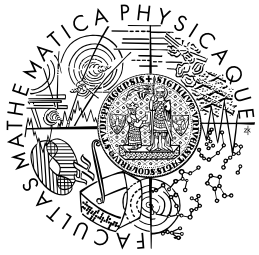

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Kateřina Průšová

L'anatra – Library for interoperability between C# and Java

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Computer Science – Software Systems

Study branch: High Performance Computing

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank Bc. Ondřej Roztočil for being my comrade in arms throughout this interoperability Odyssey and for helping me to retain my sanity even when it was proven that "*Dealing directly with JNI is a terrible experience that should be avoided at all costs.*" [20]. I would also like to thank my thesis supervisor, Mgr. Pavel Ježek, Ph.D., for helpful advice, constructive critique, and all the time he spent helping me to express my chaotic thoughts more clearly.

Title: L'anatra – Library for interoperability between C# and Java

Author: Bc. Kateřina Průšová

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: This thesis aimed at creating a software library enabling interoperability between C# and Java programming languages by enabling the use of a subset of Java features from C# code. Requirements that the library should meet were determined via analysis of potential use case scenarios and similar existing implementations. The emphasis was put on providing a user-friendly and type-safe API.

The implemented solution enables the invocation of static and instance Java methods from C#. It also allows creating Java object instances from C# code or obtaining them as return values of invoked Java methods. C# represents Java object instances as proxy classes that emulate the API of corresponding Java classes, possibly including an inheritance hierarchy between them and the interfaces they implement. Implementation of C# proxy types (classes and interfaces) is generated at compile time via the incremental source generator.

Generated proxy types are based on API provided by the implemented interoperability library. This library is mainly based on a combination of Java Native Interface (JNI) and .NET Platform Invoke (P/Invoke). However, the solution also experiments with optimizing certain invocation kinds using Foreign Function API provided by the recent Java Project Panama.

Keywords: Interoperability, C#, Java, JNI, Project Panama

Contents

1. Introduction

Both C# and Java are languages with long and rich histories, as the first version of C# was released in 2000 and the first version of Java in 1996. Moreover, both languages remain popular and widely used even today. StackOverflow's annual survey in 2023 [1] showed that 30.55 % of respondents have done extensive development work in Java over the past year or they plan to work in it over the next year. For C#, 27.62 % of respondents have reported using it or intent to use it in the following year. This puts Java and C# in seventh and eighth places, respectively, in the list of the most widely used programming languages in 2023. Both languages kept scoring similarly in the previous year of the survey as well.

As the both languages exist for significant amount of time and they are fairly popular among programmers, it is clear that a lot of code has been written in them and a lot of companies have code bases in one or both of these languages.

Both languages share similar base principles: they are garbage-collected, JITed, object-oriented languages with C-like syntax. It is clear, therefore, that it can be used in related scenarios, which may require interoperability between them. For instance, part of the system written in C# may need to be integrated with the part of the system written in Java, or C# developer may need to use a library that has been implemented in Java.

As Section 1.3 will show, there already exist some solutions for C# – Java interoperability problem. All of them are, however, either commercial and closed source, tailor-made for a particular use case and impossible to use in other scenarios, or outdated and unmaintained. There is, therefore, a need for a general, well-maintained, and well-documented C# - Java interoperability solution based on modern technologies. The objective of this thesis is to implement such a solution.

As the topic is complex and extensive, this thesis will limit itself to one of the two possible interoperability directions: direction from C# to Java. The thesis will aim at implementing a tool that should enable users to utilize code bases implemented in Java from C# applications. The opposite direction: working with constructs implemented in C# from Java programming language won't be supported. This leads to the first requirement of the thesis:

R1 Solution should support invocation direction from C# to Java. Opposite invocation direction (from Java to C#) won't be supported.

The rest of this chapter will explore potential use cases of the C# - Java interoperability tool and will analyze similar already existing implementations. Based on this analysis, a list of requirements the implemented tool should meet will be assembled.

1.1 Use cases

This section will reason about potential use cases for the C# - Java interoperability tool. The section will firstly discuss using C# Java interoperability within proprietary code bases to avoid system rewrites, and secondly, it will talk about using Java libraries from C# applications.

1.1.1 Interoperability between proprietary systems

As the introduction to this thesis mentioned, both C# and Java are and have been for a significant amount of time fairly popular programming languages, and therefore, a significant amount of code has been written in them over the years, and a lot of companies have code bases in one or both of these languages. These companies may face a situation in which they need to integrate a part of their system that is implemented in Java with another part that is implemented in C#. There are several options they may consider at that point.

The first option is a reimplementation of the part of the system from one language to the other. That can be especially tempting if the part of the system is older or a legacy system. Full system rewrites, however, tend to be demanding, time-consuming, and error-prone processes in which success is not guaranteed [2, 3, 4].

Another option a company has is to host the Java part and C# part of the system as two separate services and to manage the communication between them by some communication protocol. There exist multiple such protocols for intra-process communication (e.g., gRPC), the rest of this section, however, will work with the example of microservice-like infrastructure where individual subsystems are wrapped within a web API, and communication between them is managed via HTTPS protocol.

This approach undoubtedly has certain advantages. Many developers especially from the area of web applications backend development are familiar with microservice architectures and therefore system build as such will be easy for them to orientate in.

There are also disadvantages to consider. If the part of the system is not already a web API, it needs to be turned into one, which introduces additional infrastructural code and requires changes to existing software components. Additionally, communication via HTTPS protocol introduces performance overhead. Not all systems can afford this. Moreover, communication via HTTPS isn't, in its essence, safe. There can exist strongly typed models on both C# and Java sides of the communication; HTTPS, however, requires JSON serialization and deserialization, which loses type information, and one needs to make sure that models on both sides are kept up to date and in sync with their counterparts. A robustly designed interoperability solution can avoid all these issues.

An interoperability solution can be designed in a way that does not require modifications on the side that is being interoperated with it (Java side in our case). Interoperability solution can run both C# and Java part of the application in the same process (see Section 1.3), significantly decreasing the communication cost compared to HTTPS-based communication. It can also include the generation of strongly typed proxies that emulate Java types in C# code, allowing C# developers to work with Java types as if they were implemented in C#, decreasing the cognitive load imposed on a developer. To sum it up, interoperability can be a practical alternative to consider in the presented scenario.

1.1.2 Java libraries with no adequate alternative in .NET

A large amount of open-source and free-to-use libraries covering a huge variety of use cases have been developed for Java over the years. For some of these libraries,

no adequate free .NET alternative exists. For instance, based on our knowledge when this thesis is being written, there are no freely available .NET libraries for working with `.pdf` files. In the Java ecosystem, Apache PDFBox [5] library is available for this use case. Let's use this library as an example of a Java library that could be used from C# via our interoperability tool.

This section will examine the common usage of the Java PDFBox library, determining some requirements on the code constructs that our interoperability tool should support.

Code Snippet 1.1 demonstrates the API of Apache PDFBox library on the example of reading text from `.pdf` file as a string. Line 7 uses `Loader` class to load existing `.pdf` file. Lines 8 and 9 use `PDFTextStripper` class to obtain string content of the `.pdf`.

Code Snippet 1.1: Java: Example of Apache PDFBox library usage

```
1  import org.apache.pdfbox.Loader;
2  import org.apache.pdfbox.pdmodel.PDDocument;
3  import org.apache.pdfbox.text.PDFTextStripper;
4
5  static String ReadPdfAsText(String path) throws IOException {
6      File pdfFile = new File(path);
7      PDDocument pdf = Loader.loadPDF(pdfFile);
8      PDFTextStripper stripper = new PDFTextStripper();
9      String pdfContent = stripper.getText(pdf);
10     pdf.close();
11     return pdfContent;
12 }
```

Notice that while `.pdf` file is loaded by a static `loadPDF` method, the rest of the API requires creating instances of objects and invoking instance methods. Our interoperability solution should, therefore, allow users to do the same.

R2 As both Java and C# are object-oriented languages and object instances are a crucial part of the majority of API implemented in these languages, the solution should allow users to manipulate Java instances and invoke Java instance methods from C#.

1.2 Introduction to Interoperability tools

The previous section provided the motivation for this thesis in the form of potential use case scenarios. Section 1.3 will place the thesis in the context of already existing related implementations. To be able to reason about these implementations, however, it is necessary to first introduce basic interoperability tools that related implementations are often based on. That will be the topic of this section.

Firstly, the section will introduce Java Native Interface as a canonical means of interoperability between Java and native code (C, C++). Secondly, P/Invoke will be presented as a means of interoperability between C# and native code.

1.2.1 Java Native Interface

Java Native Interface (JNI) has been a canonical tool for Java to native calls from the very beginning of the Java platform. The first version of JNI was released with the JDK version 1.1 in 1997) [6]. In JDK release 1.2 in 1998, JNI was extended by several methods; since then, however, the interface has remained stable.

JNI only describes an API. Its implementation is a matter of each Java distribution on its own. Due to this fact, slight changes in the behavior of some JNI functions can be observed across different Java distributions, and on some distributions, particular JNI functions might not be supported at all (e.g., ART VM does not support `DefineClass` JNI function [7]).

JNI is most commonly used for Java to native direction of the invocation. However, direction from native to Java is supported as well. JNI allows programmers to embed JVM into a native application and subsequently to invoke Java methods and to manipulate Java objects from the native code. Next paragraphs will describe both directions of invocation in more detail.

Java to native invocation direction

Java language uses `native` keyword to mark Java methods that are under the hood implemented in native code. JNI is used to leverage the invocation of such methods. Code Snippet 1.2 shows a Java class containing the declaration of two `native` methods. Notice that the static initializer of the class calls `System.loadLibrary` method to load a native library that provides implementations of these methods. That is required to enable JNI to locate method implementations.

Code Snippet 1.2: Java: Example of Java class using JNI for native invocation

```
1  public class HelloJni {
2      public native int instanceNativeMethod(
3          int value, double doubleValue
4      );
5      public static native int staticNativeMethod(
6          int value, double doubleValue
7      );
8
9      static {
10         System.loadLibrary("MyNativeLibrary");
11     }
12 }
```

Unfortunately, JNI is unable to invoke arbitrary native methods. It expects the method to accept the first two parameters of particular types. The first parameter must be `JNIEnv` pointer. It allows native code to access JNI API, for instance, to carry out callbacks back to Java. This will be described in more detail in the section devoted to native to Java direction of invocation.

The second parameter differs depending on whether the corresponding Java `native` method is declared as static or non-static. Non-static methods require reference to Java object; for static methods, reference to Java class (instance of

`java.lang.Class`) is expected. The second parameter allows the native code to access the Java type that declares the native method or instance of that type. The first two compulsory parameters should be followed by a list of parameters that correspond to parameters of the Java `native` method.

Code Snippet 1.3 shows C-language native implementations of static and instance methods from Code Snippet 1.2. Notice that the types of the first two parameters match the JNI convention, and the rest of the parameters match the parameters of Java `native` methods from Code Snippet 1.2. Also, notice that the names of C-language methods consist of prefix `Java_` followed by the name of Java class declaring the `native` method (`HelloJni`) and the name of the Java `native` method itself (either `instanceNativeMethod` or `staticNativeMethod`). This naming convention allows JNI to locate the native implementations [8].

Code Snippet 1.3: C: Native implementation of Java methods

```
1  JNIEXPORT int JNICALL
2  Java_HelloJni_instanceNativeMethod(JNIEnv *env, jobject this, int
       value, double doubleValue){
3      // implementation of native method
4      ...
5  }
6
7  JNIEXPORT int JNICALL
8  Java_HelloJni_staticNativeMethod(JNIEnv *env, jclass cls, int
       value, double doubleValue){
9      // implementation of native method
10     ...
11 }
```

The requirements for the signature of native method implementation described above force developers to implement wrappers respecting the required signature around native functions they actually need to invoke from Java code. This effectively requires Java developers to write a JNI-based C-language code, which is not the most pleasant user experience.

R3 Solution should not require a user to modify a code of Java library in order to make it usable from C#.

Native to Java invocation direction

Previous section described more common use case of JNI: invocation of native methods from Java. This section will examine the direction of invocation that is more relevant for this thesis: from native to Java.

JNI API can be divided into two parts: invocation API [6, 8], which allows programmers to embed JVM into their native applications, and JNI interface functions that allow native code that already runs in the same process alongside JVM to invoke Java methods and access Java objects. JNI functions are accessible via the JNI interface pointer. Native code obtains JNI interface pointer in one of two ways:

- Either it is passed to the native method invoked from Java by JNI as the first of two compulsory arguments (see Code Snippet 1.3).

- Or JNI interface pointer is returned by the `JNI_CreateJavaVM` function, which spawns a JVM instance in the process of the native application.

Code Snippet 1.4 demonstrates creating a JVM instance inside of the C++ process. Lines 4 to 14 configure JVM, setting a version of JNI and options that would be normally passed to `java` command line command. Line 17 calls `JNI_CreateJavaVM` function that creates a JVM instance in the current process. This function returns (via its reference parameters) the handle of the JVM and JNI interface pointer.

Code Snippet 1.4: C++: Embeding JVM into C++ application

```
1  #include <jni.h>
2
3  int main(){
4      JavaVM* jvm;       // Java VM
5      JNIEnv* env;       // pointer to native method interface
6      JavaVMInitArgs vm_args;
7      JavaVMOption* options = new JavaVMOption[1]; // allows to
           specify options that would be normaly passed to java cmd
           line command
8      // specify Java class path
9      options[0].optionString =
           (char*)"-Djava.class.path=path/to/your/java/classes";
10     options[0].extraInfo = 0;
11     vm_args.version = JNI_VERSION_1_8;
12     vm_args.nOptions = 1;
13     vm_args.options = options;
14     vm_args.ignoreUnrecognized = false; // ignore unreckognized
           options
15
16     // load and initialize a Java VM, return a JNI interface
           pointer in env
17     auto rc = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
18     if (rc == JNI_OK) {
19         /* here use jvm to invoce Java methods and to manipulate
               Java objects */
20         jvm->DestroyJavaVM(); // clean up
21     }
22 }
```

Once the JNI interface pointer is obtained, it can be used to call JNI functions that allow native code to obtain handles of Java objects and methods, invoke Java methods, and manipulate Java objects via these handles. We will demonstrate this on the example of a Java class from the Code Snippet 1.5. This Java class contains the declaration of static method `myStaticMethod`.

Code Snippet 1.5: Java: Java class with static method

```
1 public class HelloJni {
2    public static double myStaticMethod(
3         int value, double longValue) {
4         return longValue + value;
5     }
6 }
```

Code Snippet 1.6 shows JNI-based C++ code that invokes this Java method. Notice that the first `FindClass` JNI function is called (via JNI interface pointer `env`) to obtain the handle of Java type. Then, the method id is obtained by `GetStaticMethodID` JNI call.

Notice that in addition to the handle of Java type that defines the method and the method name, `GetStaticMethodID` call takes the third string parameter that encodes the method signature. In our example, the Java method that should be located takes one integer and one double parameter and returns a double return value. An integer is encoded as `I`, double is encoded as `D`. Parameter types are enclosed in braces, and the return type follows these braces [8]. That gives us method signature `"(ID)D"` (see line 2 in Code Snippet 1.6).

Both `FindClass` and `GetStaticMethodID` return null if the required type or method is not found. Null checks are omitted for brevity.

Code Snippet 1.6: C++: Invocation of static method via JNI

```
1 jclass cls = env->FindClass("HelloJni");
2 jmethodID staticMethodId = env->GetStaticMethodID(
3     cls, "myStaticMethod", "(ID)D");
4 double res = env->CallStaticDoubleMethod(
5     cls, staticMethodId, 42, 0.73);
6 std::cout << "RES: " << res << std::endl; // RES: 42.73
```

The previous example demonstrated the invocation of the static Java method from C++. The following example will show how to work with an instance of a Java object. An example of a Java class defining constructor and instance method can be seen in Code Snippet 1.7.

Code Snippet 1.7: Java: Java class with instance method

```
1  public class HelloJni {
2          private final int field;
3
4          public HelloJni(int value){
5                  field = value;
6          }
7
8          public double myNonStaticMethod(int value, double
               longValue){
9                  return longValue + value + field;
10         }
11 }
```

Code Snippet 1.8 contains C++ code that creates an instance of this class and invokes a method on this instance. After the Java type handle is obtained, `GetMethodID` function is used to obtain a method id of the constructor of the Java type. Constructor is identified by special method name `<init>` and by its signature. Notice that the return type of the constructor is `V`: void.

Once the constructor method id is obtained, the constructor is invoked by `NewObject` JNI function. Then, the method ID of the instance method can be obtained, and the method can be invoked by `CallDoubleMethod` call. Notice that this call takes a Java instance handle as the first parameter, whereas `CallStaticDoubleMethod` call in Code Snippet 1.6 takes a Java type handle in order to invoke the static method.

Jni also allows to access Java fields in the similar manner.

Code Snippet 1.8: C++: Working with Java object via JNI

```
1 jclass cls = env->FindClass("HelloJni");
2 jmethodID ctorId = env->GetMethodID(cls, "<init>", "(I)V");
3 jobject instance = env->NewObject(cls, ctorId, 42);
4 jmethodID nonStaticMethodId = env->GetMethodID(cls,
     "myNonStaticMethod", "(ID)D");
5 double res = env->CallDoubleMethod(instance, nonStaticMethodId,
     42, 0.73);
6 std::cout << "RES: " << res << std::endl; // RES: 84.73
```

Local and global references

Previous code examples worked with references to Java objects and types. JNI distinguishes two main kinds of references: **local** references and **global** references. [6] JNI function introduced so far (e.g. `FindClass`, `NewObject`) return **local references** to Java objects. Local references are only valid in the dynamic context of the invocation of the native method that created them. Once the native method returns back to Java, all local references are freed. Local references are also only valid in the thread that created them. The amount of local references that can exist simultaneously may be fairly limited. The precise number depends on a particular Java distribution. However, JNI specification only requires implementations to reserve slots for 16 local references [6] at a time. When the usage of local references is more extensive, the programmer should remember to free no longer needed local references by calling `DeleteLocarRef` JNI function.

Unlike local references, global references stay valid across multiple native method invocations, and they can be used from threads other than the thread that created them. Global reference can only be obtained from a local reference by calling `NewGlobalRef` function (see Code Snippet 1.9). Global references are never freed automatically. It is the programmer's responsibility to free them when they are no longer needed.

Code Snippet 1.9: C++: Creating global JNI reference

```
1 jobject localReference = env->NewObject(cls, ctorId, 42);
2 jobject globalReference = env->NewGlobalRef(localReference);
3 env->DeleteLocalRef(localReference); // no longer needed
```

Both local and global references are GC roots of Java garbage collector. While they are valid, they therefore keep referenced Java object alive. There also exist JNI weak references that do not keep Java objects alive.

JNI functions for looking up method and field handles (such as previously show `GetStaticMethodID` function) do not return references. They return handles that are not considered expensive resources, and that can be cached across multiple native method invocations [6].

JNI interface pointer

JNI interface pointer (`JNIEnv`) is a pointer to a thread-local data that contains a pointer to a JNI function table (see Figure 1.1). Due to that, the JNI interface pointer returned by `JNI_CreateJavaVM` call can only be used by the thread that created JVM. If other threads need to call a JNI function, they first need to obtain their own JNI interface pointer by calling `AttachCurrentThread` function.

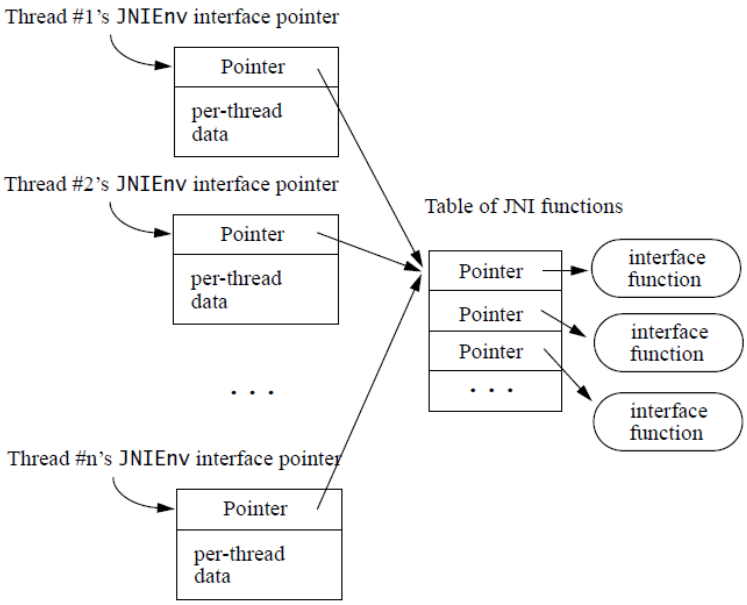

Figure 1.1: JNI interface pointer (taken from [6])

1.2.2 P/Invoke

The previous section introduced JNI as a common interoperability tool between Java and native code. This section will focus on P/Invoke (Platform Invoke) [9] as a common mean of interop between .NET and native code. P/Invoke is available since .NET Framework 1.1. [10] in form of `DllImport` attribute. This attribute can be applied to `static extern` method that matches the signature of a native method. Code Snippet 1.10 shows the declaration of `Sleep` method that matches the signature of `sleep` function from `kernel32.dll`. When the method is called, the native function will be invoked.

Code Snippet 1.10: C#: Usage of DllImport attribute

```
[DllImport("kernel32.dll")]
static extern void Sleep(uint dwMilliseconds);
```

Native method invocation using `DllImport` attribute requires IL stub to be generated at runtime based on annotated method signature and `DllImport` attribute parameters. This IL stub handles marshaling of method parameters and leverages native method invocation itself [11]. Run time code generation is problematic because it stands in the way of potential AOT compilation of .NET [11]. Due to that, .NET 7 added `LibraryImport` attribute, which uses Incremental Source Generators to generate marshaling code at compile time, getting rid of the need for runtime generated stub. `LibraryImport` attribute gets applied to `static partial` method so that Incremental Source Generator can generate method body. Code Snippet 1.11 demonstrates the usage of `LibraryImport` attribute on the example of the same `sleep` function from `kernel32.dll` that was used in Code Snippet 1.10.

Code Snippet 1.11: C#: Usage of LibraryImport attribute

```
[LibraryImport("kernel32.dll")]
static partial void Sleep(uint dwMilliseconds);
```

Under the hood, both `DllImport` and `LibraryImport` use the same mechanism to leverage native method invocation; the only difference is in the mean of generating the marshaling code. If no marshaling code is needed, the source-generated implementation of `LibraryImport` method is just a method annotated by `DllImport` attribute (see Code Snippet 1.12).

Code Snippet 1.12: C#: Source generated implementation of method from the previous code snippet

```
[System.Runtime.InteropServices.DllImportAttribute("kernel32.dll",
    EntryPoint = "Sleep", ExactSpelling = true)]
public static extern partial void Sleep(uint dwMilliseconds);
```

.NET 5 also added an option to cast native function pointers (represented as `IntPtr` type) to strongly typed delegates that can be then invoked as normal .NET delegates. Code Snippet 1.13 shows the usage example. Notice that unlike when using `LibraryImport` or `DllImport` attribute, the user is required to explicitly load the native library to obtain a function pointer.

Code Snippet 1.13: C#: Usage of unsafe function pointer

```
IntPtr kernelLib = NativeLibrary.Load("kernel32.dll");
IntPtr sleepPtr = NativeLibrary.GetExport(kernelLib, "Sleep");
unsafe
{
    var sleep = (delegate* unmanaged[Stdcall]<uint, void>)sleepPtr;
    sleep(5000);
}
```

1.3 Overview of existing implementations

The previous section introduces the most common tools for interoperability between Java and native code and .NET and native code. The following section will build on this knowledge and analyze existing implementations of interoperability between .NET and Java.

1.3.1 Java interop for Xamarin.Android and .NET MAUI

One of the scenarios that requires interoperability between C# and Java is the development of Android applications in C#.

Android operating system exposes its application framework (API for developing Android applications) in the form of Java classes [12]. The main of these classes is `android.app.Activity` class [13]. Android application consists of custom `Activity` classes that inherit base `android.app.Activity` class. `android.app.Activity` declares a set of virtual methods that a custom `Activity` developer needs to override in order to implement their functionality (e.g. `OnCreate` method that is invoked by the Android application framework to initialize the activity).

Microsoft provides a Xamarin.Android framework (and later .NET MAUI framework), that allows to develop Android applications in C# [14]. Xamarin.Android applications are run in the process that contains two execution environments running side by side (see Figure 1.2 [15]) - .NET part of the application is run by Mono Runtime, and the Java part is run by Android Runtime virtual machine (ART) (Android-specific implementation of Java virtual machine) [16]. At the core of Xamarin.Android, there is a JNI and P/invoke based (see section 1.2) interoperability engine that bridges .NET and Java worlds. .NET MAUI shares this interoperability engine.

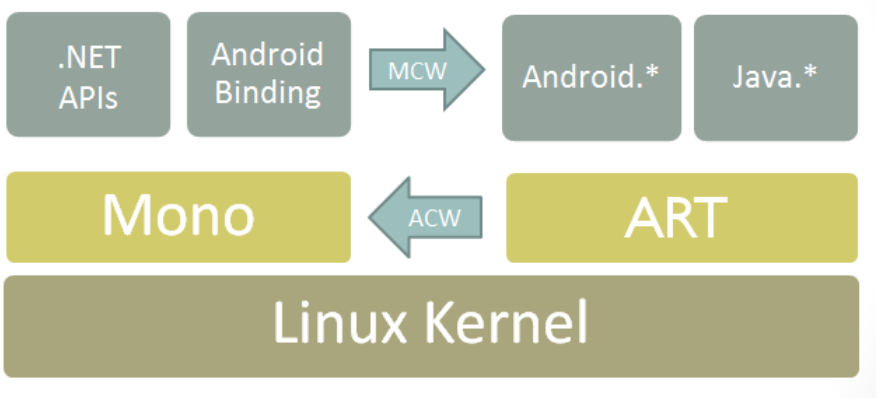

Figure 1.2: Xamarin.Android high-level architecture

Xamarin.Android contains .NET bindings of Java classes defined in Java `android` namespace [16] (including `android.app.Activity` class). That allows .NET developers to work with Java `android` classes as if they were implemented in .NET, effectively allowing them to **invoke Java methods from .NET**. In Xamarin.Android context, these .NET classes that serve as a binding of Java classes are referred to as Managed Callable Wrappers (MCW) (see Figure 1.2).

Managed Callable Wrapper of `android.app.Activity` Java class is `Android.App.Activity` .NET class. When developing an Android application with Xamarin.Android, .NET developer must inherit `Android.App.Activity` and override its virtual methods (same process as with Java). This effectively means that Xamarin.Android must enable **overriding Java methods from .NET**.

Android custom `Activity` class implementation can also contain methods that serve as event handlers for UI-induced events (e.g., button-clicked event). Java Android application runtime must be able to look up and invoke these methods; therefore, Xamarin.Android must enable **invoking .NET methods from Java**.

Last but not least, Xamarin.Android developers may need to provide .NET implementation of Java interfaces (e.g. `android.content.ComponetCallbacks` interface that sets callbacks common to multiple Android application components [17, 18]). Xamarin.Android Interop, therefore, must enable to **implement Java interfaces in .NET**.

It may seem that Xamarin.Android provides a complete and general solution to the .NET - Java interoperability problem. Interoperability engine in Xamarin.Android was, however, developed solely for the purposes of Xamarin.Android and it is strongly coupled with it. Following paragraphs will explore design and implementation of Xamarin.Android interop in more detail to identify its issue as well as aspects that may serve as an inspiration for this thesis.

Android Callable Wrappers

The previous paragraphs have explained that to enable development of Android applications in .NET, Xamarin.Andriod interop must enable:

- inheriting Java classes in .NET,
- overriding Java methods from .NET,
- invoking methods defined in .NET classes from Java,
- providing .NET implementations of Java interfaces.

The problem with these actions is that to support them, one must be able to invoke methods defined on .NET types from Java. From the point of view of the Java type system, however, no such type exists as Android Runtime (ART) does not support runtime registration of new classes (specifically, it does not support JNI `DefineClass` function) [7], Xamarin.Android needs to generate so-called Android Callable Wrappers (ACW) (sometimes also called Java Callable Wrappers (JCW)) see figure 1.2. ACW are Java stub classes that correspond to .NET classes. They are the representation of .NET classes in the Java type system.

Xamarin.Android, to some extent, emulates Java's inheritance hierarchy. It exposes `Java.Lang.Object` class [19] that is a Managed Callable Wrapper of the root class of Java inheritance hierarchy `java.lang.Object`. Android callable wrapper will, by default, be generated for every .NET class that (directly or indirectly) inherits `Java.Lang.Object` [20].

For example, for the .NET activity class in Code Snippet 1.14, the ACW in Code Snippet 1.15 will be generated as `Android.App.Activity` indirectly inherits `Java.Lang.Object` [21].

Code Snippet 1.14: C#: Example of .NET implementaion of Andriod Activity

```
public class HelloAndroid : Android.App.Activity {
    protected override void OnCreate (Bundle savedInstanceState) {
        base.OnCreate (savedInstanceState);
        SetContentView (R.layout.main);
    }
}
```

At line 27 of Code Snippet 1.15 notice the declaration of `native n_onCreate` method. This method will be bound to .NET `OnCreate` implementation that will then be invokable from Java via JNI. Similar `native` method will be generated for every .NET override of existing Java method [20]. Also, notice that ACW inherits `android.app.Activity` class, which is Java equivalent of `Android.App.Activity`: ACWs emulate the .NET inheritance hierarchy. The rest of the code in Code Snippet 1.15 serves the purposes of a native method registration process that is rather convoluted and that will be described in a bit more detail below.

Code Snippet 1.15: Java: Example generated Andriod callable wrapper

```
1  public class HelloAndroid extends android.app.Activity {
2      static final String __md_methods;
3      static {
4          __md_methods =
5          "n_onCreate:(Landroid/os/Bundle;)V" +
6          ":GetOnCreate_Landroid_os_Bundle_Handler\n" +
7          "";
8          mono.android.Runtime.register (
9          "Mono.Samples.HelloWorld.HelloAndroid, HelloWorld,
               Version=1.0.0.0, Culture=neutral, PublicKeyToken=null",
10         HelloAndroid.class,
11         __md_methods);
12     }
13
14     public HelloAndroid () {
15         super ();
16         if (getClass () == HelloAndroid.class)
17         mono.android.TypeManager.Activate (
18         "Mono.Samples.HelloWorld.HelloAndroid, HelloWorld,
               Version=1.0.0.0, Culture=neutral, PublicKeyToken=null",
19         "", this, new java.lang.Object[] { });
20     }
21
22     @Override
23     public void onCreate (android.os.Bundle p0) {
24         n_onCreate (p0);
25     }
26
27     private native void n_onCreate (android.os.Bundle p0);
28 }
```

To export .NET methods that are not overrides of existing Java methods (e.g., .NET handles of Android UI events) to ACW, `Export` attribute [22] must be used. See an example of its usage in Code snippet 1.16.

Code Snippet 1.16: C#: Example of .NET implementaion of Andriod Activity

```
1  public class HelloAndroid : Andriod.App.Activity {
2      protected override void OnCreate (Bundle savedInstanceState) {
3          base.OnCreate (savedInstanceState);
4          SetContentView (R.layout.main);
5      }
6
7      [Java.Interop.Export("ButtonClick")]
8      public void ButtonClick(View v) {
9          // implementation
10         ...
11     }
12 }
```

Native method registration

The previous paragraph introduced the concept of Android Callable Wrappers, Java stub classes that get generated for every .NET type that should be used from Java. ACWs contain `native` method declaration for every .NET method that is supposed to be callable from Java. When the Java `native` method gets called, the corresponding .NET implementation should be invoked. What remains unclear, however, is how Java (JNI) identifies the particular .NET method that should be invoked when the `native` method gets called. That is handled by the native method registration process, which will be described in this section.

Line 8 of Code Snippet 1.15 shows that the static initializer of ACW calls `mono.android.Runtime.register` method. This method is the entry point of the native method registration process. The process itself is complex and can be more easily explained on the usage of `Register` attribute [23]. `Register` attribute is an infrastructural attribute that is not intended to be used by common Android application developers. It is used while manually implementing Java bindings using Xamarin.Android JNI wrapper. The attribute serves a similar purpose as the previously mentioned `Export` attribute (exporting .NET method to ACW class), but it requires that certain infrastructural code is implemented manually, which makes it easier to understand individual components involved in the native method registration process.

Native method registration is eventually done via JNI call `RegisterNatives` [8]. Each native method being registered is passed to the JNI `RegisterNatives` function as the following struct.

Code Snippet 1.17: C: Jni representation of native method binding

```
1 typedef struct {
2     char *name;
3     char *signature;
4     void *fnPtr;
5 } JNINativeMethod;
```

The first two struct fields identify a Java method marked with `native` keyword by its name JNI type signature [8] (see Section 1.2.1). The third field is a function pointer to the method implementation (in our case, the .NET method) that should be invoked when the Java `native` method is called.

Xamarin.Android must be able to provide these 3 parameters to the JNI `RegisterNatives` function for each method it exports to ACW. That is where `Register` attribute becomes relevant. Code snippet 1.18 shows `Register` attribute usage.

Code Snippet 1.18: C#: Usage of Register attribute

```
1 partial class ManagedAdder : Adder { // suppose that Adder is MCW
      of exiting Java class
2     [Register ("add", "(II)I", "GetAddHandler")]
3     public override int Add (int a, int b) {
4         return a + b;
5     }
6 }
```

Notice that the attribute takes three constructor parameters. The first two exactly correspond to the first two fields of `JNINativeMethod` struct (see Code snippet 1.17), and they identify the Java method by its name and signature. The third field of `JNINativeMethod` struct requires a function pointer to the method implementation. However, as pointers are runtime entities, the third parameter of `Register` attribute is the name of the method that is supposed to return the pointer (.NET delegate) rather than the pointer itself. Such a method is in the context of Xamarin.Android referred to as **connector**. Connector method generated for the method from Code snippet 1.18 can be seen at line 3 of Code Snippet 1.19 [20]:

Code Snippet 1.19: C#: Generated infrastural methods required during native method registration process

```
1 partial class ManagedAdder : Adder {
2     static Delegate cb_add;
3     static Delegate GetAddHandler () {
4         if (cb_add == null)
5         cb_add = JNINativeWrapper.CreateDelegate ((Func<IntPtr,
              IntPtr, int, int, int>) n_Add);
6         return cb_add;
7     }
8
9     static int n_Add (IntPtr jnienv, IntPtr lrefThis, int a, int
          b) {
```

```
10          Adder __this = Java.Lang.Object.GetObject<Adder>(lrefThis,
                JniHandleOwnership.DoNotTransfer);
11          return __this.Add (a, b);
12      }
13  }
```

Notice that `GetAddHandler` connector method does not return a direct delegate to `Add` method, which is supposed to be invokable from Java. Instead, it returns delegate to method `n_add`, which in its body calls `Add` method. This is because the JNI `RegisterNatives` function expects that the delegate will point to a function that has JNI compatible signature [8] (see Section 1.2.1). Due to that, `n_add` **marshaller** method is needed. JNI `RegisterNatives` function will register `n_add` .NET method as a "native" implementation of Java `Add native` method enabling to invoke `n_add` from Java, transitively invoking .NET `Add` implementation.

Parameters of `Register` attribute influence generated ACW. In Code snippet 1.20 shows ACW generated for `ManagedAdder` class from Code Snippet 1.18. Notice that `mono.android.Runtime.register` method gets passed the string containing parameters of `Register` attribute: name of Java `native` method, encoding of the type signature of this method, and the name of connector method. Compare it with lines 5 and 6 in Code Snippet 1.15 that do the same for the previously mentioned `OnCreate` method.

Code Snippet 1.20: Java: ACW for ManagedAdder .NET class

```
1  public class ManagedAdder extends mono.android.test.Adder {
2      static final String __md_methods;
3      static {
4          __md_methods = "n_add:(II)I:GetAddHandler\n" +
5          "";
6          mono.android.Runtime.register (/*some irrelevant
               parameters*/, __md_methods);
7      }
8      @Override
9      public int add (int p0, int p1) {
10         return n_add (p0, p1);
11     }
12 private native int n_add (int p0, int p1);
13 }
```

Java method `mono.android.Runtime.register` is actually declared as `native` [24], and it is a callback to the .NET method in Xamarin.Android runtime. There, the method name and JNI signature of Java `native` method and the name of the .NET connector method are parsed from the passed string, the connector is invoked via Reflection, and JNI `RegisterNatives` function is called [25]. This process was referred to as "*an unholy mess of string splitting and Reflection*" by .NET Android Tech Lead Jonathan Pryor [26, 27].

The method registration process, as described above, introduces a strong implicit coupling between the ACW generator and native method registration. ACW generator must produce the correct format of the string parameter of

`mono.android.Runtime.register` method. The registration process then has to be able to parse `RegisterNatives` function from this string.

What is more, when user-friendly `Export` attribute is used instead of the internal `Register` attribute, equivalents of an infrastructural glue code such as **connector** and **marshaller** .NET methods need to be generated in runtime [26, 28], which is problematic for potential AOT compilation of .NET code using Xamarin.Android.

R4 Solution should avoid runtime code generation.

In summary, native method registration process in Xamarin.Android interop is convoluted, depending on string splitting magic, reflection and runtime generation of code. It also introduces implicit dependencies between components of Xamarin.Android that would at the first glance seem independent.

Managed Callable Wrappers

Previous sections focused mainly on Java to .NET direction of interop invocation. Xamarin.Android, however, also enables the opposite direction of invocation, mainly to provide a Xamarin.Android .NET developer access to rich variety of Java Android libraries.

Xamarin.Android is able to generate so-called **Bindings** libraries that bind each class of a particular Java library to a .NET Managed Callable wrapper class, which uses JNI to enable .NET developer to work with Java class as if it was implemented in C#. Binding libraries can be generated from JAR files using a Visual Studio template. Xamarin.Android tooling, however, currently only supports binding of Java libraries that were built for Android [15].

The process of generating the Binding library itself file is quite straightforward [15], but the potential customization of generated code is quite cumbersome. To generate Binding library Xamarin.Android will inspect the provided JAR and will generate a list of all packages, classes, and members to be bound. It stores this list to `api.xml` file, which will be later on used to generate MCWs. `api.xml` file may look as follows [29]:

Code Snippet 1.21: Example of api.xml

```
<api>
  <package name="android">
    <class
      abstract="false"
      deprecated="not deprecated"
      extends="java.lang.Object"
      extends-generic-aware="java.lang.Object"
      final="true"
          name="Manifest"
          static="false"
          visibility="public">
        <constructor deprecated="not deprecated" final="false"
                name="Manifest" static="false"
                   type="android.Manifest"
```

```
                visibility="public">
        </constructor>
        </class>
...
</api>
```

Potential changes to generated API, however, are not done via editing `api.xml` file. Rather they need to be written in one of three additional `.xml` files [29]:

- `MetaData.xml` - for renaming namespaces, classes and members, renaming or removing them
- `EnumFields.xml` - for mapping between Java int constants and C# enums
- `EnumMethods.xml` - for changing method parameters and return types from Java int constants to C# enums

Code snippet 1.22 [29] shows the example of metadata .xml.

