**How to Use Reflection**

The first step in any reflective operation is to get a Class object representing the

type to be operated on. From this, other objects, representing fields, methods, or

constructors can be accessed, and applied to instances of the unknown type.

To get an instance of an unknown type, the simplest way is to use the no-arg constructor,which is made available directly via the Class object:

**Class<?> clz = getSomeClassObject();**

**Object rcvr = clz.newInstance();**

For constructors that take arguments, you will have to look up the precise constructor

needed, represented as a Constructor object.

The Method objects are one of the most commonly used objects provided by Reflection.

We’ll discuss them in detail—the Constructor and Field objects are similar in

many respects.

**Method objects**

A class object contains a Method object for each method on the class. These are

lazily created after classloading, and so aren’t immediately visible in an IDE’s

debugger.

Let’s look at the source code from Method to see what information and metadata is

held for each method:

**private Class<?> clazz;**

**private int slot;**

***// This is guaranteed to be interned by the VM in the 1.4***

***// reflection implementation***

**private String name;**

**private Class<?> returnType;**

**private Class<?>[] parameterTypes;**

**private Class<?>[] exceptionTypes;**

**private int modifiers;**

***// Generics and annotations support***

**private transient String signature;**

***// Generic info repository; lazily initialized***

**private transient MethodRepository genericInfo;**

**private byte[] annotations;**

**private byte[] parameterAnnotations;**

**private byte[] annotationDefault;**

**private volatile** MethodAccessor methodAccessor;

This provides all available information, including the exceptions the method can

throw, annotations (with a retention policy of RUNTIME), and even the generics

information that was otherwise removed by javac.

We can explore the metadata contained on the Method object, by calling accessor

methods, but by far the single biggest use case for Method is reflexive invocation.

The methods represented by these objects can be executed by reflection using the

invoke() method on Method. An example of invoking hashCode() on a String

object follows:

**Object rcvr = "a";**

**try {**

**Class<?>[] argTypes = new Class[] { };**

**Object[] args = null;**

**Method meth = rcvr.getClass().getMethod("hashCode", argTypes);**

**Object ret = meth.invoke(rcvr, args);**

**System.out.println(ret);**

**} catch (IllegalArgumentException | NoSuchMethodException |**

**SecurityException e) {**

**e.printStackTrace();**

**} catch (IllegalAccessException | InvocationTargetException x) {**

**x.printStackTrace();**

**}**

To get the Method object we want to use, we call getMethod() on the class object.

This will return a reference to a Method corresponding to a public method on the

class.

Note that the static type of rcvr was declared to be Object. No static type information

was used during the reflective invocation. The invoke() method also returns

Object, so the actual return type of hashCode() has been autoboxed to Integer.

This autoboxing is one of the aspects of Reflection where some of the slight awkwardness

of the API can be seen—which is the subject of the next section.

**Problems with Reflection**

Java’s Reflection API is often the only way to deal with dynamically loaded code, but

there are a number of annoyances in the API that can make it slightly awkward to

deal with:

• Heavy use of Object[] to represent call arguments and other instances.

• Also Class[] when talking about types.

• Methods can be overloaded on name, so we need an array of types to distinguish

between methods.

• Representing primitive types can be problematic—we have to manually box

and unbox.

void is a particular problem—there is a void.class, but it’s not used consistently.

Java doesn’t really know whether void is a type or not, and some methods in the

Reflection API use null instead.

This is cumbersome, and can be error prone—in particular, the slight verbosity of

Java’s array syntax can lead to errors.

**One further problem is the treatment of non-public methods. Instead of using get**

**Method(), we must use getDeclaredMethod() to get a reference to a non-public**

**method, and then override the Java access control subsystem with setAccessible()**

to allow it to be executed:

**public class MyCache {**

**private void flush() {**

***// Flush the cache...***

**}**

**}**

**Class<?> clz = MyCache.class;**

**try {**

**Object rcvr = clz.newInstance();**

**Class<?>[] argTypes = new Class[] { };**

**Object[] args = null;**

**Method meth = clz.getDeclaredMethod("flush", argTypes);**

**meth.setAccessible(true);**

**meth.invoke(rcvr, args);**

**} catch (IllegalArgumentException | NoSuchMethodException |**

**InstantiationException | SecurityException e) {**

**e.printStackTrace();**

**} catch (IllegalAccessException | InvocationTargetException x) {**

**x.printStackTrace();**

**}**

However, it should be pointed out that reflection always involves unknown information.

