Java Meets Rust:
The Depths of Cross-Language Integration

Introduction

Konstantin Grechishchev

linkedin.com/in/kgrech

kmgrechis@gmail.com

Experience

- Master of computer science, 2015
- Cisco Systems Inc

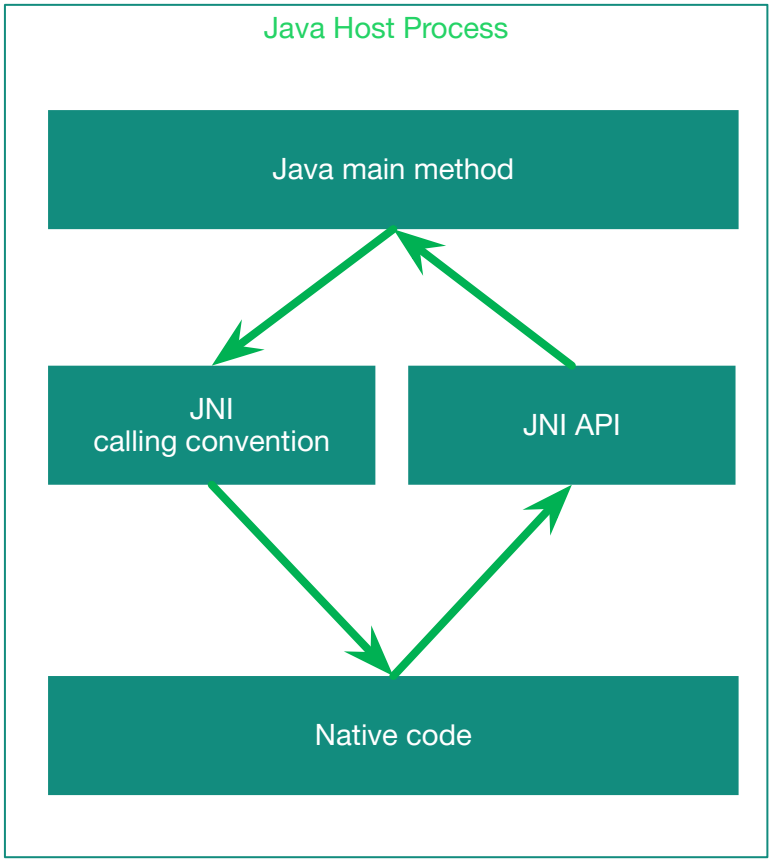
WhatsApp Software Engineer

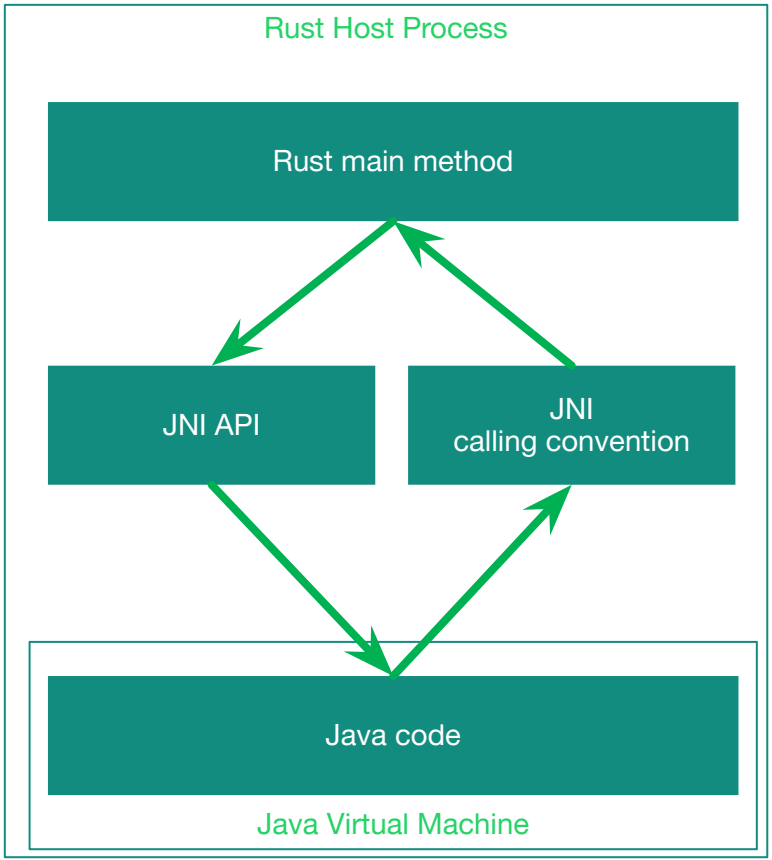
- Since 2020

Do you really need rust and Java in the same process?