Code Snippet 1.22: Example of MetaData.xml

```
<metadata>
  <!-- Normalize the namespace for .NET -->
  <attr path="/api/package[@name='com.evernote.android.job']"
    name="managedName">Evernote.AndroidJob</attr>

  <remove-node
     path="/api/package[@name='com.evernote.android.job.v14']" />
  <remove-node
     path="/api/package[@name='com.evernote.android.job.v21']" />

  <attr  path="/api/package[@name='com.evernote.android.job']
  /class[@name='JobManager']/method[@name='fo
     rceApi']/parameter[@name='p0']"
   name="name">api</attr>
</metadata>
```

Requiring a .NET developer to write multiple `.xml` to customize generated binding classes seems a highly unpleasant developer experience. It may be acceptable in the Android context as the UI part of the Android app is declared in `.xml`, and therefore, Android developers are used to it, but in the context of general .NET development, it does not seem ideal.

R5 Configuration of generated proxies should be user-friendly and in-code.

Another option is to obtain customized .NET bindings of Java classes using Xamarin.Android is to implement them manually using the .NET wrapper of JNI API [20]. This process is cumbersome and requires knowledge about JNI and an understanding of the implementation details of Xamarin.Android JNI wrappers.

Problems
Xamarin Android interop introduced in this section has several shortcomings:

- It is dependent on MONO runtime and ART virtual machine.

R6 Solution should function on modern multiplatform .NET and on common Java distributions and versions.

- Its API is designed for implementing Android applications in C#. Most common activities expected within this use case (overriding `OnCreate` method, generating MCW bindings without customization) can be achieved in an easy and user-friendly manner. Any more specific action (customizing MCW via `.xml`, using JNI manually) is highly user-unfriendly.
- It depends on the runtime generation of code, which stands in the way of potential AOT compilation.
- Native method registration is cumbersome and introduces implicit strong coupling between otherwise independent components. Today's developers can imagine that the process could be handled more elegantly using Incremental source generators.

1.3.2 IKVM.NET

IKVM.NET [30, 31] approaches the interoperability problem on a different level than the previously described Xamarin Java.Interop. Xamarin Java.Interop is an API level bridge. It allows cross-language invocation of methods and cross-language usage of objects. Using Xamarin Java.Interop, Java-side code runs on JVM, and .NET-side code runs on CLR. IKVM.NET, however, runs Java bytecode on top of .NET (originally .NET framework and Mono, now .NET as well). It is an implementation of Java for the .NET platform.

IKVM.NET can be used either in dynamic mode or in static mode. Dynamic mode reads Java class files at runtime, and JIT compiles them into CIL code. This is heavily based on reflection. IKVM.NET even contains its own fork of `System.Reflection`: `IKVM.Reflection` which extends reflection API by some functionality that IKVM.NET requires.

In static mode, JAR files are AOT compiled into .NET assemblies, which can then be used as normal .NET assemblies from .NET code.

IKVM.NET only allows Java code to run on top of .NET. It does not support the opposite direction: running .NET code on JVM [30].

IKVM.NET contains of several components [30, 31]:

- Java virtual machine implemented in .NET,
- Java class libraries precompiled to .NET using IKVM.NET,
- .NET reimplementations of native methods is Java class path,
- static (AOT) compiler of Java bytecode to CIL,
- JIT compiler of Java bytecode to CIL.

Project history and the current state
The project was started by Jeroen Frijters in 2002 [32]. In 2017 he decided to abandon the project, saying: "*I've slowly been losing faith in .NET. Looking back, I guess this process started with the release of .NET 3.5. On the Java side things don't look much better. The Java 9 module system reminds me too much of the generics erasure debacle. I hope someone will fork IKVM.NET and continue working on it.*" [33].

In 2018, Windward Studios forked IKVM from SVN to git, promising to keep supporting it and to help to build an open-source community around it [34, 35]. They, however, did not continue the development of IKVM.NET [36].

In 2022, Jerome Haltom and others moved the project to a new GitHub organization, where they continue its development since [37].

Take away
Implementing IL code level interop as IKVM.NET does seems to be an enormous task. Many aspects need to be taken into consideration to ensure that the semantics of Java byte code stay the same when it is executed on top of .NET [32]. This thesis does not aim to implement a. NET-based JVM. Rather, it will attempt to implement API level interop bridge.

R7 Interop will work on API level: allowing invocation of Java methods and usage of Java objects from C#. It won't be IL-level interop.

1.3.3 JCOBridge and JNet

JOCBridge (JVM CLR Objects Bridge) [38] is a commercial, closed-source API level interop solution. As Xamarin Java.Interop, JOCBridge hosts both JVM and CRL in the same process and uses JNI to manage communication between them. As it is closed-sourced, however, it is impossible to reason about aspects of its implementation in more detail.

JOCBridge on its own seems to heavily depend on the usage of `dynamic` objects, providing the users with late binding and lacking IntelliSense and type safety [38].

R8 Solution should provide C# proxies for Java classes with static type safety. Dynamic objects should not be used for the proxies.

This problem is addressed by JNet project [39], which provides a set of tools built on top of JOCBridge and enriches JOCBridge with additional functionality. Part of JNet is the JNetReflector command line tool [40] that statically generates .NET proxy classes based on Java classes presented in the provided JAR file. JNet project contains a set of pre-generated proxies for common classes from the Java standard library; however, if the user needs to generate a proxy for their own Java classes, they need to invoke the JNetReflector tool manually.

R9 Solution should generate .NET proxies for custom Java classes without requiring explicitly using an external tool.

Even though JNet is open-source, it cannot be used without a commercial JCOBridge layer.

1.3.4 Unity Android JNI wrappers

Unity game engine (closed-source) [41] contains a JNI wrapper allowing users to utilize Java plugins from their C# Unity scripts [42]. This wrapper is fairly low-level. It contains `AndroidJNI` class [43], which is just a P/Invoke-based (see section 1.2.2) wrapper of JNI functions (see section 1.2.1) and provides almost none additional abstraction.

Additionally, it contains a slightly higher-level `AndroidJavaClass` class that represents the concept of Java type (corresponding to `java.lang.Class`) and `AndroidJavaObject` class that serves as a type-less interface to an instance of any Java class, allowing to call instance methods (see Code Snippet 1.23) and to access instance fields.

Code Snippet 1.23: Usage of AndroidJavaObject from Unity JNI wrappers

```
1 using (AndroidJavaObject jo = new
     AndroidJavaObject("java.lang.String", text))
2 {
3         int hash = jo.Call<int>("hashCode");
4         return hash;
5 }
```

Code Snippet 1.23 shows that representing multiple Java types by single .NET types makes code stringly-typed, losing static type safety and allowing the user to attempt to invoke arbitrary methods on instances that do not implement them, which is fairly similar to the situation when dynamic objects are used for the proxies (see section 1.3.3), and it should be avoided in this thesis.

R10 To ensure static type safety, the solution should provide separate C# proxies for separate Java types. It should not be possible to represent instances of a Java type by instances of a proxy that does not correspond to that particular Java type.

1.3.5 Other commercial implementations

In addition to the previously mentioned JCOBridge (refer to Section 1.3.3), there are also other commercial tools available for interoperability between .NET and Java. This section will provide a brief overview of two such tools.

JnBridge

JnBridge [44] is a complex commercial solution. It supports an in-process API-level JNI-based interop (in the JnBridge context referred to as shared memory communication) similar to one provided by Xamarin or JOCBridge. In addition to that, however, it also supports interop between potentially multiple instances of JVM and CLR running in different processes either on the same machine or on different machines either via TCP protocol or via proprietary binary protocol. This form of communication has significant overhead over JNI-based in-memory solution [45].

JnBridge fully supports both directions of invocation: from .NET to Java and from Java to .NET: both as callbacks and as normal method invocations.

JnBridge allows user to generate proxy classes. This can be done via Visual Studio plugin (when generating .NET proxies for Java classes), Eclipse plugin (when generating Java proxies for .NET classes) or via JnBridge GUI.

It is a completely close source solution, and its public documentation provides minimal insight into its internal implementation details.

Javonet

Javonet [46] claims to be a universal interop solution for the cartesian product of languages Java (or Kotlin, Groovy, Clojure), C#(or VB.NET), Ruby, Perl, Python, and JavaScript (or TypeScript) (see Figure 1.3). It is API-level interop. It supports either in-process interop or TCP-based communication between runtimes running in different processes.

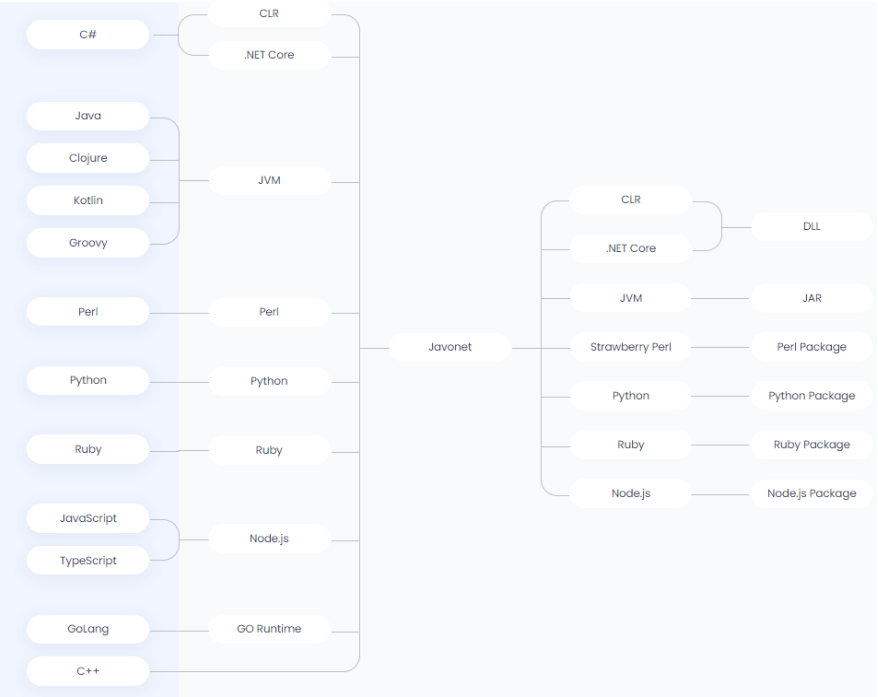

Figure 1.3: Javonet: interop between cartesian product of languages [46]

Javonet does not statically generate proxies, and it does not use dynamic objects either (see Section 1.3.3). Due to that, its API is rather cumbersome as it does not attempt to emulate working with Java objects as if they were C# objects (see Code Snippet 1.24 [46]).

R11 Solution should generate proxies .NET that emulate Java classes so that user experience is seamless, working with Java classes as if they were implemented in C#.

Code Snippet 1.24: Usage of Javonet API

```
1 Javonet.Activate("your-license-key");
2 // in memory can be replaced by Tcp to get Tcp based communication
3 var calledRuntime = Javonet.InMemory().Jvm();
4 string libraryPath = resourcesDirectory + "/TestClass.jar";
```

```
5 string className = "TestClass";
6 calledRuntime.LoadLibrary(libraryPath);
7 var calledRuntimeType = calledRuntime.GetType(className).Execute();
8 var response =
     calledRuntimeType.InvokeStaticMethod("multiplyByTwo",
     25).Execute();
9 var result = (int)response.GetValue();
```

1.3.6 Other open source solutions

In addition to the interop tools mentioned earlier there also exists a number of other open source solutions which are currently unmaintained and were left in various stages of production readiness.

J4net
J4Net [47] is API level JNI-based interop tool hosting both virtual machines in the same process. It uses reflection-based proxies on the .NET side and a similar mechanism to enable callbacks (JNI `RegisterNatives` call and Java proxies generated by external tool [48]) as Xamarin Android Java.Interop (Section 1.3.1).

It is Windows only [47] and no longer maintained [49].

JNetCall
Same as J4net, JNetCall [50] is a level JNI-based interop tool using reflection. It should be multiplatform [50], and although it is no longer maintained, it was abandoned much more recently than J4Net [51].

It is, however, completely undocumented and it seems to be created mainly for a personal needs of the author rather then as a tool intended to be used by other developers.

1.3.7 Summary

This section explored several existing implementations of interoperability between .NET and Java. Based on analyzed implementations, approaches to interoperability can be categorized by several aspects:

- **CIL level** interop (like IKVM referred to in Section 1.3.2) or **API level** interop as any other mentioned implementation

API level approaches can be further categorized:

- as **in-process** interop hosting both JVM and CRL in the same process (e.g., Xamarin interop in Section 1.3.1) or as **inter-process** interop (e.g., JnBridge referred to in Section 1.3.5).
- by their approach to **proxies**. This section showed solutions that:
  - generated statically by the external tool (e.g., JnBridge referred to in Section 1.3.5)

- used reflection-based or `dynamic` proxies (e.g. JCOBridge described in Section 1.3.3)
- did not use proxies at all (e.g., Javonet mentioned in Section 1.3.5)

According to requirement **R7** this thesis will implement **API level** interop. As in-process interop generally tends to have lower overhead and general protocols for communication between processes already exist, the thesis will implement **in-process** interop. As intuitive and easy-to-use API (**R5**, **R9**) and static type safety is required (**R8**, **R10**) **proxies will be generated statically**. Incremental source generators added to .NET in .NET 6 could be an ideal means of compile-time proxy generation.

As this section demonstrated, a certain number of API-level solutions for .NET and Java interoperability already exist. Many of them are, however, commercial and closed-source (Sections 1.3.5 and 1.3.3) and the ones that are open source are either specific to certain domain (as Xamarin Android Java.Interop (Section 1.3.1)) or long abandoned and using out-dated technologies or extremely hard to use due to lack of documentation (Section 1.3.6). It seems, therefore, that there is a need for a modern, well-documented, and maintained interop solution. What is more, none of the examined solutions utilizes Incremental source generators to generate proxies.

1.4 Goals of the thesis

After the potential use cases for the C# - Java interoperability tool have been explored and existing implementations of such or similar tools have been analyzed, the goals of this thesis can be formulated.

The objective of the thesis is to design and implement a .NET library for interoperability with Java. The library should enable users to use APIs implemented in Java from C# code bases. The resulting library should **meet requirements** **R1** – **R11** defined in this chapter.

The implemented solution should be properly **tested on a set of selected supported platforms** (support for multiple platforms is required by requirement **R6**). It should also be **tested using** a real **existing Java library**. Apache PDFBox [5] library mentioned in Section 1.1.2 seems suitable for this purpose.

2. Implementation analysis

This chapter will describe various decisions to be made while designing the interoperability tool being implemented in this thesis. For each decision, chapter will ponder alternative solutions, analyze their pros and cons and will defend the selected approach.

2.1 Differences between Java and C#

This thesis aims at implementing an interoperability tool between C# and Java programming languages. Though the languages share many similarities, there are also concepts in which they differ. We have to identify such differences that are relevant to this thesis and determine how to address them. That will be the topic of this section.

2.1.1 Primitive and value types

.NET (and therefore C#) has so-called **common type system** (CTS) [52] that unifies all types (including built-in numeric types such as `int` or `double`) so that they share the same topmost base class (the root of inheritance hierarchy) – `System.Object`. The structure of CTS is captured by Figure 2.1. Due to the common base class, value types in C# can be implicitly boxed when needed and used consistently with reference types.

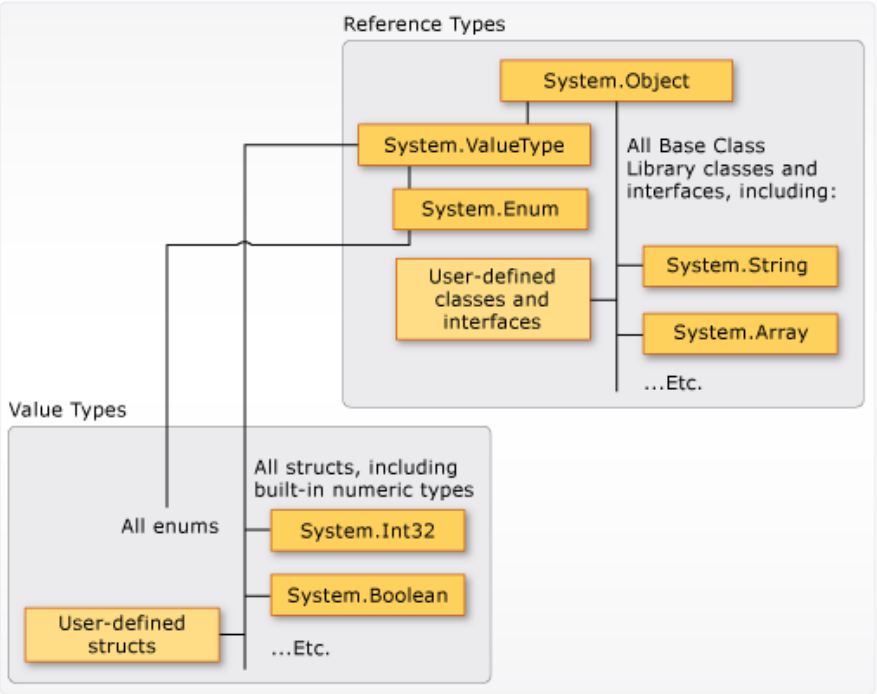

Figure 2.1: Common type system in .NET

Java, however, does not implement this type unification [53]. That has two

main consequences[1]:

1. Primitive types (such as `int` or `double`) cannot be used where reference type is expected – e.g. they cannot be passed to methods that expect `java.lang.Object`, and due to generic runtime erasure (see Section 2.1.2), they cannot be used as generic type parameters (e.g., stored in generic collections). Instead, wrapper classes (`java.lang.Integer`, `java.lang.Double`) must be used.

2. Java does not allow for user-defined value types (equivalent of C# `struct`).

In the context of our interoperability tool, this leads to the following outcomes:

O1 Built-in numeric types and their wrapper types will have to be handled separately. Wrapper types do not, in principle, differ from any other Java classes and can be, therefore, handled as such. Built-in numeric types (also called primitive types) will be analyzed in the following paragraph.

O2 Struct types should not be allowed in signatures of C# methods that emulated Java methods (from now on referred to as **proxy methods**).

Primitive types

Java defines 8 primitive types: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char` [55]. Each of them corresponds to an existing C# primitive type with a similar name – e.g., Java `int` can be represented by C# `int` (`System.Int32`) [56]. These pairs of types share the same memory representation; we can, therefore, pass values between them directly.

The only exception is `byte` type. Unlike .NET, Java does not support unsigned integer types (C# types such as `uint` and `ulong` do not have Java equivalent). Java `byte` is therefore signed while C# `System.Byte` is unsigned. We have to map Java `byte` to C# signed byte: `System.Sbyte`.

O3 Java primitive types can be represented by corresponding C# primitive types (Java `byte` must be represented by C# `sbyte`).

O4 C# unsigned integers should not be allowed in proxy method signatures.

Other common value types

C# also defines other commonly used value types, e.g. (primitive) `decimal` type [57], `System.DateTime` struct [58], `System.Guid` struct [59]. Java offers some alternatives to these types; however, as the only value type that exists in Java are 8 primitive types listed in the previous paragraph, Java equivalents of `decimal`, `DateTime` and `Guid` are reference types. They could be handled similarly to wrapper types (such as `java.lang.Integer`), but that is out of the scope of this thesis.

O5 Other C# built-in value types (such as `decimal`, `DateTime` and `Guid`) won't be allowed in proxy method signatures.

[1]Project Valhalla [54] that has been in progress for several years now aims at mitigating both of these issues.

2.1.2 Generics

Java generics significantly differ from .NET ones. Generics were added to Java in Java 5 [60]. To keep backwards compatibility with the previous Java versions, Java generics only exist at compile time – they are not represented in Java bytecode. That distinguishes them from .NET generics that are represented in CIL code and get instantiated during JIT compilation.

When Java code containing generics gets compiled to bytecode, **erasure of generic types** takes place. Generic type parameters are replaced by the most general type applicable – the type parameter upper bound (if such is imposed using `extends` keyword) [61] or `java.lang.Object` if the type parameter is not bounded. Due to that [62]:

- It is not possible to instantiate generic types with primitive types, because primitive types do not inherit `java.lang.Object`.
- It is not allowed to have static fields of type parameter types.

O6 Generics in Java are in principle different from .NET generics. Support for Java generics is out of scope of this thesis.

2.1.3 Enums

Another concept that significantly differs between C# and Java is enums. C# enums are value types with the precisely defined set of values [63]. In Java, enums are reference types with a precisely defined set of instances (see an example of a Java enum in Code Snippet 2.1). Each instance represents one enumeration constant and is accessible as an (equivalent of `static final`) field of enum type [64] (see `SUCCESS` and `ERROR` fields in the example in Code Snippet 2.1). Unlike in C#, enums in Java can define fields (for instance, `message` and `code` field in Code Snippet 2.1) and methods (`getCode` method in Code Snippet 2.1). Java enums are, therefore, more similar to C# classes than to C# enums.

Code Snippet 2.1: Example of Java enum

```
1  enum Status {
2      SUCCESS("Operation successful", 200),
3      ERROR("Operation failed", 500);
4
5      private String message;
6      private int code;
7      // constructor
8      Status(String message, int code) {
9          this.message = message;
10         this.code = code;
11     }
12     // Getter method
13     public int getCode() {
14         return code;
15     }
16 }
```

Given these differences, it is clear that we cannot represent Java enums by C# enums. It would be surely possible to design a C# proxy class that would emulate Java enum. It is, however, out of the scope of this thesis.

O7 Support for Java enums is out of the scope of this thesis. C# enums won't be allowed in proxy method signatures.

2.1.4 Multi-dimensional arrays

C# supports both **multi-dimensional arrays** (continuous block of memory allocated via one allocation, all "inner arrays" must have the same length) and **arrays of arrays** (separate allocation for each inner array, occupied memory doesn't have to be a continuous block, inner arrays may have different lengths) [65]. Code Snippet 2.2 demonstrates the difference between these two concepts using the example of allocation of a two-dimensional array and an array containing two arrays.

Code Snippet 2.2: C#: multidimensional array vs. array of arrays

```
1 int[,] twoDimensionalArray = new int[3, 4];
2
3 int[][] arrayOfArrays = new int[2][];
4 arrayOfArrays[0] = new int[4];
5 arrayOfArrays[1] = new int[3];
```

Java only supports arrays of arrays. It does not support multi-dimensional arrays [66].

O8 Our solution should support arrays of arrays. Multi-dimensional arrays should not be allowed in proxy method signatures.

2.1.5 Properties

Java does not support properties. By convention, fields are made accessible via **getter** and **setter** methods. These methods usually follow the given naming convention: `get` or `set` prefix, respectively, followed by the field name with the first character capitalized [67]. Apart from that, they do not differ in any way from any other Java methods. We will be, therefore, able to invoke them as such.

Section 2.7.2 will examine this topic in more detail.

2.1.6 Interfaces

There are not many differences between C# and Java interfaces in the latest versions of the languages. For several language versions, Java interfaces supported features that weren't supported by C# interfaces: **default**[68], **static** [68] and **private** [69] **interface methods**.

C# added these features in C# 8 [70] [71]. The concept of default and private methods does not differ much between languages, as one of the motivations for adding them to C# was to allow for easier interoperability with Java [70]. For static interface methods, the situation is different as static interface methods

in Java require default implementation in the interface, whereas C# allows for **static abstract** interface methods.

O9 We can emulate Java interfaces by C# interfaces as the two concepts are fairly similar. Support for advanced features of Java interfaces, such as static and default methods, is out of the scope of this thesis. Section 2.8.2 will discuss the support for interfaces in more detail.

2.1.7 Nested classes and static classes

Both languages support nested classes [72, 73], but the semantics of nested classes differ among languages.

In C#, the nesting of classes only influences visibility. There is no implicit connection between instances of outer and nested classes.

Java distinguishes two categories of nested classes: **static nested classes**, which have the same semantics as nested classes in C#, and non-static nested classes, also called **inner classes**. Inner class instances hold an implicit reference to an instance of the outer class. An inner class instance cannot be created without an outer class instance. This is demonstrated in Code Snippet 2.3. On line 8, notice that `Inner` class can access instance field `outerField` of `Outer` class. On line 16, notice that an instance of `Outer` class is required to create an instance of `Inner` class.

Code Snippet 2.3: Java: inner class

```
1  public class Outer {
2      public int outerField = 10;
3
4      public class Inner {
5          public int innerField = 20;
6
7          public void displayFields() {
8              System.out.println("Outer field value: " + outerField
                   + " inner field value " + innerField);
9          }
10     }
11 }
12
13 public class Main {
14     public static void main(String[] args) {
15         Outer outer = new Outer();
16         Outer.Inner inner = outer.new Inner();
17         inner.displayFields();
18         // Outer field value: 10 inner field value 20
19     }
20 }
```

O10 Java non-static nested classes differ from C# nested classes, but C# proxy class could be designed to compensate for this difference. Java static nested

classes are fairly similar to C# nested classes. The only problem to be solved to support them is the name resolution. Both categories of nested classes are out of scope for now.

Static classes
Unlike C#, Java does not allow for non-nested classes to be marked as static [74]. However, as it is going to be discussed in Sections 2.3.4 and 2.4, interoperating with Java classes as if they were static (only accessing their static members) can be easier (as it is not required to manipulate object instances).

O11 Even though Java does not allow for general classes to be static, we will allow users to choose to emulate Java class by C# static class if they only intend to access static members of underlying Java class.

2.1.8 Naming conventions

There are slight differences in naming conventions between Java [75] and C# [76]:

- **methods** – Java uses *camelCase* while C# uses *PascalCase*,
- **interfaces** – Java does not prefix interface names by capital I, while C# does
- **constants** – Java uses *SCREAMING_SNAKE_CASE* while C# uses *PascalCase*
- **packages/ namespaces** – Java uses lower case for package names while C# uses *PascalCase*

O12 Our solution should compensate for differences in naming conventions, allowing C# code that emulates Java code to respect C# naming conventions.

2.2 Supported environments

According to the requirement **R6**, the implemented solution should function on common Java distributions and their versions. This section will focus on selecting a set of Java distributions and their versions as well as a set of operating systems on which the software implemented in this thesis will be tested (for more information about automated tests, see Section 3.6).

Operating systems
The thesis software will be developed and tested on a laptop computer with **Windows 10 Pro** operating system. Apart from that, we are going to select representatives of Linux-based operating systems on which the thesis software will be tested.

We chose **Ubuntu** as the most widely used desktop distribution. Further, we opted for **CentOS Stream** as a free-to-use variant of the Red Hat Enterprise Linux, which is popular in server environments. Lastly, we opted for **Apline** Linux as it is a light-weight distribution commonly used in containers and as it uses a different C-language standard library than the previous two distributions (MUSL instead of glibc) [77].

Java versions
According to the JetBrains Developer Ecosystem survey from 2023 [78], 50 % of respondents that use Java as one of their three primary programming languages are still regularly using Java 8. That makes Java 8 the most commonly used version of Java, according to this survey. For this reason, we should test the solution on **Java 8**. Apart from that, we will run tests on all LTS versions released since Java 8: **Java 11, Java 17, and Java 21** [79]. As the newest version of Java supporting Project Panama (see Sections 2.3.4 and 2.4) that was out when Project Panama was analyzed for the purposes of this thesis was Java 20, we will also test on **Java 20**.

JDK distributions
The thesis will be tested on **Oracle OpenJDK** [80] and **Amazon Corretto OpenJDK** [81] distributions as these are among the most popular JDK distributions [82]. It will also be tested on **GrallVM** [83] JDK distribution as it is an interesting high-performance distribution whose behavior may differ in some aspects from the previous two.

2.3 Java–native interoperability tools

To enable interoperability between C# and Java, this thesis will rely on a combination of some existing tools for interoperability between Java and a native code and P/Invoke (see Section 1.2.2) that enables to invoke native code from C#. This section will focus on selecting Java–native interoperability tool to be used in this thesis.

2.3.1 Java Native Interface

Section 1.2.1 already introduced the Java Native Interface (JNI) in a sufficient amount of detail. This section will point out the advantages and disadvantages that JNI has for the purposes of this thesis.

The most commonly mentioned disadvantage of JNI is that it is highly user-unfriendly for Java developers. JNI requires users to manually write boiler-plate code for native method bindings, furthermore in C programming language. The performance and memory safety of JNI depend on the user's ability to implement this code correctly [84]. That, however, does not seem problematic for our purposes as we can abstract the majority of JNI details from a user via the C# interoperability library we will implement.

The advantages of JNI in our context are that:

- JNI itself is the native library. We can easily use JNI functionality from C# via P/Invoke.
- JNI allows for a low-level manipulation with resources such as memory and references to Java objects. That offers a certain flexibility and level of control that can be useful when implementing an interoperability library.
- JNI is an inherent part of Java and has been available since JDK version 1.1 [6]. This is an advantage as according to the requirement **R6**, our solution

should function on common Java distributions and versions, and as Section 2.2 has shown, older versions of Java (such as Java 8) are still being widely used.

- JNI allows us to spawn a JVM instance in the same process in which the C# application is running. That is really useful as it allows us to initiate interoperability interaction from the C# side code.

2.3.2 Java Native Access

Java Native Access (JNA) [85] is a community-driven library based on JNI. It removes the necessity of writing C-language binding code manually by extensive use of reflection. It is, therefore, more user-friendly and preferred among Java programmers over direct usage of JNI. However, usage of reflection imposes a performance overhead over direct usage of JNI [86].

For our purposes JNA does not present any advantage over JNI, as the fact that JNI is C-language API provides an advantage not a disadvantage in our context. JNI is therefore preferable choice for the purposes of this thesis.

2.3.3 Java Native Runtime

Java Native Runtime (JNR) [87] is a community-based library similar to JNA. It is also based on JNI and uses dynamic code generation to avoid the necessity of handwritten binding. JNR is said to be a less mature library than JNA [88]. Though it has better performance than JNA, its overhead is still worse than direct JNI usage [86]. For our purposes, JNR doesn't seem to have any advantage over JNI.

2.3.4 Project Panama

Project Panama [89] is the current openJDK project. It aims at eventually replacing JNI as a tool for Java-native interoperability that is built-in to Java programming language.

Project Panama consists of 3 parts:

- Foreign Memory API [90],
- Foreign Function API [90]
- and Vector API [91].

Vector API is not relevant to this thesis as it is not directly related to interoperability. It should enable Java programmers to explicitly specify that their code should be compiled into vector (SIMD) instructions if the underlying platform provides such [92].

The other two APIs are more relevant. **Foreign Function API** enables interoperability invocations between Java and native code. **Foreign Memory API** allows Java programmers to access native memory (in this context often called **off-heap** memory – meaning off Java managed heap) and, as such, is used while working with Foreign Function API.

Foreign Memory API

The core class of Foreign Memory API is `MemorySegment` [93]. It allows access to a continuous block of memory either on a Java managed heap or off-heap. It is designed to be safe. Each `MemorySegment` instance has its **spacial bounds** and **temporal bounds** – accesses to `MemorySegment` are checked at runtime to stop a user from accessing out-of-bounds indices or accessing `MemorySegment` after it has been freed [94].

Programmers have the option to manage the lifetime of `MemorySegment`s explicitly – unlike older `ByteBuffer` API [95], `MemorySegment` lifetime is not necessarily managed by the garbage collector. To avoid burdening programmers with having to manage the lifetime of each `MemorySegment`s separately, Foreign Memory API introduces the concept of `Arena`s. `Arena` API allows to allocate `MemorySegment`s. All `MemorySegment`s allocated via the same `Arena` share the same lifetime. There are four categories of `Arena`s:

- **Confined** – `MemorySegment`s allocated via the `Arena` are freed when the `Arena` is closed. `MemorySegment`s can only be accessed by a thread that created the `Arena`.
- **Shared** – Allocated `MemorySegment`s are freed when `Arena` is closed. `MemorySegment`s can be accessed by any thread.
- **Automatic** – Allocated `MemorySegment`s are managed by the garbage collector. They will be deallocated at some unspecified time after the `Arena` becomes unreachable. `MemorySegment`s can be accessed by any thread.
- **Global** – Allocated `MemorySegment`s are never deallocated and can be accessed by any thread.

Code Snippet 2.4 demonstrates the allocation of an off-heap memory via `Arena` API. As the used `Arena` is of **Confined** category (see line 1), allocated memory gets freed when the `Arena` is closed on line 3.

Code Snippet 2.4: Java: Foreign memory API – allocate off-heap memory via `Arena`

```
1 try(Arena offHeap = Arena.ofConfined()){
2     MemorySegment point = offHeap.allocate(POINT);
3 } // memory gets freed
```

On line 2 in Code Snippet 2.4, notice the parameter of `allocate` method: `POINT`. `POINT` is an example of `MemoryLayout` [96] – a mean Foreign Memory API provides to specify a layout of allocated `MemorySegment`s. `MemoryLayout`s allow programmers to avoid computing offsets manually each time they access a `MemorySegment`. Foreign Memory API contains `ValueLayouts` [97] of primitive types and allows to compose them into more complex `MemoryLayout`s of C-like structs and arrays. A possible definition of `POINT MemoryLayout` is shown in Code Snippet 2.5 [94].

Code Snippet 2.5: Java: Foreign memory API – MemoryLayout definition

```
1 MemoryLayout POINT = MemoryLayout.structLayout(
2     ValueLayout.JAVA_DOUBLE.withName("x"),
3     ValueLayout.JAVA_DOUBLE.withName("y"),
4 );
```

Foreign Function API

Foreign Function API builds upon Foreign Memory API. Particularly it uses:

- `MemorySegment`s to pass data between Java and the native code,
- `MemoryLayout`s to describe signatures of native functions to be invoked from Java,
- `Arena`s to manage lifetimes.

Foreign function API [90] allows both for invocations of native functions from Java (in Project Panama context, usually called **downcalls**) and for callbacks from native code back to Java (**upcalls**).

Code Snippet 2.6 [94] demonstrates **downcall** invocation of native function `extern Point* makePoint(double x, double y)` from Java via Foreign Function API. Line 1 looks up the address of `makePoint` native function. Line 3 uses `MemoryLayout` to describe the native function signature. Line 5 combines the address and the signature into a `MethodHandle`, that can be then used to invoke `makePoint` native function for Java.

Code Snippet 2.6: Java: Foreign function API – downcall

```
1 MemorySegment makePointAddress =
2     SymbolLookup().lookup("makePoint").get();
3 FunctionDesctiptor signature = FunctionDescriptor
4     .of(ADDRESS, JAVA_DOUBLE, JAVA_DOUBLE);
5 MethodHandle makePointHandle = Linker
6     .nativeLinker()
7     .downcallHandle(makePointAddress, signature);
8 MemorySegment point = makePointHandle.invokeExact(42.73, 42.73);
```

Suppose that `makePoint` native function allocates a new instance of `Point` struct (that can be described by `POINT MemoryLayout` showed in Code Snippet 2.5) and returns a pointer to it. Notice that native function invocation on line 8 in Code Snippet 2.6 returns `Point` pointer represented by a `MemorySegment`. As it was already explained, for safety reasons, each `MemorySegment` instance has spacial and temporal bounds. However, Java knows nothing about the spatial and temporal bounds of memory returned from the native code. Therefore `MemorySegment` returned from native function invocation is so called **zero-length memory segment**. It represents the correct memory address but its length is set to zero. Underlying memory, therefore, cannot be accessed from Java as that would violate the spatial bounds of the `MemorySegment`. To access the memory from Java, it is necessary to `reinterpret` the `MemorySegment` – to explicitly set its expected spacial and temporal bounds – as shown in Code Snippet 2.7.

Code Snippet 2.7: Java: Foreign function API – reinterpret zero-lenght memory segment

```
1 try(Arena offHeap = Arena.ofConfined()){
2     point = point.reinterpret(
3         POINT, // memory layout (spacial bound)
4         offHeap, // lifetime (temporal bound)
5         point -> cleanupHandle.invokeExact(point)); // cleanup
              callback
6     // access point
7 }
```

Reinterpret is a potentially dangerous operation – it allows a Java code to access a memory that potentially lays outside of actual spacial bounds of the memory region returned from the native code. Therefore, `reinterpret` method is one of Javas **restricted methods** [98] and to use it is necessary to opt-in via `--enable-native-access` command line option.

Foreign Function API – upcalls

Foreign Function API enables **upcalls** (invocations of Java methods from the native code) only as **callbacks**. Foreign function API can be used to build an **upcall stub** of the Java method. This stub can then be passed to the invocation of a native function that expects a function pointer. The native function can then call this function pointer effectively calling back to Java [99].

Code Snippet 2.8 shows how to build an **upcall stub** of Java method `javaCallback` shown in Code Snippet 2.9. Line 1 uses `MemoryLayout` to describe the method signature. Line 3 looks up the method using `MethodHandle` API [100]. Line 6 builds an upcall stub from the `MethodHandle` using `Arena` with the global scope.

Code Snippet 2.8: Java: Foreign function API – upcall stub

```
1 FunctionDescriptor signature = FunctionDescriptor
2     .of(JAVA_INT, JAVA_DOUBLE);
3 MethodHandle upcallHandle = MethodHandles
4    .lookup()
5    .findStatic(MyJavaClass.class, "javaCallback",
        signature.toMethodType());
6 MemorySegmemnt upcallStub = Linker
7     .nativeLinker()
8     .upcallStub(upcallHandle, signature, Arena.ofGlobal());
9 // pass upcallStub to native invocation
```

Code Snippet 2.9: Java: callback to be invoked from native code

```
1 public class MyJavaClass {
2     public static int javaCallback(double) { ... }
3 }
```

Summary
As the previous examples have shown, Panama Foreign Function API is highly Java-centered. Unlike JNI, it allows for interoperability invocations without the necessity to write any C-code bindings. That is a big selling point for Java programmers; however, it imposes an issue for the purposes of this thesis. As Panama API is Java API, we cannot easily access it from C# (if we could, the point of this thesis would be moot). Moreover, to enable C# to Java invocations, we are interested in upcalls that Panama only enables as callbacks. That could make it more complicated to use Panama for our purposes (Section 2.4.1 will analyze it).

Moreover, unlike JNI, the current version of Project Panama does not allow native code to access Java object [101].

On the other hand, Project Panama can also have some advantages over JNI in our context:

- Its performance is supposed to be better than the performance of JNI, especially for upcalls [101] (Section 2.4.3 will look into that).
- It is a brand new feature built-in Java. When Project Panama was analyzed for the purposes of this thesis, it was still a preview feature (in Java 20). Foreign Function and Foreign Memory API only moved from the preview in Java 22 (19th March 2024)[2] [102]. Project Panama may be the future of interoperability between Java and native code, and our solution should be prepared for that.

