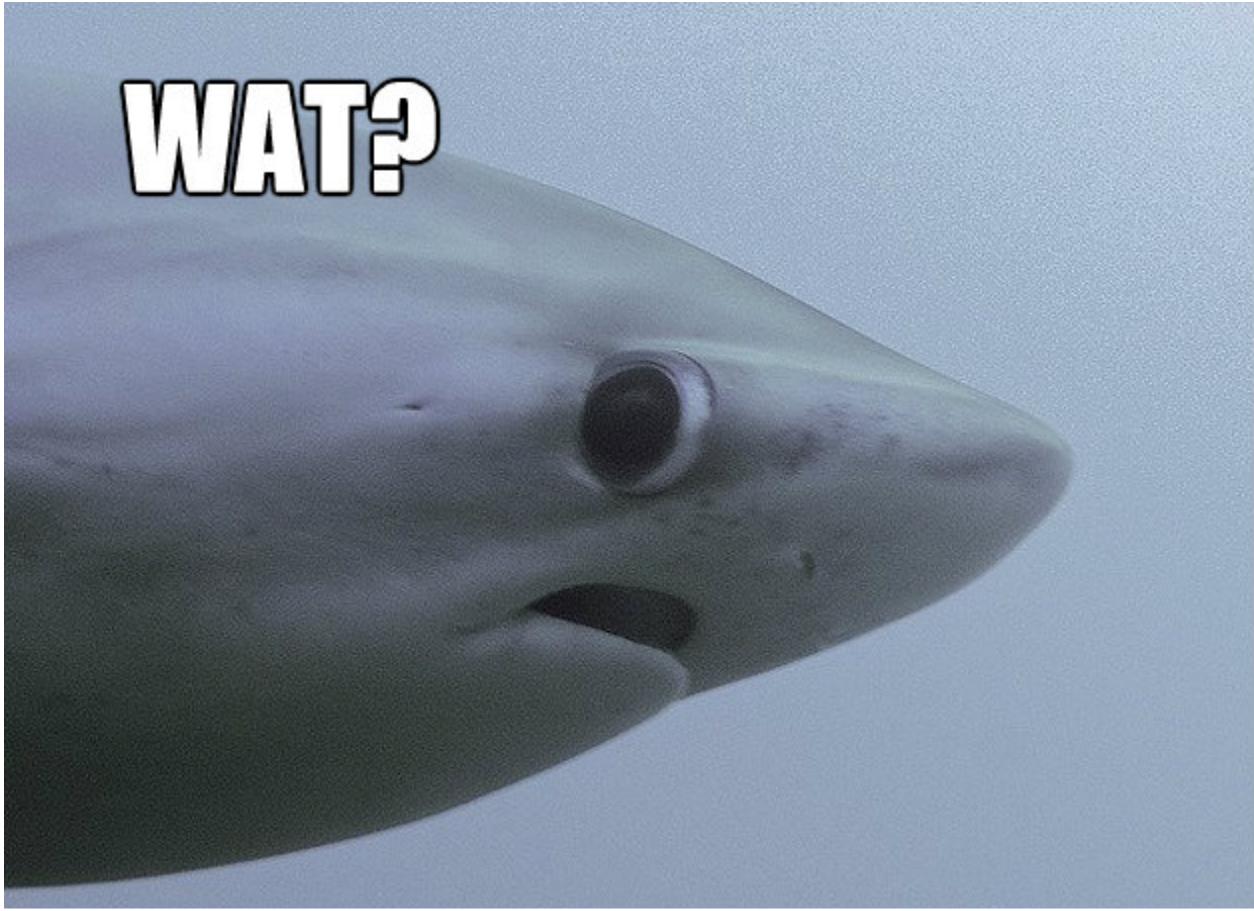

© 2017

using **compilationOptimization** =

CompilationOptimization;

© 2017

Actually:



 © 2017

NOKIA

Who's your customer?

