

COMP 150-SEN

Software Engineering Foundations

Reflection

Spring 2019

(Condensed and modified from Steve Odendahl, *Reflection: Java Technology's Secret Weapon*, JavaOne 2002)

Introduction

- Reflection makes classes, methods, and fields into *first class objects* that exist at run time
 - Can determine fields and methods of class
 - Can instantiate class given a `String` containing the name
 - Can invoke methods given a `String` with method name
 - Can create classes (in a certain way) at runtime
- Reflection does not
 - Add any expressive power to the language, in theory
 - With or without reflection, Java is Turing complete
 - Solve every problem
 - Reflection is a sledgehammer that can do certain things that are difficult to achieve any other way
 - But it should be used sparingly, preferably not at all

java.lang.Class

- An instance of `Class` represents a class
 - You can use it to get information about its fields/methods
 - Most uses of reflection start with a `Class`
 - Even primitive types have a `Class`
- Where to get one of these objects?
 - `Class<?> c = "hello".getClass();`
 - `Class<?> c = String.class;`
 - `Class<?> c = Class.forName("java.lang.String");`
 - Here, the `<?>` means the exact class is unknown, which is a conservative assumption
- What is the `Class` of a class? `Class`, of course!
 - `"hello".getClass().getClass() == Class.class;`

java.lang.reflect.Constructor

- If we want to make an instance of the class, we need to get one of its constructors first

```
Class<?> c = A.class;
Constructor<?> cons = c.getConstructor(String.class,
                                       int.class);
Object o = cons.newInstance("foo", 42);
A a = (A) o;
```

- Because we didn't indicate what class this constructor is for (parameter is `<?>`), we have to downcast the return of `newInstance`
- Omitted: lots of checked exceptions
 - `getConstructor` might raise `NoSuchMethodException`
 - `newInstance` might raise `IllegalAccessException`, ...
 - Generally, always need to use `try-catch` with reflection

Methods and Fields

```
Class<?> c = A.class;
Method<?> meth = c.getMethod("m", int.class);
A a = ...;
Object o = meth.invoke(a, 42); // call a.m(42);
```

```
Class<?> c = A.class;
Field<?> fld = c.getField("f");
A a = ...;
Object A_f = fld.get(a); // return a.f
```

- Notice that a **Method** and **Field** (from `java.lang.reflect`) describe an elt of a class
- To invoke a **Method**, pass this as the first argument
- To access a **Field**, pass this as the argument

Hello, World!

```
import java.io.*;
import java.lang.reflect.*;

public static void main(String[] args)
    throws Exception {
    Field f = System.class.getField("out");
    PrintStream out = (PrintStream) f.get(null);
    Method m = PrintStream.class.getMethod("println",
                                           String.class);
    m.invoke(out, "Hello, world!");
}
```

- Well, that doesn't seem very good!
 - Why would we ever want to use reflection?

Design Patterns with Reflection

Factory Methods

- Recall that in the factory pattern, we create objects through a method call rather than using `new` directly
- We could use reflection to create objects by name
- Example: Create objects based on names in map

```
// Assume Pawn is a class, Pawn extends Piece
HashMap<char, String> hm; // map from 'p' to 'Pawn'

String cname = hm.get('p');
Class<?> c = Class.forName(cname);
Constructor cons = c.getConstructor();
Piece p = (Piece) cons.newInstance();
```

Interpreter

- Take various actions depending on a string
 - “up” → call up()
 - “down” → call down()
- Straightforward implementation

```
if (str.equals("up")) { up(); }  
else if (str.equals("down")) { down(); }  
...
```

- Reflective implementation

```
Method m = this.class.getMethod(str);  
m.invoke(this);
```

- Concise, easy to extend

Security Warning!

- The previous example is actually a **BAD IDEA**
- What if `str` is controlled by an *adversary*?
 - Someone who is *trying* to do something bad
 - Normal users work around bugs; adversaries look for them!
- An adversary could set `str` to
 - The name of a method they shouldn't be able to call
 - The name of a nonexistent method—leads to creash
- Who might be an adversary?
 - Someone sending data over the internet
 - A local user with fewer privileges than the app
- **KEY RULE:** Never treat data that is not directly from the program as code

Testing

- Invoke all methods that begin with `test`

```
// o is an instance of some Test class  
Class c = o.class;  
Method[] meths = c.getMethods();  
for (Method m : meths) {  
    if (m.getName().startsWith("test")) {  
        m.invoke(o)  
    }  
}
```

- We'll learn about JUnit a little later, which does this

Double Dispatch

- Recall in the Visitor pattern, invoking `o1.accept(o2)` calls a method that depends on the run-time type of `o1` and the run-time type of `o2`
- We can implement this with reflection!

```
class C {
    Object m(C1a x, C1b y) { ... }
    Object m(C2a x, C2b y) { ... } ...
}
// call m method based on run-time type of x, y
Object call_m(Object x, Object y) {
    C c = ...;
    // now we look up run-time types of x and y
    Method m = c.class.getMethod("m",
                                   x.getClass(), y.getClass());
    m.invoke(c, x, y);
}
```

Dynamic Proxies

- Recall the Proxy pattern: wrap an object to add extra logic before or after calls to its methods
- Java has *dynamic proxy* support to create a wrapper on-the-fly

```
import java.lang.reflect.*;

public class MyProxy implements InvocationHandler {
    Object invoke(Object proxy, Method method,
                  Object[] args) {
        // proxy = o, method = m, args[] = { "foo" }
    } }

interface I { void m(String x); }
Class<?> interfaces = new Class<>[] { I.class };
InvocationHandler handler = new MyProxy();
I o = (I) Proxy.newProxyInstance
    (I.class.getClassLoader(), interfaces, handler);
o.m("foo");
```

Serialization

- Can use reflection to automatically serialize objects

```
Writer w = ...;
void serialize(Object o) {
    Class c = o.getClass();
    w.write(c + "\0"); // add null as terminator
    Field[] fs = c.getFields();
    for (Field f : fs) {
        w.write(f.getName() + "\0");
        serialize(w); // serialize the field value
    }
}
```

- Above code doesn't work with cycles...

Reflection Disadvantages

- Extremely verbose
- Potentially opens up security concerns
- Misses out on compile-time type checking
 - E.g., trying to invoke a method with the wrong name or wrong arg types becomes a run-time exception
- Large performance penalty compared to direct calls
 - Overhead of extra method calls plus compiler can't optimize reflective calls very well, in general
- Summary: Use with caution or not at all
 - Never use it if you don't have a very good reason