2.3.5 Summary

Previous sections analyzed several tools for interoperability between Java and the native code and identified their advantages and disadvantages. This section will choose tools that are to be used in this thesis.

JNI:

- is easy to use from C# via P/Invoke,
- enables us to spawn JVM in the process of a .NET application. Therefore, it enables us to bootstrap interoperability interaction from C# code,
- and is available in all versions of Java since Java 1.1.

JNA (Section 2.3.2) and JNR (Section 2.3.3) do not present any advantage in our context over a direct usage of JNI.

Project Panama:

- is not straight-forward to use from C#,
- does not enable us to spawn JVM in the process of .NET application,
- is only available as preview API since Java 19 [90] and as non-preview API since Java 22 [102],

[2]Vector API still remains a preview feature in Java 22.

- is a modern feature of Java and may be a direction that Java will take in the future,
- and can be more efficient than JNI for upcalls.

We will, therefore, base our interoperability tool on JNI. Section 2.4 will then explore how Project Panama can be used in this thesis and will analyze if the resulting solution outperforms the JNI-based solution.

2.4 Project Panama for C#–Java interoperability

Project Panama is a promising modern alternative to JNI. As the previous section discussed, it currently does not seem possible to base our interoperability solution solely on Panama as Panama is only supported on the newest versions of Java (violating requirement **R6**), as it doesn't allow us to spawn JVM in our process and as it does not provide support for object yet (violating requirement **R2**). We can, however, use JNI to spawn JVM instances and to handle manipulation with objects and use Panama to **optimize** method invocations that do not access object instances: **invocations of static methods with primitive parameter and return types**.

2.4.1 Static methods with primitive parameter and return types

As Section 2.3.4 explained, Panama enables the invocation of Java methods from a native code by building un **upcall stub** and passing it to a native function that expects a function pointer. We will extend this approach to leverage invocations of Java methods from C#.

When a native function receives an upcall stub pointer, it can invoke it as a normal function pointer. If we managed to obtain an upcall stub pointer from our C# interoperability tool, we could also treat it as an ordinary native function pointer – cast it to `unmanaged delegate` and invoke it via P/Invoke (see Section 1.2.2). The problem is, however, that Panama Foreign Function API is only available as Java API – and we would like to use it from our C# interoperability tool to obtain an upcall stub pointer.

We can bootstrap Panama invocation via JNI. We can implement Java method `buildPanamaStub` that given a name of a Java method to be invoked from C#, a name of a type that defines the method and the method signature, returns the upcall stub pointer. We can use JNI to invoke `buildPanamaStub` method the first time a given Java method is to be invoked from C# to obtain the upcall stub pointer. Then, each consecutive time the Java method is invoked, we can invoke the upcall stub pointer via P/Invoke (without any additional usage of JNI).

This approach only has advantages over a purely JNI-based approach if the invocation of the upcall stub via P/Invoke is significantly more efficient than the direct invocation of the Java method via JNI. Section 2.4.3 will analyze performance implications.

2.4.2 Strings

Project Panama does not currently provide dedicated support for accessing Java string instances from a native code. Due to the possibility of heap compaction, we cannot simply pass Java string reference to C# and vice versa. We can, however, pass strings by copying them to unmanaged memory and passing a pointer to the unmanaged copy.

The problem is that with this approach, arguments and return values passed across the language border do not match the signature of actual Java method being invoked – method expects string, but C# would like to pass a pointer to an unmanaged string copy represented by `long` type[3].

Before an actual user-defined Java method is invoked, preprocessing of string arguments is required, reading string content from an unmanaged memory and creating `java.lang.String` instance. Symmetrically, when the Java method returns a string, return value postprocessing must be carried out, allocating an unmanaged string copy and returning a pointer to it to C#. According to requirement **R3**, our solution should not require users to modify Java-side code in order to make it usable from C#; we, therefore, cannot rely on users to carry out these transformations.

Luckily, Java `MethodHandles` API [100] provides the solution. Code Snippet 2.8 has shown that to build an upcall stub of the Java method, one must first obtain a `MethodHandle` of the Java method. `MethodHandles` API allows to combine `MethodHandle`s into more complex invocable structures. Among others, it offers `collectArguments` method that allows to set another `MethodHandle` (`filter`) that will be used to preprocess argument(s) of original `MethodHandle` before the original method is invoked. `collectArguments` method produces a new `MethodHandle` with signature of the original `MethodHandle`, but with specifies parameter types replaced by parameter types of `filter` `MethodHandle`.

As a `filter`, we can apply a method that takes an address of a string copy in an unmanaged memory (`long`) and returns a managed Java string read from that address. Resulting `MethodHandle` will therefore have all occurrences of string type among its parameters replaced by `long` type. We can then build an upcall stub from this modified `MethodHandle` – resulting signature will directly match what C#-side caller expects, and string arguments preprocessing will automatically take place each time the upcall stub is invoked from C#.

Symmetrically, we can use `MethodHandles.filterReturnValue` method [100] with a `filter` allocating a copy of `java.langString` in unamanaged memory to ensure return value postprocessing.

Take away

Though the described approach allows us to pass string values between Java and C#, it does not allow us to treat Java strings as objects on the C# side – e.g., invoke instance methods on string instances. It may be possible to extend the approach further to be feasible in a broader range of scenarios, but for the purposes of this thesis, we will abandon these attempts. We will handle strings via JNI, as the approach seems less fragile and bug-prone than the manual copying of strings between managed and unmanaged memory. We will keep in mind that

[3] Java `MemorySegment` API represents addresses as `long`s [93]

Panama could potentially be used to work with strings in future versions of the project.

2.4.3 Benchmark: JNI vs Project Panama

This section compares the performance of the interoperability approach based on Project Panama described in Section 2.4.1 with a purely JNI-based approach. Based on the results, it will decide whether Project Panama is worth using in this thesis.

The author did a more detailed analysis of the problem as the final project for Performance Evaluation of Computer Systems class [103]; this section contains the summary of the obtained results. All the benchmarks were executed in August 2023 on the following platform:

- **CPU**: 12th Gen Intel(R) Core(TM) i7-12700H, 2300 Mhz, 14 Core(s), 20 Logical Processor(s)
- **RAM**: 32 GB
- **OS**: Microsoft Windows 10 Pro, Version 10.0.19045 Build 19045
- **Java version**: openjdk 20.0.1 2023-04-18
- **.NET version**: .NET SDK 7.0.203 (.NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2)
- **C++ compiler**: cl C/C++ Optimizing Compiler Version 19.35.32217.1 for x64
- **Power management**: all experiments were run on plug-in laptop with high performance mode enabled

Java – C++ interoperability: downcalls vs upcalls

The first set of benchmarks aims at verifying the claim [101] that Project Panama should outperform JNI on **upcalls from a native code to Java**.

Figure 2.2 captures invocation times of 1000 invocations of static method taking ten int parameters via JNI (blue and green) and via Project Panama (red and yellow). Both upcall[4] and downcall[5] directions of invocations are captured. To disregard the influence of a warm up (e.g. caused by JIT-ting of Java bytecode) on the the measurement, displayed samples are taken from the middle of the longer run. Iterations 5000 to 6000 are displayed.

Figure 2.2 shows that Project Panama is significantly faster for upcall invocations than JNI. The mean upcall invocation time via JNI is 75.96 ns, whereas, via Project Panama, it is 25.07 ns. That confirms the assumption that Project Panama is more efficient on upcalls than JNI, and it makes Project Panama a promising tool for this thesis, as the direction of invocation we are interested in is upcalls (requirement **R1**).

[4] native to Java
[5] Java to native

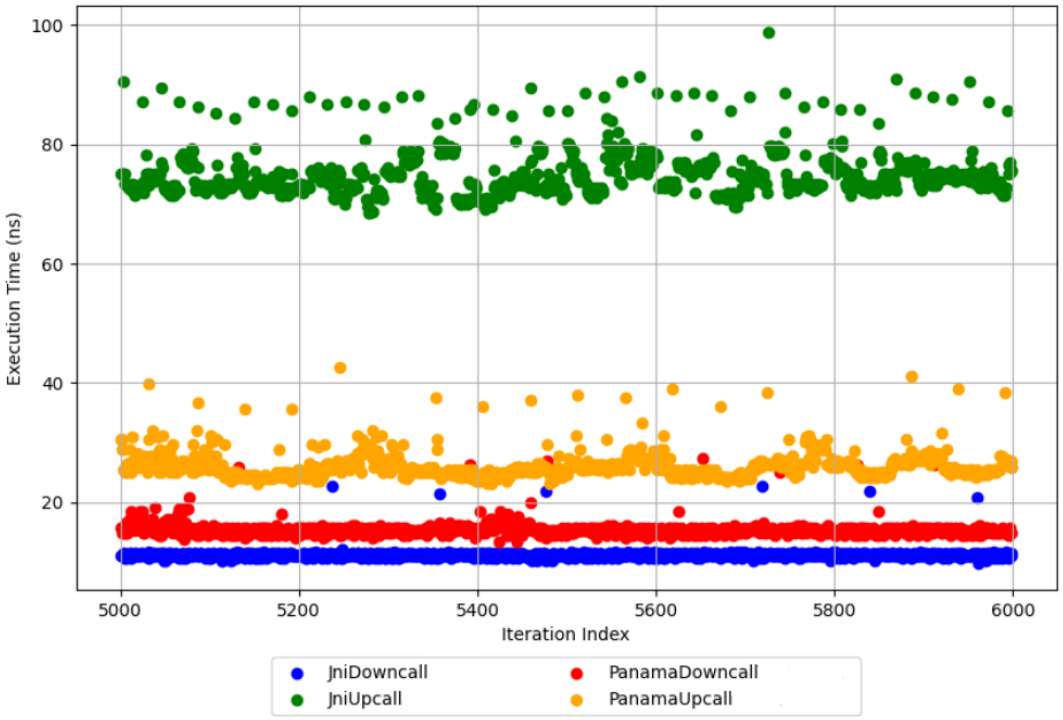

Figure 2.2: Upcall vs downcall invocation times, Java – C++ interop, method taking 10 int parameters

On downcall both Project Panama and JNI perform similarly. The mean downcall invocation time via JNI is 10.07 ns and via Project Panama is is 12.24 ns. No matter the technology used, downcalls are more efficient than upcalls.

C# – Java interoperability: static methods with primitive parameter types

The second set of benchmarks compares the performance of JNI and Panama on invocation of **static Java methods with primitive parameter and return types from C# using the interoperability tool being implemented in this thesis**. Benchmark explores the influence of a number of method parameters (methods with **0, 1, 5** and **10 parameters** were measured) and of parameter types (methods taking `int`, `long`, `float`, `double` or `bool` parameters were measured) on the invocation time.

For all explored combinations of parameter types and counts, Project Panama performed better than the JNI-based solution. Figure 2.3 demonstrates it by comparing mean invocation times of Java methods taking 0, 1, 5, and 10 double parameters via JNI and via Project Panama. Attachment A.2 then contains a table of mean invocation times of all observed methods, collected using Benchmark .NET [104].

As Figure 2.3 shows, invocation times using JNI depend on the number of the method parameters. The more parameters the method being invoked is accepting the longer the JNI mean invocation time (the difference in mean invocation time between a method taking zero parameters and a method taking ten `double` parameters is approximately 100 ns). Panama invocation times do not seem to be affected in this manner.

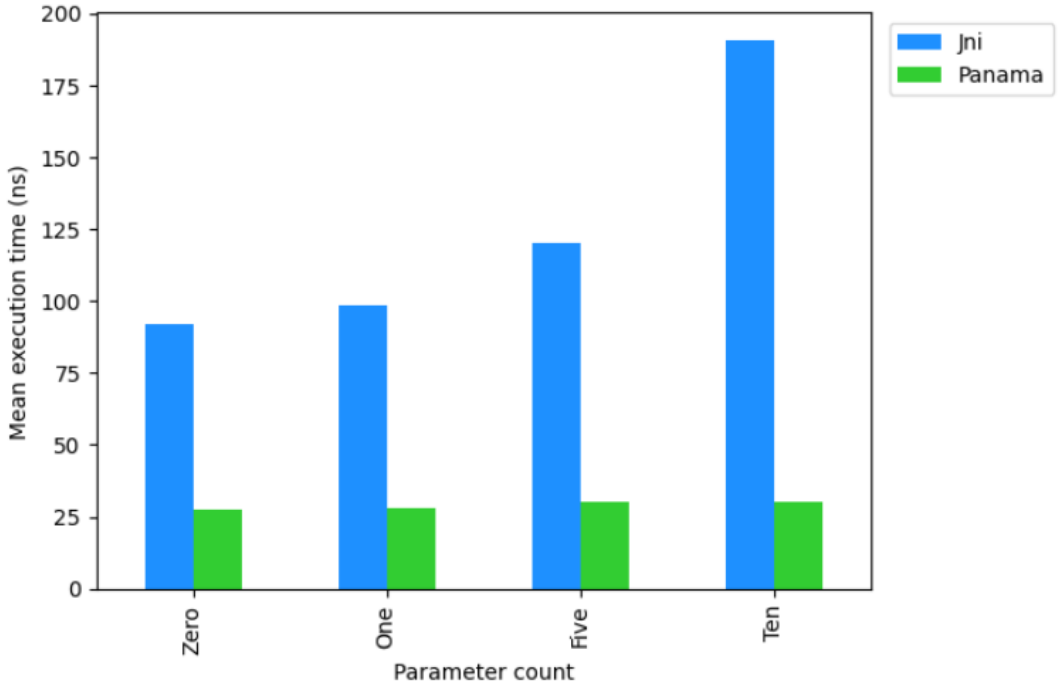

Figure 2.3: Influence of number of parameters on mean invocation time – double parameters

Table 2.1 shows statics considering **invocations of methods taking int parameters** collected via Benchmark .NET. It demonstrates that, unlike the Project Panama-based approach, **JNI carries out heap allocations to pass primitive type arguments from C# to Java**.

Method	Mean	StdDev	Allocated
IntParamlessMethodJni	67.90 ns	0.472 ns	-
IntMethodIntParamJni	74.83 ns	0.261 ns	32 B
FiveIntParamsJni	98.19 ns	0.371 ns	64 B
TenIntParamsJni	135.71 ns	0.782 ns	104 B
IntParamlessMethodPanama	20.05 ns	0.129 ns	-
IntMethodIntParamPanama	20.36 ns	0.024 ns	-
FiveIntParamsPanama	19.14 ns	0.102 ns	-
TenIntParamsPanama	22.35 ns	0.092 ns	-

Table 2.1: Performance comparison between JNI and Project Panama on methods taking int parameters

Figure 2.4 explores the influence of parameter types on a mean invocation time. JNI performance seems to be slightly affected by parameter type, while the Project Panama-based approach seems indifferent to types of primitive parameters passed.

To sum it up, the Project Panama-based approach is more efficient than JNI on all observed combinations of parameter types and counts. Unlike JNI, it does not carry out heap allocations to pass arguments, and the invocation time does not

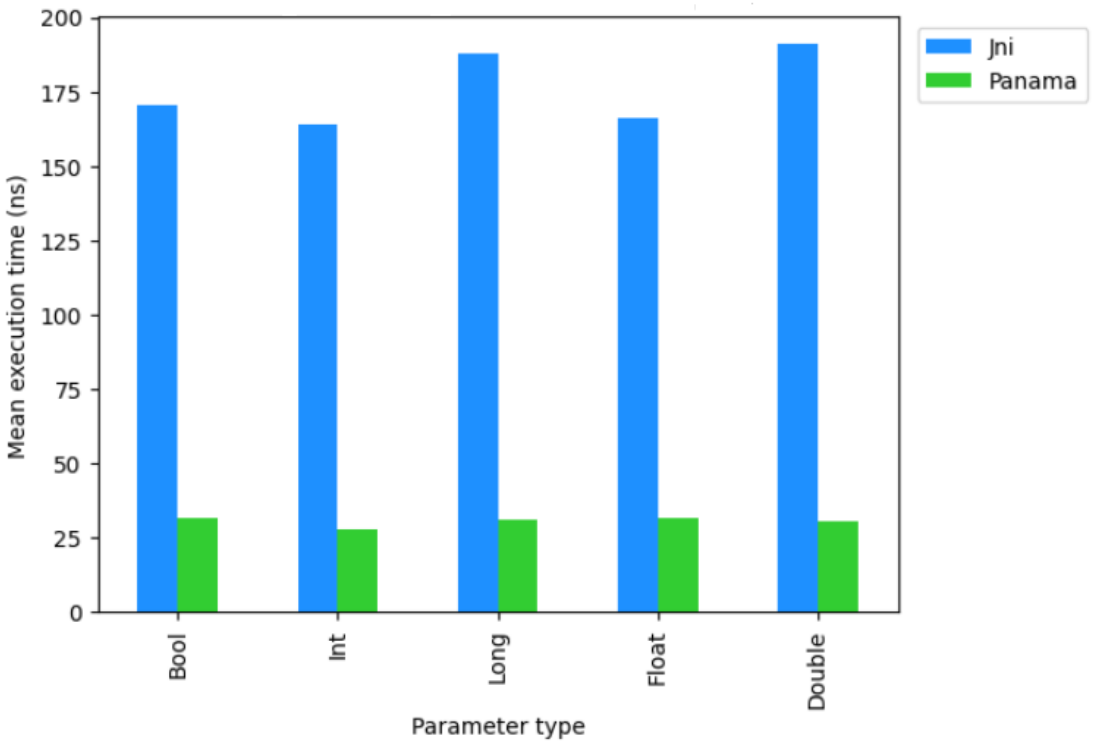

Figure 2.4: Influence of primitive parameter types on mean invocation time – 10 parameters method

depend on the number of arguments passed. **Project Panama seems to have a significant performance advantage over JNI on invocations of static methods with primitive parameter and return types**. We will, therefore **incorporate Project Panama as opt-in performance optimization** for these kinds of invocations. That will also encourage us to design the solution in a modular enough way so that if the Java ecosystem moves away from JNI and towards Project Panama in the future, it will be easier to modify our solution to use Project Panama more broadly.

2.5 JNI layer

This section will analyze a set of decisions that had to be made while designing JNI-based layer of the interoperability tool being implemented.

2.5.1 CreateJVM – locating JVM library

As was already mentioned, JNI can be used to create a JVM instance in the process where the .NET application is running. To make it possible, however, `jvm` native library (`jvm.dll` on Windows, `libjvm.so` on Linux) must be loaded. This library comes with each Java version and distribution present on a hosting machine. We need to let the user specify which Java version and distribution should be used. The process of looking up native libraries to be loaded, however, differs between Windows and Linux platforms (and we would like to provide a solution working on both platforms – requirement **R6**).

On Windows, we can make libraries loadable by setting their location to `PATH` environment variable [105]. `PATH` environment variable can be modified by running a .NET application via `Environment.SetEnvironmentVariable` call [106]. This modification will affect consecutive loads of native libraries carried out by the application.

The situation on Linux is different. To enable for a library to be dynamically loaded, the path to the library must be set in `LD_LIBRARY_PATH` environment variable [107]. This environment variable, however, must be set before the application starts. Later changes of the variable won't have an effect on a running application [108] – `Environment.SetEnvironmentVariable` method, therefore, cannot be used to set `LD_LIBRARY_PATH` for running .NET application.

On Linux, we, therefore, need a user to set `LD_LIBRARY_PATH` to contain a path to `jvm` library to be used before an application using our interoperability tool is launched. On Windows, we can read it either from an in-code variable (e.g., the parameter of `CreateJavaVM` method) or from `JAVA_HOME` environment variable [109] and set it to `PATH` environment variable. Section 4.1 will describe the configuration process in a bit more detail.

2.5.2 JNIEnv pointer and multi threading

As Section 1.2.1 described, the JVM instance can be spawned in the current process via calling `JNI_CreateJavaVM` function. The majority of JNI implementations, however, only allow to spawn one JVM instance per the whole lifetime of the process [8, 110, 111].

The thread that calls `JNI_CreateJavaVM` function obtains `JNIEnv` pointer that allows it to call other JNI functions (see Section 1.2.1). `JNIEnv` pointer is, however, only valid in a thread that obtained it. If other threads need to call JNI functions, they have to attach themselves to the JVM instance via calling `AttachCurrentThread` or `AttachCurrentThreadAsDeamon` JNI functions that return `JNIEnv` valid in the given thread.

Taking this into account, we can keep `JNIEnv` pointer in `ThreadLocal` [112] field. In the main thread, the field can be initialized by `CreateJavaVm` call; in other threads, the first access of the field can initialize it via `AttachCurrentThread` call.

The problem is that non-deamon attached threads will keep JVM from exiting [8] when `DestroyJavaVM` function is called.

A thread can detach itself from the JVM instance by calling `DetachCurrentThread` JNI function. This function, however, must be called from the thread that is to be detached – in our case, that is going to be a user's thread. We, therefore, have to encourage or force the user's thread to detach itself.

Solution: dispose pattern

We could represent the current's thread `JNIEnv` pointer by a `IDisposable` object and implement `Dispose` method that calls `DetachCurrentThread` JNI function. Usage would then look as shown in Code Snippet 2.10.

Code Snippet 2.10: C#: Using Dispose pattern to detach threads from JVM instance

```
using(JavaVM.JNIThreadContext ctx = new()) {
    // call Java methods
    ...
}
```

This approach has the following problems:

- It moves responsibility for detaching threads to a user, increasing the cognitive load imposed on them.
- Only threads that did not create a JVM instance need to attach and detach themselves. Usage of the interoperability library would differ among the main thread and the other threads – creating inconsistency in the API.
- Usage with a `using` block may encourage users to attach and detach threads repeatedly, which may not be efficient.

More importantly, however, this approach is not feasible in the context of **asynchronous programming**. Asynchronous programming in C# is achieved using `async` and `await` keywords. Put very simply, a method containing `async` keyword in its signature will get broken down into parts between two consecutive occurrences of `await` keyword in the method body. By default, these pieces will be executed by `ThreadPool` threads [113]. The two consecutive parts of the same method can be potentially executed by the same `ThreadPool` thread; however, often, they are executed by different `ThreadPool` threads (especially when `ConfigureAwait(false)` is called).

That might be problematic in our context because if the first part of the method explicitly attached its thread to JVM, the second part will expect to be run in a thread that is already attached, but that may not be the case if the second part of the method got scheduled to a different `ThreadPool` thread. That makes the approach requiring a thread to be attached and detached explicitly (using `Dispose` pattern) unfeasible.

To attach a tread to a JVM instance, we can use the initializer of the `ThreadLocal` field storing the `JNIEnv` pointer – it will get called the first time the field is accessed by a tread [112]. The question is how to ensure that a thread detaches itself.

Solution: hooking to thread exit event

We need to make sure that a thread will detach itself before it exits [8]. We can hook to the Thread Exit event and make sure that a clean-up callback detaching the thread is invoked.

There is no standard mechanism in .NET to hook to a Thread Exit event. There is, however, NuGet package `UnmanagedThreadUtils` [114] providing this functionality.

Using this approach, attaching and detaching threads can be completely transparent to a user and the usage of the library stays consistent between the tread that created JVM instance and other threads.

ThreadPool threads

The problem that remains to be addressed is `ThreadPool` [115] threads. A lifetime of `ThreadPool` threads is managed neither by us nor by a user. We, therefore, cannot ensure that all attached `ThreadPool` threads will exist – and therefore detach themselves before the VM is supposed to exit.

We can solve this by checking if a thread that is about to attach itself to JVM is `ThreadPool` thread (via `Thread.CurrentThread.IsThreadPoolThread` property [116]) and if so, attach it as a daemon (via `AttachCurrentThreadAsDeamon` JNI function). That way, `ThreadPool` threads won't keep the JVM instance alive.

2.5.3 Strongly typed references

As Section 1.2.1 described, JNI distinguishes between local and global object references. Local references are freed automatically when a native call returns; global references must be freed explicitly using `DeleteGlobalRef` JNI function [8].

JNI, however, does not distinguish between local and global references on the type system level (e.g., both `NewLocalRef` and `NewGlobalRef` JNI functions return `jobject` type, that represents a reference to Java object, but the type does not hold information about what kind of reference is it). This is a common source of issues when it comes to JNI programming, as mismatched reference types lead to an undefined behavior (often crash).

We would like to design our inner API in a more type-safe manner. We would also like to make sure that allocated global references will eventually get freed via `DeleteGlobalRef` JNI function.

Our interoperability library is going to contain a P/Invoke-based wrapper of the JNI function table. P/Invoke unmanaged delegates allow us to strongly type native functions pointers – we can, therefore, decide C# signatures of JNI function wrappers as long as P/Invoke is able to marshal between types in our signatures and types in signatures `jni.h` header.

We have the following options for how to represent JNI references (which are nothing more than C pointers):

- `IntPtr` type that would not hold any type information about reference whatsoever,
- `struct` wrapping `IntPtr` with `StructLayout` set to `LayoutKind.Sequential` – allows us to add a type information by representing different reference kinds by different `struct` types,
- `SafeHandle`s – `SafeHandle` class is intended to wrap unmanaged resources [117]. It is integrated into P/Invoke – P/Invoke is able to marshal between native pointers and C# `SafeHandle`s. `SafeHandle`s are designed to manage the lifetime and ownership of unmanaged resources held from the managed code. To use `SafeHandle`s, one should override `ReleaseHandle` method to specify how their unmanaged resources should be cleaned up. `SafeHandle` class should ensure that the handle won't be closed while it is used by a P/Invoke invocation and that the cleanup will run even if the application ends unexpectedly [118].

JNI local references do not require explicit clean-up – we can represent them by sequential layout `structs`. Global references can be represented by `SafeHandle`s that can manage the lifetime of references. We must ensure, however, that we will never create `SafeHanlde` instance wrapping local reference, as clean up using `DeleteGlobalRef` function would crash.

2.5.4 Type and method handles

Apart from local and global references to objects discussed in the previous section, JNI also requires us to work with handles of Java types and methods.

As Section 1.2.1 described, to invoke the Java method via JNI, one must first obtain a handle of defining Java type via `FindClass` JNI function. `FindClass` function in fact returns a local reference to an `java.lang.Class` instance that represents the required type [6]. As the number of Java types used by an application is going to be limited, we can create a global reference from the returned local reference and cache it in a static field.

When we hold the reference to a Java type, we can use it to look up a method handle via `GetMethodID` JNI function. This function returns **method ID**. Method ID is not a resource whose lifetime we can control. Method IDs are managed by JVM and are valid until the defining class gets unloaded [6]. We can therefore cache obtained method IDs in our interoperability library.

2.6 Primitive type arrays

Primitive type arrays differ significantly from other Java collection types. Because of runtime generic erasure in Java (see Section 2.1.2), unboxed primitive types (such as `int` and `double`) cannot be used as generic type parameters. Therefore, they cannot be stored in any generic collections (such as `ArrayList` [119]). Instead, boxed variants of primitive types (`Integer`, `Double`) must be used. Java arrays, however, do not suffer from this limitation. They can contain primitive types directly while having a C-array-like memory layout – consecutive blocks of memory containing primitive elements one next to another. Due to this special position that arrays hold among other Java types, JNI provides a dedicated API to work with them.

The goal of this section is to design a C# proxy class based on JNI API for primitive arrays. The proxy should allow C# programmers to work with Java primitive type arrays, allowing them to:

- read and write elements of Java arrays from C#,
- obtain array reference as a return value of Java method invoked from C#,
- create a new instance of Java array from C#.

Proxy API should provide as seamless user experience as possible (which is in accordance with requirement **R11**) while avoiding unnecessary overhead. Particularly, the following properties are desirable:

1. From the user's point of view, the array proxy should behave like a normal collection data structure, abstracting the fact that it crosses the language

border. Changes made to the array from C# should be visible to Java-side code and vice versa.

2. API of the proxy should be user-friendly and aligned with what would be expected from such a proxy: it would be desirable for a proxy to be generic (with type parameter specifying array element type) and to implement an indexer.

3. Solution should minimize copying of array elements as much as possible.

The rest of this section will attempt to fulfill these requirements to the best extent possible within the limitations imposed by JNI and C# programming language.

2.6.1 Selection of JNI functions

This section will analyze various functions that JNI provides for accessing primitive array elements, identifying the variant offering the best user experience while minimizing the overhead of copying.

Each of the functions described in the rest of this section exists in 8 variants: one for each Java primitive type. Behavior of the type variants, however, does not, in principle differ, and therefore, we will not distinguish between them in the rest of this section. Names of JNI primitive array functions contain a name of primitive type functions work with. The rest of this section will use `[PrimitiveType]` placeholder instead of the name of a particular primitive type in JNI function names to indicate that the reasoning applies to all the type variants of the function.

Get/ReleaseArrayElements JNI functions

The first pair of JNI functions to discuss are `Get[PrimitiveType]ArrayElements` and `Release[PrimitiveType]ArrayElements` functions.

`Get[PrimitiveType]ArrayElements` returns a pointer to elements of Java array. According to JNI specification [8] `Get[PrimitiveType]ArrayElements` function either pins Java array on Java managed heap and returns a direct pointer to it, or it allocates an unmanaged buffer, copies array elements into the buffer and returns a pointer to this copy.

`Release[PrimitiveType]ArrayElements` function either unpins the Java array (if it has been pinned) or copies the content of the buffer to the original Java array and frees the buffer (when the array was copied). Either way, the pointer to array elements is no longer valid after `Release[PrimitiveType]ArrayElements` function returns.

As our goal is to provide seamless propagation of array elements changes between C# and Java while minimizing an overhead of copying elements, **pinning Java array on Java heap and providing C# code a direct access to it would be ideal**. The problem is, however, that `Get[PrimitiveType]ArrayElements` JNI functions do not let a user decide if a Java array will be pinned or copied.

As was already mentioned in Section 1.2.1, JNI only describes an interface. Implementation of the interface is a matter of each Java distribution. JNI specification [8] suggests that if JVM implementation supports pinning

`Get[PrimitiveType]ArrayElements` function should return a direct pointer to a pinned array. It, however, leaves JVM the possibility of not pining an array to accommodate JVMs that do not support pinning. It seems, however, that today's JVM implementations do not follow this recommendation.

It is not easy to look up information about the implementation details of JNI by particular Java distributions. IBM's J9 VM documentation explicitly states that their implementation of `Get[PrimitiveType]ArrayElements` JNI function always copies array elements [120] for other JVM implementations. However, available information is fairly limited.

`Get[PrimitiveType]ArrayElements` function informs a user if it pinned or copied a Java array by returning `isCopy` flag. We carried out an experiment, observing values of `isCopy` flag when `Get[PrimitiveType]ArrayElements` JNI function is called with various sizes of an array on several Java distributions and versions. Concretely, we experimented with:

- **primitive element types:** int, double, signed byte,
- **array sizes:** 10, 100, 1000, 10000, 100000 elements,
- **Java distributions and versions:**
  - openJdk 21
  - openJdk 20
  - openJdk 17
  - openJdk 11
  - openJdk 8
  - graal VM 21
  - correto VM 21

In all observed scenarios, `Get[PrimitiveType]ArrayElements` function always performed copying. It is important to realize, however, that observing this result, we cannot conclude that pinning does not occur with any other combination of Java distribution and array size.

Get/ReleaseArrayCritical JNI functions

The other pair of functions JNI provides to access elements of primitive arrays are `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical`. According to JNI specification [8] `GetPrimitiveArrayCritical` function is more likely to return a direct pointer to a Java array even when JVM does not support pinning [6]. Specification, however, does not guarantee that a direct pointer will be returned. Copying can still occur [7].

Furthermore, there are strict restriction on actions that can be performed while critical array reference is held. Most importantly, no other JNI functions can be called before the reference is released. **That makes `Critical` alternative of array functions unusable in our context**.

[6] e.g., by temporarily disabling garbage collection while the array reference is held

[7] For instance documentation of J9 VM [120] states that `GetPrimitiveArrayCritical` function will return a direct pointer to Java array unless Java array is too big and is not stored in one consecutive block of memory.

Get/SetArrayRegion JNI functions

The last pair of functions to discuss are `Get[PrimitiveType]ArrayRegion` and `Set[PrimitiveType]ArrayRegion`. `Get[PrimitiveType]ArrayRegion` function takes JNI reference to Java array and pointer to a buffer, and it copies a specified region of elements of Java array to the provided buffer.
`Set[PrimitiveType]ArrayRegion` takes the same parameters but copies in the opposite direction: from the provided buffer to the Java array.

The advantage of these functions over
`Get/Release[PrimitiveType]ArrayElements` functions is that
`Get/Set[PrimitiveType]ArrayRegion` function allows us to specify our own buffer (so we can copy elements directly to or from C# array) and that they allow us to specify array region to be copied (avoiding the necessity to copy whole array when it is not required). They, however, can never take advantage of the possibility of pinning: copying is guaranteed to occur.

Working with copies of Java arrays makes it problematic to propagate array changes between Java and C#. Java arrays are reference types. Therefore, any Java method invoked from C# may access any array (the method can hold some reference to an array even if it does not take an array as its parameter). To make sure that changes that C# code makes to an array are always visible to Java, it would be necessary to copy all arrays C# holds a reference to Java before an arbitrary Java method is invoked. Symmetrically, to ensure that C# code always sees array changes Java carried out, it would be necessary to reload all arrays C# holds a reference to every time an arbitrary Java method returns to C#. That would introduce an unbearable amount of overhead. In contrast, obtaining a direct reference to a pinned Java array removes this synchronization struggle as well as the overhead of copying array elements. We cannot, however, enforce Java array pinning via JNI.

Considering these facts, it is necessary to somewhat relax requirements on the seamlessness of change propagation, as it is impossible to completely abstract change propagation from users without introducing an unacceptable amount of overhead. Users know their use case better, and they know when they need changes to be propagated across the language border. We need to design an array proxy API to allow them to do so as easily as possible.

Solution: Get/SetRange methods

We can build proxy API upon the usage of `Get[PrimitiveType]ArrayRegion` and `Set[PrimitiveType]ArrayRegion` JNI functions, allowing the user to copy specific ranges of elements between managed C# arrays and Java arrays. Names of API methods must, however, make it obvious that copying is taking place and that changes made to an array of elements returned from Java won't affect the Java array itself. We opted for `GetCopyOfArray` and `GetCopyOfRange` for the names of methods copying elements from Java array to C# array and for `SetRange` for a method coping elements from C# array to Java. Code Snippet 2.11 demonstrates the potential usage of this API.

Code Snippet 2.11: C#: Array API based on Get/Set[PrimitiveType>

```
1 //returns reference to array [0,0,0,0,0] from Java
2 using JavaPrimitiveArrayProxy<int> arrayProxy =
     JavaStaticClassProxy.ReturnArrayFromJava();
3 int[] valuesToSet = [1, 2, 3];
4 arrayProxy.SetRange(valuesToSet, startIndex:2);
5 int[] contentOfJavaArray  = arrayProxy.GetCopyOfArray(); // [0, 0,
     1, 2, 3]
```

Solution: Indexer

As we aim to provide an intuitive API, we would like to allow users to access elements of Java arrays via the C# indexer.

`Get<PrimitiveType>ArrayRegion` and `Set<PrimitiveType>ArrayRegion` JNI functions could be potentially used to implement an indexer on primitive array proxy type. Such an indexer would, however, have to carry out separate JNI calls for each primitive array element access (read or write), which would be inefficient.

We could use `Get[PrimitiveType]ArrayElements` function to obtain a pointer to elements of Java array (either pined or copied) and implement the indexer via pointer arithmetic using this pointer. The problem is that if `Get[PrimitiveType]ArrayElements` function copies, changes made via this indexer would be visible to Java–side code until `Release[PrimitiveType]ArrayElements` function is called.

This motivates us not to store array elements pointer directly in C# primitive array proxy class but rather represent it by a separate class. This class may implement `IDisposable` interface so that the user can obtain an instance in a `using` block, and `Dispose` method will be automatically called when the block is left invoking `Release[PrimitiveType]ArrayElements` under the hood. A potential usage example can be seen in Code Snippet 2.12.

As Code Snippet 2.12 shows, using this approach, the indexer can be implemented on class representing array elements (`JavaPrimitiveArrayElements` class in Code Snippet 2.12) rather than on array proxy class itself. That provides us with another advantage. User will only be able to obtain an instance of `JavaPrimitiveArrayElements` class by calling `LoadJavaArrayElements` method that gains access to an element by invoking `Get[PrimitiveType]ArrayElements` function. Therefore, we can be sure that when the user accesses array elements via indexer, array elements have already been made available to C# code.

Code Snippet 2.12: C#: Array API based on Get<PrimitiveType>ArrayElements JNI function

```
1 using JavaPrimitiveArrayProxy<int> arrayProxy =
     JavaStaticClassProxy.ReturnArrayFromJava();
2 using (JavaPrimitiveTypeArray<int>.JavaPrimitiveArrayElements
     arrayElements = arrayProxy.LoadJavaArrayElements()) {
3     for (int i = 0; i < arrayProxy.Lenght; i++) {
4         arrayElements[i] = i;
5     }
6 }
```

2.6.2 Generic proxy over non-generic JNI functions

The previous section decided which JNI function should be used to implement primitive array proxy and how to build proxy API methods above them. One more problem, however, remains to be solved. JNI is a C-language API, and as such, it has no concept of generics. As the previous section mentioned, each of the discussed JNI functions actually exists in 8 variants – one for each Java primitive type. These variants differ both in function parameter types and in function names (as primitive type name is a part of function name). Our goal is to build a user-friendly C# API based on this set of non-generic functions. As discussed in the introduction to this section, such API should preferably be generic.

It would be easy to implement non-generic array proxies for each primitive type separately. This design would result in a set of proxy classes such as `JavaIntArrayProxy` and `JavaDoubleArrayProxy` and so on. The resulting code would contain a lot of duplicities.

Duplicities could be avoided by moving array access logic to a generic parent class. Parent class could define a set of abstract delegates and use these delegates to implement array access logic. Non-generic child class could be implemented for every primitive element type, implementing abstract delegates as invocations of JNI primitive array functions corresponding to particular primitive type. This approach would avoid code duplicities but it would not result in generic API we would prefer.

The goal, therefore, is to **implement generic primitive array proxy that under the hood uses different sets of JNI functions based on the type parameter provided**. Such a thing would be easy to achieve using C++ templates as they support **explicit specialization** - a custom implementation of a template for a specific type. C# generics, however, do not support this feature [121].

We could attempt to carry out a dispatch based on type parameter in runtime using **switch expression** as demonstrated in Code Snippet 2.13. This Code Snippet, however, would not build as C# cannot cast `int[]` or `double[]` to an array of type parameter type.