- Maybe you don’t?
- Can you keep them as separate processes?
- Ways of interprocess communication
  - Rest
  - GRPC/Thrift/etc
  - TCP/UWP/Web Sockets
  - Files
  - Database
  - Message queues
  - Operation system IPC mechanisms
- Same process, but different architecture?
  - CRUX?

Rust and Java in the same process





How to pass complex data?

- Use JNI APIs to create or traverse java objects (jni-rs,
  - Freedom to define interface according to your preference
  - Could be broken by minifier and obfuscator
  - Requires aligned changes on java and rust
  - A lot of work!
- Generate FFI interface (UniFFI - kotlin)
  - Restricted to the capabilities of the framework
  - Not so many options available (write yours?)
- Serialize data on one side and deserialize on other (protobuf or even json!)
  - Thin JNI layer
  - Write once and only modify the schema later
  - Slightly less efficient (profile before jurging!)
  - Boilerplate code to serialize and deserialize

Meet Toy JNI!

- Define protobuf schema
- Generate bindings for both java and rust
- Pass arrays of bytes over JNI interface
- Implement wrappers to make it more convenient!
- Done!
- Simply extend schema when required

```
syntax = "proto3";

package toy_jni;
option java_package = "com.github.kgrech.toyjni.proto";
option java_multiple_files = true;

message Request {
  string message = 1;
  uint64 response_delay = 2;
}

message Response {

  message Success {
    string message = 1;
  }

  message Error {
    string error_message = 1;
  }

  optional Success success = 1;
  optional Error error = 2;
}
```

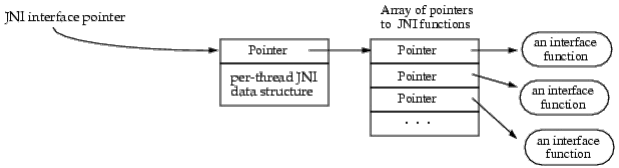
JNI Calling Convention

- Oracle Java Native Interface Specification. Chapter 2: Design Overview
- Java:

```
package pkg;
class Cls {
     native double f(int i, String s);
     static {
         // Consider compiling all-rust dependencies in single library
         System.loadLibrary("libname");
     }
}
```

- C:

```
#include <jni.h>
JNIEXPORT jdouble JNICALL Java_pkg_Cls_f (
     JNIEnv *env,        /* interface pointer */
     jobject obj,        /* "this" pointer */
     jint i,             /* argument #1 */
     jstring s)          /* argument #2 */
{
     /* Obtain a C-copy of the Java string */
     const char *str = (*env)->GetStringUTFChars(env, s, 0);
     /* process the string */
     /* Now we are done with str */
     (*env)->ReleaseStringUTFChars(env, s, str);
     return ...
}
```



Java calling Rust (jni crate)

- Declare the native method

```
private native byte[] nativeCall(byte[] request);
```

- Define the implementation in rust

```
#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_sync_JNIBridge_nativeCall<'a>(
    env: JNIEnv<'a>,
    j_object: JObject<'a>,
    request: JByteArray<'a>,
) -> JByteArray<'a> {
    handle_error(env, |env| {
        let request = Request::decode_from_java(env, &request)?;
        // Do something with request
        let response = if request.message == "Hello, Rust!" {
            Response::success("Hello, Java!".into())
        } else {
            Response::error(format!("Unable to respond to '{}'", request.message))
        };
        response.encode_to_java(env)
    })
}
```

Decoding data efficiently!