To some degree, we just have to live with some of this verbosity as the price

of dealing with reflective invocation, and the dynamic, runtime power that it gives

to the developer.

As a final example in this section, let’s show how to combine reflection with custom

classloading to inspect a class file on disk and see if it contains any deprecated

methods (these should be marked with @Deprecated):

**public class CustomClassloadingExamples {**

**public static class DiskLoader extends ClassLoader {**

**public DiskLoader() {**

**super(DiskLoader.class.getClassLoader());**

**}**

**public Class<?> loadFromDisk(String clzName)**

**throws IOException {**

**byte[] b = Files.readAllBytes(Paths.get(clzName));**

**return defineClass(null, b, 0, b.length);**

**}**

**}**

**public void findDeprecatedMethods(Class<?> clz) {**

**for (Method m : clz.getMethods()) {**

**for (Annotation a : m.getAnnotations()) {**

**if (a.annotationType() == Deprecated.class) {**

**System.out.println(m.getName());**

**}**

**}**

**}**

**}**

**public static void main(String[] args)**

**throws IOException, ClassNotFoundException {**

**CustomClassloadingExamples rfx =**

**new CustomClassloadingExamples();**

**if (args.length > 0) {**

**DiskLoader dlr = new DiskLoader();**

**Class<?> clzToTest = dlr.loadFromDisk(args[0]);**

**rfx.findDeprecatedMethods(clzToTest);**

**}**

**}**

**}**

**null;**

**}**

**return null;**

**}**

**};**

**Channel c =**

**(Channel) Proxy.newProxyInstance(Channel.class.getClassLoader(),**

**new Class[] { Channel.class }, h);**

**c.isOpen();**

**c.close();**

**Dynamic Proxies**

One last piece of the Java Reflection story is the creation of dynamic proxies. These

are classes (which extend java.lang.reflect.Proxy) that implement a number of

interfaces. The implementing class is constructed dynamically at runtime, and forwards all calls to an invocation handler object:

**InvocationHandler h = new InvocationHandler() {**

**@Override**

**public Object invoke(Object proxy, Method method, Object[] args)**

**throws Throwable {**

**String name = method.getName();**

**System.out.println("Called as: "+ name);**

**switch (name) {**

**case "isOpen":**

**return false;**

**case "close":**

**return null;**

**}**

**return null;**

**}**

**};**

**Channel c =**

**(Channel) Proxy.newProxyInstance(Channel.class.getClassLoader(),**

**new Class[] { Channel.class }, h);**

**c.isOpen();**

**c.close();**

Proxies can be used as stand-in objects for testing (especially in test mocking

approaches).

Another use case is to provide partial implementations of interfaces, or to decorate

or otherwise control some aspect of delegation:

**public class RememberingList implements InvocationHandler {**

**private final List<String> proxied = new ArrayList<>();**

**@Override**

**public Object invoke(Object proxy, Method method, Object[] args)**

**throws Throwable {**

**String name = method.getName();**

**switch (name) {**

**case "clear":**

**return null;**

**case "remove":**

**case "removeAll":**

**return false;**

**}**

**return method.invoke(proxied, args);**

Proxies are an extremely powerful and flexible capability that are used within many

Java frameworks.

Method Handles

In Java 7, a brand new mechanism for introspection and method access was introduced.

This was originally designed for use with dynamic languages, which may

need to participate in method dispatch decisions at runtime. To support this at the

JVM level, the new invokedynamic bytecode was introduced. This bytecode was not

used by Java 7 itself, but with the advent of Java 8, it was extensively used in both

lambda expressions and the Nashorn JavaScript implementation.

Even without invokedynamic, the new Method Handles API is comparable in power

to many aspects of the Reflection API—and can be cleaner and conceptually simpler

to use, even standalone. It can be thought of as Reflection done in a safer, more

modern way.

**MethodType**

In Reflection, method signatures are represented as Class[]. This is quite cumbersome.

By contrast, method handles rely on MethodType objects. These are a typesafe

and object-orientated way to represent the type signature of a method.

They include the return type and argument types, but not the receiver type or name

of the method. The name is not present as this allows any method of the correct signature

to be bound to any name (as per the functional interface behavior of lambda

expressions).