- End user
- Developers are also end users!
- As well as them:

 © 2017

NOKIA

Source: Wikipedia Commons

NOKIA

Compilation optimization – build system

- Ninja! - 5-10% vs make
- Reordering sources and targets
- Use newer compiler
- Use less generated code

© 2017

Compilation optimization - cleaning your code

- Clean up your inclusions!

© 2017

Cleaning your code

```
//#include <boost/iterator/iterator_traits.hpp>


int main(int argc, char* argv[])
{
   return 0;
}
```

0.07s~0.08s (usr + sys)

© 2017

Cleaning your code

```
#include <boost/iterator/iterator_traits.hpp>


int main(int argc, char* argv[])
{
   return 0;
}
```

0.25s~0.3s ! (usr + sys)

© 2017

- Clean up your inclusions

 © 2017

NOKIA

Cleaning your code

- Never include more than you need!
- Use fast meta-programming libraries:
  - Like Kvasir, meta, mp11
  - Go to Odin Holmes presentation tomorrow ;) (9:15, Stage 7)!
  - **boost::mpl – go home!**
- Tools: metashell, Templight

© 2017

Q's! And A's!

© 2017
NOKIA

Thank You!

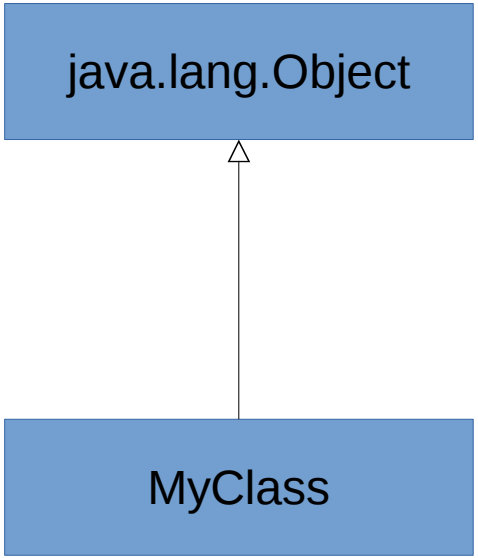
https://github.com/nokia/cppJNI

michal.los@nokia.com

 © 2017

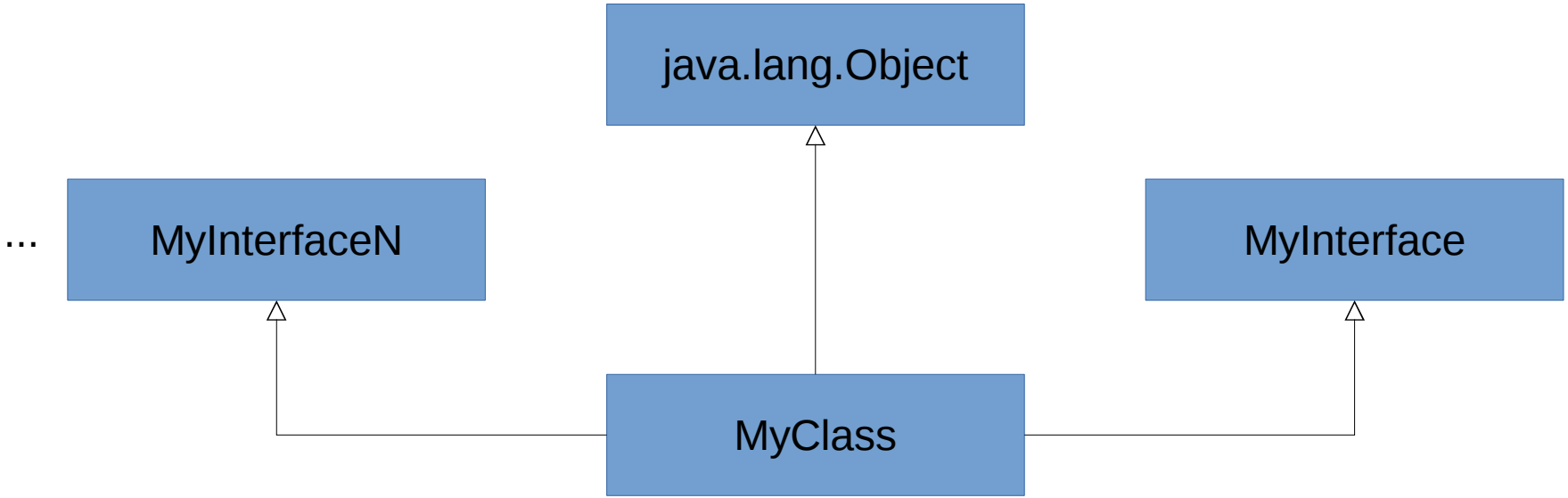
NOKIA

Java Inheritance



© 2017
NOKIA

Java Inheritance



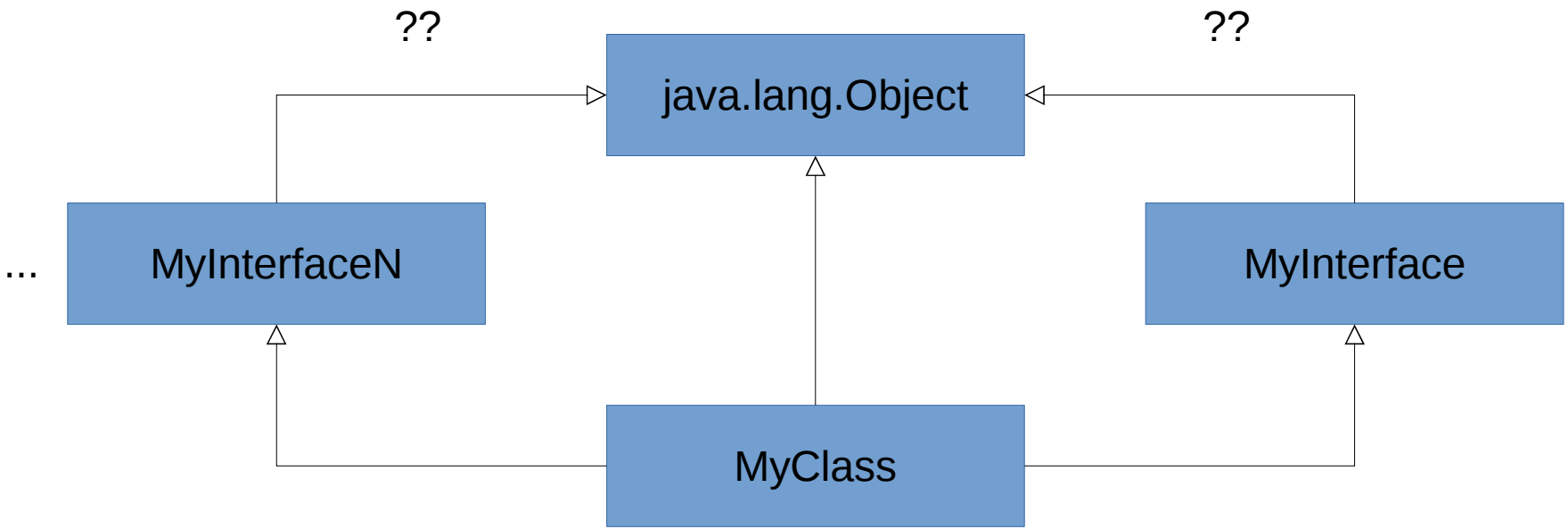
© 2017
NOKIA

Java Inheritance

```
MyInterface x = new MyClass();
x.hashCode(); //??
```