- Oracle Java Documentation. Chapter 4: JNI Functions
  - GetByteArrayRegion - **copies** a region of a primitive array into a buffer.
  - GetByteArrayElements - returns the body (**maybe a copy**) of the primitive array, valid until ReleaseByteArrayArrayElements().
  - GetPrimitiveArrayCritical/ReleasePrimitiveArrayCritical - similar to GetByteArrayElements, but asks JVM to avoid copy. There are restrictions on what could be done between Get and Release calls.
- JNI crate provides nice wrappers:

```
fn decode_from_java(env: &mut JNIEnv, j_byte_array: &JByteArray) -> anyhow::Result<Self>
{
    unsafe {
        let guard = env
            .get_array_elements_critical(j_byte_array, ReleaseMode::NoCopyBack)
            .with_context(|| "Failed to access byte array")?;
        let slice: &[i8] = &*guard;
        let u8_slice: &[u8] =
            std::slice::from_raw_parts(slice.as_ptr() as *const u8, slice.len());
        Self::decode(u8_slice).with_context(|| "Unable to decode request")
    }
}
```

Encoding the data

- Oracle Java Documentation. Chapter 4: JNI Functions
  - NewByteArray - construct a new primitive array object
  - SetByteArrayRegion - **copies** a region of a primitive array from a buffer.

- JNI crate provides a nice wrapper for both:

```
fn encode_to_java<'a>(&self, env: &JNIEnv<'a>) -> anyhow::Result<JByteArray<'a>> {
    let protobuf_bytes: Vec<u8> = self.encode_to_vec();
    env.byte_array_from_slice(protobuf_bytes.as_slice())
        .with_context(|| "Error converting result to java byte array")
}
```

Calling Java Back

- Let's define a function to be called from rust:

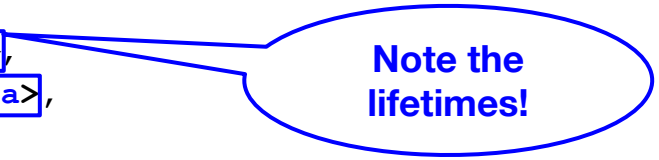
```
public byte[] onJniCallback(byte[] request) { // Called from native code
    Request deserializedRequest = Request.parseFrom(request);
    Response response = handleCallbak(deserializedRequest);
    return response.toByteArray();
}
```

- Oracle Java Documentation. Chapter 4: JNI Functions
  - Call<type>Method Routines, Call<type>MethodA Routines, Call<type>MethodV Routines used to call a Java instance method from a native method

```
let data = response.encode_to_java(env)?;
let response = env
    .call_method(
        j_object,
        "onJniCallback",
        "([B)[B", // Method signature
        &[JValue::from(&data)],
    )
    .with_context(|| "Error calling onJniCallback method")?;
```

JNIEnv lifetimes

```
pub extern "system" fn Java_com_github_kgrech_toyjni_sync_JNIBridge_nativeCall<'a>(
    env: JNIEnv<'a>,
    j_object: JObject<'a>,
    request: JByteArray<'a>,
) -> JByteArray<'a>
```



- Oracle Java Documentation. Chapter 5: The Invocation API
- JNIEnv is a pointer to native method interface (main set of of the APIs). **Valid only in the current thread**
- Could be obtained:
  - As a parameter to any call made from Java
  - From JavaVM *jvm using AttachCurrentThread/AttachCurrentThreadAsDaemon JNI APIs
- Lifetime ensures JVM is not terminated and the object is not garbage collected
- This won't compile:

```
pub extern "system" fn nativeCall<'a>(env: JNIEnv<'a>, j_object: JObject<'a>) {
    std::thread::spawn(move || {
        callback(env, j_object); // env and j_object are neither Send nor 'static!
    });
}
```

How do we pass the Java object to another thread?

- Oracle Java Documentation. Chapter 4: JNI Functions
  - NewGlobalRef - Creates a new global reference, must be explicitly disposed of by DeleteGlobalRef().
  - GetJavaVM - Returns the Java VM interface (used in the Invocation API) associated with the current thread

```
let global_ref: GlobalRef = env.new_global_ref(j_object)
let jvm:        JavaVM    = env.get_java_vm()
```

Have no lifetime.
Implement Send.