Code Snippet 2.13: C#: Runtime dispach base on type parameter - does not build

```
1 internal TElem[] GetArrayRegion(int start, int len) {
2     return typeof(TElem) switch {
3         Type type when type == typeof(int) =>
              (TElem[])_env.GetIntArrayRegion(
4             _arrayInstance, start, len),
5         Type type when type == typeof(double) =>
              (TElem[])_env.GetDoubleArrayRegion(
6             _arrayInstance, start, len),
7         /* ... */
8     };
9 }
```

Even if the approach from Code Snippet 2.13 worked, it would introduce an overhead of runtime dispatch to each invocation of the JNI primitive array

function. It would be better to minimize the amount of times this dispatch occurs.

We can represent JNI primitive array functions by generic delegates, store these delegates in the static dictionary by the type they work with, and retrieve particular delegate from the dictionary during the static initialization of primitive array proxy based on specified type parameter. This approach is demonstrated in Code Snippet 2.14.

Code Snippet 2.14: C#: type dispatch during static initialization

```
1 private static PrimitiveArrayJniFunctions<TElem>
      _primitiveArrayJniFunctions =
      (PrimitiveArrayJniFunctions<TElem>)
2 JNIPrimitiveArrayFunctionTable.Table[typeof(TElem)];
```

This way the dispatch will occur once per static initialization of generic type, therefore once per each specialization of this type, which is much more efficient then dispatch during each invocation.

The only problem that remains to be solved is how to store different generic specializations of delegates in the same collection. Code Snippet 2.14 has already hinted at the solution. We can implement a generic class encapsulating the set of delegates, and we can inherit this class from the non-generic parent as is demonstrated in Code Snippet 2.15

Code Snippet 2.15: C#: Class wrapping generic delegates of JNI primitive array functions

```
1  internal abstract class PrimitiveArrayJniFunctionsBase { }
2
3  internal class PrimitiveArrayJniFunctions<TElem> :
       PrimitiveArrayJniFunctionsBase {
4      internal NewPrimitiveArrayGlobalDelegate<TElem>
5          NewPrimitiveArray { get; }
6      internal GetArrayElementsDelegate<TElem>
7          GetArrayElements { get; }
8      internal ReleaseArrayElementsDelegate<TElem>
9          ReleaseArrayElements { get; }
10     internal GetArrayRegionDelegate<TElem>
11         GetArrayRegion { get; }
12     internal SetArrayRegionDelegate<TElem>
13         SetArrayRegion { get; }
14 }
```

Every specialization of the generic delegate wrapper class will, therefore, inherit the parent and can, therefore, be stored in the collection of parent type. This collection can be initialized as follows (Code Snippet 2.16).

Code Snippet 2.16: C#: Dictionary of JNI primitive array function delegates

```
internal static readonly Dictionary<Type,
    PrimitiveArrayJniFunctionsBase> Table = new()
{
    { typeof(int), new PrimitiveArrayJniFunctions<int>(
        NewIntArrayGlobal, GetIntArrayElements,
        ReleaseIntArrayElements, GetIntArrayRegion,
        SetIntArrayRegion )},
    { typeof(double), new PrimitiveArrayJniFunctions<double>(
        NewDoubleArrayGlobal, GetDoubleArrayElements,
        ReleaseDoubleArrayElements, GetDoubleArrayRegion,
        SetDoubleArrayRegion )},
         ...
}
```

As Code Snippet 2.14 showed, when the value is retrieved from this dictionary in static initializer of primitive array proxy, we must cast it back to generic child type. As we, however, have complete control over static initialization of the readonly dictionary, this does not seem to be an issue.

2.7 Proxy design

The previous sections focused on low-level aspects of building a JNI-based interoperability engine: accessing JNI functions from different threads, dealing with JNI references, calling methods, and allowing users to work with arrays. According to the thesis requirement **R11**, however, the implemented solution should also provide a seamless user experience while working with arbitrary Java objects via generating .NET proxies of Java classes. And according to the requirement **R5**, the configuration of such proxies should be user-friendly and in-code. This section will be devoted to designing such a user-friendly proxy API.

According to the requirement, **R9** solution should not require the user to use an external tool to generate .NET proxies. To fulfill this requirement, we can use **incremental source generators** added to .NET in .NET 6 [122] [8]. Incremental source generator allows us to hook into the compilation process of .NET assembly and generate additional code based on the code already contained in the assembly. The generated code will be added to the compilation. We can use an incremental source generator to generate C# proxies of Java classes that the user wants to work with.

Source generators work conveniently in combination with partial methods (that were tweaked in .NET 5 to accommodate source generation [123]) and partial types. Users can define a signature of a partial method and have the body of the method generated by a source generator. Usually, a source generator library also defines dedicated **marker attributes** that are used to annotate partial methods and types whose implementation should be generated by the particular source generator [124].

[8] Incremental source generators are an improvement upon source generators that were added to .NET in .NET 5.

We are going to use this approach to allow users to invoke Java methods from C#. The user will define a partial method whose name and signature match the Java method they want to invoke. User will also annotate this method by `JavaImport`[9] marker attribute. The incremental source generator will locate the method by the marker attribute and will generate the method implementation that will use our interoperability library to leverage the interop invocation. Implementation of the source generator will be analyzed in more detail in Section 2.9

Source-generating bodies of partial methods are the user-friendly way to emulate Java methods from C#. To allow users to work with Java objects from C#, we would, however, also like to support other language features such as fields and constructors. It would be nice to emulate Java constructors by C# constructors and Java fields by C# properties, as that would result in an intuitive and easy-to-use API. C# constructors and properties, however, cannot be marked as partial [125]. Thus, the approach described for methods cannot be directly applied to them. This section will address this limitation. It will analyze alternative ways how to allow users to specify that the generated proxy class should emulate a given constructor or field implemented on the underlying Java class. The last part of this section will then look at how to best represent strings in C# proxy method signatures.

2.7.1 Emulating Java constructors

As the introduction of this section mentioned, we would like to **emulate constructors of Java classes by constructors of C# proxy classes**. Concretely, if the Java class defines a constructor that takes `int` and `long` parameters (as shown in Code Snippet 2.17), we would emulate this constructor by the constructor of the proxy type with the same signature (demonstrated in Code Snippet 2.18).

Code Snippet 2.17: Java: Java class with constructor

```
1 public class Coffee {
2     public Coffee(int abraka, long dabra) { ... }
3 }
```

Code Snippet 2.18: C#: Goal: proxy class emulating Java constructor by C# constructor

```
1 [JavaClassProxy]
2 public class CoffeeProxy {
3     public CoffeeProxy(int abraka, long dabra) {
4         // use JNI to invoke Java constructor
5         ...
6     }
7 }
```

[9]The name of the attribute is inspired by `LibraryImport` and `DllImport` attributes that are a part of P/Invoke (ses Section 1.2.2).

As the constructor cannot be partial, we cannot let the user specify the constructor signature, and then the source generates the constructor implementation (as we can do with ordinary methods). The question is then: how can we let a user specify the signature of a particular constructor of the underlying Java class they want to invoke from C#?

The designed solution should also support specifying **multiple constructors with different signatures for a proxy class**.

Marker attribute parameters

Our solution already uses marker attributes to direct source generation. We use `JavaImport` attribute to mark proxy methods and `JavaClassProxy` attribute (see Code Snippet 2.18) to mark proxy classes. We could use an attribute applied to a non-static proxy class to specify constructor parameters[10].

To specify the constructor signature, the user needs to be able to specify a list of pairs: **parameter type** and **parameter name**. We can define `CtorParameter` structure to represent this pair (as shown in Code Snippet 2.19). We would like to define `JavaImportCtor` attribute that would take an array of these structures as its parameter. Such attribute could be applied to proxy class as shown in Code Snippet 2.20.

Code Snippet 2.19: C#: Structure to specify constructor parameter

```
1 public record struct CtorParameter(Type parameterType, string
      parameterName);
```

This approach would allow us to specify multiple constructor signatures for a single proxy class as we can allow for multiple instances of `JavaImportCtor` attribute to be applied to a proxy class definition[11].

Code Snippet 2.20: C#: Does not build: unsupported type of attribute parameter

```
1 [JavaImportCtor(
2     new CtorParameter(typeof(int), "abraka"),
3     new CtorParameter(typeof(long), "dabra"))]
4 [JavaClassProxy]
5 public partial class CoffeeProxy
6 {
7 }
```

The problem with this approach, however, is that Code Snippet 2.20 is not valid C# code. C# only allows attribute parameters to be primitive types, enums, strings, and instances of `System.Type` type or one-dimensional arrays of these types [126]. We are trying to provide an array of custom `CtorParameter` structures as the attribute parameter, which is not supported.

We could try to work around this problem by changing the definition of `JavaImportCtor` attribute so that it takes two separated arrays – an array of parameter types (array of `System.Type` instances) and an array of parameter

[10] We don't want to `JavaClassProxy` attribute directly to specify constructor parameters as this attribute can also be applied to static classes, for which constructors do not make sense.

[11] By applying `AttributeUsage` attribute to `JavaImportCtor` attribute definition and setting `Allowmultiple` field to `true`).

names (array of strings). That way, it would meet the type restriction imposed on attribute parameter types. This version of the attribute could be applied to a proxy class as shown in Code Snippet 2.21.

Code Snippet 2.21: C#: Attempt to design ctor attribute

```
1 [JavaImportCtor(
2     new Type[] {typeof(int), typeof(long)},
3     new string[] {"abraka", "dabra"})]
4 [JavaClassProxy]
5 public partial class CoffeeProxy{
6 }
```

Though the code in Code Snippet builds and allows users to specify the signatures of Java constructors, it does not provide a pleasant user experience as it is easy to mismatch parameter name and parameter type when they are defined separately.

Factory method

As the previous paragraph shows, specifying Java constructor signature via attribute parameters does not lead to a user-friendly API. Let's try to approach the problem differently.

C# partial methods are very convenient for specifying signatures of Java methods to emulate. C# constructor itself cannot be partial, so it cannot be used in this manner. Users can, however, specify a signature of partial **factory method**. An example of such a method is shown in Code Snippet 2.22. On line 3, notice that the factory `Create` method is annotated by `JavaImportCtor` attribute. Thanks to that, the source generator can distinguish factory methods from partial methods emulating ordinary Java methods (annotated by `JavaImport` attribute).

Code Snippet 2.22: C#: Definition of partial factory mehtod

```
1 [JavaClassProxy]
2 public partial class CoffeeProxy {
3     [JavaImportCtor]
4     public static partial CoffeeProxy Create(
5         int abraka, long dabra);
6 }
```

The factory method itself can be used to create instances of a proxy class. Our goal is, however, to emulate Java constructors by constructors of C# proxy classes. To achieve that, we can **generate a constructor taking the same parameters as the factory method.** That is demonstrated in Code Snippet 2.23. The generated constructor (line 9) handles JNI invocations necessary for creating a new instance of an underplaying Java class (JNI implementation details are omitted from the example as they are not relevant to this section). The generated body of the factory just invokes the generated constructor (line 5).

Code Snippet 2.23: C#: Source generated implementation of factory method and related constructor

```
public partial class CoffeeProxy {
    // generated implementation of factory method
    public static partial CoffeeProxy Create(
        int abraka, long dabra) {
            return newCoffeeProxy(abraka, dabra);
        }

    // generated ctor
    public CoffeeProxy(int abraka, long dabra) {
        /* invoke Java ctor via JNI */
        ...
    }
}
```

Using this approach, users can choose if they will create instances of proxy classes via the factory method or via the constructor. That may be viewed as a slight disadvantage as a redundant factory method is added to a proxy API. This approach, however, allows users to specify signatures of Java constructors in a user friendly manner that is consistent with specifying signatures of ordinary proxy methods. It also allows to define multiple constructors for single proxy class. The advantages of the approach, therefore, seem to compensate for this slight disadvantage.

The last problem to consider is that `JavaImportCtor` attribute specifying the factory method should only be applied to the static method defined in the non-static class, where the return type of the method matches the class in which the method is defined. However, as incremental source generators can emit diagnostics that will manifest themselves as compiler errors or warnings, we can ensure the desired usage of our attributes quite easily (see Section 3.5.2).

2.7.2 Emulating Java fields

This section will focus on designing an API for accessing Java fields from C#. As in C#, fields in Java usually are not part of a public API of a class. An exception can be constants: `public final fields`, which will be discussed later in this section. C# usually exposes private fields via properties. As Section 2.1.5 explained, in Java, the concept of properties does not exist. Instead, fields are exposed via getter and/or setter methods.

Java getter and setter methods

From the point of view of JNI, getter and setter methods do not in any way differ from any other Java methods. Our solution is therefore able to emulate them as methods annotated by `JavaImport` attribute. That allows C# programmer to access Java fields via their setter and getter methods. That, however, is not an API C# programmer would expect.

It would be more user-friendly for C# programmers if we managed to emulate Java getter and setter methods by C# property. We could require a user to an-

notate methods that are emulating getter and setter by a different attribute than `JavaImport` attribute: for instance `JavaImportGetter` and `JavaImportSetter` attributes as shown in Code Snippet 2.24. Based on these methods (their names and signatures), we could potentially generate C# property.

Code Snippet 2.24: C#: Methods emulationg Java getter and setter methods

```
1 [JavaImportGetter]
2 public partial int GetAbraka();
3
4 [JavaImportSetter]
5 public partial void SetAbraka(int value);
```

This approach provides a slightly more user friendly API for C# programmers. It, however, does not enable any new functionality. We therefore won't included in the scope of the thesis. It may be implemented as a future improvement.

Accessing Java fields directly

In some contexts, it may make sense to access Java fields from C# directly (not by invoking getter or setter methods). An example of such a situation may be accessing `public final field` (Java constant) or working with Java API that does not comply with the best practices and exposed fields as a part of public API (according to requirement **R3** our solution should not require a user to modify a Java-side code in order to be able to work with it from C#).

In general, our solution consists of the **incremental source generator** and the **interoperability library** that provides the logic that source-generated code uses to handle interop invocations. Using the interoperability library directly, we can implement the C# property that accesses the Java field. An example of such a property is shown in Code Snippet 2.25. Line 1 obtains a JNI handle (by calling `GetFieldId` JNI function) that can be used to access a field called `Abraka` of Java class whose type is represented by `_javaType` variable. The second parameter of `GetFieldId` function specifies that the type of the field is `int` (see Section 1.2.1 for an explanation of how JNI encodes type signatures). The rest of Code Snippet 2.25 defines a getter and a setter that use JNI to access the field represented by the obtained JNI field handle.

Code Snippet 2.25: C#: Property emulating Java field

```
1  private static Lazy<JniField> _jni_abraka = new Lazy<JniField>(
2      () => _javaType!.GetFieldId("Abraka", "I"));
3
4  public int Abraka {
5      get {
6          return Instance.GetIntField(_jni_abraka.Value);
7      }
8      set {
9          Instance.SetIntField(_jni_abraka.Value, value);
10     }
11 }
```

To implement the code in Code Snippet, 2.25, the user would be required to have some knowledge of JNI API (for instance, they need to know that before accessing a field, a field handle must be obtained via `GetFieldId` function and have to understand JNI type signatures). That is surely not a user-friendly API. We would like to use an incremental source generator to generate the code in Code Snippet 2.25 automatically.

If C# allowed for **partial properties**, a user could specify a name and a type of a Java field as a partial property, and we could source generate the implementation of the property as shown in Code Snippet 2.25. Unfortunately, C# does not support partial properties [123]. There is an open issue in the .NET repository on GitHub [127], asking for support for partial properties for the purposes of source generation, but it has not yet been implemented.

We could try to let the user to **specify a name and type of Java field** they want to access **by defining a C# field of corresponding name and type** (as shown in Code Snippet 2.26) and to source generate property such as the one in Code Snippet 2.25 based on this field.

Code Snippet 2.26: C#: Specify Java field by C# field

```
1 [JavaImportField]
2 private int _abraka;
```

Similar approach is used by **MVVM Community Toolkit** [128] to implement **observable properties** [129]. There a user specifies a field as shown in Code Snippet 2.27 [129] and based on it a source generator generates a property shown in Code Snippet 2.28 [129].

Code Snippet 2.27: C#: MVVM Community Toolkit: backing field for observable property

```
1 [ObservableProperty]
2 private string? name;
```

Code Snippet 2.28: C#: MVVM Community Toolkit: generated property

```
1 public string? Name
2 {
3     get => name;
4     set => SetProperty(ref name, value);
5 }
```

The difference between MVVM Comunity Toolkit observable property example and our situation is that observable properties actually use the user defined field as their backing field. Therefore the field is not just uselessly taking up additional memory in proxy instances and its value is in sync with the value of generated property.

In our scenario, the generated property would not use the user-defined field as its backing field. The backing field of the property would be the Java field accessed via JNI API. The C# field would be only used to specify the name and the type of the Java field and, therefore, of the generated property. In runtime, the field should never be used; it would just take up memory in proxy

class instances. Moreover, there is no way to keep the value of the C# field in sync with the actual value of the Java field. Therefore, the field may be confusing for a user. That is not an efficient or user-friendly option.

Another option would be to **access Java field via getter and setter methods defined on C# side**. The user would specify signatures of partial methods, and the source generator would generate method bodies using JNI to access the Java field (in the same way as the property in Code Snippet 2.25 accesses it).

As discussed previously, the resulting API would not be what a C# programmer would expect. Moreover, if Java-side code happens to define getter and setter methods for the given field, C#-side getter and setter methods accessing Java fields via JNI could be mixed up with C# proxies of Java-side getter and setter methods (that invoke Java-side getter/setter via JNI). Therefore, the whole API could easily become convoluted.

As none of the described options seems like a reasonable alternative and as there is a hope that some of the future versions of C# will introduce partial properties that will allow for a more user-friendly way of accessing Java fields from C#, the **dedicate API for accessing Java fields won't be in the scope of this thesis**. If the user has a real need to access the Java field directly, they can manually implement the property shown in Code Snippet 2.25.

2.7.3 String parameter and return types

The next topic to discuss is how to represent strings in proxy method signatures. Java and .NET use the same string representation: UTF-16 encoding, endianness depending on the underlying architecture, and most importantly, strings in both languages are immutable [130, 131, 132]. We can use these similarities to our advantage.

Passing C# string to Java

As for arrays, JNI provides dedicated functions to work with strings. To pass a C# string as a parameter to the Java method, a new instance of `java.lang.String` with the same content must be allocated on the Java heap via `NewString` JNI function. This function takes as a parameter a pointer to a buffer from which it initializes the content of the new Java string instance. As Java and C# use the same string encoding, we can let JNI copy directly from the pinned instance of managed C# string, avoiding a need for additional copying (e.g., to an unmanaged buffer).

However, using this approach, we still have to copy the string once (from C# heap to Java heap). Passing a string from C# to Java can, therefore, be considered an expensive operation. That leads to the question of how to represent strings in JNI method signatures.

Disadvantages of representing `java.lang.String` by `System.String`

The most intuitive for a user would be to emulate Java `java.lang.String` by C# `System.String`. That would, however, require us to allocate a new string instance on the Java heap each time a C# string is passed to Java, even if an (immutable) string with the same content has already been allocated to be passed from C# to Java. That is not an efficient solution.

One possible solution would be to implement a string cache as a part of our library. Once a string instance is allocated on the Java heap via `NewString` function, the returned JNI reference can be used as an argument of a Java method an unlimited amount of times before it is freed. It can, therefore, be cached. The user could pass a string as C# `System.String` to a method annotated by `JavaImport` attribute. We could check if the cache already contains JNI reference to a `java.Lang.String` instance with the same content, and if so, reuse the reference. This would be possible without changing code semantics because strings in both languages are immutable.

The problem with this approach is determining the parameters of the cache. We are implementing a general tool that should allow users to work with arbitrary Java code from C#. Therefore, We can'te, make any assumptions about the common string workload a user will have because these workloads can widely differ between use cases. For some use cases, the cache may be useless as they won't pass the same strings repeatedly (in which case the cache would unnecessarily keep alive two copies of each string – Java copy and C# copy). For others, the optimal size of the cache may differ. We could make the cache (its size and if it is used at all) configurable, but that seems unnecessarily complex for the purposes of this thesis.

Representing `java.lang.String` by `JavaLangStringProxy`

Instead, we can let the user decide if it makes sense to reuse string references in their context. We can represent Java strings by `JavaLangStringProxy` C# proxy class. This class will hold JNI reference to `java.lang.String` instance allocated on the Java heap. Such reference can either be obtained by allocated new Java string from C# via `NewString` JNI function or as a return value of a string Java method called from C#. Users will be able to use an instance of `JavaLangStringProxy` as an argument of the Java method an arbitrary amount of times (possibly caching the reference if they need it).

Representing strings by `JavaLangStringProxy` instances is convenient for more advanced users who want to take advantage of some form of caching of string references. It may, however, present unnecessary cognitive load for other users as it requires them to use `JavaLangStringProxy` type in order to be able to pass strings between C# and Java.

We can provide a more user-friendly API by supporting both `JavaLangStringProxy` and `System.String` types in proxy method signatures. Internally, we will still work with strings as `JavaLangStringProxy` instances; we can, however, implement conversion operators between `System.String` and `JavaLangStringProxy`, allowing the user to pass `System.String` instance to Java proxy method and converting it to `JavaLangStringProxy` (allocating string instance on the Java heap) internally.

Consider the Java method `javaMethod` shown in Code Snippet 2.29.

Code Snippet 2.29: Java: Example of method with string type in its signature

```
1 public static String javaMethod(
2     String abraka, String dabra) { ... }
```

Due to the approach described above, the user can choose which occurrences of `java.lang.String` in the Java method signature are to be represented by

`System.String` type and which by `JavaLangStringProxy` type. That may result in various potential signatures of the C# proxy method invoking `javaMethod`. Code Snippet 2.30 shows the variant where the user opted for representing all strings by `System.String` type. Code Snippet 2.31 captures a variant where the user wants to take advantage of string reference caching for strings passed as parameter `abraka` (and therefore represents it by `JavaLangStringProxy` type) but does not need to cache strings passed as `dabra` parameter.

Code Snippet 2.30: C#: Signature of proxy method represnting strings via dedicated proxy

```
1 [JavaImport]
2 public static partial string JavaMethod(
3     string abraka, string dabra) { ... }
```

Code Snippet 2.31: C#: Signature of proxy method represnting strings via dedicated proxy

```
1 [JavaImport]
2 public static partial string JavaMethod(
3     JavaLangStringProxy abraka, string dabra) { ... }
```

This approach allows users to reuse string references if it makes sense in their context while still allowing for intuitive proxy method signature for less proficient users or in contexts where string reference caching does not provide any benefit.

2.8 Emulating Java type system

The previous section focused on designing a user-friendly API for C# proxies of Java classes. This section will focus on how to make these proxies the best possible C# representation of the Java-side type system they are emulating.

2.8.1 Creating proxy instances

Requirement **R10** states that it should not be possible to represent instances of a Java type by instances of C# proxy type that does not correspond to that Java type. In other words, if Java class `Coffee` is represented by C# proxy `CoffeeProxy`, then from the user point of view, it should not be possible to create an instance of `CoffeeProxy` class that would hold the reference to any other Java type than `Coffee` class (or some class inherited from `Coffee` class).

Two conflicting aspects need to be considered in this context:

1. Proxies of Java classes will be **defined in users assembly**, and their implementation will be source generated by incremental source generator. As the generated code will be a part of the user assembly, it **can only access public members of the interoperability library**.

2. Proxies must be able to **emulate Java methods returning object**. JNI returns objects in the form of (local or global) references (see Section 1.2.1).

JNI references are going to be wrapped by proxy instances for users' convenience. Generated proxy code must, therefore, be able to **create instances of** (other) **proxy classes**.

To sum up these two requirements: **generated proxy code must be capable of creating proxy instances from JNI object references using only public API of interoperability library**.

At the same time, however, once we expose the possibility of creating a proxy instance from a given JNI reference to a user, a user can easily and potentially accidentally violate requirement **R10**, creating a proxy instance with mismatched Java type. We cannot, however, easily move the proxy instance creation inside the interoperability library (without using reflection) as **constructors cannot be a part of the interface contract**.

Another related problem to take into consideration is **support for arrays of objects**. Arrays of objects will be represented by the generic proxy class that will be a part of the interoperability library (similar to the primitive type array discussed in Section 2.6). The array element type (which should always be a proxy type) will be specified by a generic type parameter. When an element of an object array is being accessed, it is necessary to create an instance of the proxy type to return to a user. The proxy type is, however, only specified as a type parameter, and **C# does not enable to require other than parameterless constructor via type parameter constraints**.

To sum up the problem, we need to be able to generate a code that uses only the public API of the interoperability library, and given JNI reference to a Java object, it is able to create an instance of a user-defined proxy whose type is only specified as a generic type parameter. At the same time, we want to deter users from creating proxy instances wrapping arbitrary JNI references as that would violate requirement **R10**.

Solution: Static abstract interface methods

To tackle these problems, we can take advantage of **static abstract interface methods**. Static abstract interface methods were added to C# in C# 11 [133]. Since then, it has been possible to have static methods as part of an interface contract.

We can define an interface that will require proxy classes to implement the static `CreateProxyInstance` method. We can ensure that every proxy class will implement this interface by source-generating base list and interface implementation.

`CreateProxyInstance` method is part of the interface contract and, therefore, can be invoked from the interop library. Moreover, the static interface method can be invoked on a generic type parameter, and therefore, we can use it in the interoperability library to implement a generic method accessing an object array element and returning it as an instance of a proxy class.

Other advantage of this approach is that it moves the responsibility for the process of proxy instance creation to proxy itself while keeping instance creation API consistent among proxy type. This allows us to implement specialized logic for creating instances of proxies implemented manually in library (such as object array proxy) while being able to invoke this logic in the same manner as code generated logic of user defined proxies.

Solution: `JniInternalMarker`
What remains to be solved is that as with any other interface method, static interface method `CreateProxyInstance` is part of a public interface contract, and therefore, it is accessible to a user (potentially violating requirement **R10**). We can, however, discourage a user from invoking this method by not allowing a user to obtain instances of types that `CreateProxyInstance` method expects as parameters. We can add an additional parameter of `JniInternalMarker` type. `JniInternalMarker` type will be a public singleton type provided by our library. Its singleton instance, however, will be internal to our library. Therefore, the type itself may appear in the user's code (e.g., in `CreateProxyInstance` method signature), but the user will be unable to obtain an instance of this type (by standard means). A potential definition of such type is shown in Code Snippet 2.32.

Code Snippet 2.32: C#: Public singleton of which user will never obtain an instance

```
1 [EditorBrowsable(EditorBrowsableState.Never)]
2 public sealed class JniInternalMarker {
3     internal static JniInternalMarker Instance = new();
4     private JniInternalMarker() { }
5 }
```

Users can still obtain an instance of `JniInternalMarker` type via reflection. That would, however, require an active attempt on a user's part to try to use the library in a manner that was not intended. It does not seem necessary to try to prevent this behavior. Our goal is to discourage a well-meaning user from accidentally breaking our invariants. If users are actively trying to break things for themselves, consequences are their to bear.

2.8.2 Returning interface

The previous section focused on creating instances of user defined proxies, allowing us to emulate Java methods returning object, to access elements of Java object arrays (and potentially to access Java object fields). This section will focus on supporting interfaces in the same scenarios.

From the point of view of this section, interface types differ from class types because, unlike class type, **interface type does not determine a runtime type to return**. If we are emulating a Java method that has Java `Coffee` class type as its return type, we know that any instance returned at runtime from Java will be an instance of `Coffee` (or some class inherited from `Coffee` class). On the C# side, we can, therefore, represent returned values by `CoffeeProxy` proxy class.

The situation is different when working with interface types. We can emulate Java interface (e.g., `Shape`) by C# interface (e.g., `IShape`) (the two concepts do not significantly differ, see section 2.1.6). At runtime, when invoking the Java method returning `Shape` type, however, we have to return an instance of some C# proxy class that implements `IShape` proxy interface. The problem is that we do not know what type it should be. Statically, we only know that the type returned from Java implements the Java `Shape` interface; we don't know which C# proxy should be used to represent this type.

Default interface proxy

The previous section focused on creating instances of user-defined proxies, allowing us to emulate Java methods returning objects to access elements of Java object arrays (and potentially to access Java objects We can solve this problem by generating an artificial class proxy for each interface proxy – a **default interface proxy** class. This proxy won't represent any existing Java class. It will be used to wrap returned JNI references to a Java class implementing an interface. Given that Java `Shape` interface is implemented as shown in Code Snippet 2.33, default C# proxy class for this interface can be `ShapeDefaultProxy` shown in Code Snippet 2.34. In Code Snippet 2.34, notice that `ShapeDefaultProxy` implements interface methods via JNI invocations. As `ShapeDefaultProxy` instances will at runtime wrap reference to existing Java class that implements Java interface, JNI will handle the virtual method dispatch and invoke methods on correct Java method.

Code Snippet 2.33: Java: Example of Java interface

```
1  public interface Shape {
2      int intInterfaceMethod(int a);
3      double doubleInterfaceMethod(double a);
4  }
```

Code Snippet 2.34: C#: Default proxy class for Java interface

```
1  public interface ShapeDefaultProxy {
2      public int IntInterfaceMethod(int a) {
3          /*handle method invocation using JNI */
4          ...
5      }
6      public double DoubleInterfaceMethod(double a) {
7          /*handle method invocation using JNI */
8          ...
9      }
10     /* other JNI-based code */
11     ...
12 }
```

Described approach, however, **looses a part of a type information**. It could happen that the Java type returned at runtime already has a dedicated C# proxy type. We would, however, represent the returned reference by the default interface proxy. From the C# type system point of view, however, this default interface proxy is not related in any way to the existing dedicated proxy type. It **would not be impossible to cast the default proxy to dedicated proxy type**, even though both would hold a reference to the same Java type at runtime.

The best effort to capture correct runtime type

Equipped with incremental source generators and JNI, we can improve upon the solution.

JNI provides `GetObjectClass` function [8] that, given an object instance reference, returns a reference to `java.lang.Class` instance representing the runtime type of the instance. We can apply this JNI function at runtime to **get a runtime type of the object returned from the Java method**. What remains to be solved is how to create an instance of C# proxy type corresponding to this Java type if such C# proxy type exists.

Using Roslyn API in the source generator, we can find all the classes in a given assembly that implement a given interface (in our example `IShape` interface representing Java `Shape` interface). We can use this information while source generating interface proxy to be able to map Java runtime type to C# delegate that creates a C# proxy instance of a correct proxy type. As was described in Section 2.8.1, each C# proxy class will implement `CreateProxyInstance` method, which creates a proxy class instance and can be therefore used to instantiate this delegate.

Mapping can be implemented by a static dictionary generated as a part of the C# interface proxy type. An example of such a dictionary is shown in Code Snippet 2.35. Notice that the dictionary maps Java type name (that can be obtained via JNI invocations given JNI object reference) to a delegate to `CreateProxyInstance` method.

Code Snippet 2.35: C#: Generated dictionary enableling to match runtime Java return type of methods returning interface type

```
1  public interface IShapeProxy : IJavaInterface<IShapeProxy> {
2      private static readonly Dictionary<string, ProxyCreator>
           _interfaceImplementations = new() {
3          {
4              "javasources.testclasses.Circle",
5              new ProxyCreator(Circle.CreateProxyInstance)
6          },
7          {
8              "javasources.testclasses.Square",
9              new ProxyCreator(Square.CreateProxyInstance)
10         },
11     };
12 }
```

The described approach allows us to match the runtime type of the Java instance represented statically by interface type with the correct C# proxy type. It will only work, however, **if such proxy type exists in the C# assembly that defines proxy of the given interface**. **We still need to provide the default interface implementation proxy** (as shown in Code Snippet 2.34) **as a fall-back** when C# proxy class corresponding to the runtime type of Java object is not present.

`CreateProxyInstance` method for interface proxies

To be able to work with interface proxies in the same manner (described in Section 2.8.1) as with other proxy types, we need an interface proxy to implement the static `CreateProxyInstance` method. This method can apply the steps described

previously in this section to return an instance of the best possible proxy class that implements a given interface.

2.9 Incremental source generator

The last part of the solution is an incremental source generator that is going to generate implementation of proxy classes and methods. This section discusses some design decisions made while developing it.

2.9.1 Generating via string or Roslyn API

An incremental source generator allows us to hook into the compilation process of a user-implemented assembly, inspect a user-written code using Roslyn API, and generate an additional source code that gets added to the compilation (as if it was written by the user). Generated source code is always added to the compilation in the form of a string. We are, however, given an option to either generate it directly as a string or to use Roslyn API to generate a syntax tree that gets converted to a string directly before it gets added to the compilation.

Though generating source code via string concatenation can be readable and easy to understand in simple scenarios, in our context, it would result in confusing and hard-to-maintain implementation of the source generator. We need to generate relatively complex code that is heavily parameterized based on user-implemented input. Roslyn API allows for a more modular design of the source generator and results in an implementation that is easier to maintain. We will, therefore, opt to generate source code in the form of Roslyn AST rather than concatenating strings.

2.9.2 Method overloading

As Section 2.5.4 described, to invoke Java methods, a generated proxy class will need to hold the JNI method IDs of these methods. These IDs are not expensive resources and are valid while the corresponding Java class is loaded. Therefore, the proxy class can cache them in static fields, as demonstrated in Code Snippet 2.36.

Code Snippet 2.36: C#: Static field caching JNI method ID in generated C# proxy class

```
1 private static readonly Lazy<JniStaticMethod> jnimyMethod = new
    (() => _javaType!.GetStaticMethodID("myMethod",
    "(ILmy/classes/MyClass;)V"));
2 public static partial void MyMethod(int number,  MyClass myClass)
    { ... }
```

In order to produce a valid C# code, the source generator must produce a unique name for every method ID field in a given class. Code Snippet 2.36 demonstrates using the name of a related Java method in the method ID field name. This approach, however, is not general enough: Java methods can be overloaded, allowing for multiple methods with the same name defined in the given type.

Solution – encoding parameter types
We can address this problem by encoding method parameter types into method field names. We can use the name mangling strategy similar to one that JNI uses to specify signatures of methods to be looked up. This strategy has first been mentioned in Section 1.2.1 and is also demonstrated in Code Snippet 2.36. The second parameter of `GetStaticMethodID` method is a string representation `myMethod` method signature. Each Java primitive type is represented by a dedicated letter; for reference types, the full name is encoded. Considering this, we can source generate method handle field related `MyMethod` e.g., called `jnimyMethod_I1Lmy_classes_yClass1` (replacing characters that are not allowed in identifiers).

This approach works well in common cases, but unfortunately, it is not a completely general solution. According to a specification [134] Java methods can accept up to 255 arguments. We did not find any formally specified limitation on the length of the C# identifier. We have experimentally found out that when a C# solution is being built from Visual Studio, there is either no limit on the identifier length or the limit is so high that we did not manage to reach it, and then it is irrelevant in our context. When a C# solution is being built in the command line via `dotnet build` command, however, the limit of the identifier length seems to be 1023 bytes in UTF-8 encoding. Java method accepting a lot of non-primitive parameters can exceed this limit when its argument types are encoded into a field name.

Solution – counter
We can try to achieve uniqueness in field names differently – by appending a value of a counter to a method name to form a unique field name. The counter, however, cannot be global, as an incremental source generator pipeline should be stateless and as we cannot be sure that all methods defined in one proxy type will be processed by the same invocation of the source generator `transform` method (see Section 3.5 for an explanation of the incremental source generator implementations). The proxy class is partial; the user can potentially separate the proxy class definition into multiple parts, and then each of these parts would get processed separately. We, therefore, cannot use a local per-proxy-class counter.

Solution – GUID
Instead of appending a counter value, we can append a generated GUID to a Java method name to produce unique field name. The disadvantage of this approach is that it makes source generation non-deterministic, which is not recommended as it interferes with incremental source generator caching.

In common cases, we can use the approach of encoding method parameter types into a field name when the resulting field name doesn't overflow the length limit of 1023 bytes. If the length limit should be exceeded, GUID can be used instead. As Java methods accepting a large number of arguments that the field name limit should be exceeded are going to be rare, it does not seem necessary to optimize for them. We only need to make sure that the code produced by the source generator is valid, even in these corner cases.

2.9.3 Incremental source generator caching

Incremental source generators extensively use caching, which is useful because, especially in IDEs, they can be rerun often. Incremental source generators are implemented as stateless pipelines (see Section 3.5 for more details), and outputs of each pipeline stage are cached. If some pipeline stage produced the same outputs in two consecutive runs of an incremental source generator, later stages of the pipeline can be skipped in the second of the runs [122]. This provides incremental source generators with a significant performance advantage over earlier introduced source generators.

However, in order for an incremental source generator to be able to detect that the stage produced the same output as the last time it was run, the incremental source generator has to be able to **compare** data transfer objects (DTOs) returned by individual pipeline stages **by value** [135].

Solution: EqualityComparer

Comparison of pipeline stages results by value can be achieved by implementing `EqualityComparer`s [136] for DTOs types returned by pipeline stages. This `EqualityComparer`s must be set to particular pipeline stages via `WithComparer` method [137].

This approach, however, can be error-prone as it creates separate comparison semantics for DTOs that are only used in the source generator pipeline. Moreover, implementation of `EqualityComparer` for more complex DTOs is rather cumbersome.

Solution: Records

Another option is to implement DTOs in a way that they are by themselves comparable by value. To achieve this, we can use **record types** [138] as they are comparable by value by default. We just need to make sure that our records do not contain fields of any complex types that are comparable by reference. For example, we cannot pass Roslyn **syntax nodes** or **type symbols** between pipeline stages as they are very likely to stand in the way of caching. Instead, we need to extract data that we need in later stages of the pipeline into fields of simpler types.

Another issue comes with having fields of **collection types** in our DTOs because collections are, by default, comparable by reference. Luckily, Andrew Lock comes up with the solution in his blog post [135], where he suggests implementing an array wrapper that implements value comparison semantics [139]. In this blog series, he also shows how to test that caching actually hits between incremental source generator pipeline stages [140] when the approach of using DTOs that are inherently comparable by value is used. We can adopt this approach in source generator unit tests.

As the approach of making DTOs inherently comparable by value seems less error prone and easier to maintain, we will opt for it.

3. Implementation documentation

The previous chapter went through a set of decisions that had to be made while designing software implemented in this thesis. This chapter will describe the main aspects of the software architecture that was created as the result of these decisions.

The resulting solution enables interoperability invocations from C# to Java and manipulation with Java objects from a C# code. It focuses on making these interoperability interactions as seamless as possible from the user's perspective through extensive usage of the incremental source generator. Figure 3.1 shows the high-level view of components that play role in the interoperability interaction. Let's describe them in more detail.

