Dynamic features of Java

https://www.cesarsotovalero.net/blog/the-dynamic-features-of-java.html#challenges-of-using-dynamic-features

The existence of dynamic features built-in within the language allows Java developers to dynamically transform their program executions at runtime. For example, using the Java Reflection API, one can inspect and interact with otherwise static language constructs such as classes, fields, and methods, e.g., to instantiate objects, set fields and invoke methods. These dynamic language features are helpful, but their usage also hinders the accuracy of static analysis tools. This is due to the undecidability of resolving and analyzing code that is not reachable at compile time. As I mentioned in a previous blog post, the promising **GraalVM compiler** performs Ahead of Time Compilation (AOT) through static analysis on Java bytecode. However, the presence of dynamic features in most Java programs is a fundamental challenge for GraalVM. Consequently, recognizing these features is key to understand the current limitations of AOT. This blog post covers the fundamental dynamic features of Java and the reasons why they pose a significant challenge for GraalVM and static analysis tools in general.

Dynamic Class Loading

Java makes possible declaring [custom class loaders](Class loaders.md). We can use a custom `ClassLoader` to compile a `.java` source file from a file system and then load the compiled `.class` at runtime. This mechanism allows programmers to load classes from arbitrary locations, e.g., from an external file system or over the network.

```
public class CustomClassLoader extends ClassLoader {

  @Override
  protected Class<?> findClass(String name) throws ClassNotFoundException {
    byte[] content;
    try {
      content = compile(this.getClass().getClassLoader(), name);
    } catch (Exception e) {
      throw new ClassNotFoundException();
    }
    return defineClass(name, content, 0, content.length);
  }

  private byte[] compile(ClassLoader classLoader, String name) throws IOException
{
    String path = name.replace(".", "/");
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    compiler.run(null, null, null, classLoader.getResource(path +
".java").getFile());
    File file = new File(classLoader.getResource(path + ".class").getFile());
    FileInputStream input = new FileInputStream(file);
    byte[] content = new byte[(int) file.length()];
    input.read(content);
    input.close();
```

```
    return content;
  }
}
```

This can support a class being loaded outside the classpath (a file in the host file system), therefore known only at **run-time**. Also the name of the class and the method to call can be determined at run-time as shown by:

```
@Test
public void findClass() {
  CustomClassLoader ccl = new CustomClassLoader();
  try {
    Class<?> target = ccl.findClass("dynamicClassLoading.Target");
    int magic = (Integer) target.getMethod("magic").invoke(target.newInstance());
    Assert.assertEquals(42, magic);
  } catch (Exception e) {
    fail("ClassNotFoundException");
  }
}
```

Dynamic Proxies

Dynamic proxies allow one single class with one single method to service multiple method calls to arbitrary classes with an arbitrary number of methods. It is **like a facade**, but one that **can pretend to be an implementation of any interface**. Under the cover, it routes all method invocations to a single handler: the `invoke()` method.

A proxy class is a **class created at runtime that implements a specified list of interfaces, known as proxy interfaces**. A proxy instance is an instance of a proxy class. Each proxy instance has an associated **invocation handler**, which implements the interface `InvocationHandler`. When a method is invoked on a proxy instance, **the method invocation is encoded and dispatched to the invoke method of its invocation handler** The invocation handler processes the encoded method invocation as appropriate and the result that it returns will be returned as **the result of the method invocation on the proxy instance**.

We have the follwoing interface/class we want to proxy.

```
public interface MyInterface {
  String foo(String s);
}

public class MyInterfaceImpl implements MyInterface {
  @Override
  public String foo(String s) {
    return s + " from foo(String)";
  }
}
```

We create an `InvocationHandler` implementation

```
public class MyProxyInvocationHandler implements InvocationHandler {
  @Override
  public Object invoke(Object proxy, Method method, Object[] arg) {
    return target((String) arg[0]);
  }
  public String target(String s) {
    return s + " from target(String)";
  }
}
```

And an application that creates a proxy as an instance of `MyInterface` and that a run-time intercepts the call to the proxy

```
public class DynamicProxyApplication {
  public String execute() {
    MyInterface proxy = (MyInterface) Proxy.newProxyInstance( //it is created at
run-time
        MyInterface.class.getClassLoader(), //that is resolved at run time
        new Class[]{MyInterface.class},
        new MyProxyInvocationHandler()
    );
    return proxy.foo("hello");
  }
}
```

The test confirms that the result is from the proxy and not from the `MyInterfaceImpl` class.

```
@Test
public void testExecute() {
  DynamicProxyApplication dpa = new DynamicProxyApplication();
  Assert.assertEquals("hello from target(String)", dpa.execute());
}
```