- JavaVM is a JNI interface pointer (JVM lifecycle and threads API) and denotes a Java VM.
- Could be obtained:
  - If you start the JVM by calling JNI_CreateJavaVM()
  - From JNIEnv env using GetJavaVM method

Storing Java Object inside Rust Struct

- Why?
  - To keep it for a call done in background
- Can’t store j_object: JObject**<’a>**
  - It has a lifetime. Because Java will GC collect it!
- Use NewGlobalRef API

```
let global_ref: GlobalRef = env.new_global_ref(j_object)?;
```

- Global references must be explicitly disposed of by calling the DeleteGlobalRef method.
  - Rust wrapper calls it called on drop
- Can use as_obj to get JObject back for API call when needed

```
let j_object: JObject = global_ref.as_obj();
```

- Do not forget to attach the thread!

Rust Part of the Callback (1)

- Protect JObject from being collected garbage collected and get JVM pointer:

```
pub struct CallbackContext {
    jvm: JavaVM,
    global_ref: GlobalRef,
}

impl CallbackContext {
    pub fn new(env: &JNIEnv, j_object: &JObject) -> anyhow::Result<Self> {
        let global_ref = env
            .new_global_ref(j_object)
            .with_context(|| "Unable to create global reference for java object repo")?;
        let jvm: JavaVM = env
            .get_java_vm()
            .with_context(|| "Unable to JVM reference")?;
        Ok(Self { jvm, global_ref })
    }
}
```

Rust Part of the Callback (2)

- CallbackContext implements Send trait, so we can pass it to another thread safely

```
pub extern "system" fn nativeCall<'a>(env: JNIEnv<'a>, j_object: JObject<'a>) {
   let context = CallbackContext::new(env, &j_object)?;
   std::thread::spawn(move || {
       std::thread::sleep(
          Duration::from_millis(request.response_delay)
       );
       let response = context.callback(Request::message(
                "Hello, Java!".into())
           )
           .expect("Callback error!");
       println!("Response from callback: {response:?}")
   });
}
```

Beware: Attaching a thread

- Oracle Java Documentation. Chapter 5: The Invocation API
- AttachCurrentThread API attaches the current thread to a Java VM.
  - You can think about it as “set required thread-local variables”
  - Returns a JNI interface pointer in the JNIEnv argument.
  - Attaching a thread that is already attached is a no-op, can’t attach one thread to two VMs.
  - You must attach thread in order to call any JNI API (rust enforces it via JNIEnv lifetime)

Rust Part of the Callback (3)

- Attach the thread and make the call

```
impl CallbackContext {
    pub fn callback(&self, response: Request) -> anyhow::Result<Response> {
        // Attach current thread, guard derefs to JEnv
        let mut guard = self.jvm.attach_current_thread();
        let data = response.encode_to_java(&guard)?;
        // Call java back
        let response = guard // Derefs to &JNIEnv
            .call_method(
                self.global_ref.as_obj(),
                "onJniCallback",
                "([B)[B",
                &[JValue::from(&data)],
            );

        // Convert response to JObject
        let j_byte_array: JByteArray = response.l().into();
        Response::decode_from_java(&mut guard, &j_byte_array)
    }
}
```

Lets go async?

Async method to call

- We want to call async_handle_request method from Java:

```
async fn async_handle_request(request: Request) -> Response {
    if request.response_delay > 0 {
        // Emulate async IO
        tokio::time::sleep(Duration::from_millis(request.response_delay)).await;
    }
    let response = if request.message == "Hello, Rust!" {
        Response::success("Hello, Java!".into())
    } else {
        Response::error(format!("Unable to respond to '{}'", request.message))
    };
    response
}
```

Can’t await in synchronous methods

```
#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_async_JNIBridge_nativeCall<'a>(
    env: JNIEnv<'a>,
    j_object: JObject<'a>,
    request: JByteArray<'a>,
) -> JByteArray<'a> {
    handle_error(env, |env| unsafe {
        let request = Request::decode_from_java(env, &request)?;
        let response = async_handle_request(request).await;
        response.encode_to_java(env)
    })
}
```

- Compile error: “only allowed inside `async` functions and blocks”.
- We need to access an async runtime!