A type signature for a method is represented as an immutable instance of Method

Type, as acquired from the factory method MethodType.methodType(). For example:

**MethodType m2Str = MethodType.methodType(String.class); *// toString()***

***// Integer.parseInt()***

**MethodType mtParseInt =**

**MethodType.methodType(Integer.class, String.class);**

***// defineClass() from ClassLoader***

**MethodType mtdefClz = MethodType.methodType(Class.class, String.class,**

**byte[].class, int.class,**

**int.class);**

This single piece of the puzzle provides significant gains over Reflection, as it makes

method signatures significantly easier to represent and discuss. The next step is to

acquire a handle on a method. This is achieved by a lookup process.

**Method Lookup**

Method lookup queries are performed on the class where a method is defined, and

are dependent on the context that they are executed from. In this example, we can

see that when we attempt to lookup the protected Class::defineClass() method

from a general look up context, we fail to resolve it with an IllegalAccessExcep

tion, as the protected method is not accessible:

**public static void lookupDefineClass(Lookup l) {**

**MethodType mt = MethodType.methodType(Class.class, String.class,**

**byte[].class, int.class,int.class);**

**try {**

**MethodHandle mh =**

**l.findVirtual(ClassLoader.class, "defineClass", mt);**

**System.out.println(mh);**

**} catch (NoSuchMethodException | IllegalAccessException e) {**

**e.printStackTrace();**

**}**

**}**

**Lookup l = MethodHandles.lookup();**

**lookupDefineClass(l);**

We always need to call MethodHandles.lookup()—this gives us a lookup context

object based on the currently executing method.

Lookup objects have several methods (which all start with find) declared on them

for method resolution. These include findVirtual(), findConstructor(), and

findStatic().

One big difference between the Reflection and Method Handles APIs is access control.

A Lookup object will only return methods that are accessible to the context

where the lookup was created—and there is no way to subvert this (no equivalent of

Reflection’s setAccessible() hack).

Method handles therefore always comply with the security manager, even when the

equivalent reflective code does not. They are access-checked at the point where the

lookup context is constructed—the lookup object will not return handles to any

methods to which it does not have proper access.

The lookup object, or method handles derived from it, can be returned to other

contexts, including ones where access to the method would no longer be possible.

Under those circumstances, the handle is still executable—access control is checked

at lookup time, as we can see in this example:

**public class SneakyLoader extends ClassLoader {**

**public SneakyLoader() {**

**super(SneakyLoader.class.getClassLoader());**

**}**

**public Lookup getLookup() {**

**return MethodHandles.lookup();**

**}**

**}**

**SneakyLoader snLdr = new SneakyLoader();**

**l = snLdr.getLookup();**

**lookupDefineClass(l);**

With a Lookup object, we’re able to produce method handles to any method we have

access to. We can also produce a way of accessing fields that may not have a method

that gives access. The findGetter() and findSetter() methods on Lookup produce

method handles that can read or update fields as needed.

**Invoking Method Handles**

A method handle represents the ability to call a method. They are strongly typed

and as typesafe as possible. Instances are all of some subclass of

java.lang.invoke.MethodHandle, which is a class that needs special treatment

from the JVM.

There are two ways to invoke a method handle—invoke() and invokeExact().

Both of these take the receiver and call arguments as parameters. invokeExact()

tries to call the method handle directly as is, whereas invoke() will massage call

arguments if needed.

In general, invoke() performs an asType() conversion if necessary—this converts

arguments according to these rules:

A primitive argument will be boxed if required.

• A boxed primitive will be unboxed if required.

• Primitives will be widened is necessary.

• A void return type will be massaged to 0 or null, depending on whether the

expected return was primitive or of reference type.

• null values are passed through, regardless of static type.

With these potential conversions in place, invocation looks like this:

**Object rcvr = "a";**

**try {**

**MethodType mt = MethodType.methodType(int.class);**

**MethodHandles.Lookup l = MethodHandles.lookup();**

**MethodHandle mh = l.findVirtual(rcvr.getClass(), "hashCode", mt);**

**int ret;**

**try {**

**ret = (int)mh.invoke(rcvr);**

**System.out.println(ret);**

**} catch (Throwable t) {**

**t.printStackTrace();**

**}**

**} catch (IllegalArgumentException |**

**NoSuchMethodException | SecurityException e) {**

**e.printStackTrace();**

**} catch (IllegalAccessException x) {**

**x.printStackTrace();**

**}**

Method handles provide a clearer and more coherent way to access the same

dynamic programming capabilities as Reflection. In addition, they are designed to

work well with the low-level execution model of the JVM and thus hold out the

promise of much better performance than Reflection can provide.