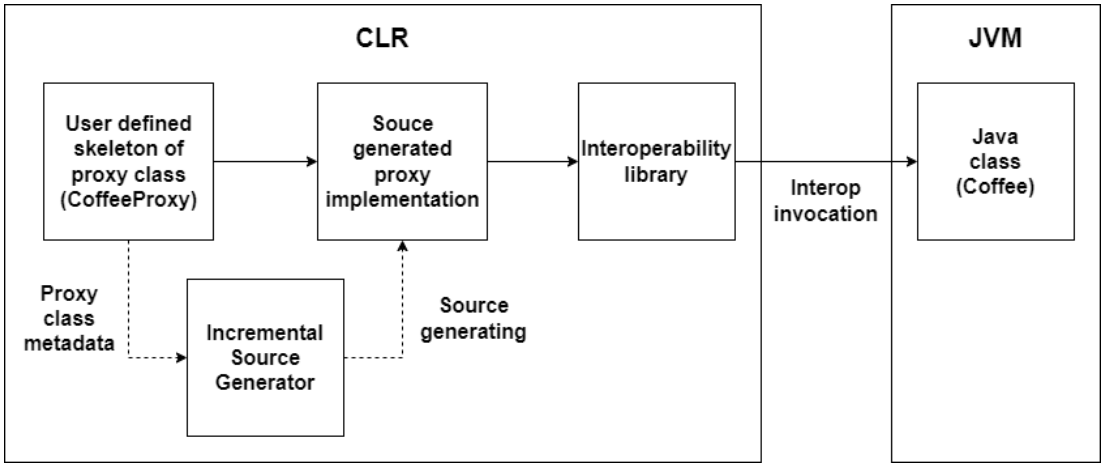

Figure 3.1: High-level view of components playing a role in interoperability invocations

Suppose that a user wants to access `Coffee` **Java class** shown in Code Snippet 3.1 from C#. User will define a **skeleton of a C# proxy class** (for example, as shown in Code Snippet 3.2) – left side of Figure 3.1. **Incremental source generator** will locate the proxy class skeleton by applying **marker attributes** (`JavaClassProxy`, `JavaImport` in Code Snippet 3.2), it will analyze it using Roslyn API and will **generate its implementation**. This implementation will use the API provided by the **interoperability library** to carry out interoperability invocations to Java-side code.

Code Snippet 3.1: Java: Example of Java class

```
1 public class Coffee {
2     public static Coffee brew(int gramsOfCoffee) { ... }
3 }
```

Code Snippet 3.2: C#: Example of proxy class

```
[JavaClassProxy]
public static unsafe partial class CoffeeProxy {
    [JavaImport]
    public static partial CoffeeProxy  Brew(int gramsOfCoffee);
}
```

For more details about the usage see Chapter 4.

3.1 Solution structure

The described behavior is implemented as a .NET solution (`.sln`) consisting of 13 .NET projects (`.csproj`), including two testing projects and a benchmarking project. The solution depends on a small amount of Java code enabling usage of Project Panama and providing Java classes the solution can be tested on.

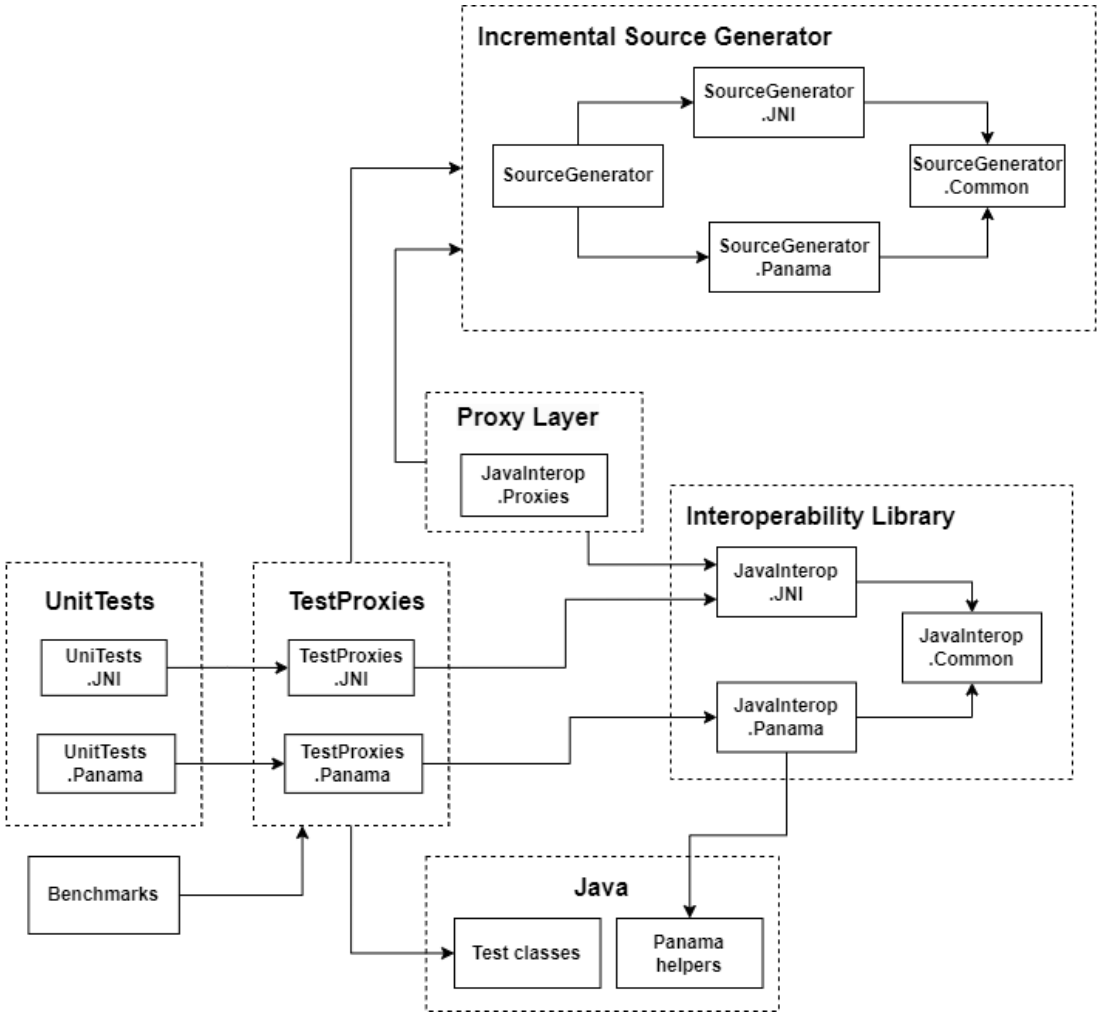

Figure 3.2: Overall solution structure

Figure 3.2 captures the high-level structure of the solution (on the level of individual projects).

The core of the solution is **Interoperability library** that consists of three projects:

- `JavaInterop.JNI` – contains JNI-based interoperability logic designed in Sections 2.5 and 2.6. This project handles invocations of Java methods and manipulation with Java objects via JNI API.
- `JavaInterop.Panama` – enables invocations of static Java methods with primitive parameter types using Project Panama, providing opt-in optimization for these invocations over JNI-based solution. This project depends on a small amount of Java code – `Panama Helpers` – that enables the C#-side part of the library to use features of Java-based Project Panama. Section 2.4 focused on designing this part of the solution.
- `JavaInterop.Common` – contains a bit of shared code, mainly **Marker attributes** used by an incremental source generator (see Section 2.7).

The API provided by the interoperability library is used by the **Incremental source generator**. Incremental source generated composes this API into source generated C# proxies of Java classes. Source generation is separated into two phases:

1. **Scanning a user-written code** and collecting metadata about proxy classes that should be generated. This phase is handled by `JavaImport.SourceGenerator` project.
2. **Source generating C# code** based on collected metadata. This phase is handled by `JavaInterop.SourceGenerator.JNI` for JNI-based proxy classes and by `JavaInterop.SourceGenerator.Panama` for proxies using Project Panama.

An interface between these two phases is provided by `JavaInterop.SourceGenerator.Common` project that defines a set of data transfer objects (DTOs) used to pass proxy classes metadata from the scanning phase to the generating phase. This project also contains a bit of shared source-generating logic. An incremental source generator was designed in Section 2.9.

`JavaInterop.Proxies` project contains predefined proxies of a few commonly used Java classes intended to be used by a user. The following sections will describe these components in more detail.

Other projects shown in Figure 3.2 serve for testing and benchmarking purposes. Section 3.6 looks at testing in more detail.

As the implemented solution is a functional prototype of production-ready software, some `TODO`s were intentionally left in the code to mark places of possible future improvements. Section 5.2 analyses these possible improvements.

3.2 Interoperability library – JNI

The first component to describe is the JNI-based part of the interoperability library implemented by `JavaInterop.JNI` project.

This component consist of several layers of abstractions. This section will describe them starting in JNI-based core of the library and progressing up to user-level API.

3.2.1 JavaVM and JniEnv

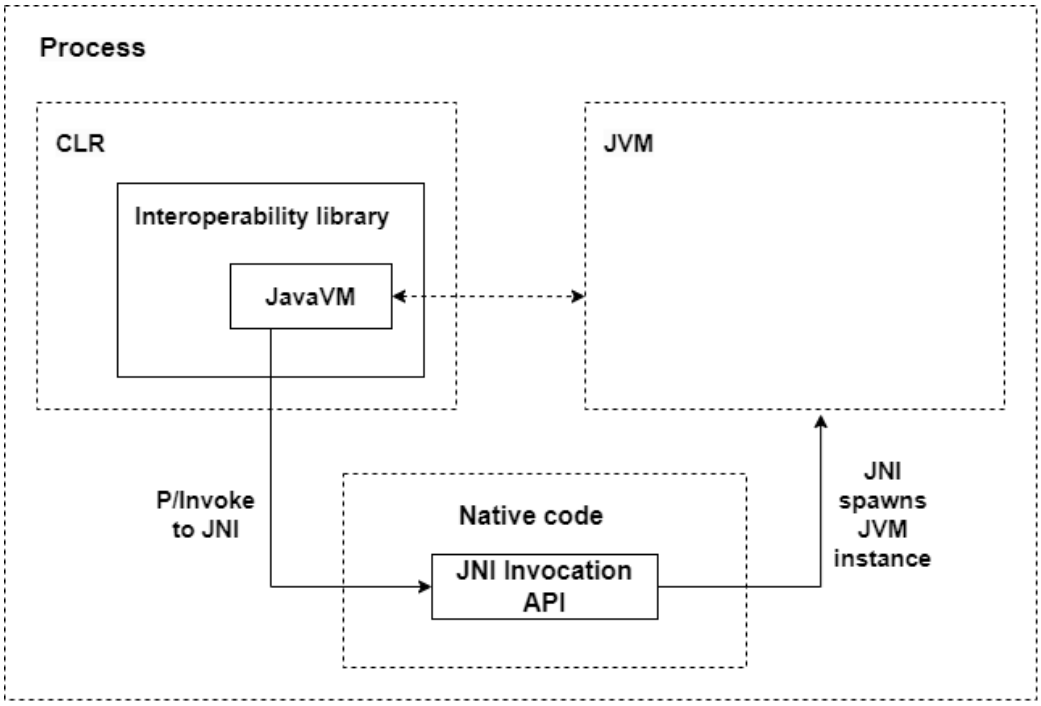

Figure 3.3: Using JNI invocation API to spawn JVM instance

Before any interoperability invocation from C# to Java can occur, a JVM instance must be spawned in the process where a .NET application is running. JVM instance can be spawned via **JNI Invocation API** [8]. Figure 3.3 captures steps that our interoperability library carries out in order to spawn a JVM instance.

Interoperability library represents JVM instance via `JavaVM` class. `JavaVM` class uses P/Invoke (see Section 1.2.2) to access JNI Invocation API (which is native-code API). When `CreateJavaVM` function from JNI Invocation API [8] is called, the JVM instance spawned in the current process. Once the JVM instance is created, it can be accessed via `JavaVM` class. As at most one JVM instance is allowed per the lifetime of the process (see Section 2.5.2), `JavaVM` is static.

The majority of JNI API is provided in the form of functions in **JNI Function Table**. This table is accessible via the JNI interface pointer, commonly referred to as `JNIEnv` pointer (see Section 1.2.1). `JNIEnv` pointer is only valid in a thread that obtained it. The thread that spawns the JVM instance will obtain `JNIEnv` pointer returned from `CreateJavaVM` function. Other threads have to attach themselves to the JVM instance using JNI Invocation API to obtain `JNIEnv` pointer.

Within our C# interoperability library, `JavaVM` class is responsible for attaching threads to the JVM instance it manages. It contains `ThreadLocal` instance of `JNIEnv` class. `JNIEnv` class is C# representation of `JNIEnv` pointer. Its `ThreadLocal` instance gets initialized by `CreateJavaVM` call in the thread that spawned JVM and by `AttachCurrentThread` call in other threads.

Once a thread holds its `JNIEnv` instance, it can be used to invoke Java methods

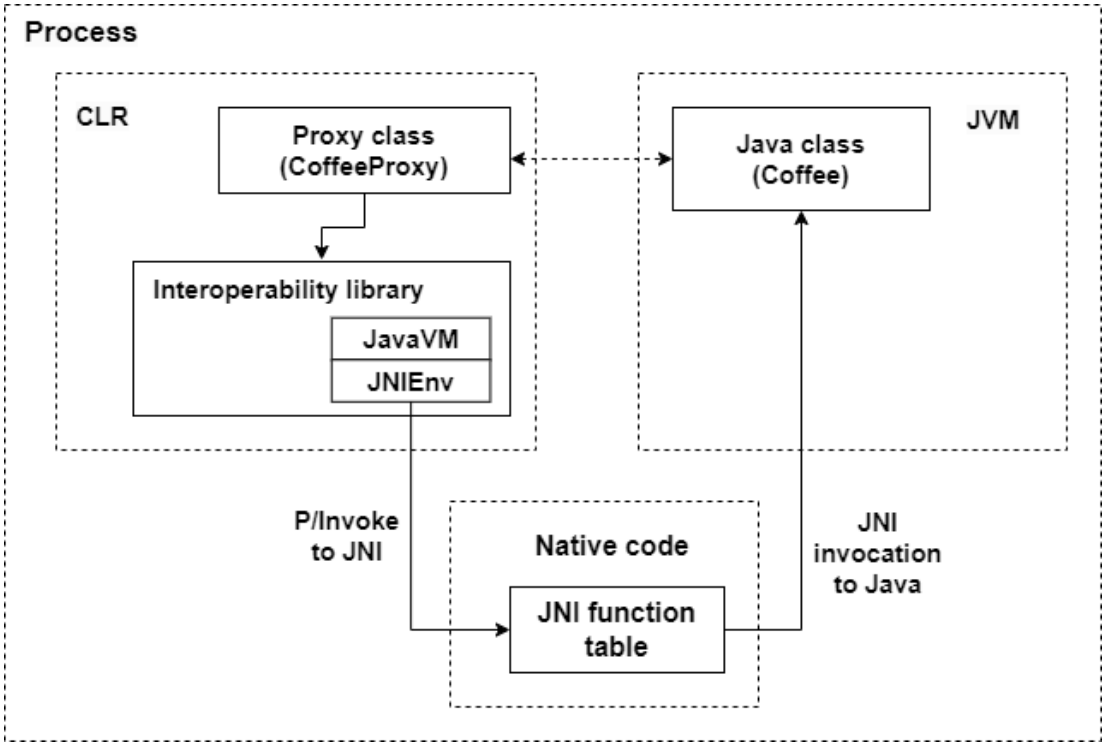

Figure 3.4: Invocation of Java code using JNI

via JNI. This process is captured by Figure 3.4. First, P/Invoke is used to invoke a function in the JNI Function Table. The invoked JNI function then calls the required Java method (or accesses an instance of a Java object).

As Figure 3.5 shows, `JNIEnv` class accesses the JNI Function Table via a set of P/Invoke-based `unmanaged delegates`. It also contains an additional layer of abstractions in the form of wrapper methods that invoke unmanaged delegates and abstract other parts of the library from low-level details related to JNI interoperability invocations. For example, they handle exceptions that occur in Java-side code and propagate them as C# exceptions; they also manage low-level details of passing strings and arrays to the native JNI code (pinning, stack-allocating).

3.2.2 Safe handle layer – typing JNI references

JNI functions access Java objects via JNI references. On the type system level, JNI distinguishes multiple types of references depending on the type of object the reference is pointing to. The full list can be seen in the first column of Table 3.1[1].

Apart from these reference types, JNI also distinguishes two orthogonal kinds of references (see Section 1.2.1 for more details):

- **local references** that:
  - are only valid in a thread that created (obtained) them,
  - are only valid until the control flow returns to Java,
  - do not require explicit cleanup.

[1]JNI defines all these types as `typedef` aliases of the same pointer type.

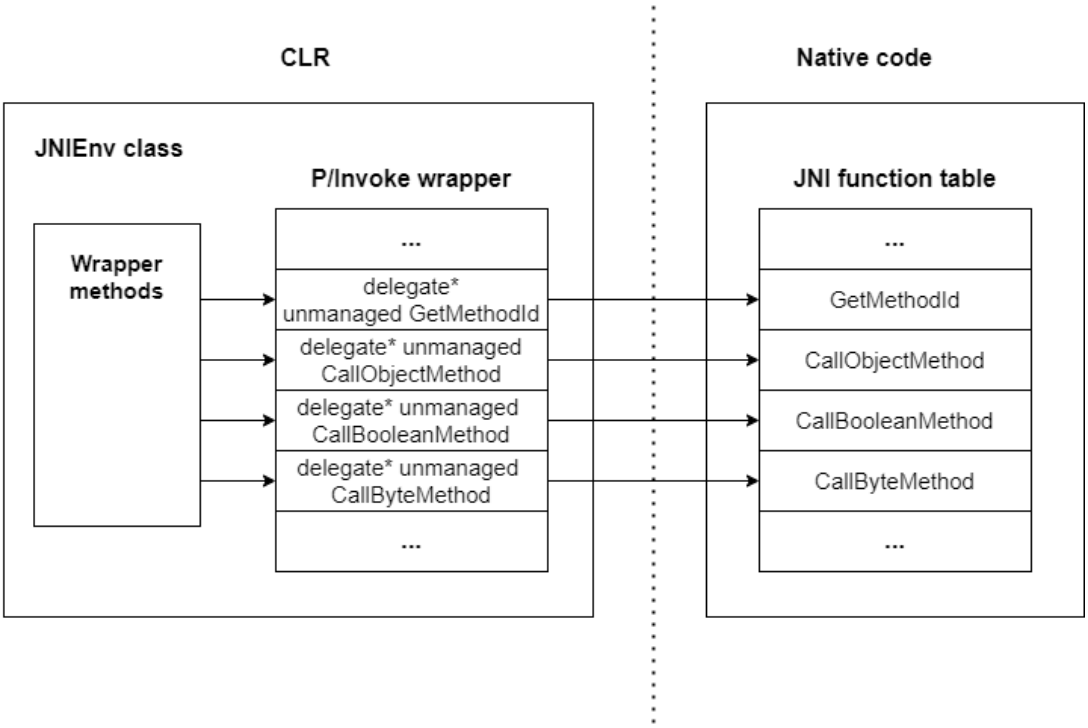

Figure 3.5: Layer of abstraction wrapping JNI Function Table

- **global references** that:
  - are valid in any thread,
  - are valid until they are explicitly freed,
  - requires explicit cleanup.

JNI, however, does not distinguish between local and global references on the type system level. Any of the types from the first column of Table 3.1 can represent either local or global reference. That proves to be tricky, as some JNI functions will crash if an incorrect reference kind is passed.

Our library aims at typing JNI references more safely. We have a certain freedom when it comes to typing JNI references. We can specify signatures of unmanaged delegates invoking JNI functions as we please as long as the P/Invoke marshaller is able to marshall between JNI types and types we provided (see more in Section 2.5.3). The rest of this section will talk about local and global references in more detail.

Local references

As Section 2.5.3 decided, JNI local references are represented by sequential layout `structs` wrapping `IntPtr` (example is shown in Code Snippet 3.3).

Code Snippet 3.3: C#: Representation og JNI local reference

```
1 [StructLayout(LayoutKind.Sequential), NativeCppClass]
2 public struct JniObjectLocalRef {
3     internal IntPtr _handle;
4 }
```

JNI type	C# local reference	C# global reference
jobject	JniObjectLocalRef	JniObjectSafeHandle
jclass	JniClassLocalRef	JniClassSafeHandle
jthrowable	JniThrowableLocalRef	JniThrowableSafeHandle
jstring	JniStringLocalRef	JniStringSafeHandle
jarray		JniArraySafeHandle
jbooleanArray	JniArrayLocalRef<bool>	JniPrimitiveArraySafeHandle<bool>
jbyteArray	JniArrayLocalRef<sbyte>	JniPrimitiveArraySafeHandle<sbyte>
jcharArray	JniArrayLocalRef<char>	JniPrimitiveArraySafeHandle<char>
jshortArray	JniArrayLocalRef<short>	JniPrimitiveArraySafeHandle<short>
jintArray	JniArrayLocalRef<int>	JniPrimitiveArraySafeHandle<int>
jlongArray	JniArrayLocalRef<long>	JniPrimitiveArraySafeHandle<long>
jfloatArray	JniArrayLocalRef<float>	JniPrimitiveArraySafeHandle<float>
jdoubleArray	JniArrayLocalRef<double>	JniPrimitiveArraySafeHandle<double>
jobjectArray	JniObjectArrrayLocalRef	JniObjectArraySafeHandle

Table 3.1: Mapping between JNI reference types and interoperability library types

Several similar struct were implemented to represent individual types of JNI references. The second column of Table 3.1 shows the full list. The table also captures the mapping between these types and JNI types in the first column.

All JNI functions apart from `NewGlobalRef` function return local reference. For some of these functions, it is clear what type of local reference is returned – e.g., it is clear that `NewIntArray` returns a reference to an int array (`jintArray`). The type returned from some other JNI functions, however, is less clear. For instance, `CallObjectMethod` JNI function is used to invoke arbitrary non-static Java method returning arbitrary non-primitive type – returned type can be string, array, user-defined class, etc. As we represent local references by structs and structs do not allow for polymorphism, we introduce a separate type of local reference – `JniLocalRef` – to represent these undetermined local references. Section 3.2.5 will talk about typing in these cases.

Global references

To represent global references, we use instances of classes inherited from `SafeHandle` class [117]. `SafeHadle` is a standard library class intended to wrap unmanaged resources. It manages the clean up of these resources. It is integrated into P/Invoke – P/Invoke is able to marshal between native pointers and C# `SafeHandle`s.

We implement several classes inherited from `SafeHandle` class to represent individual types of JNI references. The third column of Table 3.1 lists these types. Figure 3.6 captures the inheritance hierarchy between them. Polymorphism presents an advantage for us because it allows us to pass more specialized types to unmanaged delegates that expect `JniObjectSafeHandle`. That, for example, allows us to invoke instance methods on string instances.

Clean up of `JniObjectSafeHandle` and inherited classes invokes `DeleteGlobalRef` JNI function. This function will crash if a local reference gets passed to it. To make sure that a safe handle instance wrapping local

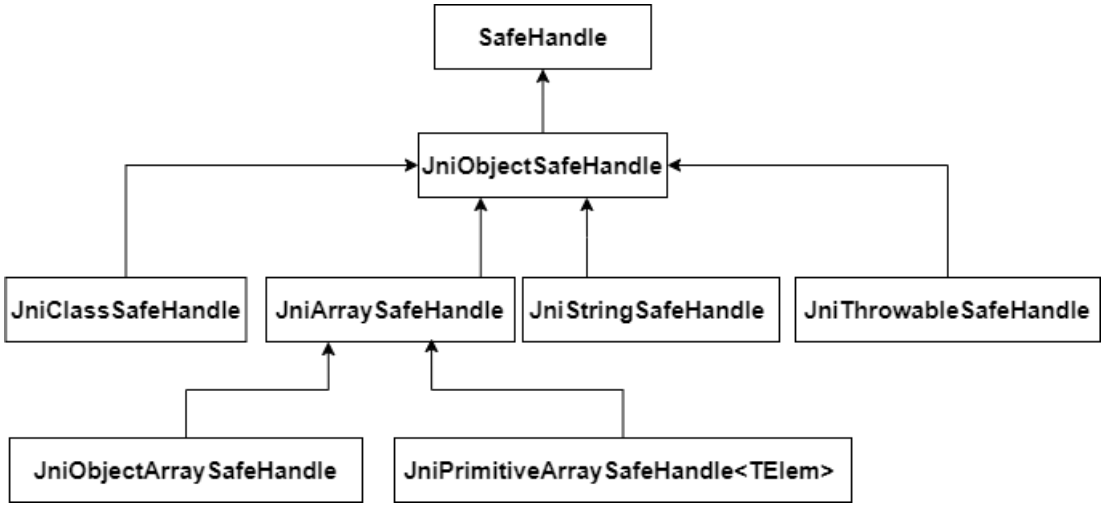

Figure 3.6: Safe handle inheritance hierarchy

reference never gets created (as cleanup of such safe handle would crash), we obtain a global reference from (local) reference directly in the constructor of `JniObjectSafeHandle` class.

3.2.3 JavaObject layer – abstracting implementation details

All proxies of Java classes a user gets to interact with under the hood wrap JNI global references, represented by safe handles. The code of proxy classes is generated by an incremental source generator and gets added to the user's assembly. It can, therefore, only access public methods and types of the interoperability library. To avoid leaking implementation details of our library, to avoid allowing users to manipulate with safe handles directly, and to keep safe handles implementation as slim as possible (safe handles only represent a reference, they do not implement any additional functionality), we have wrapped safe handles into one additional layer of an abstraction – `JavaObject` layer.

Figure 3.7 shows all the layers of abstraction implemented inside the interoperability library. In the core of the interoperability library is `JNIEnv` class that manages JNI invocations via strongly typed unmanaged delegates. These delegates work with JNI global references as with safe handles. Safe handles are further wrapped to another layer of abstraction to hide implementation details from a user – **`JavaObject` layer**. Users will interact with source-generated proxy classes on **Proxy Layer** that use API provided by **`JavaObject` layer** to emulate the API of Java classes they access.

There are two classes on `JavaObject` layer that deserve closer attention:

- `JavaObject` class represents a notion of a Java object. It wraps `JniObjectSafeHandle`. It enables invoking instance methods and accessing instance fields of the Java instance it represents.
- `JavaType` class represents a notion of Java type. It wraps `JniClassSafeHandle`. It allows the invocation of static Java methods and

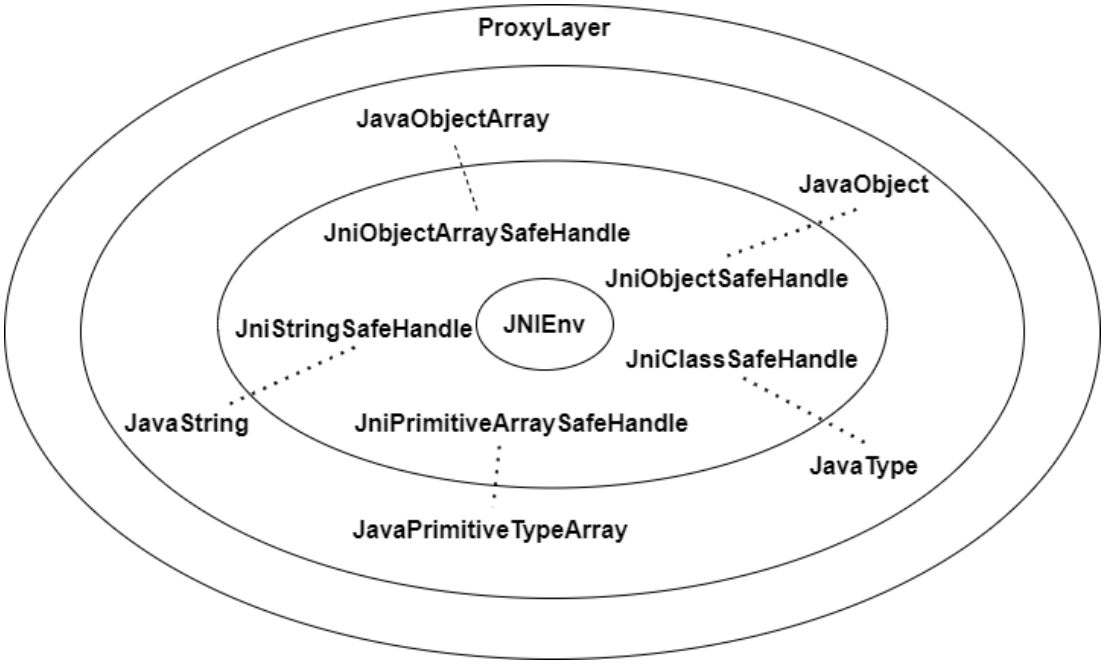

Figure 3.7: Layers of abstraction of interoperability library

accessing static Java fields. It also allows to look up both static and non-static Java methods and fields that are to be used from C# (before the method is to be called via JNI, it has to be looked up - read more in Section 1.2.1).

3.2.4 Proxy layer – providing pleasant user experience

The last layer of abstractions shown in Figure 3.7 is the **proxy layer**. This layer provides API that users of our library should interact with. It consists of individual proxy classes (such as `CoffeeProxy` from Code Snippet 3.2) that emulate the API of existing Java classes and enable a user to access it from C#.

Each C# proxy type corresponds to exactly one existing Java type. That distinguishes proxies from `JavaObject` class that can represent a reference to an instance of any Java class and holds no type information about the Java type it references. On the other hand **the type of proxy class determines the type of Java class it emulates**[2].

Figure 3.8 captures the relation between `JavaObject`, `JavaType`, C# proxy instances, and Java instances. Each C# proxy instance contains a field of `JavaObject` type that references the Java instance the proxy instance works with. The proxy class then contains a static field of `JavaType` type that references Java type (`java.lang.Type` instance) the proxy type emulates.

The majority of proxy classes will be source-generated based on user-written code. Section 3.4 will be devoted to source-generated proxies. There are, however, a few proxies implemented manually in the interoperability library:

[2]The relation is not necessarily bijection between C# proxies and Java classes that are to be accessed from C# as a user can define multiple C# proxy types referencing the same Java type if they so please. All instances of a particular Java type should, however, reference instances of the same Java type (or possibly types inherited from this Java type).

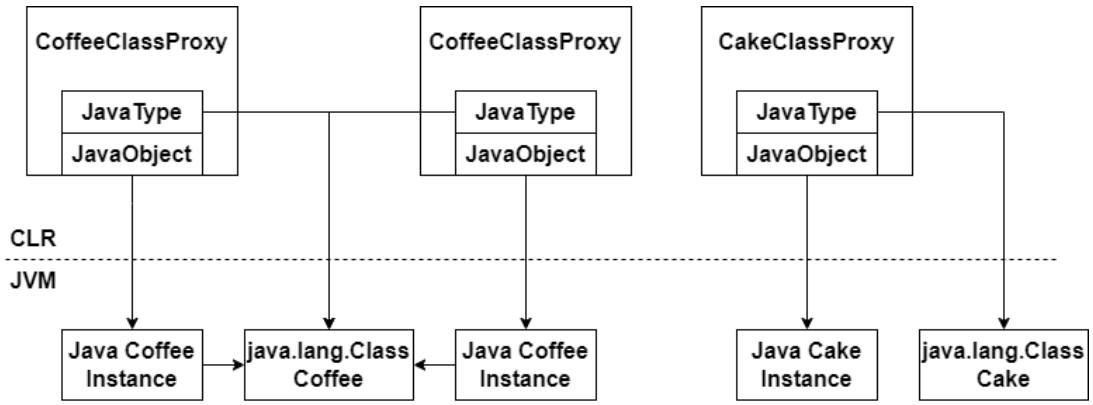

Figure 3.8: Relation between `JavaObject`, `JavaType`, C# proxy instances and Java instances

- `JavaLangObjectProxy` – represents `java.lang.Object` type, which is the root of Java inheritance hierarchy. Each (user-defined or in-library implemented) C# proxy of a Java class (directly or indirectly) inherits `JavaLangObjectProxy` class.
- `JavaLangStringProxy` – represents `java.lang.String` type and implements some string-specific functionality, e.g., conversion between Java strings and C# string.
- `JavaPrimitiveArrayProxy` – represents a reference to a Java primitive type array. Accepts a generic type parameter, specifying a type of an array elements. The source generator emits diagnostics to ensure that only supported primitive types can be specified (see Section 3.5.2) [3]. This class does not hold array elements (therefore, it does not implement an indexer). It enables access to blocks of an array via `GetCopyOfRange`/`SetRange` methods or to obtain access to all array elements – returned as `IDisposable JavaPrimitiveArrayElements` instance implementing an indexer. Section 2.6 justifies related design decisions.
- `JavaObjectArrayProxy` – represents a Java array of Java objects. Accepts generic type parameter specifying a type of an array an elements, which has to be a Java class proxy type. Supports multidimensional arrays by passing `JavaObjectArrayProxy` as the type parameter. That imposes a complication for the source generator because it has to be able to handle types that are parameterized by types. Unlike for primitive type arrays, underlying JNI API for object arrays is relatively straightforward, allowing the implementation of an indexer directly on `JavaObjectArrayProxy` type.

3.2.5 Creating proxy instances

A user of our interoperability tool will work with Java object instances via instances of generated C# proxy classes. Therefore, when a Java method returning

[3] C# itself does not allow to specify granular enough generic constraint. `unmanaged` [141] constrain is close, but not all primitive types it includes can be mapped to existing Java primitive types (see Section 2.1.1).

an object is called via an invocation of the corresponding C# proxy method, it is necessary to create an instance of the correct C# proxy class to return to a user. Section 2.8 analyzed some aspects that make creating proxy instances tricky. This Section will provide a high-level description of the solution implemented based on this analysis.

The solution is explained using the example of `makeCoffee` method (see Code Snippet 3.4) returning an instance of `Coffee` class. Figure 3.9 captures the whole process of invoking a method that returns a proxy instance.

Code Snippet 3.4: Java: Example of Java class

```
1 public class CoffeeMachine {
2         public static Coffee makeCoffee() { ... }
3 }
```

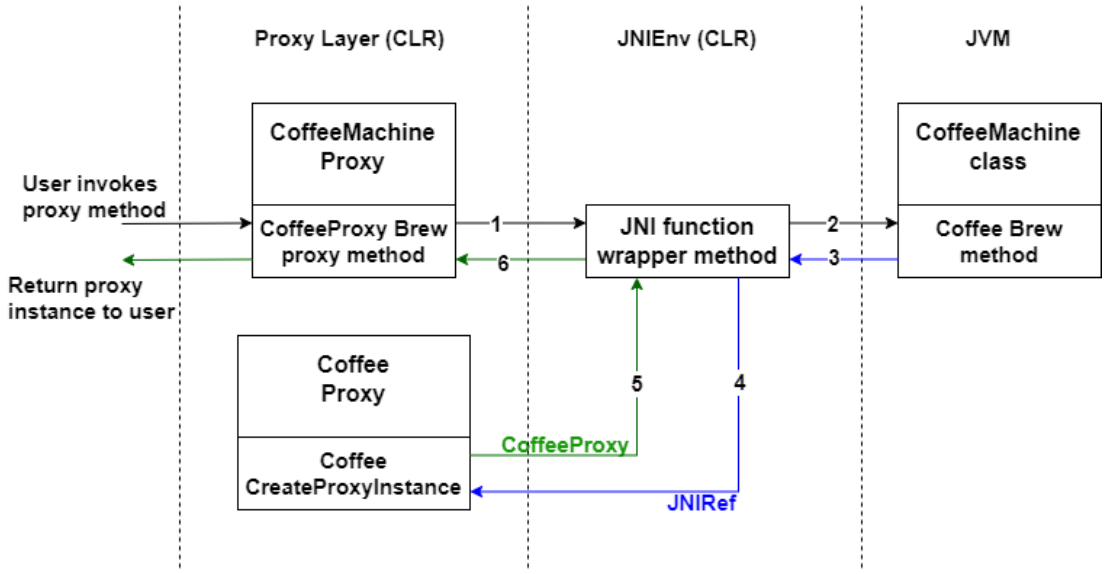

Figure 3.9: Returning an instance of a proxy class

To access the Java `makeCoffee` method, a user has to declare a partial proxy method (similar to the method shown in Code Snippet 3.2). Source-generated implementation of this proxy method uses `JNIEnv` class (step 1 in Figure 3.9) to invoke the Java method via JNI (step 2). JNI function returns `Coffee` class instances as JNI reference (step 3). This JNI reference needs to be wrapped in the correct class proxy instance.

Each proxy type implements `IJavaProxy<TProxy>` interface that requires static `CreateProxyInstance` method. Proxy classes implement this method to create their own instance from a given JNI reference. That moves the responsibility for the proxy instance creation to proxy classes themselves. For source-generated proxy classes, the implementation of `CreateProxyInstance` method can be source-generated, and in-library proxy classes can implement a custom logic if required.

Wrapper methods (see Figure 3.5 for an explanation) of JNI functions returning Java object instances are generic by a C# proxy type corresponding to the returned Java type. They can, therefore, invoke `CreateProxyInstance` method

of a particular proxy type (step 4 in Figure 3.9) as static interface methods can be invoked on generic type parameters. Instances created by `CreateProxyInstance` method can be returned to a user.

Interfaces

Our solution also supports invocations of Java methods returning interface types. As Section 2.8.2 explained, interface return type further complicates the situation as it is not immediately clear an instance of which C# proxy classes should be used to wrap a returned object reference.

Our solution represents Java interfaces by C# proxy interfaces. When a Java method returning an interface is to be invoked, a type specified as a type parameter of the `JNIEnv` wrapper method will be an interface proxy type (`ICoffee` type in the example in Figure 3.10). As Section 2.8.2 decided, interface proxy types also implement static `CreateProxyInstance` method. That gets invoked from the `JNIEnv` wrapper method (step 4 in Figure 3.10, previous steps are the same as in the previous case). Source-generated implementation of this method uses `InterfaceProxyFactory` to create an instance of a correct proxy type (step 5).