Storing the runtime

- Option 1: make it global and store in OnceCell.
- Option 2: Associate the runtime pointer with java object
  - Java

```
public class JNIBridge {
    private final long runtime;
    public JNIBridge() { // Can't have native constructor
        this.runtime = init();
    }
    private native long init();
}
```

  - Rust

```
#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_async_JNIBridge_init<'a>(
    _env: JNIEnv<'a>, _j_object: JObject<'a>,
) -> jlong {
    let runtime = Arc::new(Runtime::new().expect("Unable to init runtime"));
    Arc::into_raw(runtime) as jlong // Convert arc to pointer, forget it and cast to long
}
```

Accessing the runtime

- Access the field back:
- Get<Type>Field APIs are the way to go

```
/// Converts the value of the field into Arc without incrementing the Arc's ref counter.
pub unsafe fn get_field<T>(
    env: &mut JNIEnv,
    j_object: &JObject,
    field_name: &str,
) -> anyhow::Result<Arc<T>> {
    let field_value: JValueOwned = env
        .get_field(j_object, field_name, Primitive::Long.to_string())
        .with_context(|| format!("Can't get long value of the '{field_name}' field"))?;
    let j_long_value: jlong = field_value
        .j()
        .with_context(|| format!("Wrong type of the '{field_name}' field."))?;
    let ptr = j_long_value as *const c_void;
    Ok(Arc::from_raw(ptr.cast()))
}
```

Cleanup

- Java

```
public class JNIBridge implements AutoCloseable {
    @Override
    public native void close();
}
```

- Rust

```
const RUNTIME_FIELD: &str = "runtime";

#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_async_JNIBridge_close<'a>(
    mut env: JNIEnv<'a>,
    j_object: JObject<'a>,
) {
    unsafe {
        let _runtime = get_field::<Runtime>(&mut env, &j_object, RUNTIME_FIELD);
        // drop(_runtime); is called implicitly
    }
}
```

Accessing the runtime

- Borrow instead

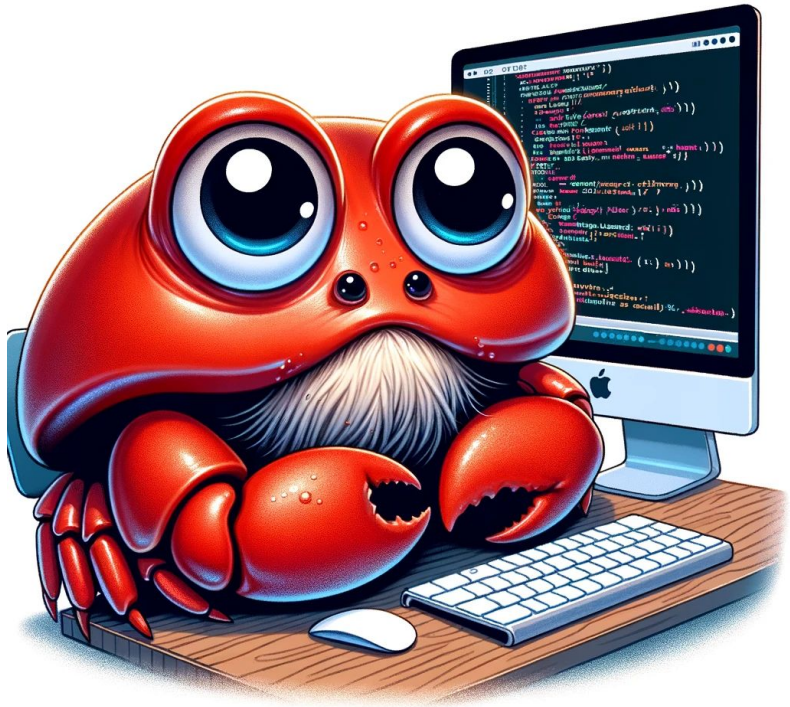
```
/// Converts the value of the field into Arc and increments the Arc's ref counter
pub unsafe fn borrow_field<T>(
    env: &mut JNIEnv,
    j_object: &JObject,
    field_name: &str,
) -> anyhow::Result<Arc<T>> {
    let handle: Arc<T> = get_field(env, j_object, field_name)?;
    let clone = handle.clone(); /// Increment ref-counter
    std::mem::forget(handle); /// Forget the Arc again!
    Ok(clone)
}
```