The `invokedynamic`

The JVM invokedynamic instruction was introduced in Java 7. It makes it possible to **resolve method calls dynamically at runtime**. This gives the user more control over the method dispatch process by using a user-defined bootstrap method that computes the call target. While the original motivation behind `invokedynamic` was to provide support for dynamic languages like Ruby, its main application (in the JDK 8) is to **provide support for lambda expressions**.

```
public class LambdaExample {
   private static final String HELLO = "Hello";
```

```
    public static void main(String[] args) throws Exception {
       Runnable r = () -> System.out.println(HELLO);
       Thread t = new Thread(r);
       t.start();
       t.join();
    }
}
```

The lambda expression is assigned to a variable of type `Runnable`. This means that the **lambda evaluates to a reference to an object that has a type that is compatible with `Runnable`**.

Essentially, this object's type will be some subclass of Object that has defined one extra method (and has no fields). The extra method is understood to be the `run()` method expected by the `Runnable` interface.

Before Java 8, such an object was represented only by an instance of a **concrete anonymous class that implemented `Runnable`**. In fact, in the initial prototypes of Java 8 lambdas, **inner classes were used as the implementation technology**.

The long-range future roadmap for the JVM could contain future versions where more sophisticated representations of lambdas could be possible. Fixing the representation to use explicit inner classes would prevent a different representation from being used by a future version of the platform. This is undesirable and so, instead, Java 8 and Java 9 use a more sophisticated technique than hardcoding inner classes. The bytecode for the previous lambda example is as follows:

```
public static void main(java.lang.String[]) throws java.lang.Exception;
   Code:
      0: invokedynamic #2, 0 // InvokeDynamic
                                          // #0:run:()Ljava/lang/Runnable;
      5: astore_1
      6: new #3 // class java/lang/Thread
      9: dup
      10: aload_1
      11: invokespecial #4 // Method java/lang/Thread."<init>":
                                       // (Ljava/lang/Runnable;)V
      14: astore_2
      15: aload_2
      16: invokevirtual #5 // Method java/lang/Thread.start:()V
      19: aload_2
      20: invokevirtual #6 // Method java/lang/Thread.join:()V
      23: return
```

`invokedynamic` also benefits dynamic language implementers by supporting dynamically changing call site targets -- a call site, more specifically, a dynamic call site is an invokedynamic instruction. Furthermore, because the JVM internally supports invokedynamic, this instruction can be better optimized by the JIT compiler.

JNI - Java Native Interface

Sometimes we need to use code that's natively-compiled for a specific hardware architecture.

There could be many reasons for needing to use native code, for example:

- To handle some hardware.
- To improve the performance of a very demanding process.
- To reuse an existing native library instead of rewriting it in Java. For these purposes, the JDK introduces the **Java Native Interface (JNI)** as a foreign function interface. It works as a **bridge between the bytecode running in the Java Virtual Machine (JVM) and the native code**. Thus, JNI allows code running on the JVM to call and be called by native applications. Using JNI, one can call methods written in C/C++ or even access assembly language functions from Java.

Native methods are declared using the `native` keyword. They are called just like regular Java methods. `System.loadLibrary()` is used to **load the shared native library** (e.g, a .so file on Linux or .dll file on Windows).

Here's a simple example:

```
package com.example;

public class JniExample {
    static {
        System.loadLibrary("native"); //load the shared native libs
    }
    public static void main(String[] args) {
        new JniExample().sayHello();
    }
    private native void sayHello();
}
```

Create the C definition `.h` file

```
javac -h . JniExample.java
```

The output of the `JniExample.h` is

```
JNIEXPORT void JNICALL Java_com_example_JniExample_sayHello(JNIEnv *, jobject);
```

The two parameters passed to our function; a pointer to the current `JNIEnv`; and also the Java object that the method is attached to, the instance of our `HelloWorldJNI` class.

Create the implementation `.cpp` file

Now, we have to create a new .cpp file for the implementation of the `sayHello` function.

```
JNIEXPORT void JNICALL Java_com_example_JniExample_sayHello(JNIEnv* env, jobject
thisObject) {
  std::cout << "Hello from C++ !!" << std::endl;
}
```

Compile the `.cpp` into the object `.o` file

```
g++ -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/darwin JniExample.cpp -
o JniExample.o
```

Add compiled file into a dynamic library

```
g++ -dynamiclib -o libnative.dylib JniExample.o -lc
```

Run the Java code

```
java -cp . -Djava.library.path=. com.example.JniExample
```

Pitfalls using *JNI*

No platform portability

The native code makes the application depend on the underlying compiler platform. In case of requiring the support for different Operating Systems, the application needs to be compiled separately for each of those. Thus, using JNI means losing the **"write once, run anywhere"** feature of Java.