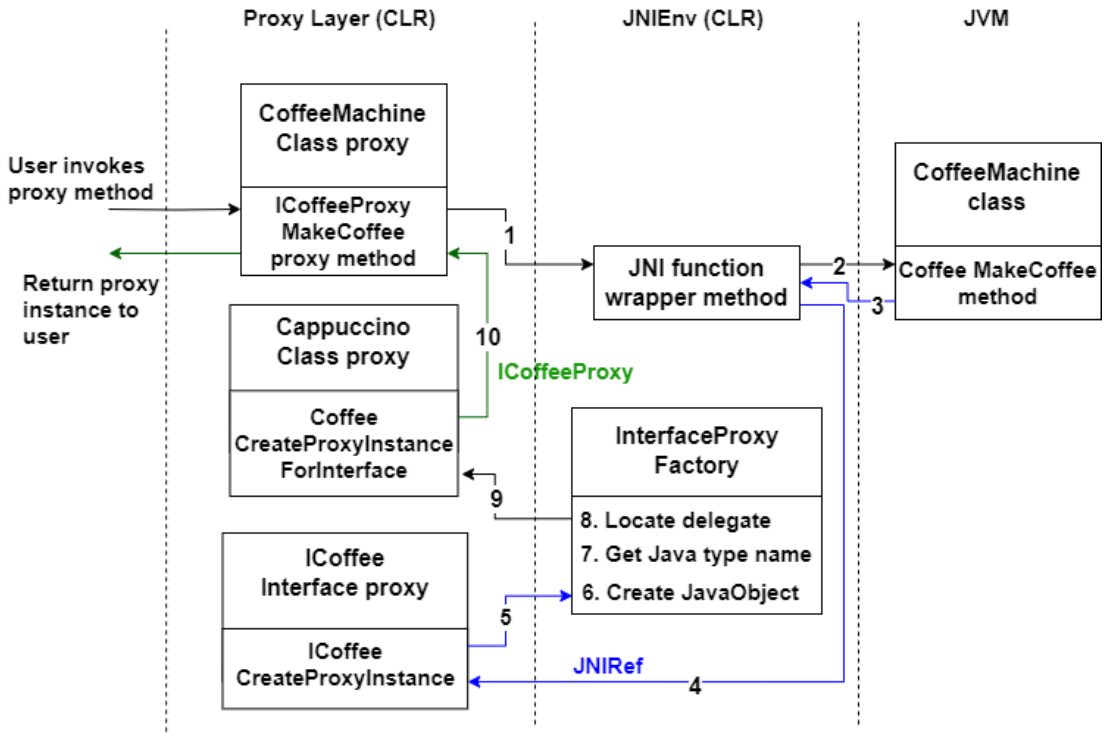

Figure 3.10: Returning Java instance wrapped in an interface type

Source-generated interface proxy contains a dictionary (such as the one shown in Code Snippet 3.5) that maps Java type names to C# delegates to `CreateProxyInstanceForInterface` methods defined by corresponding proxy classes. This dictionary contains one record for each C# proxy class defined in the given assembly that implements the proxy interface (in our example, suppose that Java classes `Cappuccino` and `Espresso` implement Java `Coffee` interface and that C# assembly contains proxies of these Java classes that implement `ICoffee` proxy interface). `CreateProxyInstanceForInterface` method serves the same purpose

as `CreateProxyInstance` method and is also required by `IJavaProxyInterface`. The difference between methods will be explained later in this section.

Code Snippet 3.5: C#: Source generated dictionary that maps Java type names to delegates creating instances of corresponding C# proxy classes – example for ICoffee interface

```
1  private static readonly Dictionary<string, ProxyCreator>
       _interfaceImplementations = new() {
2      {
3          "javasources.testclasses.Cappuccino",
4          new ProxyCreator(
5              CappuccinoProxy.CreateProxyInstanceForInterface)
6      },
7      {
8          "javasources.testclasses.Espresso",
9          new ProxyCreator(
10             EspressoProxy.CreateProxyInstanceForInterface)
11     },
12 };
```

`InterfaceProxyFactory` needs to find out the Java name of a runtime type of JNI reference it obtains (step 7) to look up a correct `CreateProxyInstanceForInterface` delegate in the dictionary (step 8). To find out the Java type name, it first needs to create `JavaObject` instance from the JNI reference (step 6). Once `InterfaceProxyFactory` holds the correct delegate, it can invoke the corresponding `CreateProxyInstanceForInterface` method (step 9) that will create an instance of a proxy class that can be then returned to a user.

Unlike `CreateProxyInstance`, `CreateProxyInstanceForInterface` method accepts `JavaObject` as its parameter. Therefore, `JavaObject` instance created to figure out the Java type name can be passed to it, and no redundant instance needs to be created. On the other hand, `CreateProxyInstance` method accepts JNI reference directly, and the method implementation is free to wrap it in an instance of a more specific type inherited from `JavaObject` if necessary.

3.3 Interoperability library – Panama

As section 2.3.4 explained, Project Panama is a brand new feature in Java that could potentially replace JNI for interoperability invocations between Java and a native code. The current version of Project Panama does not provide support for working with Java objects. We managed, however, to use Project Panama to enable invocations of static Java methods with primitive parameter and return types from C#. As Section 2.4.3 shows, the performance of the solution using Project Panama is superior to the one of the purely JNI-based solution. Project Panama can, therefore, be used as an opt-in optimization for this specific kind of invocation when a new enough version of Java is used. This section explains how.

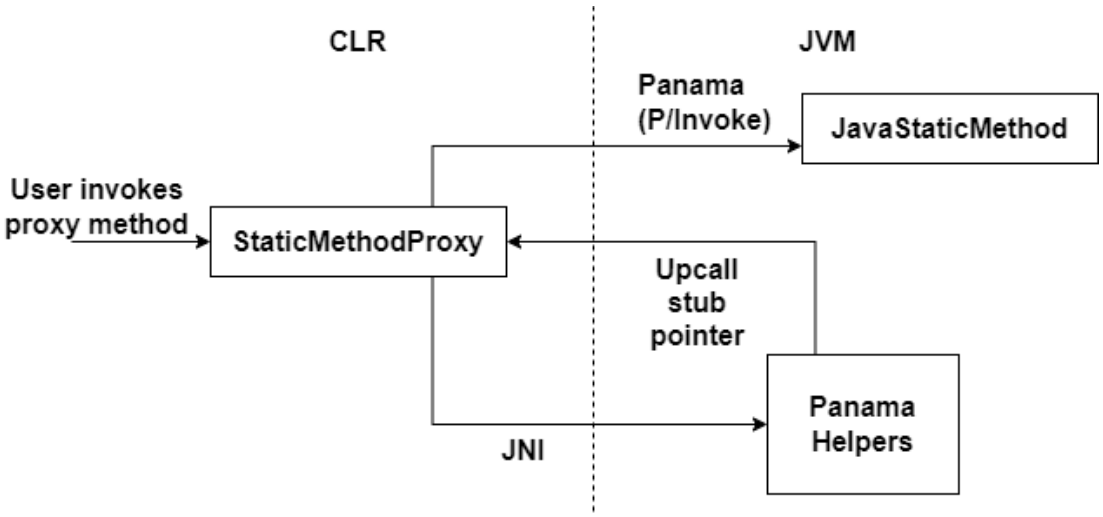

Figure 3.11: The first invocation of a particular Java method using Project Panama

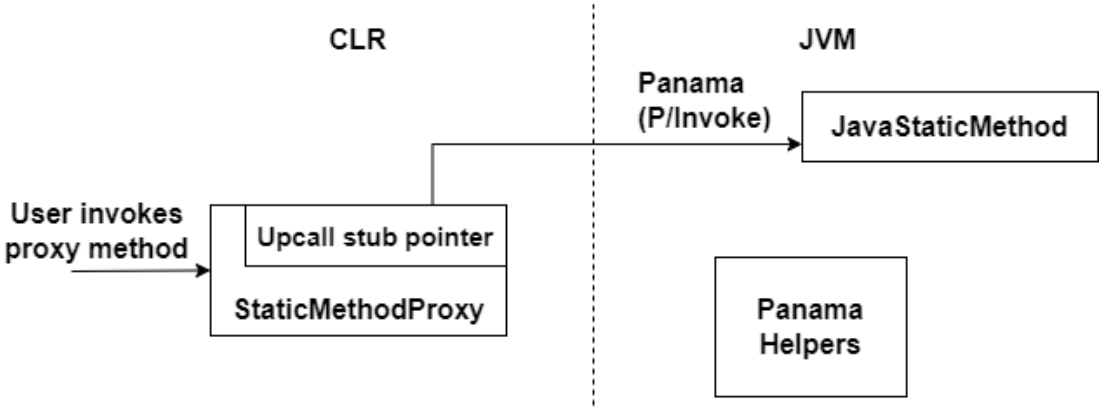

Figure 3.12: The second and any other consecutive invocation of a particular Java method using Project Panama

Figure 3.11 demonstrates the process of the first invocation of a particular Java method from C# using Project Panama. To be able to invoke the Java method via Project Panama, we first need to obtain a pointer to an **upcall stub** of this method. As Project Panama is only available as Java API, an upcall stub can only be built from Java code. We, therefore, need to bootstrap the process via JNI invocation[4] of Java method that builds an upcall stub and returns a pointer to it (number 1 and 2 in Figure 3.11).

Once a C# code holds un upcall stub pointer, P/Invoke can be used to invoke this function pointer, effectively invoking the Java method the upcall stub belongs to (number 3 in Figure 3.11).

An upcall stub pointer can be cached on the C# side. Any consecutive invocation of the given Java method can, therefore, avoid the JNI bootstrapping step and directly invoke the upcall stub pointer via P/Invoke (as Figure 3.12 shows).

[4]As the Java method that creates an upcall stub accepts non-primitive (string) parameters, Panama cannot be used to invoke it.

3.4 Generated Proxies

This section will describe significant aspects of different kinds of source-generated proxies. It will build on what was explained in previous sections of this chapter, providing an overview of how the concepts introduced are composed into generated proxy classes.

3.4.1 Static class proxy

First, let's introduce a more straightforward case of **static proxy classes**. Even though Java does not support non-nested static classes, we allow a user to mark their C# proxy class of (non-static) Java class as static as far as it only accesses static members of the Java class. That allows for simpler proxy classes in this simpler scenario.

As Sections 3.3 mentioned, our solution allows for the invocation of static methods with primitive parameter and return types either via JNI or via Project Panama. This section will first describe the JNI-based proxy; then it will look at the Panama-based proxy. Both kinds of proxies will be introduced using the example of the Java class shown in Code Snippet 3.6. Real project would usually contain only one of these proxies, depending on if a user opted-in the usage of Project Panama.

Code Snippet 3.6: Java: Example of simple Java class that can be represented by a static proxy class

```
1 public class JavaStaticClass {
2     public static void voidParamlessMethod() { ... }
3 }
```

JNI-based static proxy

Code Snippet 3.7 shows an example of JNI-bases static proxy of Java class from Code Snippet 3.6. Any JNI-based proxy class contains `_javaType` field of `JavaType` type that references the Java type that the given proxy class corresponds to (line 3). This field gets initialized in the static constructor of the proxy class. `JavaType` constructor uses JNI to obtain a handle of the corresponding Java type.

For each Java method that should be accessible from C#, a proxy class contains the method ID field (`jni_voidParamlessMethod`) field in Code Snippet 3.7. These fields are defined as `Lazy`, and the first access initializes them via JNI invocation, looking up the Java method to be invoked (line 10).

Proxy method of static Java method (`VoidParamlessMethod` in our example) then uses `_javaType` field and method ID field (`jni_voidParamlessMethod`) to invoke Java method via JNI (lines 13 to 16).

Code Snippet 3.7: C# JNI-based proxy of Java static class

```
1  public static unsafe partial class JavaStaticClassJNI {
2      // java type reference
3      private static readonly JavaType _javaType;
4
5      static JavaStaticClassJNI() {
6          _javaType = new
               JavaType("javasources/testclasses/JavaStaticClass");
7      }
8
9      // java method ID
10     private static readonly Lazy<JniStaticMethod>
           jni_voidParamlessMethod = new(() =>
           _javaType!.GetStaticMethodID("voidParamlessMethod", "()V"));
11
12     // java method proxy
13     public static partial void VoidParamlessMethod() {
14         _javaType.CallStaticVoidMethod(
15             jni_voidParamlessMethod.Value, args:
                   Span<JniValue>.Empty);
16     }
17 }
```

Panama-based static proxy

Code Snippet 3.8 shows a Panama-based proxy of the same Java class (shown in Code Snippet 3.6). Notice that no `JavaType` field is required. Instead of JNI-based method ID, unmanaged delegate `panamaPtr_voidParamlessMethod` (line 5) is used to invoke the Java method. This delegate is initialized in the method that uses(`VoidParamlessMethod`) it via `BuildPanamaStub` invocation, that under the hood uses JNI to invoke Panama-based Java method that returns an upcall stub pointer. `panamaPtr_voidParamlessMethod` field cannot be `Lazy`, as unmanaged delegate types cannot be used as generic type parameters.

After the delegate is initialized, `VoidParamlessMethod` invokes it, effectively invoking the Java method (line 18).

Code Snippet 3.8: C# Panama-based proxy of Java static class

```
1  public static unsafe partial class StaticClassProxyPanama {
2      private const string _javaTypeName =
           "javasources.testclasses.JavaStaticClass";
3
4      // upcall stub pointer
5      private static delegate* unmanaged<void>
           panamaPtr_voidParamlessMethod;
6
7      // java method proxy
8      public static partial void VoidParamlessMethod() {
9          if (panamaPtr_voidParamlessMethod == default) {
```

```
10              nint _stubAddr = PanamaCallbackHelpers
11                  .BuildPanamaStub(
12                      _javaTypeName, "voidParamlessMethod", "()V")
13                  .Address;
14              panamaPtr_voidParamlessMethod = (delegate*
                    unmanaged<void> )_stubAddr;
15          }
16
17          // invoke unmanaged delegate
18          panamaPtr_voidParamlessMethod();
19      }
20  }
```

3.4.2 Non-static class proxy

Code Snippet 3.9 shows an example of Java `Coffee` class defining a constructor. Such a class cannot be emulated by static proxy; instead, non-static proxy has to be used. Code Snippet 3.10 demonstrates user-written **non-static partial skeleton of C# proxy class** emulating Java `Coffee` class. Notice that instead of defining a constructor, a proxy class defines **static factory method** `CreateMe`. The factory method is used to determine the signature of the constructor to be generated (see Section 2.7.1 for an explanation). The name of the factory method is not significant; it suffices that the method is annotated by `JavaImportCtor` attribute.

Code Snippet 3.9: Java: class defining constructor

```
1  public class Coffee {
2      public Coffee(int gramsOfCoffee) { ... }
3  }
```

Code Snippet 3.10: C#: non-static proxy skeleton

```
1  [JavaClassProxy]
2  public unsafe partial class Coffee {
3      [JavaImportCtor]
4      public static partial Coffee CreateMe(int gramsOfCoffee);
5  }
```

Code Snippet 3.11 shows a proxy class implementation that would get generated based on the skeleton from Code Snippet 3.10. Some implementation details were omitted for brevity.

On line 2, notice that the proxy class gets generated so that it inherits `JavaLangObjectProxy` type (see section 3.2.4) and implements `IJavaProxy` interface. Lines 16 and 20 implement methods required by this interface – `CreateProxyInstance`, `CreateProxyInstanceForInterface` (Section 3.2.5 described the purpose of these methods) and `GetJavaType` method that is used when the user works with Java object array from C#.

On lines 4 and 5, notice that apart from `_javaType` field referencing Java type, the non-static proxy also contains non-static `_javaObject` field referencing

Java object instance a given proxy instance corresponds to. This field gets initialized in instance constructors. Two utility constructors were omitted from the example: private constructor used by `CreateProxyInstance` and `CreateProxyInstanceForInterface` methods and protected constructor used when another proxy class inherits the given proxy class.

Code Snippet 3.11: C#: non-static generated proxy

```
1  public unsafe partial class Coffee
2      : JavaLangObjectProxy, IJavaProxy<Coffee> {
3      // java type and java instance references
4      private static readonly JavaType _javaType;
5      private readonly JavaObject? _instance;
6
7      static Coffee() {
8          _javaType = new JavaType("javasources/testclasses/Animal");
9          _jniCtor_I = _javaType.GetMethodID("<init>", "(I)V");
10     }
11
12     // utility constructors omitted
13     ...
14
15     // implementation of IJavaProxy interface
16     static Coffee IJavaProxy<Coffee>.CreateProxyInstance(
17         JniInternalMarker marker, JniLocalRef javaObjectPtr)
18     { ... }
19
20     public static new Coffee CreateProxyInstanceForInterface(
21         JniInternalMarker marker, JavaObject javaObject)
22     { ... }
23
24     static JavaType
           IJavaProxy<Coffee>.GetJavaType(JniInternalMarker marker)
25     { ... }
26
27     // constructor generated bases Brew factory method omitted here
28     private static JniMethod _jniCtor_I;
29     ...
```

Static constructor of the `Coffee` proxy class initializes `_jniCtor_I` field (line 9 in Code Snippet 3.11). This field represents the JNI method ID of the emulated Java constructor represented by factory method `CreateMe` (Code Snippet 3.10). Code Snippet 3.12 shows the generated implementation of `CreateMe` factory method and the constructor generated based on this method (with the same signature). Constructor uses `_javaType` and `_jniCtor_I` fields to invoke Java constructor via JNI (line 6). The factory method just invokes the generated constructor.

Code Snippet 3.12: C#: generated proxy constructor

```
1  // ctor generated based on the factory method
2  private static JniMethod _jniCtor_I;
3  public Coffee(int gramsOfCoffee) :
       base(JniEmptyCtorMarker.Instance) {
4      JniValue[] argsArr = { JniValue(gramsOfCoffee) };
5      Span<JniValue> args = new Span<JniValue>(argsArr);
6      _instance = _javaType.InvokeCtor(jniCtor_I, args);
7  }
8
9  // generated body of the factory method
10 public static partial Coffee CreateMe(int gramsOfCoffee) {
11     return new Coffee(gramsOfCoffee);
12 }
```

If `Coffee` class defined a non-static method, it would have been emulated similarly to the static method shown in Code Snippet 3.7. The only difference would be using `_javaObject` field instead of `_javaType` field to invoke the Java method.

3.4.3 Interface proxies

Code Snippet 3.13 shows an example of a user-written skeleton of C# proxy emulating Java interface. Code Snippet 3.14 captures the generated implementation of the interface proxy based on this skeleton.

Code Snippet 3.13: C#: Skeleton of interface proxy

```
1  [JavaInterfaceProxy]
2  public partial interface ICoffee {
3      [JavaImport]
4      void Drink();
5  }
```

On line 1 in Code Snippet 3.14, notice that the generated interface type extends `IJavaInterface` interface. `IJavaInterface` extends `IJavaProxy` interface, therefore it also requires static `CreateProxyInstance`, `CreateProxyInstanceForInterface` and `GetJavaType` methods. The generated interface provides a default implementation of these methods. They were, however, omitted from the Code Snippet 3.14 as they have already been shown in Code Snippet 3.11.

Apart from these methods, `IJavaInterface` requires one additional method – `GetProxyTypeRecord` method. This method is used by `InterfaceProxyFactory` to obtain a delegate to `CreateProxyInstanceForInterface` method of a particular class proxy when the Java method with an interface return type is called (see Section 3.2.5).

`GetProxyTypeRecord` tries to look up the delegate in `_interfaceImplementations` dictionary (initialization of the dictionary is omitted from the Code Snippet 3.14 because it has already been demonstrated in

Code Snippet 3.5). If the dictionary does not contain a record for a given Java type name, `_defaultInterfaceImplementation` field is returned. This field contains a delegate to `CreateProxyInstanceForInterface` method of the **default interface implementation proxy** class. Such proxy class is generated along with each proxy interface. It does not correspond to any existing Java class. It is used to represent return values of the Java method returning interfaces if no more appropriate proxy class is available. Otherwise, the default interface implementation proxy does not significantly differ from other non-static proxy classes.

Code Snippet 3.14: C#: Generated interface proxy

```
1  public partial interface ICoffee : IJavaInterface<ICoffee> {
2      private static JavaType _javaType = new
           JavaType("javasources/testclasses/Coffee");
3
4      // dictionary of interface implementations -- initialization
           omitted
5      private static readonly Dictionary<string, ProxyCreator>
           _interfaceImplementations = new() { ... }
6
7      // identify default proxy class implementing this interface
8      private static readonly ProxyCreator
           _defaultInterfaceImplementation =
           new(CoffeeDefaultProxy.CreateProxyInstance);
9
10     // implementation of IJavaInterface interface
11     static ProxyCreator
           IJavaInterface<ICoffee>.GetProxyTypeRecord(string
           javaTypeName) {
12         if (_interfaceImplementations.TryGetValue(javaTypeName,
               out ProxyCreator typeRecord)) {
13                 return typeRecord;
14         }
15         return _defaultInterfaceImplementation;
16     }
17
18     // implementation of IJavaProxy interface omitted
19     ...
20 }
```

3.5 Incremental source generator

The implementation of class and interface proxies shown in the previous section is generated by an incremental source generator. Incremental source generator allows us to hook into the process of compilation of a user-implemented assembly, inspect a user-written code using Roslyn API, and generate additional source code that gets added to the compilation [142, 122] (as if a user implemented it).

Incremental source generators are an improvement upon source generators that were added to .NET in .NET 5 [143]. Source generators tended to slow down IDEs, as the whole source generator had to be carried out during each compilation, which can happen on each keystroke in IDE. Therefore, .NET 6 introduced incremental source generators [122] that are implemented as stateless pipelines. The output of each pipeline stage gets cached, which allows skipping the following pipeline stages if some pipeline stage produces the same output as it produced during the previous run. That improves the source generator's performance.

Incremental source generator is implemented as a class that implements `IIncrementalGenerator` interface. This interface requires one method – `Initialize` – that is called exactly once and sets up the incremental source generator pipeline [122].

Figure 3.13 shows a typical structure of the incremental source generator pipeline. Usually, one of the first methods called in the pipeline is `CreateSyntaxProvider` method[5]. This method allows the incremental source generator to analyze a user-written code. It takes two parameters: a delegate to `predicate` method and a delegate to `transform` method [144].

Figure 3.13: Common structure of incremental source generator pipeline

`predicate` delegate will be invoked for every syntax node. It should return `true` for syntax nodes that are potentially relevant for the future pipeline phases. It is going to be invoked very often. Therefore, it should be fast, even if it will produce occasional false positives – e.g., it should not access the semantic model, and the filtering should only be based on the syntax tree.

`transform` delegate gets invoked for syntax nodes for which `predicate` returned `true`. It can access the semantic model and perform additional checks to filter out `predicate`'s potential false positives. Then, it usually extracts relevant data from the syntax node and its related semantic information (`ITypeSymbol`) to a custom DTO that gets passed to the following pipeline stages.

`CreateSyntaxProvider` method returns `IncrementalValuesProvider` [145] specialized to whatever type the `transform` delegate returns. The following stages of the pipeline can carry out further transformation, eventually producing `IncrementalValuesProvider<string>` that contains a generated source code and can be passed to `RegisterSourceOutput` method to add this source code to the compilation.

3.5.1 Source generator structure

The previous section introduced a general concept of an incremental source generator pipeline. Now, let's describe how this pipeline is implemented in our

[5]Unless the generator does not need to analyze a user-written code, for instance, because it generates source code based only on additional files [122]

`JavaInteropSourceGenerator`.

The pipeline is separated into two phases: **scanning** phase and **generating** phase. **Scanning phase** analyzes a user-written code, locates proxy types and methods, and collects all the required metadata to generate their implementations. **Generating phase** uses this metadata to generate the implementation and adds it to the compilation.

To decouple these phases, the implementation of `JavaInterop` incremental source generator is separated into four projects as shown in Figure 3.14. The main source generator project `JavaInterop.SourceGenerator` defines the incremental source generator pipeline and carries out the scanning phase. The generating phase is separated into two projects – `JavaInterop.SourceGenerator.JNI` and `JavaInterop.SourceGenerator.Panama` – each of which is responsible for generating the code of proxies based on the given interoperability technology (JNI or Panama). The interface between the scanning and generating phase is given by a set of data transfer objects (DTOs) that are defined by `JavaInterop.SourceGenerator.Common` project.

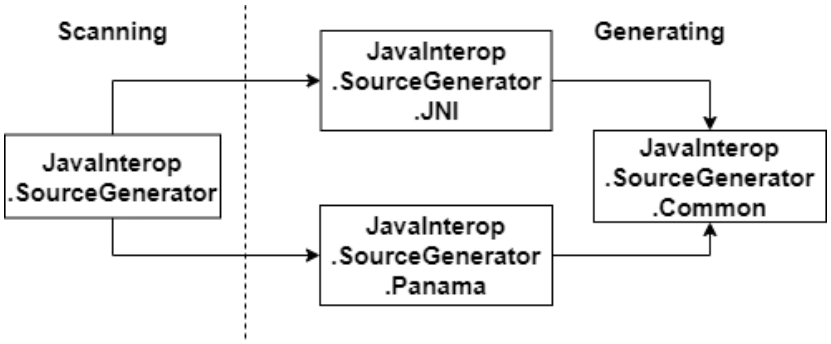

Figure 3.14: Architecture of `JavaInteropSourceGenerator`

Separation into the scanning and generating phase allows for potential other sources of metadata DTOs than is the current scanning of a user-written C# code via Roslyn API. As a future improvement, a parser of Java `.class` files could be added to the solution, allowing `JavaInterop` source generator to inspect Java bytecode. That would allow for generating of C# proxy classes based directly on Java code they interoperate with – without requiring a user to implement partial proxy skeletons. Or, when a user implements these partial skeletons, we could provide static (compile time) control that the skeletons match types and methods that actually exist in a provided Java bytecode.

3.5.2 Scanning phase

Figure 3.15 shows the whole incremental source generator pipeline of `JavaInteropSourceGenerator`. The scanning phase starts with our implementation of incremental source generator `predicate` delegate (see introduction of Section 3.5) – `CouldBeJavaImportAttribute` method. This method gets called for every syntax node in a user's assembly, and it provides the first filter to determine if the given syntax node could be relevant for the following stages of the pipeline. It returns `true` if the given syntax node syntactically corresponds to an annotation by one of **marker attributes** defined by our library –

`JavaClassProxyAttribute`, `JavaImportCtorAttribute`, `JavaImportAttribute` or `JavaInterfaceProxyAttribute`.

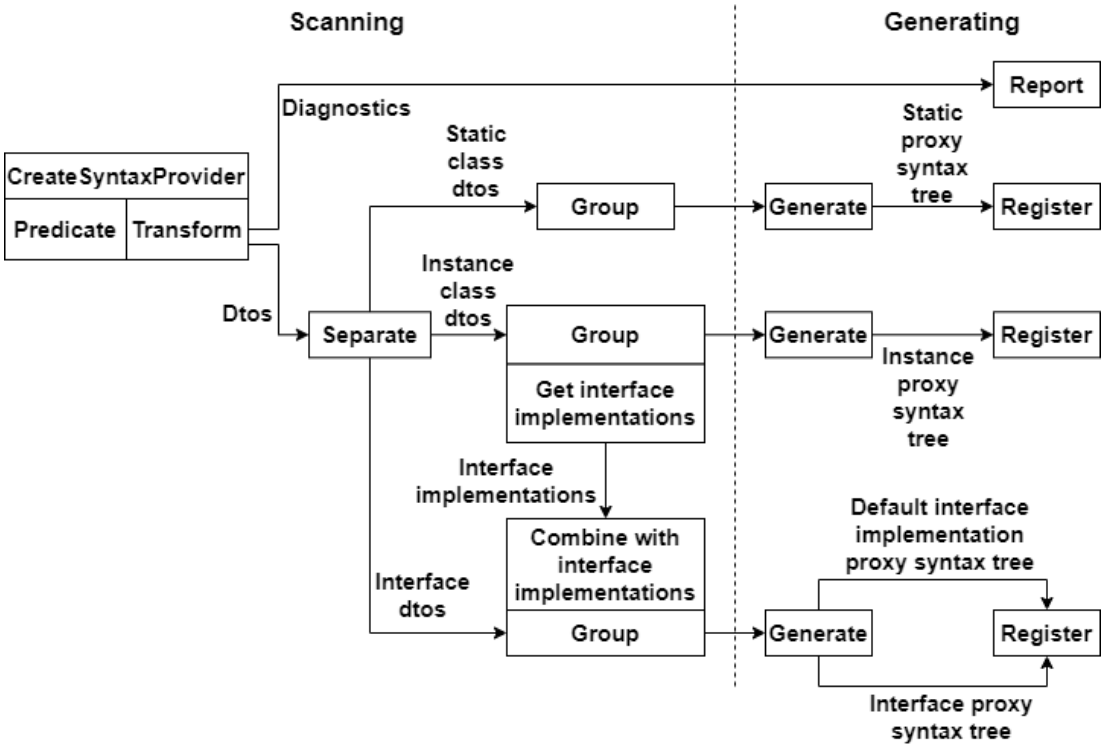

Figure 3.15: JavaInterop incremental source generator pipeline

`transform` delegate is implemented by `GetJavaProxyTypesMetadataOrDiagnostics` method. This method checks that the syntax node it receives matches the marker attribute not only syntactically but also semantically. Then it retrieves the syntax node that represents a declaration annotated by the attribute (either class, interface, method or constructor declaration depending on an attribute type) and gets its semantic information.

If the syntax node represents a type declaration (class or interface declaration), the given invocation of `GetJavaProxyTypesMetadataOrDiagnostics` method processes the whole type and all relevant proxy methods and constructors it contains, collecting all metadata required by following pipeline stages. If the syntax node is a method or constructor declaration, `GetJavaProxyTypesMetadataOrDiagnostics` invocation only validates if the method or the constructor is defined in the type annotated by our marker attribute. If so, it is ignored because it will be processed by `GetJavaProxyTypesMetadataOrDiagnostics` processing the annotated type. If not, the method or constructor is not considered valid, and `GetJavaProxyTypesMetadataOrDiagnostics` method returns `Diagnostics` object that will be emitted as a compiler error during the generation phase. That will inform a user about the incorrect usage of our attributes (see the upper part of Figure 3.15).

`GetJavaProxyTypesMetadataOrDiagnostics` method also carries out other validations – e.g., it checks that the proxy types and methods are partial and that proxy method signatures only contain supported types. If some of the requirements are violated, a diagnostic is returned. If the error is severe enough,

the given type or method is not propagated to the further pipeline stages (and, therefore, is omitted from the source generation).

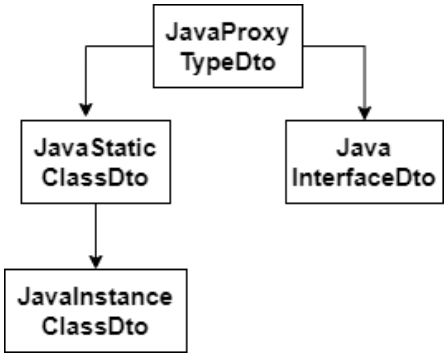

Figure 3.16: Generator static classes

As Section 3.4 described, we are distinguishing several kinds of type proxies – **static class proxies**, **instance class proxies** and **interface proxies**[6]. Each of these proxy kinds will require slightly different metadata to be generated. The incremental source generator pipeline, however, requires us to collect metadata of all proxy kinds by the same method. Therefore, this method must provide some kind of polymorphism in its outputs. We solved this by implementing an inheritance hierarchy between DTOs types representing metadata of individual proxy types – as shown in Figure 3.16.

As Section 2.9.3 described, all DTOs are implemented to be comparable by value and, therefore, compatible with incremental source generator caching mechanism.

Separating

Once metadata DTOs are returned by `GetJavaProxyTypesMetadataOrDiagnostics`, they are separated by their types (corresponding to proxy kinds) by the following stage of the pipeline (`separate` step in Figure 3.15). Each proxy kind is, from that point on, processed separately.

Grouping

As proxy types are partial, the definition of a user-defined proxy skeleton can be separated into multiple parts that will be processed by multiple invocations of `GetJavaProxyTypesMetadataOrDiagnostics` method and will therefore produce multiple metadata DTOs, each containing a part of the methods defined in the proxy type. Before a source generation can proceed, it is necessary to "group" proxy methods so that only one type DTO exists per proxy type and that it contains metadata of all proxy members defined in that type. This is done by **group** stage of the pipeline (see middle part of Figure 3.15). Grouping slightly differs for individual proxy kinds and is therefore carried out after **separate** phase.

[6]Future versions of the project can potentially add more kinds, e.g., enum proxies or inner class proxies

Interface implementations
As Section 3.4.3 showed, to generate an implementation of interface proxy, we need to have a list of proxy classes in the given assembly that implements the proxy interface. As the list is needed when an interface proxy is being generated, it should be a part of `JavaInterfaceDto` metadata. It, however, cannot be collected when `GetJavaProxyTypesMetadataOrDiagnostics` method processes the interface proxy itself because that would require scanning the whole assembly each time an interface is processed. Instead, when the instance proxy class is being processed, information about the proxy interfaces that it implements is stored. The later stage of the pipeline then retrieves this information from instance class proxy DTOs and passes it to the stage that processes proxy DTOs (see the middle part of Figure 3.15).

3.5.3 Generation phase

The generation phase receives proxy types metadata DTOs produced by the scanning phase. It uses Roslyn API to build the syntax tree of the proxy types implementations (implementations were demonstrated in Section 3.4). Then, it uses Roslyn API to convert these syntax trees to string and to add them to a compilation (see the right part of Figure 3.15).

Source code generation is mostly the process of building syntax trees based on proxy types of metadata DTOs. This process is handled by a set of static classes, each responsible for generating a particular part of the proxy implementation. The overview of these static `Generator` classes is shown in Code Snippet 3.17. Arrows in this Figure indicate that one class uses the other to obtain a part of the generated AST.

Implementing the source generation via a set of static classes rather than by instance methods of proxy metadata DTOs allows us to decouple data from logic and to decouple the scanning and generation phases.

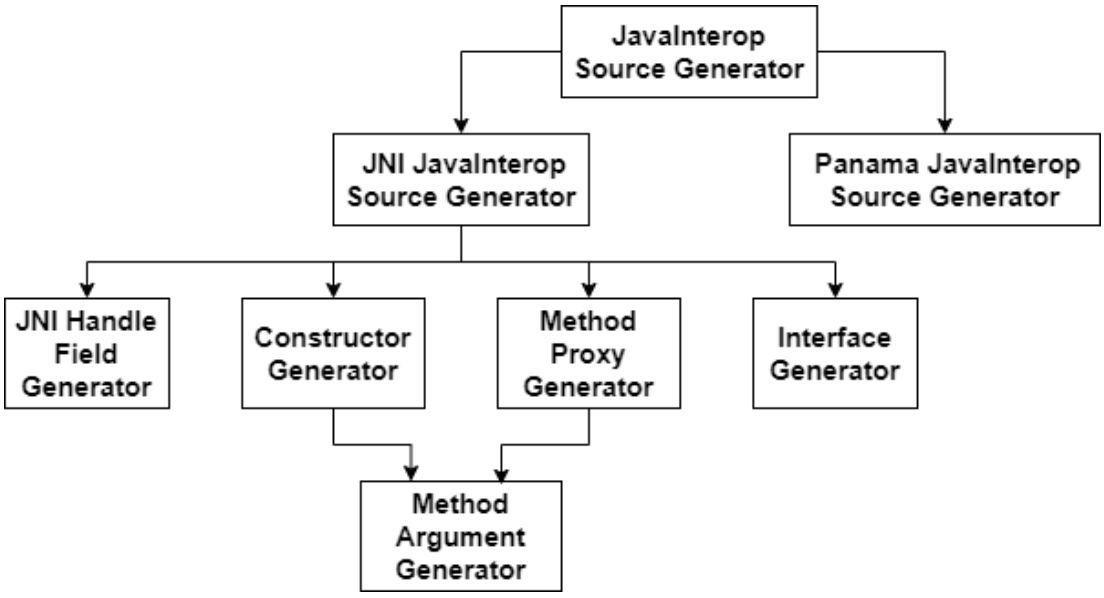

Figure 3.17: Static generator classes

3.6 Unit tests

Unit tests of the solution are separated into two projects – `JavaInterop.UnitTests` and `JavaInterop.UnitTests.Panama`. Our tool only supports Project Panama on Java versions 20, 21, and 22. Separation into two projects allows us to easily run general tests on all tested versions of Java and Panama-based tests only on Java versions where we expect them to function.

GitHub actions were used [146] to test the solution on multiple Java versions, Java distributions, and operating systems. Table 3.2 captures test results on all tested combinations of Java version, distribution, and operating system. As the table shows, the solution currently passes tests on the majority of tested environments. It only fails with older versions of OpenJDK on the CentOS operating system, where the solution fails to locate Java sources to be interoperated with. The solution, however, functions on the CentOS operating system with newer Java versions and functions with older Java versions on other platforms. Problems encountered with this particular combination of the operating system and Java version do not seem to be crucial and can be fixed during future improvements.

Java version / OS and distribution	8	11	17	20	21	22
Ubuntu OpenJdk	✓	✓	✓	✓	✓	✓
Ubuntu Corretto	–	–	–	–	✓	–
Ubuntu GrallVM	–	–	–	–	✓	–
Alpine OpenJdk	✓	✓	✓	✓	✓	–
Alpine Correto	–	–	–	–	✓	✓
CentOS OpenJdk	✗	✗	✗	✓	✓	✓
CentOS Corretto	–	–	–	–	✓	–
CentOS GrallVM	–	–	–	–	✓	–
Windows 10 Pro OpenJdk	✓	–	–	✓	✓	✓

Table 3.2: Test result for various environments

4. User documentation

This chapter describes the intended usage of the tool implemented in this thesis. The tool enables interoperability between C# and Java code. Its intended user is a developer developing a C# application that needs to interoperate with some code base implemented in Java programming language.

This chapter is written in the form of tutorials. Each section introduces a particular feature of the library and instructs a user on how to use that feature. All the code presented in examples throughout this chapter can be found in the thesis attachments in folder `/src/LanatraDemo/` and can be run by running the following command in that folder (tutorial number must be an integer from 1 to 8):

```
dotnet run --project LanatraDemo <tutorial-number>
```

4.1 Setup

The tool will be distributed in the form of NuGet package `Lanatra.JavaInterop.nupkg`. For the purposes of submission in the thesis, the NuGet package has not yet been published online, but it has been built, and it is available locally as an attachment of the thesis (in folder `/src/LanatraDemo/artifacts/`). To use the tool, one must add a package reference to `Lanatra.JavaInterop.nupkg` by adding the following to the `.csproj` file:

```
<ItemGroup>
    <PackageReference Include="Lanatra.JavaInterop"
       Version="1.0.0" PrivateAssets="None" />
</ItemGroup>
```

To use a local NuGet package, one must configure the local package source via `nuget.config` file [147]. An example of such a file can be found as a thesis attachment in folder `/src/LanatraDemo/`.

4.1.1 Tutorial 0 – Create Java VM

Before a user can interoperate with a Java code from C#, they have to create a JVM instance in the process of running a .NET application. That can be done via calling `JavaVM.Create` method provided by our tool. In order for the method to succeed, our tool needs to be able to locate the Java distribution to be used. The way of setting the path to the Java distribution, however, differs between Linux and Windows platforms.