- Now we can call it this way:

```
let runtime: Arc<Runtime> = borrow_field(env, &j_object, RUNTIME_FIELD)? ;
```

Beware: passing the raw pointer to Java

- **Unsafe**
  - Ensure there is no code flow allowing user API input to change long value
  - Consider OnceCell<GlobalContext> for sensitive cases
- **Optimizations**
  - Java would complain that the field is not used (annotate with @Keep?)
- **Access by name**
  - Ensure the field is not renamed during obfuscation
- **Concurrency**
  - Use Box::into_raw and Box::from_row if you guarantee exclusive thread safe access from Java
  - Use Arc<T> if the value is read only
  - Otherwise, consider Arc<Mutex<T>> or Arc<RwLock<T>>



Blocking method

- Simple block-on:

```
#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_async_JNIBridge_nativeCall<'a>(
    env: JNIEnv<'a>,
    j_object: JObject<'a>,
    request: JByteArray<'a>,
) -> JByteArray<'a> {
    handle_error(env, |env| unsafe {
        let request = Request::decode_from_java(env, &request)?;
        let runtime: Arc<Runtime> = borrow_field(env, &j_object, RUNTIME_FIELD)?;
        let response = runtime.block_on(async_handle_request(request));
        response.encode_to_java(env)
    })
}
```

- Java thread is blocked
  - Bad idea to block main thread in Android or coroutine dispatcher threads
  - Could end up spawning too many threads

Use Java abstraction for delayed computation

- Java has Future and CompletableFuture class
- Return CompletableFuture from the native code

```
private native CompletableFuture<byte[]> nativeCall(byte[] request);
```

- Create instance using FindClass and NewObject APIs. env.new_object is a wrapper for both

```
const FUTURE_CLASS: &str = "java/util/concurrent/CompletableFuture";
const FUTURE_CONSTRUCTOR: &str = "()V";

pub struct ToyJniFuture {
    /// Pointer to java future object. GlobalRef garbage collection
    java_future: GlobalRef,
}

impl ToyJniFuture {
    pub fn new(env: &mut JNIEnv) -> anyhow::Result<Self> {
        let java_future_obj = env.new_object(FUTURE_CLASS, FUTURE_CONSTRUCTOR, &[])?;
        let java_future = env.new_global_ref(java_future_obj)?;
        Ok(Self { java_future })
    }
}
```

Beware: Classloaders

- Java Classloaders are associated with a caller class, not with a thread!
- If you create a thread in the rust code, it has no stack frames from java.
- The "system" class loader to be used
- Your app might have custom associated classloader
- Options:
  - Do your FindClass lookups in the method called by JVM
  - Cache a reference to the ClassLoader object somewhere, then manually call the loadClass method

- See more: Why didn't FindClass find my class?

Completing the future

- When ready, call complete method:

```
const COMPLETE_METHOD_NAME: &str = "complete";
const COMPLETE_METHOD_SIGNATURE: &str = "(Ljava/lang/Object;)Z";
impl ToyJniFuture {
    pub fn complete(&self, env: &mut JNIEnv, object: &JObject) -> anyhow::Result<bool> {
        let response: JValueOwned = env
            .call_method(
                self.java_future.as_obj(),
                COMPLETE_METHOD_NAME,
                COMPLETE_METHOD_SIGNATURE,
                &[JValue::from(object)],
            )
            .with_context(|| "Error calling complete")?;
        response
            .z()
            .with_context(|| "Method returned non-bool value"))
    }
}
```