Difficult to implement

JNI is a low-level interface and is not easy to use. For example, sometimes there isn't even a direct conversion between types, so we'll have to write our equivalent.

Difficult to debug

JNI adds a layer of complexity to the application. It's difficult to debug runtime errors. A simple error can lead to a complete system crash (e.g., a segmentation fault).

No garbage collection

From the native side, resources should be handled manually. Thus, in case of forgetting to free memory, leaks might occur, hindering performance and compromising security.

Extra layer of communication

JNI adds a costly layer of communication between the code running into the JVM and the native code. JNI applications need to convert the data exchanged between Java and C++ in a marshaling/unmarshaling process.

No thread safety

JNI is not thread-safe. Thus, it is not possible to use the same JNI environment (JNIEnv *) from multiple threads.

Java Reflection API

The Java Reflection API allows inspecting, modifying, and instantiating otherwise static language elements such as classes, fields, and methods at runtime. This dynamic feature is handy when we don't know their names at compile time. For example, to **dynamically instantiate objects, set fields, or invoke methods**.

For example, we can **instantiate classes by calling constructors of any class and even instantiate objects at runtime**. This is made possible by the `java.lang.reflect.Constructor` class.

The following class `Instantiation` illustrates the use of **dynamic class instantiation** via reflection:

```
public class Instantiation {

  /**
   * Interprocedural instantiation.
   * The class name is supplied externally.
   */
  public void instantiateInterprocedural() throws Exception {
    BufferedReader br = new BufferedReader(
        new FileReader(this.getClass().getClassLoader()
          .getResource("class.txt").getFile())
    );
    Class<?> c = Class.forName(br.readLine());
    c.getConstructor(Instantiation.class).newInstance(this);
  }

  /**
   * Intraprocedural instantiation.
   * The class name is supplied internally.
   */
  public void instantiateIntraprocedural() throws Exception {
    String className = new StringBuilder("tegraT").reverse().toString();
    Class<?> c = Class.forName(
      "se.kth.instantiation.DynamicInstantiation$" + className
    );
    c.getConstructor(Instantiation.class).newInstance(this);
  }

  class Target {
    public Target() {}
    public Target(String c) {}
```

```
    }
}
```

We can invoke methods of any class (even `private` methods) at runtime. This is made possible by the `java.lang.reflect.Method` class.

```
public class Invocation {

  /**
   * Interprocedural invocation.
   * The method name is supplied externally.
   */
  public void invokeMethodInterprocedural() throws Exception {
    BufferedReader br = new BufferedReader(
        new FileReader(this.getClass().getClassLoader()
          .getResource("method.txt").getFile())
    );
    String methodName = br.readLine();
    Method m = Invocation.class.getDeclaredMethod(methodName, null);
    m.invoke(this, null);
  }

  /**
   * Intraprocedural invocation.
   * The method name is provided through a series of transformations.
   */
  public void invokeMethodIntraprocedural() throws Exception {
    String methodName = new StringBuilder("TEGRAT")
      .reverse().toString().toLowerCase();
    Method m = Invocation.class.getDeclaredMethod(methodName, null);
    m.invoke(this, null);
  }

  public void target() {}
  public void target(String a) {}
}
```

Deserialization

Serialization converts an object into a byte stream (i.e., a sequence of bytes). Deserialization is the opposite: it converts a byte stream into an object. Serialized objects are typically used to save the application's state, store objects in a database, or transfer them over a network. We can serialize objects on one platform and deserialize them on another. The serializability of a class is enabled by implementing the `java.io.Serializable` interface.

```
public class Deserialization implements Serializable {
```

```
  public byte[] serialize(Object obj) throws Exception {
    ByteArrayOutputStream ba = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(ba);
    oos.writeObject(obj);
    oos.close();
    return ba.toByteArray();
  }

  public void deserialize(byte[] obj) throws Exception {
    ObjectInputStream ois = new ObjectInputStream(
      new ByteArrayInputStream(obj)
    );
    HelloInterface foo = (HelloInterface) ois.readObject();
    foo.hello();
    ois.close();
  }

  interface HelloInterface {
    void hello();
  }

  class Hello implements HelloInterface, Serializable {
    @Override
    public void hello() {
      System.out.println("hello");
    }
  }
}
```

Deserialization is a potentially dangerous operation. For example, the notable Apache log4j library uses deserialization to read configuration files. This use of deserialization was exploited (CVE-2022-23302) in 2022, causing a global disturbance on most Java based systems.

As a rule of thumb, deserialization of untrusted data is inherently dangerous and should be avoided. Untrusted data should be carefully validated according to the “Serialization and Deserialization” section of the Secure Coding Guidelines for Java SE. As pointed out by Oracle, Serialization Filtering describes best practices for defensive use of serial filters.