Setting path to Java distribution – Linux
On Linux, it is necessary to set two environment variables before our tool can be used. `JAVA_HOME` environment variable must be set to the root directory of the Java distribution to be used. `LD_LIBRARY_PATH` environment variable must be set

to the location of `libjvm.so` file, which is usually located in `JAVA_HOME/lib/server` folder. The setup can, therefore look for example as follows:

```
export JAVA_HOME=/usr/lib/jvm/java-21-openjdk
export LD_LIBRARY_PATH=/usr/lib/jvm/java-21-openjdk/lib/server
```

After setting these environment variables `JavaVM.Create` call should succeed, and it should produce an output as follows:

Code Snippet 4.1: C#: Create Java VM

```
JavaVM.Create();
// Class path: ...
// Using java version 21.0.3
```

Setting path to Java distribution – Windows

On Windows, it is only necessary to specify the path to JAVA_HOME. Users can either set it via environment variable as on Linux, or they can specify it via a parameter of `JavaVM.Create` method as it is demonstrated in Code Snippet 4.2. The parameter of `JavaVM.Create` method is considered to have a higher priority than setting JAVA_HOME environment variable. On Linux, this parameter cannot be used.

Code Snippet 4.2: C#: Setting JAVA_HOME via `JavaVM.Create` parameter

```
JavaVM.Create(pathToJavaHome:
    "C:\\Users\\Name\\.jdks\\openjdk-20.0.1_windows-x64\\jdk-20.0.1");
// Class path: ...
// Using java version 20.0.1
```

Apart from locating Java executable, our tool also needs to be able to locate Java code with which a user wishes to interoperate. A user has to specify Java Class Path via a parameter of `JavaVM.Create` method to contain a path to `.jar` files or to directories containing `.class` files a user wishes to use from C#. Multiple paths can be specified, as it is demonstrated in Code Snippet 4.3. Apart from user-specified paths, our tool automatically adds the content of `AppContext to the classpath.BaseDirectory` property [148].

Code Snippet 4.3: C#: Setting Java class path

```
JavaVM.Create(javaClassPaths:
    ["path\\to\\jar-files\\my-javalib1.jar",
    "path\\to\\jar-files\\my-javalib2.jar",
    "path\\to\\folder\\with\\class-files\\"]);
// Class path: -Djava.class.path=-Djava.class.path=
// SolutionDir\project\bin\configuration\dotnet-version\
// :path\to\jar-files\my-javalib1.jar
// :path\to\jar-files\my-javalib2.jar:
// path\to\folder\with\class-files\
// Using java version ...
```

Passing JVM options
`JavaVM.Create` method also enables specifying options for Java VM instance being created. Users can specify the same set of options that can be applied to `java` command when it is run in the command line. Code Snippet 4.4 shows the example.

Code Snippet 4.4: C#: Setting JVM options

```
1 JavaVM.Create(jvmOptions: [
2     "-verbose:class", "-XX:+PrintCompilation"]);
```

4.2 Static proxy classes

Once a Java VM instance is created, we can call some Java methods. To be able to invoke a Java method from C#, a user must first define C# partial **proxy class** that will emulate the Java class containing the method to be invoked. Then, they must define a partial **proxy method** that will emulate the method itself. Both a class and a method must be annotated with specific attributes defined in our library. The rest of this chapter will describe all the processes in much more detail. Once a user assembly contains correctly annotated partial classes and methods, the incremental source generator will be used to generate their implementation, leveraging the interop invocation.

Java does not allow non-nested classes to be static. Our tool, however, allows proxy classes to be static as long as they only access static members of the corresponding Java classes. This use case is simpler than full-fledged working with Java objects, and therefore, we will start tutorials by describing it.

4.2.1 Tutorial 1 – Invoke static Java method

Suppose that a user has Java class `CoffeeMachine` defining a few static Java methods as shown in Code Snippet 4.5 and they want to invoke these methods from a C# code.

Code Snippet 4.5: Java: Java class with static methods

```
1  package javalib;
2
3  public class CoffeeMachine {
4      public static void turnOn() { ... }
5      public static void refill(
6          byte gramsOfCoffee, int millilitersOfWater) { ... }
7      public static byte getGramsOfCoffee() { ... }
8
9      // other methods and fields are omitted
10     ...
11 }
```

To use this Java class from C#, a user has to define a partial proxy class as shown in Code Snippet 4.6. Notice that:

- C# **namespace** name matches Java **package** name (apart from the casing, which follows conventions of respective languages),
- C# proxy **class name** matches Java class name apart from the fact that it contains an additional (and optional) `Proxy` suffix,
- C# proxy class is **annotated** by `JavaClassProxy` attribute,
- proxy class contains **proxy methods** that:
  - are **partial**
  - are **annotated** by `JavaImport` attribute
  - have **names and signatures** that match respective Java methods – Java primitive types are represented by corresponding C# primitive types (e.g C# `int` by Java `int`). The only exception among primitive types is Java `byte` type that is signed and is therefore represented by C# `sbyte` type (see methods `Refill` and `GetGramsOfCoffee`).

Our tool will **resolve the full name of Java class** corresponding to the declared proxy class from the full name of the proxy class. C# namespace name determines Java package; casing in the namespace name does not play a role because, by convention, Java package names are lowercase. If a proxy class name contains suffix `Proxy`, this suffix won't be considered when looking up the corresponding Java class. **Java methods** are located by their names and signatures. Naming conventions for methods differ between Java and C# in the casing of the first letter. Upper-case first letter of C# method names will be considered lower-case when looking up Java method.

Code Snippet 4.6: C#: Basic proxy class containig static proxy methods

```
1  using JavaInterop.Common.Attributes;
2  namespace Javalib;
3
4  [JavaClassProxy]
5  public static partial class CoffeeMachineProxy {
6      [JavaImport]
7      public static partial void TurnOn();
8      [JavaImport]
9      public static partial void Refill(
10        sbyte gramsOfCoffee, int millilitersOfWater);
11     [JavaImport]
12     public static partial sbyte GetGramsOfCoffee();
13 }
```

Once a user has implemented this proxy, they can invoke proxy methods as shown as shown in Code Snippet 4.7 effectively calling Java methods from C#.

Code Snippet 4.7: C#: Using static proxy class

```
CoffeeMachineProxy.TurnOn();
sbyte gramsOfCoffee =  CoffeeMachineProxy.GetGramsOfCoffee(); \\ 0
CoffeeMachineProxy.Refill(gramsCoffee: 42, millilitersWater: 500);
gramsOfCoffee = CoffeeMachineProxy.GetGramsOfCoffee(); \\ 42
```

Method generation mode

In the example in Code Snippet 4.6 all proxy methods were explicitly annotated by `JavaImport` attribute. That gives a user more flexibility: the proxy class could potentially contain non-proxy methods implemented in C#.

However, a user may often want to implement a proxy class that directly represents the corresponding Java class. Then, all methods declared in the proxy class will be proxy methods of some Java methods, and it would be unnecessarily verbose to annotate each of them separately. In that case, a user can set `javaImportMode` parameter of `JavaClassProxy` attribute to `Declared`, and all the methods declared in the proxy class will be automatically considered proxy methods without having to be annotated with `JavaImport` attribute. Code Snippet 4.8 demonstrates that. The proxy class declared in Code Snippet 4.8 is semantically equivalent to the one from Code Snippet 4.6.

Notice also that `CoffeeMachine` proxy class in Code Snippet 4.8 does not contain optional `Proxy` suffix in its name, and yet it will be resolved to the same Java class as `CoffeeMachineProxy` class shown in Code Snippet 4.6.

Code Snippet 4.8: C#: Setting MethodGenerationMode

```
using JavaInterop.Common.Attributes;
namespace Javalib;

[JavaClassProxy(javaImportMode: JavaImportMode.Declared)]
public static partial class CoffeeMachine {
    public static partial void TurnOn();
    public static partial void Refill(sbyte gramsOfCoffee, int
       millilitersOfWater);
    public static partial sbyte GetGramsOfCoffee();
}
```

Specifying classes and methods names

So far, the names of Java methods and classes corresponding to C# proxies were resolved automatically based on the names of proxies. The described name resolution mechanism requires a user to declare their namespace hierarchy so that it mirrors the Java package structure and names their classes and methods so that the names match Java names. Though that may make the code using proxy classes easier to read in some cases, sometimes a user may wish not to respect a structure and naming imposed by Java.

In that case, a user is given the option to specify the full name of the corresponding Java class explicitly via parameters of `JavaClassProxy` attribute and to specify the name of the corresponding Java method via the parameter of

`JavaImport` attribute. Then, they are free to name their C# classes and methods as they please. Code Snippet 4.9 demonstrates that. Notice that names specified as attribute parameters directly correspond to Java methods and class names from Code Snippet 4.5. `CoffeeMachineWithChangedNames` proxy class is equivalent to both `CoffeeMachine` and `CoffeeMachineProxy` classes shown previously in this section.

Code Snippet 4.9: C#: Specifying Java class and method names explicitly

```
1  using JavaInterop.Common.Attributes;
2  namespace LanatraDemo.Tutorials.T1.Static.Proxies;
3
4  [JavaClassProxy(javaTypeFullName:"javalib.CoffeeMachine")]
5  public static partial class CoffeeMachineWithChangedNames {
6      [JavaImport(javaMethodName: "turnOn")]
7      public static partial void Start();
8      [JavaImport(javaMethodName: "refill")]
9      public static partial void AddCofeeAndWater(sbyte gramsCoffee,
           int millilitersWater);
10     [JavaImport(javaMethodName: "getGramsOfCoffee")]
11     public static partial sbyte GetRemainingCoffee();
12 }
```

Remember that in order for Java (and therefore for our tool) to be able to locate a given Java class, **Java Class Path** combined with the **package name** must form the whole path to the folder where `.class` file implementing the class is located. Otherwise, an exception will be thrown the first time the corresponding C# proxy class is used.

4.2.2 Tutorial 2 – Enable Panama

If a user uses our tool with Java version 20, 21, or 22, an opt-in optimization based on Project Panama is available for certain kinds of invocations. To enable the optimization, a user has to opt-in twice – once by a parameter of `JavaVM.Create` method and the second time on the assembly level via a parameter of `JavaImportAssembly` attribute. Code Snippet 4.10 demonstrates that. If the optimization is opted-in but a sufficient Java version is unavailable at runtime, an exception will be thrown.

Code Snippet 4.10: C#: Opt-in usage of Project Panama

```
1  [assembly: JavaImportAssembly(
2      jniPanamaUsage: JniPanamaUsageOption.Panama)]
3  ...
4  JavaVM.Create(enablePanama: true);
```

Once Panama optimization is enabled, an invocation of each method that meets the following criteria will be optimized using Project Panama:

- method is declared in an assembly annotated as shown in Code Snippet 4.10

- method is static,
- it only accepts primitive type parameters,
- it has a primitive return type,
- it is declared in the static proxy class.

Methods that do not fulfill these criteria won't be optimized (their invocation will be leveraged via JNI as usual). The optimization can also be opted-out for individual methods via `optOutPanama` parameter of `JavaImport` attribute. Code Snippet 4.11 provides an overview of these situations. If Panama optimization is enabled for an assembly containing `PanamaCoffeeMachine` proxy class, methods `TurnOn` and `Refill` will be optimized because their signatures allow for it. `GetGramsOfCoffee` method won't be optimized because optimization is opted-out for it, and `GetStatus` method won't be optimized because the string return type is not supported by Panama optimization.

Code Snippet 4.11: C#: Proxy class using Project Panama

```
1  [JavaClassProxy(javaTypeFullName: "javalib.CoffeeMachine")]
2  public static partial class PanamaCoffeeMachine {
3      // uses Panama
4      [JavaImport]
5      public static partial void TurnOn();
6      // uses Panama
7      [JavaImport]
8      public static partial void Refill(sbyte gramsOfCoffee, int
           millilitersOfWater);
9      // uses JNI, because Panama is opted-out
10     [JavaImport(optOutPanama: true)]
11     public static partial sbyte GetGramsOfCoffee();
12     // uses JNI, because Panama does not support method signature
13     [JavaImport]
14     public static partial string? GetStatus();
15 }
```

4.3 Objects

Java is an object oriented language, therefore when interoperating with Java code it will often be necessary to manipulate with Java object instances. This section will described how it can be done using our tool.

4.3.1 Tutorial 3 – Working with Java object instances

Suppose that Java `CoffeeMachine` class shown in Code Snippet 4.5 defines static `MakeCoffee` method that returns an instance of `Coffee` Java class shown in Code Snippet 4.12.

Code Snippet 4.12: Java: Example of Java class

```
public class Coffee {
    public Coffee addSugar() { ... }
    public Coffee addMilk(){ ... }
    public void printStatus() { ... }
    ...
}
```

To invoke a Java method that returns (or takes) an instance of a Java class, a user has to define a non-static C# proxy class for the given Java class. Non-static proxy class for `Coffee` Java class class shown in Code Snippet 4.12 can look as demonstrated in Code Snippet 4.13. Notice that apart from not being static, the proxy class does not differ in any way from the static proxies introduced in the previous section.

Code Snippet 4.13: C#: Non-static proxy class

```
namespace Javalib;

[JavaClassProxy(javaImportMode: JavaImportMode.Declared)]
public partial class CoffeeProxy {
    public partial CoffeeProxy AddSugar();
    public partial CoffeeProxy AddMilk();
    public partial void PrintStatus();
}
```

Then the proxy method of Java `MakeCoffee` method returning a `Coffee` instance can be defined as shown in Code Snippet 4.14.

Code Snippet 4.14: C#: Proxy method returning object instance

```
[JavaClassProxy]
public static partial class CoffeeMachine  {
    [JavaImport]
    public static partial CoffeeProxy MakeCoffee(sbyte
       useGramsOfCoffee);
}
```

Code Snippet 4.15 demonstrates the usage. Call to `MakeCoffee` proxy method returns a reference to an instance of Java `Coffee` class represented by C# `CoffeeProxy` proxy class. Instance methods can then be invoked on this proxy instance, effectively invoking methods on the corresponding instance of Java class, which is demonstrated in Code Snippet 4.15. `AddSugar` and `AddMilk` methods modify Java `Coffee` instance and `PrintStatus` method displays these changes.

Code Snippet 4.15: C#: Working with Java object instance

```
CoffeeProxy coffee = CoffeeMachineProxy.MakeCoffee(27);
coffee.PrintStatus();
// [Java] Amount of coffee: 27 grams
// [Java] Contains sugar: No
// [Java] Contains milk: No
```

```
6
7  coffee.AddSugar()
8      .AddMilk();
9      .PrintStatus();
10 // [Java] Amount of coffee: 27 grams
11 // [Java] Contains sugar: Yes
12 // [Java] Contains milk: Yes
```

Invoking Java constructors

In the previous example, a C# code obtained an instance of Java `Coffee` class as a return value of static `MakeCoffee` method. Such a factory method, however, often won't be available in Java API, and then a user may need to create a Java object instance by invoking a constructor.

Suppose that Java `Coffee` class defines constructors as shown in Code Snippet 4.16 – the parameter-less constructor and the constructor taking one int parameter.

Code Snippet 4.16: Java: Example of Java class with constructors

```
1  public class Coffee {
2      // methods shown previously omitted
3      ...
4      public Coffee() { ... }
5      public Coffee(int gramsOfCoffee) { ... }
6  }
```

To be able to invoke these constructors, a user must add static factory methods to `CoffeeProxy` class definition as shown in Code Snippet 4.17. These factory methods must:

- be static,
- be annotated by `JavaImportCtor` attribute,
- accept the same parameter types as the desired constructor,
- return defining type (`CoffeeProxy` type in our example).

A user can choose a name of the factory arbitrarily (the example in Code Snippet 4.17 uses the name `Brew`).

Code Snippet 4.17: Java: Example of Java class with constructors

```
1  namespace Javalib;
2
3  [JavaClassProxy(javaImportMode: JavaImportMode.Declared)]
4  public partial class CoffeeProxy {
5      // methods shown previously omitted
6      ...
7      [JavaImportCtor]
8      public static partial CoffeeProxy Brew();
```

```
9       [JavaImportCtor]
10      public static partial CoffeeProxy Brew(int gramsOfCoffee);
11  }
```

Based on factory methods annotated with `JavaImportCtor`, the incremental source will generate a C# constructors accepting the same parameter types as factory methods accept. Invoking these constructors effectively invokes Java constructors with the corresponding signature. Users can, therefore, create Java object instances from C# by invoking C# proxy constructors, as shown in Code Snippet 4.18.

Code Snippet 4.18: C#: Invoke Java constructors

```
1  CoffeeProxy coffee1 = new CoffeeProxy();
2  CoffeeProxy coffee2 = new CoffeeProxy(gramsOfCoffee: 35);
```

Though using generated constructors to create proxy instances is preferable, a user can also create proxy instances using the static factory method (`Brew` method in our example).

4.3.2 Tutorial 4 – Proxies with inheritance

A commonly used concept in object-oriented languages is inheritance. Our tool is able to emulate inheritance between Java classes by inheritance between C# proxy classes.

Suppose that a user needs to work with Java classes `Animal`, `Cat`, and `Fish` that inherit one another (shown in Code Snippet 4.19). Notice that `Animal` base class defines abstract method `makeSound`, which inherited classes implement.

Code Snippet 4.19: Java: Classes with inheritance

```
1  package javalib;
2
3  public abstract class Animal {
4      public abstract void makeSound();
5  }
6
7  public class Cat extends Animal {
8      @Override
9      public void makeSound() {
10         System.out.println("Meow");
11     }
12 }
13
14 public class Fish extends Animal {
15     @Override
16     public void makeSound() {
17         System.out.println("<quiet>");
18     }
19 }
```

A user can represent these Java classes by C# proxy classes as shown in Code Snippet 4.20. Proxy classes mirror the Java inheritance hierarchy. Notice, however, that the base proxy class `AnimalProxy` is not abstract (unlike Java `Animal` class) and that it defines the non-abstract proxy method `MakeSound`. Also, notice that inherited proxy classes `CatProxy` and `FishProxy` do not provide any override of `MakeSound` method.

For the purposes of our example, `CatProxy` and `FishProxy` classes define static factory method `CreateMe`, based on which parameterless constructors of these classes get generated (as was explained in Section 4.3.1). These constructors will emulate default parameterless constructors of Java `Cat` and `Fish` classes and allow us to create instances of these classes from C#.

Code Snippet 4.20: C#: Proxies with inheritance

```
1  namespace Javalib;
2
3  [JavaClassProxy(javaImportMode:JavaImportMode.Declared)]
4  public partial class AnimalProxy {
5      public partial void MakeSound();
6  }
7
8  [JavaClassProxy]
9  public partial class CatProxy : AnimalProxy {
10     [JavaImportCtor]
11     public static partial CatProxy CreateMe();
12 }
13
14 [JavaClassProxy]
15 public partial class FishProxy : AnimalProxy {
16     [JavaImportCtor]
17     public static partial FishProxy CreateMe();
18 }
```

Code Snippet 4.21 shows the usage of proxy classes from Code Snippet 4.20. Line 2 calls `MakeSound` proxy method, which invokes the correct Java implementation of the method based on the proxy runtime type.

Code Snippet 4.21: C#: Using proxy classes with inheritance

```
1  List<AnimalProxy> animals = [new CatProxy(), new FishProxy(), new
       CatProxy()];
2  animals.ForEach(animal => animal.MakeSound());
3  // Meow
4  // <quiet>
5  // Meow
```

4.3.3 Tutorial 5 – Working with interfaces

Suppose a user needs to invoke a Java method with Java interface as its return or parameter type. For that, it is necessary to be able to represent the concept

of Java interfaces in C# code.

Code Snippet 4.22 shows Java class `UIManager` that defines two methods: `addElement` method that accepts a parameter of `UIElement` type and `getElementById` that returns `UIElement` type. Supposes that `UIElement` type is **Java interface** defined as shown in Code Snippet 4.23.

Code Snippet 4.22: Java: Class working with Java interface

```
1  package javalib;
2
3  public class UIManager {
4      public static void addElement(UIElement element) {
5          System.out.println("Adding element");
6          element.draw();
7          ...
8      }
9      public static UIElement getElementById(int id) { ... }
10 }
```

Code Snippet 4.23: Java: Example of Java interface

```
1  package javalib;
2
3  public interface UIElement {
4      void draw();
5      int getId();
6  }
```

To be able to call Java `addElement` and `getElementById` methods from C#, a user must first define a **C# proxy of Java interface** `UIElement`. That can be done as shown in Code Snippet 4.24. Java interface is represented by C# interface that is annotated by `JavaInterfaceProxy` attribute. The Proxy interface declares methods that correspond to methods declared by the Java interface. Methods are not marked as partial but are annotated by `JavaImport` attribute.

Code Snippet 4.24: C#: Proxy of Java interface

```
1  namespace Javalib;
2
3  [JavaInterfaceProxy]
4  public partial interface IUIElementProxy {
5      [JavaImport]
6      void Draw();
7      [JavaImport]
8      int GetId();
9  }
```

Suppose that Java provides 2 classes that implement `UIElement` interface: `Button`, `Label`. User can represent these classes as C# proxy classes that implement C# proxy interface `IUIElementProxy`. Example is shown in Code Snippet 4.25.

Code Snippet 4.25: C#: Proxy class implementing an interface

```
1 namespace Javalib;
2
3 [JavaClassProxy(javaImportMode: JavaImportMode.Declared)]
4 public partial class ButtonProxy : IUIElementProxy {
5     [JavaImportCtor]
6     public static partial ButtonProxy Create(int id);
7     public partial void Draw();
8     public partial int GetId();
9 }
```

Suppose that a user has also defined a proxy class for Java `UIManager` class itself, using `IUIElementProxy` interface for the parameter type of `AddElement` method and the return type of `GetElementById` method. Then, they can work with Java `UIManager` class from C# code as shown in Code Snippet 4.26.

Code Snippet 4.26: C#: Usage of proxy interface

```
1 UIManagerProxy.AddElement(new ButtonProxy(_button1Id));
2 UIManagerProxy.AddElement(new LabelProxy(_labelId)));
3
4 IUIElementProxy element1 =
     UIManagerProxy.GetElementById(_button1Id);
5 element1.Draw(); // drawing Button
6 Debug.Assert(element1 is ButtonProxy);
7
8 IUIElementProxy element2 = UIManagerProxy.GetElementById(_labelId);
9 element2.Draw(); //  drawing Label
10 Debug.Assert(element2 is LabelProxy);
```

A user does not have to implement proxy classes for all Java classes that implement an interface. If the Java method returning interface type at runtime returns an instance of Java class for which C# assembly does not define a dedicated proxy class, **default interface implementation proxy** will be used to represent this instance on the C# side. Code Snippet 4.27 demonstrates this, supposing that `GetElementById` call on line 1 returns an instance of `Image` Java class that implements `UIElement` interface but no `ImageProxy` class is implemented in the current C# assembly. Invocation of `Draw` method on line 2 will invoke correct Java implementation of the method provided by `Image` Java class, even though the C# side code has no knowledge about the existence of this class.

Code Snippet 4.27: C#: Default interface implementation

```
1 IUIElementProxy element3 = UIManagerProxy.GetElementById(_imageId);
2 element3.Draw(); // drawing Image
3 Debug.Assert(element3 is UIElementDefaultProxy);
```

4.3.4 Tutorial 6 – Working with strings

Suppose a user wants to invoke Java method `concat` shown in Code Snippet 4.28, that accepts string parameters and returns a string return value.

Code Snippet 4.28: Java: Method working with strings

```
1 public static String concat(String s1, String s2){
2     return s1 + s2;
3 }
```

When defining a proxy method for Java `concat` method, a user can choose if they are going to represent Java strings (`java.lang.String`) by C# string (`System.String`) or by the dedicated proxy type – `JavaLangStringProxy`. This decision can be made for each occurrence of the string type in the method signature separately. Potential definitions of the proxy method invoking Java `concat` method can, therefore, look as shown in Code Snippet 4.29. The first proxy method variant uses C# string to represent all occurrences of a string in the Java method signature; the second one uses `JavaLangStringProxy`, and the third uses their combination.

On line 2, notice that when the C# string is used to represent the Java string return type, the C# string must be nullable. This is because Java has no concept of nullable annotations, and any string returned from Java can potentially be null.

Code Snippet 4.29: C#: Variants of proxy method of Java method working with strings

```
1  [JavaImport]
2  public static partial string? Concat(string s1, string s2);
3
4  [JavaImport(javaMethodName:"concat")]
5  public static partial JavaLangStringProxy ConcatProxy(
6      JavaLangStringProxy s1, JavaLangStringProxy s2);
7
8  [JavaImport(javaMethodName: "concat")]
9  public static partial JavaLangStringProxy ConcatCombined(
10     string s1, JavaLangStringProxy s2);
```

The advantage of using C# string is the user convenience and the more intuitive signature of a proxy method. It is, however, less effective alternative as each time a string is passed from C# to Java a new string instance must be allocated on the Java heap. `JavaLangStringProxy` represents a reference to an existing Java string instance and can be therefore passed to proxy methods repeatedly without the necessity of further allocations. That makes `JavaLangStringProxy` a preferable option for representing strings in situations when the same string instances are passed between languages repeatedly. Otherwise more intuitive approach using C# string can be used.

4.3.5 Tutorial 7 – Working with arrays

Suppose that a user wants to obtain a reference to a Java primitive type array and modify the array from C#. We will describe this situation in the example of `int` array. Arrays of other Java primitive types would be handled similarly.

Given Java methods returning and taking `int` array shown in Code Snippet 4.30, the user can define corresponding C# proxy methods as shown in Code Snippet 4.31. Notice that the Java `int` array gets represented by the generic `JavaPrimitiveArrayProxy` type in proxy method signatures. The generic type parameter specifies an array element type – `int` in our example.

Code Snippet 4.30: Java: example of methods working with primirtive type array

```
1 public class ArrayUtils{
2     public static int[] returnJavaArray() {
3         return new int[]{1, 2, 3, 4, 5};
4     }
5     public static void printArray(int[] array) { ... }
6 }
```

Code Snippet 4.31: C#: Proxy methods working with Java int array

```
1 [JavaClassProxy(javaImportMode:JavaImportMode.Declared)]
2 public static partial class ArrayUtils {
3     public static partial JavaPrimitiveArrayProxy<int>
          ReturnJavaArray();
4     public static partial void PrintArray(
5         JavaPrimitiveArrayProxy<int> array);
6 }
```

`JavaPrimitiveArrayProxy` type represents a reference to a Java array. It does not, however, hold the array elements, so it does not implement an indexer. To access array elements, a user must first load them by calling `LoadJavaArrrayElements` method as shown on line 2 in Code Snippet 4.32. Once a user holds array elements, they access them via an indexer, reading and modifying them. Once the dispose method is called on an object representing array elements, changes made to the array are propagated to Java.

Code Snippet 4.32: C#: Accessing elements of Java primitive type array

```
1  using JavaPrimitiveArrayProxy<int> array =
       ArrayUtils.ReturnJavaArray();
2  using(var arrayElements = array.LoadJavaArrayElements()) {
3      // do some computation with array elements
4      for (int i = 0; i < arrayElements.Length; i++)  {
5          if (arrayElements[i] % 2 == 0)
6              arrayElements[i] += 42;
7      }
8  } // here array changes get propagated to Java
9
10 ArrayUtils.PrintArray(array); // [JAVA] 1 44 3 46 5
```

`JavaPrimitiveArrayProxy` also allows to access Java array elements by obtaining their copy stored in C# array via `GetCopyOfArray` method. That is demonstrated in Code Snippet 4.33. On line 2, a new Java array is allocated on the Java heap from C# code. The Java `modifyArray` method can then modify this array. Afterwards, C# can observe the changes by obtaining a copy of Java array elements (line 4).

Code Snippet 4.33: C#: Getting copy elements of Java primitive type array

```
1 // allocate new array on Java heap
2 using JavaPrimitiveArrayProxy<int> array = new([1, 2, 3, 4, 5]);
3 ArrayUtils.ModifyArray(array);
4 int[] copy = array.GetCopyOfArray(); // [C#] 1 44 3 46 5
```

Object Arrays

Conciser Java method `petAnimals` (shown in Code Snippet 4.34) that accepts an array of instances of `Animal` class (defined in Code Snippet 4.19). Suppose that a user wants to invoke this Java method from C#.

Code Snippet 4.34: Java: Example of Java method accepting objetc array paramater

```
1 public static void petAnimals(Animal[] animals){
2     for (int i = 0; i < animals.length; i++ ){
3         animals[i].makeSound();
4     }
5 }
```

Java object array (such as `Animal[]`) can be represented by generic `JavaObjectArrayProxy` type in C#. The generic type parameter specifies the proxy type that represents Java object array element type (`AnimalProxy` from Code Snippet 4.20 in our example). Therefore, the C# proxy method of Java `petAnimals` method may look as shown in Code Snippet the method from the Code Snippet 4.34 may look as shown in Code Snippet 4.35.

Code Snippet 4.35: C#: Proxy method accepting object array paramater

```
1 [JavaImport]
2 public static partial void
    PetAnimals(JavaObjectArrayProxy<AnimalProxy> animals);
```

Code Snippet 4.36 shows an example of invoking `PetAnimals` proxy method. First, `JavaObjectArrayProxy` is created from the C# array of proxy classes, effectively allocating a new object array on the Java heap. Then, this array proxy can be passed to the `PetAnimals` method.

As lines 5 to 7 of Code Snippet 4.36 demonstrate, `JavaObjectArrayProxy` defines an indexer that can be used to access individual elements of the Java object array.

Code Snippet 4.36: C#: Using object array proxy

```
using JavaObjectArrayProxy<AnimalProxy> animals = new(
    [new CatProxy(), new FishProxy(), new CatProxy()]);
ArrayUtils.PetAnimals(animals); // Meow <quiet> Meow

animals[0].MakeSound(); // Meow
animals[1].MakeSound(); // <quiet>
animals[2].MakeSound(); // Meow
```

Multidimensional Arrays

Our library also allows a user to work with multidimensional Java arrays. Suppose that a user wants to invoke following Java method `flatten2DArray` accepting two-dimensional array of `int` and returning its flatten (one-dimensional) variant (Code Snippet 4.37).

Code Snippet 4.37: Java: Example of Java method accepting 2D int array

```
public static int[] flatten2DArray(int[][] array2D){ ... }
```

Our library also allows a user to work with multidimensional Java arrays. Suppose that a user wants to invoke the following Java method `flatten2DArray` accepting a two-dimensional array of `int` and returning its flatten (one-dimensional) variant (Code Snippet 4.37).

Code Snippet 4.38: C#: Proxy method accepting 2D int array parameter

```
[JavaImport]
public static partial JavaPrimitiveArrayProxy<int> Flatten2DArray(
    JavaObjectArrayProxy<JavaPrimitiveArrayProxy<int>> array2D);
```

`Flatten` method proxy can be used as shown in Code Snippet 4.39. Line 4 creates a 2D array proxy from 3 `int` array proxies created previously. Line 6 passes this 2D array to the Java `flatten` method that returns it flatten as 1D `int` array.

Code Snippet 4.39: C#: Using 2D array proxy

```
using JavaPrimitiveArrayProxy<int> arr1 = new
   JavaPrimitiveArrayProxy<int>([1, 2, 3]);
using JavaPrimitiveArrayProxy<int> arr2 = new
   JavaPrimitiveArrayProxy<int>([4, 5]);
using JavaPrimitiveArrayProxy<int> arr3 = new
   JavaPrimitiveArrayProxy<int>([6, 7, 8, 9]);
using JavaObjectArrayProxy<JavaPrimitiveArrayProxy<int>> array2D =
   new([arr1, arr2, arr3]);

JavaPrimitiveArrayProxy<int> flatten =
   ArrayUtils.Flatten2DArray(array2D); // 1 2 3 4 5 6 7 8 9
```

Arrays with more dimensions and multidimensional arrays of objects can be used similarly.

4.4 Tutorial 8 – Using existing library

Section 1.1.2 introduced the Apache PDFBox library as an example of an existing Java library that could be used via our tool from a C# code. Code Snippet 1.1 demonstrated the usage of the library on the example of reading text from `.pdf` file as a string. Code Snippet 4.40 contains a copy of this example for a reader's convenience.

Using our tool we can now implement the same functionality of reading text from `.pdf` file via Apache PDFBox library from C# code. Code Snippet 4.41 demonstrates that. Notice That once proxy classes are defined, Java library can be used from C# code in the same manner as it is used from Java code (compare Code Snippets 4.40 and 4.41). Code Snippet only captures usage of proxy classes, their definitions are omitted for brevity. The example from Code Snippet 4.41 and definitions of all proxy classes that appear in it can be found in thesis attachments (in folder `/src/LanatraDemo/LanatraDemo/`).

Code Snippet 4.40: Java: Example of Apache PDFBox library usage

```
1  import org.apache.pdfbox.Loader;
2  import org.apache.pdfbox.pdmodel.PDDocument;
3  import org.apache.pdfbox.text.PDFTextStripper;
4
5  static String ReadPdfAsText(String path) throws IOException {
6      File pdfFile = new File(path);
7      PDDocument pdf = Loader.loadPDF(pdfFile);
8      PDFTextStripper stripper = new PDFTextStripper();
9      String pdfContent = stripper.getText(pdf);
10     pdf.close();
11     return pdfContent;
12 }
```

Code Snippet 4.41: C#: Using Apacha PDFBox library from C#

```
1  using Org.Apache.Pdfbox.Loader;
2  using Org.Apache.Pdfbox.Pdmodel.PDDocument;
3  using Org.Apache.Pdfbox.Text.PDFTextStripper;
4
5  public static string? ReadPdfAsText(string filePath) {
6      FileProxy file = new FileProxy(filePath);
7      PDDocumentProxy pdf = LoaderProxy.LoadPDF(file);
8      PDFTextStripperProxy stripper = new PDFTextStripperProxy();
9      string? pdfContent = stripper.GetText(pdf);
10     pdf.Close();
11     return pdfContent;
12 }
```

5. Conclusion

The conclusion of this thesis first evaluates whether the thesis managed to meet its goals. Then, it suggests several possible extensions that could be implemented to improve the developed software in the future.

5.1 Evaluation of thesis goals

The objective of the thesis was to design and implement a .NET library for interoperability with Java that should enable users to use APIs implemented in Java from C# code bases. This library should:

- fulfill requirements **R1** – **R11** defined in the thesis introduction,
- be tested on the set of supported platforms selected in Section 2.2,
- and be tested using selected existing Java library Apache PDFBox [5].

Let's first go through requirements **R1** – **R11** and evaluate whether they have been met.

R1 **Solution should support invocation direction from C# to Java. Opposite invocation direction (from Java to C#) won't be supported.**

As multiple sections of this thesis have shown, the implemented solution allows a user to invoke static and instance Java methods from C#. It also allows invoking Java constructors and manipulating Java objects from C#. Invocation of C# methods from Java is not supported, even though Section 1.3.1 hinted at how this invocation direction could possibly be implemented.

R2 **As both Java and C# are object-oriented languages and object instances are a crucial part of the majority of API implemented in these languages, the solution should allow users to manipulate Java instances and invoke Java instance methods from C#.**

As was demonstrated in Section 4.3.1, our solution allows users to manipulate Java object instances from a C# code – to create them, to pass them to methods or return them from methods, and to invoke instance methods on them. As Section 4.3.2 mentioned, the solution also allows the user to emulate Java inheritance hierarchy by inheritance between C# proxy classes.

R3 **Solution should not require a user to modify a code of a Java library in order to make it usable from C#.**

The solution does not impose any particular requirements on Java methods and classes it accesses (in the manner in which JNI restricts signatures of native methods it is able to invoke from Java – Section 1.2.1). However, as some features of Java programming language are not yet supported, we are not able to access all arbitrary Java constructs. For instance, our solution

does support Java enums; it is therefore not capable of invoking methods that accept enum parameters. Users may want to circumvent this limitation by implementing a Java adapter layer wrapping unsupported features into methods with supported signatures. To sum it up, our solution conceptually meets requirement **R3**, but as its current version lacks support for certain Java features, a user can be motivated to implement an adapter layer on the Java side to make a particular Java code usable from C#.

R4 **Solution should avoid runtime code generation.**

No part of the solution depends on a runtime code generation. Incremental source generator (Section 3.5) is used to generate code at compile time.

R5 **Configuration of generated proxies should be user-friendly and in-code.**

A user configures generated proxy classes by implementing their partial skeletons and annotating them by marker attributes. No external configuration is required. Section 2.7 was devoted to designing certain aspects of generated proxies in a user-friendly manner.

R6 **Solution should function on modern multiplatform .NET and on common Java distributions and versions.**

Section 2.2 analyzed a set of Java versions, Java distributions, and operating systems on which the solution should function. As section 3.6 described, GitHub Actions were used to test the solution on these platforms. The solution passes tests on the majority of tested platforms.

R7 **Interop will work on API level: allowing invocation of Java methods and usage of Java objects from C#. It will not be IL-level interop.**

The solution does not attempt to interpret Java bytecode on CLR. Rather, it uses JNI or Foreign Function API from Project Panama to invoke Java methods and manipulate Java objects.

R8 **Solution should provide C# proxies for Java classes with static type safety. Dynamic objects should not be used for the proxies.**

The solution does not use dynamic objects. It represents Java classes by compile time generated C# proxy classes, providing static type safety.

R9 **Solution should generate .NET proxies for custom Java classes without requiring to use external tool explicitly.**

User is not required to use any external tool to generate proxy classes. Proxy classes are generated by incremental source generator during the build of the project that uses our library.

R10 **To ensure static type safety, the solution should provide separate C# proxies for separate Java types. It should not be possible to represent instances of a Java type by instances of a proxy that does not correspond to that particular Java type.**

Each generated C# proxy class corresponds to a particular Java class. Instances of the proxy class can only represent instances of that given Java class (or Java classes inherited from that class). The user is strongly discouraged from creating proxy instances representing instances of mismatched Java types. Section 2.8 focused on designing this aspect of the solution.

R11 Solution should generate proxies .NET that emulate Java classes, so that user experience is seamless, working with Java classes as if they were implemented in C#.

Chapter 4 demonstrated the usage of our solution. Solution emulates Java classes by C# proxy classes, Java methods by C# proxy methods, and Java constructors by C# constructors. As Section 4.4 shows, C# code that uses Java library via our tool reasonably resembles equivalent Java code that uses the library directly from Java.

Given the above evaluation, we consider **requirements R1 – R11** to be reasonably fulfilled, and therefore, we consider the first goal of the thesis to be met.

The next goal was to **test the solution on the set of selected supported platforms**. As Section 3.6 described a reasonable amount of unit tests was implemented and these tests were run on the selected set of platforms using GitHub actions. We therefore consider this goal to be met.

Lastly, the solution should **be tested using existing Java library** Apache PDFBox [5]. Code Snippet 4.41 in Section 4.4 demonstrated that the implemented solution enables a user to use this Java library from C# code for a simple use case of reading `.pdf` file text as a string.

Writing any text to `.pdf` file using Apache PDFBox library, however, requires specifying text font via calling `setFont` method on `PDPageContentStream` instance as shown in Code Snippet 5.1 [149]. This method takes a parameter of `PDFont` type, which is the abstract type, and usually, an instance of inherited `PDType1Font` is provided [149] as shown in Code Snippet 5.1. Constructor of `PDType1Font`, however, accepts parameter of `Standard14Fonts.FontName` type (line 2 in Code Snippet 5.1), which is a nested Java enum. Here, our ability to emulate Java features fails us because the current version of the library does not support either enums or nested Java types.