- Note that we need &mut JNIEnv and JObject

Callback context

```
pub struct CallbackContext { // Send + Sync + 'static
    jvm: JavaVM, // Keep the JVM pointer
    future: ToyJniFuture, // And the future
}

impl CallbackContext {
    pub fn callback(&self, response: Response) -> anyhow::Result<()> {
        let mut guard = self // Attach current thread to the JVM
            .jvm
            .attach_current_thread()
            .with_context(|| "Unable to attach current thread to the JVM")?;
        if let Some(success) = response.success {
            let data = success.encode_to_java(&guard)?;
            self.future.complete(&mut guard, &data)?;
        } else { todo!()}
        Ok(())
    }
}
```

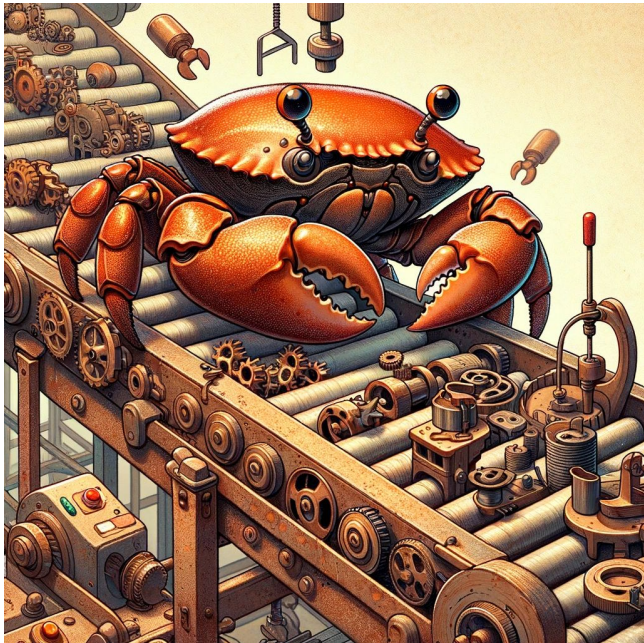
Implement native call

- Implement Java_com_github_kgrech_toyjni_async_JNIBridge_nativeCall:

```
#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_async_JNIBridge_nativeCall<'a>(...)
-> JObject<'a> {
    handle_error_exceptionally(env, |env| unsafe {
        let request = Request::decode_from_java(env, &request)?;
        let runtime: Arc<Runtime> = borrow_field(env, &j_object, RUNTIME_FIELD)?;
        let future = ToyJniFuture::new(env)?;
        let local_ref = future.local_reference(env)?;
        // If required, load required classes here and include them to CallbackContext
        let callback_context = CallbackContext::new(env, future)?;
        runtime.spawn(async move {
            let response = async_handle_request(request).await;
            callback_context
                .callback(response)
                .expect("Unrecoverable callback error");
        });
        Ok(local_ref)
    })
    .unwrap_or_default()
}
```

Beware: JMV calls in the async runtime thread

- Requires attachment and detachment of the async executor thread to JVM
- Could cause performance bottlenecks
- Consider:
  - Attach dedicated thread to VM
  - Dispatch computation results over the sync channel to this thread
  - Loop over result queue and call complete methods



Error handling

- Implement Java_com_github_kgrech_toyjni_async_JNIBridge_nativeCall:

```
#[no_mangle]
pub extern "system" fn Java_com_github_kgrech_toyjni_async_JNIBridge_nativeCall<'a>(...)
-> JObject<'a> {
    handle_error_exceptionally(env, |env| unsafe {
        let request = Request::decode_from_java(env, &request)?;
        let runtime: Arc<Runtime> = borrow_field(env, &j_object, RUNTIME_FIELD)?;
        let future = ToyJniFuture::new(env)?;
        let local_ref = future.local_reference(env)?;
        // If required, load required classes here and include them to CallbackContext
        let callback_context = CallbackContext::new(env, future)?;
        runtime.spawn(async move {
            let response = async_handle_request(request).await;
            callback_context
                .callback(response)
                .expect("Unrecoverable callback error");
        });
        Ok(local_ref)
    })
    .unwrap_or_default()
}
```