© 2017
NOKIA

Java Inheritance



© 2017
NOKIA

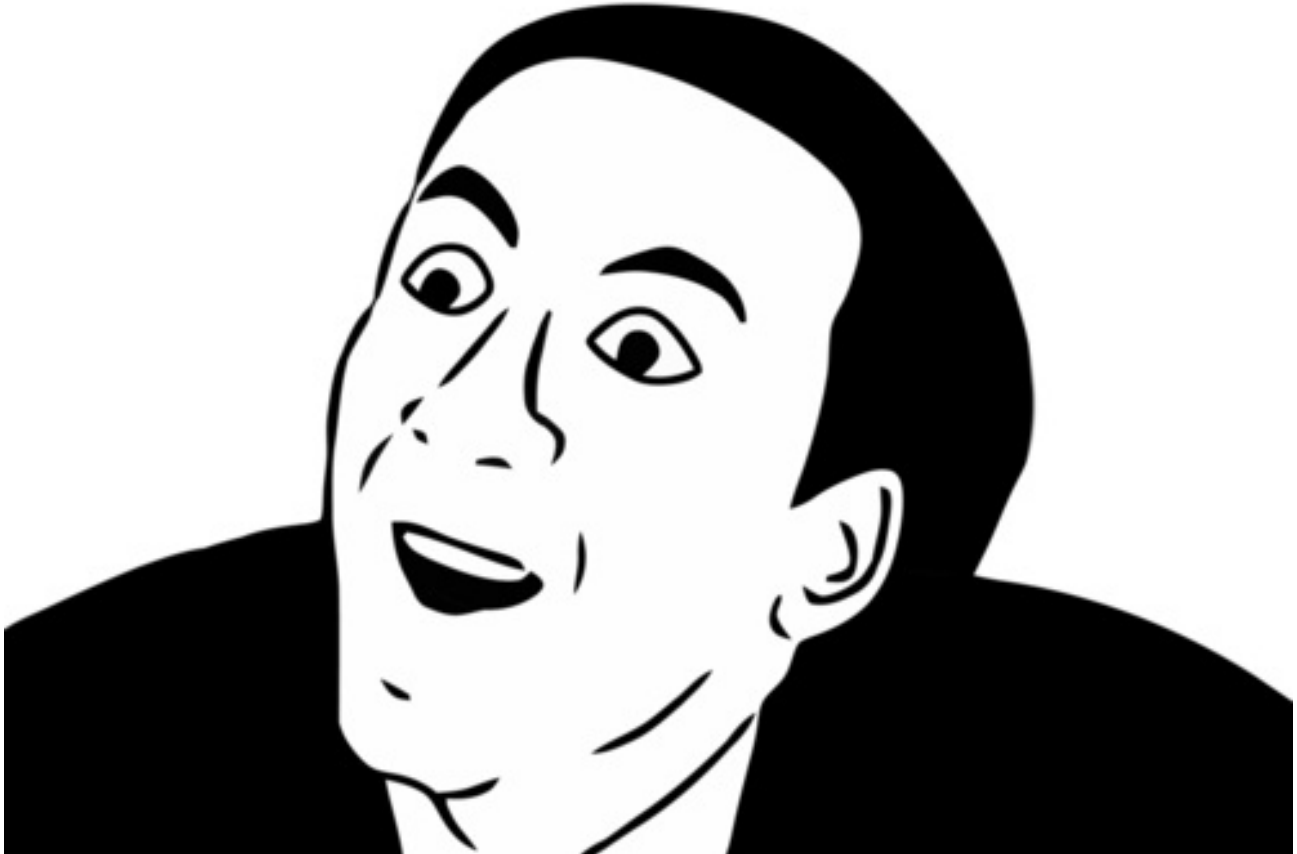
9.2 Interface Members

[...]

If an interface has no direct superinterfaces, then the interface **implicitly declares** a public abstract member method m with signature s, return type r, and throws clause t corresponding to each public instance method m with signature s, return type r, and throws clause t **declared in Object**, unless a method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.

 © 2017

NOKIA



© 2017

Java inheritance

```
template<typename T>
struct MyClass:
    virtual java::lang::Object<T>,
    virtual java::lang::Comparable<T>,
    ...
{
/*...*/
```


© 2017