Code Snippet 5.1: Java: Using Apacha PDFBox to write text to .pdf file

```
1 PDPageContentStream contentStream = new PDPageContentStream(pdf,
    firstPage);
2 PDType1Font font = new
    PDType1Font(Standard14Fonts.FontName.COURIER);
3 contentStream.setFont(font, 14);
4 contentStream.beginText();
5 contentStream.newLineAtOffset(50, 700);
6 contentStream.showText("abraka dabra");
```

Our solution, therefore, is usable with a real Java library, however, only to a limited extent. To enable users to use arbitrary Java API from C#, we would have to provide support for a broader range of Java features. Enums, nested

types and field accesses are missing features with the highest priority to be supported. However, **we managed to create functional and carefully tested prototype that can be easily extended into a full-fledged solution**.

5.2 Possible future improvements

This section analyzes potential extensions that could be implemented to improve the software developed in this thesis in the future.

5.2.1 `.class` file parser

Using a current version of the solution, a user must define a partial skeleton of C# proxy for every Java class and method they want to work with from C#. That can be cumbersome. Incremental source generator could be extended to read Java `.class` files and generated C# proxy classes based directly on existing Java classes.

Apart from analyzing user-written assembly, an incremental source generator can also read so-called **additional files** – non C# external files whose content can be used during source generation. Our `JavaInteropSourceGenerator` incremental source generator could, therefore, read Java `.class` or `.jar` files as additional files. Then a C# parser of `.class` file format would be required to parse provided `.class` files[1] and to build DTOs that are expected as an input of the generating phase (see Section 3.5.3) of `JavaInteropSourceGenerator`. That way, `JavaInteropSourceGenerator` could generate C# proxy classes based on the content of provided `.class` files.

Additional files

In order for a file to be accessible as an additional file of a source generator, it has to be registered in `.csproj` file of the project that uses the source generator (as shown in Code Snippet 5.2). Our current solution expects that the path to Java sources to be used is specified at runtime via `javaClassPath` parameter of `JavaVM.Create` method (see Section 4.1.1). The approach to this configuration would have to be changed.

Code Snippet 5.2: Registering Java library as additional file of incremental source generator in .csproj of a project that uses the generator

```
1 <ItemGroup>
2     <AdditionalFiles Include="path/to/javalib.jar" />
3 </ItemGroup>
```

Once additional files are registered, the incremental source generator can access them in its pipeline. Code Snippet 5.3 captures a part of incremental source generator `Initialize` method that sets up the pipeline step accessing additional files [151].

[1] `.jar` format is just a `.zip` archive of `.class` files [150]

Code Snippet 5.3: Acceess additional files from the source generator

```
context.AdditionalTextsProvider
    .Where(text => text.Path.EndsWith(
        ".jar",  StringComparison.OrdinalIgnoreCase))
    ... // process additional files
```

`.class` file parser

In order for `.class` file parser to be usable from the source generator, it has to be compliant with `netstandard2.0`[2]. To our knowledge, no such implementation of `.class` file parser is currently available. `.class` file parser could be potentially implemented using Katai Struct C# Runtime library [152, 153] or implemented from scratch, according to `.class` format specification [134].

Generation mode

It does not seem efficient to, by default, generate C# proxy classes for all Java classes present in a given `.jar` file. Users should be allowed to configure which Java classes they intend to use from C#. According to requirement **R5**, this configuration should be user-friendly and in-code. We can let users specify which Java classes should have their C# proxy generated by defining a partial proxy class (as it is done now). We can, however, add new `GenerationMode` called, e.g., `JavaDeclared` (`GenerationMode` parameter of `JavaClassProxy` attribute was introduced in Section 4.2.1). This mode will use `.class` file parser to obtain metadata for all public methods and constructors defined in the corresponding Java type and will generate proxy methods for these Java methods without requiring a user to specify method signatures manually. Users will still be allowed to specify proxy method signatures explicitly if they need to adjust the signature in some way (e.g., use `JavaLangStringProxy` instead of C# string to represent some string parameter – see Section 4.3.4).

If a user specifies signatures of proxy methods explicitly, `.class` file parser could be used to provide static checks across the language border to make sure that a signature of the C# proxy method is compatible with the signature of the corresponding Java method.

5.2.2 Callbacks from Java

In accordance with requirement **R1**, our solution only supports the invocation of Java methods from C#. Invocation of C# methods from Java is not supported – not even in the form of callbacks.

Callbacks could be enabled by allowing C# classes to inherit Java classes or to implement Java interfaces. Then, a proxy method that expects a proxy of a Java class or Java interface could accept a C# class inherited from that proxy class or implementing that proxy interface. Java method corresponding to that proxy method can then invoke methods required by the Java interface or by the Java parent class, effectively invoking their C# implementations.

[2]Because the source generator itself has to target `netstandard2.0` to be able to function in Visual Studio IDE [122].

Even though our solution does not currently support this, Section 1.3.1 analyzed, how is this functionality implemented by Xamarin.Android Java.Interop.

Summary of Xamarin.Android approach to callbacks

To enable Java to invoke C# methods, it is necessary to generate Java "proxy" class that corresponds to the C# class defining these methods (in Xamarin.Android, these Java classes are called Android Callable Wrappers (ACW) or Java Callable Wrappers (JCW)). This class must contain declarations of `native` methods, whose signatures correspond to C# methods to be invoked (see Code Snippet 1.15 for the example).

These Java `native` methods then have to have their C# implementation registered via `RegisterNatives` JNI function [8].

To make the registration possible, one additional layer of abstraction is required. JNI imposes restriction on the signature of the method that can be used as an implementation of Java `native` method – it has to accept the first two parameters of expected types (see Section 1.2.1). Therefore, the arbitrary C# method cannot be used as the implementation of the Java `native` method. JNI compatible C# wrapper method must be generated (in the context of Xamarin.Android, this method is referred to as **marshaller**).

Implications for our solution

In our context, **marshaller** methods could be easily generated at compile time by the incremental source generator. `RegisterNatives` JNI function can be potentially called from the source-generated static constructor of the type that inherits a proxy class or implements the proxy interface. The problem is source-generating Java proxy classes.

Ideally, we would like to generate them via incremental source generator, as it can inspect user-written assembly and collect metadata about what Java classes and what Java `native` methods need to be generated. The problem is, however, that the source generator is not allowed to produce any other output than a code that gets added to the .NET compilation. It cannot emit external files such as `.java` or `.class` files containing Java proxies.

There are two alternative solutions:

- JNI `DefineClass` function could be used. This function takes a bytecode of a Java class and loads this class to JVM. However, as we can only invoke at runtime when a JVM instance exists, it means that the bytecode of Java proxy classes would be generated at runtime, which violates requirement **R4**. Moreover, not all Java distributions are guaranteed to support the JNI `DefineClass` function (e.g., ART VM does not support it [7]).

- External tool could be implemented to scan a user-written assembly and to generate an implementation of Java proxy classes (possibly source code, but preferably bytecode). The disadvantage of this approach is that a user-written assembly would have to be scanned twice – via an external tool and an incremental source generator. Furthermore, a user would be required to explicitly use external tools, which violates requirement **R9**.

Support for callbacks would undoubtedly make our tool more general and usable in more scenarios. Therefore, one of the requirements mentioned above will probably have to be relaxed so that support for callbacks can be provided.

5.2.3 Support more Java features

The current version of the solution only supports a subset of Java features. In the future we could add support for example for:

- **Accessing Java field** – as Section 2.7.2 described, the current version of our JNI-based interoperability library is capable of accessing Java fields from the C# code base. However, as C# does not currently support partial properties, we postponed designing a Java field proxy that would make this functionality accessible for a user (Section 2.7.2 explained the reasoning behind this decision).

- **Java enums** – Section 2.1.3 described that enums in Java conceptually differ from C# enums. Java enums are reference types. C# proxy could be designed to represent Java enums. This proxy would probably be a C# class rather than a C# enum type. If support for partial properties was added to C#, they could be used to represent individual enumeration variants of Java enum type.

- **Nested classes** – as Section 2.1.7 described, Java distinguishes two kinds of nested classes – static and non-static (inner) nested classes. Static nested classes are semantically equivalent with C# nested classes, and support for them would require only slight changes in `JavaInteropSourceGenerator`. Non-static classes are conceptually different and support for them would require designing a dedicated proxy.

- **Generics** – as Section 2.1.2 explained, generics in Java are implemented using **runtime erasure**. They, therefore, only exist at compile time; at runtime, generic type parameters are replaced by the most general type applicable. It should, therefore, be possible to invoke **generic** Java **methods** from C# via JNI if we specify a method signature so that occurrences of type parameter type are replaced by the most general type applicable.

  Runtime erasure is also carried out for Java **generic classes**. Therefore, at runtime, they should be indistinguishable from non-generic classes, and it should be possible to manipulate them from C# via JNI. The obvious problems are:

  - designing C# proxy class so that it emulates generic API over non-generic Java bytecode and non-generic JNI API,
  - obtaining information about generic signature of a class or a method by parsing `.class` file. It seems that even though the bytecode itself does not capture the concept of generics, `.class` file contains metadata, including signatures of generic types and methods before the erasure for the purposes of reflection and debugging [134]. This metadata could be potentially read by `.class` file parser and used to generate generic C# proxies.

5.2.4 Other improvements

This section list a several more miscellaneous possible improvements.

- **Runtime dispatch generation mode** – the current version of the solution decides whether a given Java method will be invoked via JNI or Project Panama at a compile time. Project Panama can, however, only be used if a supported Java version is available at runtime. If such a Java version is not available, an attempt to call the Panama-based proxy method will throw an exception. It could be helpful to provide a mixed method proxy that will check the available Java version at runtime and, based on it, decide which interoperability technology (JNI or Panama) should be used to leverage the invocation. This adds a small overhead to every method invocation and doubles the size of generated method proxies. It would, however, enable users to use our tool so that an application using the tool can be deployed to the platform with any Java version, and Panama-based optimization can be used where it is applicable.

  This feature was not a priority for the thesis as it does not conceptually bring anything new. However, it could be useful for the practical use of our tool.

- **Fix lookup of Java sources for older Java versions on CentOS** – as Section 3.6 stated, the implemented solution has been tested on a variety of Java versions, distributions, and operating systems. In the majority of them, the tests pass without any issues. However, with older versions of OpenJDK Java on the CentOS operating system, the solution has a problem with locating Java sources it should interoperate with. For the purposes of the thesis, this does not seem to be a major issue as the solution passes tests on the majority of tested environments. However, during the project's future development, the cause of the issue should be identified and fixed.

- **Incremental source generator: caching for diagnostics** – Section 2.9.3 analyzed how to design DTOs that get passed between stages of incremental source generator pipeline in the way that incremental source generator can take advantage of caching of stages results. In this thesis, we designed proxy types DTOs to comply with incremental source generator caching, and we implemented tests ensuring that caching is actually used when these DTOs are passed between pipeline stages.

  Apart from proxy classes DTOs, our incremental source generator pipeline also works with `Diagnostic` objects, which allows us to emit compiler errors when a user attempts to use marker attributes in a way that is not intended. These `Diagnostics` objects, however, do not meet conditions listed in Section 2.9.3, which would allow incremental source generator to cache them. As this was discovered in the final phase of the development, time did not allow us to refactor the mechanism of emitting diagnostics in a way that would comply with caching. Therefore, an incremental source generator will be able to take advantage of caching when a user-written code is correct and does not cause any of our custom diagnostics. If diagnostics are emitted, caching is unlikely to work. The diagnostic emitting mechanism should be refactored in the future.

Bibliography

[1] Stack overflow 2023 developer survey. https://survey.stackoverflow.co/2023/#most-popular-technologies-language. [cit. 23/06/2024].

[2] Greg Utas. The software rewrite. https://www.codeproject.com/Articles/5283862/The-Software-Rewrite. [cit. 23/06/2024].

[3] Xingru Chen, Deepika Badampudi, and Muhammad Usman. Reuse in contemporary software engineering practices – an exploratory case study in a medium-sized company. *e-Informatica Software Engineering Journal*, 16(1):220110, September 2022. [cit. 23/06/2024].

[4] Erik Dietrich. The myth of the software rewrite. https://daedtech.com/the-myth-of-the-software-rewrite/. [cit. 23/06/2024].

[5] Apache pdfbox library. https://pdfbox.apache.org/. [cit. 23/06/2024].

[6] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999. [cit. 25/06/2024].

[7] MSDN – Android Callable Wrappers for Xamarin.android. https://learn.microsoft.com/en-us/previous-versions/xamarin/android/platform/java-integration/android-callable-wrappers. [cit. 25/06/2024].

[8] Java Native Interface Specification. https://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html. [cit. 25/06/2024].

[9] MSDN – Platform Invoke. https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke. [cit. 25/06/2024].

[10] MSDN – DllimportAttribute. https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.dllimportattribute?view=net-8.0. [cit. 25/06/2024].

[11] MSDN – Source generation for platform invokes. https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke-source-generation. [cit. 25/06/2024].

[12] Java T Point – Android Operating System. https://www.javatpoint.com/android-operating-system. [cit. 25/06/2024].

[13] Android for Developers – Activity class. https://developer.android.com/reference/android/app/Activity. [cit. 25/06/2024].

[14] MSDN – What is Xamarin. https://learn.microsoft.com/en-us/xamarin/get-started/what-is-xamarin. [cit. 25/06/2024].

[15] MSDN – Xamarin – Binding a Java Library. https://learn.microsoft.com/en-us/previous-versions/xamarin/android/platform/binding-java-library/. [cit. 25/06/2024].

[16] MSDN – Xamarin Architecture. https://learn.microsoft.com/en-us/xamarin/android/internals/architecture. [cit. 25/06/2024].

[17] MSDN – IComponentCallbacks interface. https://learn.microsoft.com/en-us/dotnet/api/android.content.icomponentcallbacks?view=net-android-34.0. [cit. 25/06/2024].

[18] Android for Developers – ComponentCallbacks. https://developer.android.com/reference/android/content/ComponentCallbacks. [cit. 25/06/2024].

[19] MSDN – Java.Lang.Object Class. https://learn.microsoft.com/en-us/dotnet/api/java.lang.object?view=net-android-34.0. [cit. 25/06/2024].

[20] MSDN – Working with JNI and Xamarin.Android. https://learn.microsoft.com/en-us/previous-versions/xamarin/android/platform/java-integration/working-with-jni. [cit. 25/06/2024].

[21] MSDN – Activity class. https://learn.microsoft.com/en-us/dotnet/api/android.app.activity?view=net-android-34.0. [cit. 25/06/2024].

[22] MSDN – ExportAttribute Class. https://learn.microsoft.com/en-us/dotnet/api/java.interop.exportattribute?view=net-android-34.0. [cit. 25/06/2024].

[23] MSDN – RegisterAttribute Class. https://learn.microsoft.com/en-us/dotnet/api/android.runtime.registerattribute?view=net-android-34.0. [cit. 25/06/2024].

[24] .NET for Android GitHub – mono.android.Runtime Java class. https://github.com/dotnet/android/blob/d8f818d00611588a065982c61f6002ff023f8beb/src/java-runtime/java/mono/android/Runtime.java. [cit. 25/06/2024].

[25] .NET for Android GitHub – AndroidRuntime C# class. https://github.com/dotnet/android/blob/05ca38c7248b9af926c3c314bb99bf5cc22c3bd3/src/Mono.Android/Android.Runtime/AndroidRuntime.cs. [cit. 26/06/2024].

[26] .NET for Android GitHub – wiki – blueprint. https://github.com/xamarin/xamarin-android/wiki/Blueprint. [cit. 26/06/2024].

[27] GitHub – Jonathan Pryor. https://github.com/jonpryor. [cit. 26/06/2024].

[28] .NET for Android GitHub – Mono.Android.Export. https://github.com/dotnet/android/tree/main/src/Mono.Android.Export. [cit. 26/06/2024].

[29] MSDN – Java Bindings Metadata. https://learn.microsoft.com/en-us/previous-versions/xamarin/android/platform/binding-java-library/customizing-bindings/java-bindings-metadata. [cit. 26/06/2024].

[30] GitHub – IKVM.NET. https://github.com/ikvmnet/ikvm. [cit. 26/06/2024].

[31] Ikvm.net. https://ikvm.org/. [cit. 26/06/2024].

[32] Lang NEXT 2012 IKVM NET Building a Java VM on the NET Framework. https://www.youtube.com/watch?v=cuXLA7lrmSU. [cit. 26/06/2024].

[33] IKVM.NET WEBLOG. https://web.archive.org/web/20170718180358/http://weblog.ikvm.net/2017/04/21/TheEndOfIKVMNET.aspx. [cit. 26/06/2024].

[34] GitHub – Windward IKVM. https://github.com/windward-studios/ikvm. [cit. 26/06/2024].

[35] David Thielen. Windward Studios – IKVM is alive & well. https://www.windwardstudios.com/blog/ikvm-is-alive-well. [cit. 26/06/2024].

[36] GitHub – Windward IKVM – commits. https://github.com/windward-studios/ikvm/commits/master/. [cit. 26/06/2024].

[37] GitHub organization – IKVM. https://github.com/ikvmnet. [cit. 26/06/2024].

[38] JVM CLR Object Bridge. https://www.jcobridge.com/features/. [cit. 26/06/2024].

[39] GitHub – JNet. https://github.com/masesgroup/JNet/tree/master. [cit. 26/06/2024].

[40] GitHub – JNet – JNet: Reflector. https://github.com/masesgroup/JNet/blob/master/src/documentation/articles/usageReflector.md. [cit. 26/06/2024].

[41] Unity engine. https://unity.com/. [cit. 26/06/2024].

[42] Unity Documentation – Call Java and Kotlin plug-in code from C# scripts. https://docs.unity3d.com/Manual/android-plugins-java-code-from-c-sharp.html. [cit. 26/06/2024].

[43] Unity Documentation – AndroidJNI. https://docs.unity3d.com/ScriptReference/AndroidJNI.html. [cit. 26/06/2024].

[44] JNBridge. https://jnbridge.com/software/jnbridgepro/overview. [cit. 26/06/2024].

[45] JNBridge – userguide. https://jnbridge.com/guides/usersguide.pdf. [cit. 26/06/2024].

[46] Javonet. https://www.javonet.com/. [cit. 26/06/2024].

[47] GitHub – jni4net. https://github.com/jni4net/jni4net/tree/master. [cit. 26/06/2024].

[48] Pavel Šavara. How calling from java to .net works in jni4net. https://zamboch.blogspot.com/2009/11/how-calling-from-java-to-net-works-in.html. [cit. 26/06/2024].

[49] GitHub – jni4net – commits. https://github.com/jni4net/jni4net/commits/master/. [cit. 26/06/2024].

[50] GitHub – JNetCall. https://github.com/xafero/JNetCall/. [cit. 26/06/2024].

[51] GitHub – JNetCall – commits. https://github.com/xafero/JNetCall/commits/main/. [cit. 26/06/2024].

[52] MSDN – The C# type system. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/. [cit. 29/06/2024].

[53] Java Documentation – Package java.lang. https://docs.oracle.com/javase%2F9%2Fdocs%2Fapi%2F%2F/java/lang/package-summary.html. [cit. 29/06/2024].

[54] Project valhalla. https://openjdk.org/projects/valhalla/. [cit. 29/06/2024].

[55] Java Documentation – Primitive Data Types. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. [cit. 29/06/2024].

[56] MSDN – Built-in types (C# reference). https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types. [cit. 29/06/2024].

[57] MSDN – Decimal Struct. https://learn.microsoft.com/en-us/dotnet/api/system.decimal?view=net-8.0. [cit. 29/06/2024].

[58] MSDN – DateTime Struct. https://learn.microsoft.com/en-us/dotnet/api/system.datetime?view=net-8.0. [cit. 29/06/2024].

[59] MSDN – Guid Struct. https://learn.microsoft.com/en-us/dotnet/api/system.guid?view=net-8.0. [cit. 29/06/2024].

[60] Petr Hnětynka. Pokročilé programování v jazyce Java, lecture 2. https://d3s.mff.cuni.cz/cz/teaching/nprg021/, 2021. [cit. 29/06/2024].

[61] Java Documentation – Bounded Type Parameters. https://docs.oracle.com/javase/tutorial/java/generics/bounded.html. [cit. 29/06/2024].

[62] Java Documentation – Restrictions on Generics. https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html. [cit. 29/06/2024].

[63] MSDN – Enumeration types (C# reference). https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum. [cit. 29/06/2024].

[64] Java Documentation – Enum Types. https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html. [cit. 29/06/2024].

[65] MSDN – C# Arrays. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/arrays. [cit. 29/06/2024].

[66] Java for c# programmers. https://web.archive.org/web/20230602194107/https://www.kynosarges.org/JavaCSharp.html. [cit. 29/06/2024].

[67] Hans Bergsten. *JavaServer Pages, 3rd Edition*. O'Reilly Media, Inc, 2003. [cit. 30/06/2024].

[68] Java Documentation – Default Methods. https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html. [cit. 30/06/2024].

[69] Java T Point – Java 9 Private Interface Methods. https://www.javatpoint.com/java-9-interface-private-methods. [cit. 30/06/2024].

[70] MSDN – C# default interface methods. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods. [cit. 30/06/2024].

[71] MSDN – Static abstract members in interfaces. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/static-abstracts-in-interfaces. [cit. 30/06/2024].

[72] MSDN – Nested Types (C# Programming Guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/nested-types. [cit. 30/06/2024].

[73] Java Documentation – Nested Classes. https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html. [cit. 30/06/2024].

[74] Java T Point – Why We Use Static Class in Java. https://www.javatpoint.com/why-we-use-static-class-in-java. [cit. 30/06/2024].

[75] Java T Point – Java Naming Convention. https://www.javatpoint.com/java-naming-conventions. [cit. 30/06/2024].

[76] MSDN – C# identifier naming rules and conventions. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names. [cit. 30/06/2024].

[77] Docker – How to Use the Alpine Docker Official Image. https://www.docker.com/blog/how-to-use-the-alpine-docker-official-image/. [cit. 30/06/2024].

[78] JetBrains Developer Ecosystem survey – Java. https://www.jetbrains.com/lp/devecosystem-2023/java/. [cit. 30/06/2024].

[79] Oracle Java SE Support Roadmap. https://www.oracle.com/java/technologies/java-se-support-roadmap.html. [cit. 30/06/2024].

[80] Openjdk. https://openjdk.org/. [cit. 30/06/2024].

[81] Amazon corretto. https://aws.amazon.com/corretto/?filtered-posts.sort-by=item.additionalFields.createdDate&filtered-posts.sort-order=desc. [cit. 30/06/2024].

[82] What Is the State of the Java Ecosystem in 2023. https://www.itprotoday.com/programming-languages/what-state-java-ecosystem-2023. [cit. 30/06/2024].

[83] Introduction to GraalVM. https://www.graalvm.org/22.0/docs/introduction/. [cit. 30/06/2024].

[84] Does Java 18 finally have a better alternative to JNI. https://developer.okta.com/blog/2022/04/08/state-of-ffi-java. [cit. 30/06/2024].

[85] GitHub – Java Native Access. https://github.com/java-native-access/jna. [cit. 30/06/2024].

[86] GitHub – Java Native Benchmark. https://github.com/zakgof/java-native-benchmark. [cit. 30/06/2024].

[87] GitHub – JNR-FFI. https://github.com/jnr/jnr-ffi. [cit. 30/06/2024].

[88] GitHub – JNR-FFI – Comparison to Similar Projects. https://github.com/jnr/jnr-ffi/blob/master/docs/ComparisonToSimilarProjects.md. [cit. 30/06/2024].

[89] OpenJDK – Project Panama: Interconnecting JVM and native code. https://openjdk.org/projects/panama/. [cit. 30/06/2024].

[90] JEP 424: Foreign Function & Memory API (Preview). https://openjdk.org/jeps/424. [cit. 30/06/2024].

[91] Jep 426: Vector api (fourth incubator). https://openjdk.org/jeps/426. [cit. 30/06/2024].

[92] The Vector API in JDK 17. https://www.youtube.com/watch?v=1JeoNr6-pZw. [cit. 30/06/2024].

[93] Java Documentation – Interface MemorySegment. https://docs.oracle.com/en/%2Fjava/%2Fjavase/%2F22/%2Fdocs/%2Fapi/%2F/%2F/java.base/java/lang/foreign/MemorySegment.html. [cit. 30/06/2024].

[94] Project Panama - Foreign Function & Memory API. https://youtu.be/kUFysMkMS00. [cit. 30/06/2024].

[95] Java Documentation – Class ByteBuffer. https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html. [cit. 30/06/2024].

[96] Java Documentation – Interface MemoryLayout. https://docs.oracle.com/en%2Fjava%2Fjavase%2F22%2Fdocs%2Fapi%2F%2F/java.base/java/lang/foreign/MemoryLayout.html. [cit. 30/06/2024].

[97] Java Documentation – Interface ValueLayout. https://docs.oracle.com/en%2Fjava%2Fjavase%2F22%2Fdocs%2Fapi%2F%2F/java.base/java/lang/foreign/ValueLayout.html. [cit. 30/06/2024].

[98] Java Restricted Methods. https://docs.oracle.com/en/java/javase/22/core/restricted-methods.html. [cit. 30/06/2024].

[99] Java Documentation – Interface Linker. https://docs.oracle.com/en%2Fjava%2Fjavase%2F22%2Fdocs%2Fapi%2F%2F/java.base/java/lang/foreign/Linker.html. [cit. 30/06/2024].

[100] Java Documentation – Class MethodHandle. https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html. [cit. 30/06/2024].

[101] Project Panama: Interconnecting the Java Virtual Machine and Native Code. https://youtu.be/M57l4DMcADg. [cit. 30/06/2024].

[102] Oracle Releases Java 22. https://www.oracle.com/pl/news/announcement/oracle-releases-java-22-2024-03-19/. [cit. 30/06/2024].

[103] Petr Tůma. Performance Evaluation of Computer Systems. https://d3s.mff.cuni.cz/teaching/nswi131/. [cit. 30/06/2024].

[104] Benchmarkdotnet. https://benchmarkdotnet.org/index.html. [cit. 30/06/2024].

[105] MSDN – Dynamic-link library search order. https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order. [cit. 30/06/2024].

[106] MSDN – Environment.SetEnvironmentVariable. https://learn.microsoft.com/en-us/dotnet/api/system.environment.setenvironmentvariable?view=net-8.0. [cit. 30/06/2024].

[107] Linux manual page – ld.so. https://man7.org/linux/man-pages/man8/ld.so.8.html. [cit. 30/06/2024].

[108] GitHub – Dotnet runtime issue #34711 – Using the LD LIBRARY PATH to load local libraries does not work on Linux. https://github.com/dotnet/runtime/issues/34711. [cit. 30/06/2024].

[109] Administering Data Integration for Oracle Enterprise Performance Management Cloud. https://docs.oracle.com/en/cloud/saas/enterprise-performance-management-common/diepm/epm_set_java_home_104x6dd63633_106x6dd6441c.html. [cit. 30/06/2024].

[110] J9 VM – Support for multiple Java virtual machines. https://www.ibm.com/docs/en/i/7.5?topic=api-support-multiple-java-virtual-machines. [cit. 30/06/2024].

[111] Android Developers – JNI tips. https://developer.android.com/training/articles/perf-jni. [cit. 30/06/2024].

[112] MSDN – ThreadLocal Class. https://learn.microsoft.com/en-us/dotnet/api/system.threading.threadlocal-1?view=net-8.0. [cit. 30/06/2024].

[113] Stephen Toub. Microsoft .NET Blog – ConfigureAwait FAQ. https://devblogs.microsoft.com/dotnet/configureawait-faq/, 2019. [cit. 30/06/2024].

[114] NuGet – UnmanagedThreadUtils. https://www.nuget.org/packages/UnmanagedThreadUtils/. [cit. 30/06/2024].

[115] MSDN – ThreadPool Class. https://learn.microsoft.com/en-us/dotnet/api/system.threading.threadpool?view=net-8.0. [cit. 30/06/2024].

[116] MSDN – Thread.IsThreadPoolThread Property. https://learn.microsoft.com/en-us/dotnet/api/system.threading.thread.isthreadpoolthread?view=net-8.0. [cit. 30/06/2024].

[117] MSDN – SafeHandle Class. https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.safehandle?view=net-8.0. [cit. 30/06/2024].

[118] MSDN – System.Runtime.InteropServices.SafeHandle class. https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-runtime-interopservices-safehandle. [cit. 30/06/2024].

[119] Java Documentation – Class ArrayList. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. [cit. 30/06/2024].

[120] J9 VM – Copying and pinning. https://www.ibm.com/docs/kk/sdk-java-technology/8?topic=jni-copying-pinning. [cit. 30/06/2024].

[121] MSDN – Differences Between C++ Templates and C# Generics (C# Programming Guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/differences-between-cpp-templates-and-csharp-generics. [cit. 30/06/2024].

[122] GitHub – Roslyn – Incremental Generators. https://github.com/dotnet/roslyn/blob/main/docs/features/incremental-generators.md. [cit. 30/06/2024].

[123] MSDN – Extending Partial Methods. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/extending-partial-methods. [cit. 30/06/2024].

[124] Andrew Lock. .NET Escapades – Customising generated code with marker attributes. https://andrewlock.net/creating-a-source-generator-part-4-customising-generated-code-with-marker [cit. 30/06/2024].

[125] MSDN – Partial Classes and Methods (C# Programming Guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods. [cit. 30/06/2024].

[126] MSDN – Error BC30045. https://learn.microsoft.com/en-us/dotnet/visual-basic/misc/bc30045. [cit. 30/06/2024].

[127] GitHub – Dotnet csharplang issue #34121 – Please allow partial properties. https://github.com/dotnet/csharplang/discussions/3412. [cit. 30/06/2024].

[128] MSDN – Introduction to the MVVM Toolkit. https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/. [cit. 30/06/2024].

[129] MSDN – ObservableProperty attribute. https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/generators/observableproperty. [cit. 30/06/2024].

[130] Java Documentation – java.lang.String. https://docs.oracle.com/javase/8/docs/api/java/lang/String.html. [cit. 30/06/2024].

[131] MSDN – Strings and string literals. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/. [cit. 30/06/2024].

[132] MSDN – String Class. https://learn.microsoft.com/en-us/dotnet/api/system.string?view=net-8.0. [cit. 30/06/2024].

[133] MSDN – Tutorial: Explore C# 11 feature - static virtual members in interfaces. https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/static-virtual-interface-members. [cit. 30/06/2024].

[134] The class File Format. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html. [cit. 30/06/2024].

[135] Andrew Lock. .NET Escapades – Avoiding performance pitfalls in incremental generators. https://andrewlock.net/creating-a-source-generator-part-9-avoiding-performance-pitfalls-in-incre [cit. 01/07/2024].

[136] MSDN – EqualityComparer<T> Class. https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.equalitycomparer-1?view=net-8.0. [cit. 01/07/2024].

[137] MSDN – IncrementalValueProviderExtensions.WithComparer Method. https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.incrementalvalueproviderextensions.withcomparer?view=roslyn-dotnet-4.7.0. [cit. 01/07/2024].

[138] MSDN – Records (C# reference). https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record. [cit. 01/07/2024].

[139] GitHub – andrewlock, EquatableArray.cs. https://github.com/andrewlock/StronglyTypedId/blob/master/src/StronglyTypedIds/EquatableArray.cs. [cit. 01/07/2024].

[140] .NET Escapades – Testing your incremental generator pipeline outputs are cacheable. https://andrewlock.net/creating-a-source-generator-part-10-testing-your-incremental-generator-pi [cit. 01/07/2024].

[141] MSDN – Unmanaged types (C# reference). https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/unmanaged-types. [cit. 30/06/2024].

[142] MSDN – Source Generators. https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/source-generators-overview. [cit. 30/06/2024].

[143] Phillip Carter. Microsoft .NET Blog – Introducing C# Source Generators. https://devblogs.microsoft.com/dotnet/introducing-c-source-generators/, 2020. [cit. 30/06/2024].

[144] MSDN – SyntaxValueProvider. https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.syntaxvalueprovider?view=roslyn-dotnet-4.7.0. [cit. 30/06/2024].

[145] MSDN – IncrementalValuesProvider. https://learn.microsoft.com/en-us/dotnet/api/microsoft.codeanalysis.incrementalvaluesprovider-1?view=roslyn-dotnet-4.7.0. [cit. 30/06/2024].

[146] GitHub Actions documentation. https://docs.github.com/en/actions. [cit. 30/06/2024].

[147] MSDN – nuget.config reference. https://learn.microsoft.com/en-us/nuget/reference/nuget-config-file. [cit. 30/06/2024].

[148] MSDN – AppContext.BaseDirectory. https://learn.microsoft.com/en-us/dotnet/api/system.appcontext.basedirectory?view=net-8.0. [cit. 30/06/2024].

[149] Java T Point – PDFBox Tutorial. https://www.javatpoint.com/pdfbox-tutorial. [cit. 30/06/2024].

[150] Java Documentation – JAR File Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html. [cit. 30/06/2024].

[151] Incremental Roslyn Source Generators: Using Additional Files. https://www.thinktecture.com/en/net/roslyn-source-generators-using-additional-files/. [cit. 30/06/2024].

[152] GitHub – Kaitai Struct: runtime library for C# / .NET. https://github.com/kaitai-io/kaitai_struct_csharp_runtime. [cit. 30/06/2024].

[153] Kaitai Struct – .class file format: C# parsing library. https://formats.kaitai.io/java_class/csharp.html. [cit. 30/06/2024].

A. Attachments

A.1 Overview of electronic attachments

.
- **bin** - binary version of the implemented solution
- **src** - source code of the implemeted solution and demo examples
  - **JavaInterop** - source code of the solution
    - **BuildScripts** - scripts used during a build
    - **Docker** - scripts and dockerfiles used for testing in GitHub actions
    - **JavaInterop** - JNI part of the interoperability library
    - **JavaInterop.Benchmarks** - benchmarks
    - **JavaInterop.Common** - shared part of interoperability library
    - **JavaInterop.DebugConsoleApp** - debugging console app, mainly enabeling stopping on breakpoints in incremental source generator
    - **JavaInterop.NuGet** - project enabling packaging solution as NuGet package
    - **JavaInterop.Panama** - Panama part of the interoperability library
    - **JavaInterop.Proxies** - proxy classes
    - **JavaInterop.SourceGenerator** - the main part of incremental source generator
    - **JavaInterop.SourceGenerator.Common** - shared part of incremental source generator
    - **JavaInterop.SourceGenerator.JNI** - JNI focused part of incremental source generator
    - **JavaInterop.SourceGenerator.Panama** - Panama focused part of incremental source generator
    - **JavaInterop.Proxies** - JNI based proxy classes used for testing
    - **JavaInterop.Proxies.Panama** - Panama based proxy classes used for testing
    - **JavaInterop.UnitTests** - unit tests
    - **JavaInterop.UnitTests.Panama** - Panama unit tests
    - **javasources** - Java source code used by the solution
      - **panama** - helpers usage of Project Panama
      - **testclasses** - Java classes used to test the solution
  - **LanatraDemo** - source code of user documentation tutorials
    - **artifacts** - folder containing solution packaged as NuGet package
    - **javalib** - custom Java classes used in demo examples
    - **LanatraDemo** - main demo examples
    - **LanatraDemo.Panama** - Panama based demo examples
    - **pdfFiles** - .pdf files used in examples
- **tex** - folder containing LaTeXsource code of the thesis text
  - **images** - thesis text images
- **readme.txt** - file describing structure of thesis attachments

└── **thesis.pdf** - text of this thesis in PDF/A format

A.2 Results of JNI vs Project Panama benchmark

Method	Mean	StdDev	Allocated
VoidParamlessMethodJni	45.52 ns	0.237 ns	-
VoidParamlessMethodPanama	20.35 ns	0.032 ns	-
IntParamlessMethodJni	67.90 ns	0.472 ns	-
IntMethodIntParamJni	74.83 ns	0.261 ns	32 B
FiveIntParamsJni	98.19 ns	0.371 ns	64 B
TenIntParamsJni	135.71 ns	0.782 ns	104 B
IntParamlessMethodPanama	20.05 ns	0.129 ns	-
IntMethodIntParamPanama	20.36 ns	0.024 ns	-
FiveIntParamsPanama	19.14 ns	0.102 ns	-
TenIntParamsPanama	22.35 ns	0.092 ns	-
LongParamlessMethodJni	67.18 ns	0.196 ns	-
LongMethodLongParamJni	76.51 ns	0.329 ns	32 B
FiveLongParamsJni	91.91 ns	0.330 ns	64 B
TenLongParamsJni	144.08 ns	0.774 ns	104 B
LongParamlessMethodPanama	17.79 ns	0.175 ns	-
LongMethodLongParamPanama	20.73 ns	0.009 ns	-
FiveLongParamsPanama	21.81 ns	0.013 ns	-
TenLongParamsPanama	22.67 ns	0.113 ns	-
FloatParamlessMethodJni	70.73 ns	0.484 ns	-
FloatMethodFloatParamJni	78.06 ns	0.293 ns	32 B
FiveFloatParamsJni	108.87 ns	0.833 ns	64 B
TenFloatParamsJni	138.43 ns	0.367 ns	104 B
FloatParamlessMethodPanama	19.64 ns	0.014 ns	-
FloatMethodFloatParamPanama	22.42 ns	0.020 ns	-
FiveFloatParamsPanama	22.73 ns	0.036 ns	-
TenFloatParamsPanama	23.12 ns	0.011 ns	-
DoubleParamlessMethodJni	71.78 ns	0.181 ns	-
DoubleMethodDoubleParamJni	76.73 ns	0.323 ns	32 B
FiveDoubleParamsJni	96.89 ns	0.478 ns	64 B
TenDoubleParamsJni	145.46 ns	1.191 ns	104 B
DoubleParamlessMethodPanama	20.92 ns	0.037 ns	-
DoubleMethodDoubleParamPanama	21.53 ns	0.019 ns	-
FiveDoubleParamsPanama	22.68 ns	0.045 ns	-
TenDoubleParamsPanama	23.90 ns	0.030 ns	-
BoolParamlessMethodJni	68.60 ns	0.118 ns	-
BoolMethodBoolParamJni	84.37 ns	0.310 ns	32 B
FiveBoolParamsJni	121.82 ns	0.788 ns	64 B
TenBoolParamsJni	146.51 ns	0.399 ns	104 B
BoolParamlessMethodPanama	21.13 ns	0.013 ns	-
BoolMethodBoolParamPanama	21.22 ns	0.086 ns	-
FiveBoolParamsPanama	23.32 ns	0.043 ns	-
TenBoolParamsPanama	22.27 ns	0.008 ns	-