Throwing Exceptions

- Calling unwrap causes a panic in rust and likely to crash the JVM :(
- Throw and ThrowNew API are the way to go

```
const TOY_JNI_EXCEPTION: &str = "com/github/kgrech/toyjni/ToyJNIException";
pub fn throw_exception(env: &mut JNIEnv, error: impl Display) {
    // Check if there is an existing Java exception
    match env.exception_check() {
        Ok(true) => {
            // If there is an existing exception, return here to re-throw it
        }
        Ok(false) => {
            let result = env.throw_new(TOY_JNI_EXCEPTION, error.to_string());
            if let Err(err) = result {
                env.fatal_error(format!("Error throwing {TOY_JNI_EXCEPTION}: {err}"))
            }
        }
        Err(err) => env.fatal_error(format!("Exception check failed: {err}")),
    }
}
```

Passing exception to the future (1)

- Convert error the the exception and pass it
- Modify CallbackContext::callback:

```
impl CallbackContext {

    pub fn callback(&self, response: Response) -> anyhow::Result<()> {
        // Existing code to attach current thread to the JVM

        if let Some(success) = response.success {
            ...
        } else if let Some(error) = response.error {
            let exception = create_exception(&mut guard, error.error_message)?;
            self.future.complete_exceptionally(&mut guard, &exception)?;
        }
        Ok(())
    }
}
```

Creating Exceptions

- It is possible to create the instances of exceptions same as other Java objects
- FindClass and NewObject APIs could be used. env.new_object is a wrapper for both

```
const TOY_JNI_EXCEPTION: &str = "com/github/kgrech/toyjni/ToyJNIException";
const TOY_JNI_EXCEPTION_CONSTRUCTOR: &str = "(Ljava/lang/String;)V";

pub fn create_exception<'a>(env: &mut JNIEnv<'a>, error: impl Display) ->
                                                        anyhow::Result<JThrowable<'a>> {
    let java_sting = env.new_string(error.to_string())
        .with_context(|| "Error converting error string to Java string")?;
    let j_object = env.new_object(
            TOY_JNI_EXCEPTION,
            TOY_JNI_EXCEPTION_CONSTRUCTOR,
            &[JValue::from(&java_sting)],
        )
        .with_context(|| format!("Error creating {TOY_JNI_EXCEPTION}"))?;
    Ok(j_object.into())
}
```

Passing exception to the future (2)

- Call the completeExceptionally(Throwable ex) method
- The same pattern as earlier:

```
const COMPLETE_EXCEPTIONALLY_METHOD_NAME: &str = "completeExceptionally";
const COMPLETE_EXCEPTIONALLY_METHOD_SIGNATURE: &str = "(Ljava/lang/Throwable;)Z";

impl ToyJniFuture {
    pub fn complete_exceptionally(&self, env: &mut JNIEnv, object: &JThrowable) ->
                                                                    anyhow::Result<bool> {
        let response = env
            .call_method(
                self.java_future.as_obj(),
                COMPLETE_EXCEPTIONALLY_METHOD_NAME,
                COMPLETE_EXCEPTIONALLY_METHOD_SIGNATURE,
                &[JValue::from(object)],
            )
            .with_context(|| "Error calling completeExceptionally")?;
        response.z().with_context(|| "method returned non-bool value")
    }
}
```

How it looks in java?

```
public class JNIBridge {

    public CompletableFuture<Response.Success> nativeCall(Request request)
                                                                throws ToyJNIException {
        byte[] serializedRequest = request.toByteArray();
        CompletableFuture<byte[]> future = nativeCall(serializedRequest);
        return future.thenApply(response -> { // Could be dispatched to executor
            Response.Success deserializedResponse;
            try {
                deserializedResponse = Response.Success.parseFrom(response);
            } catch (Exception e) {
                throw new RuntimeException("Protobuf schema mismatch", e);
            }
            return deserializedResponse;
        }/*, executor*/);
    }

    private native CompletableFuture<byte[]> nativeCall(byte[] request)
                                                                throws ToyJNIException;
}
```

Thank you! Q&A

- Oracle JNI Specification
- The Java™ Native Interface
- https://www.linkedin.com/in/kgrech/
- kmgrechis@gmail.com

https://github.com/kgrech/toy-jni