

Reflection: In practice via Java

CPSC 501: Advanced Programming Techniques
Fall 2020

Jonathan Hudson, Ph.D
Instructor
Department of Computer Science
University of Calgary

Wednesday, August 5, 2020



Here comes the code

Java Packages

- The reflection classes (**metaobjects**) are in two packages:
 - **java.lang**
 - **Object**
 - **Class**
 - **java.lang.reflect**
 - **Method**
 - **Field**
 - **Constructor**
 - **etc.**

Stay classy

Object class

- `java.lang.Object`

- Is the **root superclass** of every object in a program
- **Not a metaobject** (helps us get to it)
- Each base-level object keeps a reference to its class object
 - Accessed with the method:
`public final Class getClass()`
 - E.g.
`Object myObj = new ...`
`Class classObject = myObj.getClass();`

Class metaobject

- `java.lang.Class`
 - Is the class of metalevel *class objects*
 - Has many useful reflective methods to:
 - Create new instances
 - Find methods, constructors, and fields of a class
 - Traverse the inheritance hierarchy
 - etc.

Getting Class

- Finding class objects (3 ways)
 1. For an already instantiated base-level object, use **getClass()**
 - Demonstrated previously
 2. If you know the class object at compile time, use the ***class literal* .class**
 - E.g. **Class classObject = Color.class;**
- ...

Getting Class (cont'd)

3. If the class name is represented as a **String** (usually at runtime), use the method:

public static Class.forName(String className)

- If not already loaded, dynamically loads the class from bytecode in the **.class** file
- If the class is in a named package, use the fully qualified name
- To work, the classpath must be set properly
- E.g.

```
String name = "java.io.File";
```

```
Class classObject = Class.forName(name);
```


Class Usage

- Java uses class objects (instances of Class) to represent the types of all entities:
 - Ordinary objects (like previously demonstrated)
 - We need to have meta idea of what everything is in language
 - That even includes no OO things (java has holdovers from c-like design)
- But also
 1. Primitives
 - int, float, char, etc.
 2. Arrays
 3. Interfaces

1.Primitives

- Although primitives are not objects, Java uses class objects to represent their type
 - Use a class literal to specify the class object
 - E.g. **int.class**, **double.class**
 - **void.class** represents the void return type
 - To check if primitive, use **isPrimitive()** on the class object
 - E.g. **if (classObject.isPrimitive()) ...**

2.Arrays

- Java arrays are objects whose classes are created at runtime by the JVM
 - A new class for each element type and dimension
 - Use a class literal to specify the class object
 - E.g. **int[].class, Object[].class**
 - To check if an array, use **isArray()**
 - To find the base type of an array, use **public Class getComponentType()**

3.Interfaces

- Each declared interface is represented with a class object
 - Can be specified with a class literal
 - E.g. **Collection.class**
 - Can be queried for supported methods and constants
 - To check if an interface, use **isInterface()**

Methods, man

Get a method

- Methods for a class or interface are represented with metaobjects of the type **java.lang.reflect.Method**
- Methods can be found at runtime by querying the class object
 - To find a **public** method (either declared or inherited), use **Method getMethod(String name, Class[] parameterTypes)**
 - E.g.
Method m;
m = classObject.getMethod("setColor", new Class[] {Color.class});
- If no parameters, use null or zero-length array for the 2nd argument

Get a method (cont'd)

- The above **getMethod** can access every method attached to a class
 - Including those inherited
- **Use getDeclaredMethod(...)** to find a method explicitly declared by the class (i.e. not inherited)
 - In addition it also returns methods of all visibilities: **public, protected, package,** and **private**

Get ALL methods

- To find *all* **public** methods of a class (either declared or inherited), use **Method[] getMethods()**
 - E.g.
Method mArray[] = classObject.getMethods();
- To find all declared methods of any visibility, use **Method[] getDeclaredMethods()**

Method Parts

- A Method object can be queried with:
 - **String getName()**
 - **Class getDeclaringClass()**
 - **Class[] getExceptionTypes()**
 - **Class[] getParameterTypes()**
 - **Class getReturnType()**
 - **int getModifiers()**
 - The returned int can be decoded with methods in Modifier class

Method acting

Dynamic Method Call

- To call a method dynamically, use
Object invoke(Object obj, Object[] args)
- E.g.

```
Object myObj = new String("Hello, world");  
Class classObject = myObj.getClass();  
Integer l = new Integer(5);  
Method m = classObject.getMethod("substring", new Class[]{int.class});  
Object result = m.invoke(myObj, new Object[]{l});  
System.out.println(result);
```

Dynamic Method Call (cont'd)

- If there are no arguments, use null or zero-length array for the 2nd parameter
- If a static method, use null for the 1st parameter

Dynamic Method Call (primitive para)

- Primitives are passed as parameters by putting them into a “wrapper object”
 - E.g. (next page)

```
Object myObj = new String("Hello, world");
Class classObject = myObj.getClass();
int i = 10;
Method m = classObject.getMethod("indexOf", new Class[]{int.class});
Object result = m.invoke(myObj, new Object[]{new Integer(i)});
System.out.println(result);
```

Dynamic Method Call (return primitive)

- If a method normally returns a primitive, **invoke()** will return the primitive in a wrapper object
- Since typed as **Object**, you must cast it to the correct type
- Then unwrap it using a **xxxValue()** method

- Note: Java 5.0 introduced automatic boxing and unboxing
- E.g. (next page)

Dynamic Method Call (return primitive) - Example

```
Object myObj = new String("Hello, world");  
Class classObject = myObj.getClass();  
int code;  
Method m = classObject.getMethod("hashCode", null);  
code = ((Integer) m.invoke(myObj, null)).intValue();  
System.out.println(code);
```

Turtles all the way up

Superclass

- To find the *superclass object of a class* object use **Class getSuperClass()**
 - E.g.
Class superclassObject = classObject.getSuperClass();
- Returns null if **classObject** represents a *primitive type, void, an interface, or Object class*
- Returns *class object* for **Object** if an array

Interfaces

- Use **Class[] getInterfaces()** on a class object to find all interfaces that the class directly implements
 - If used on a class object that represents an interface, then returns the direct superinterfaces

Fielders choice

Fields

- Fields for a class or interface are represented with metaobjects of the type **java.lang.reflect.Field**
- Fields can be found at runtime by querying the class object
 - To find a public field (either declared or inherited), use

Field getField(String name)

Fields (cont'd)

- E.g.
Field f = classObject.getField("id");
 - May throw **NoSuchFieldException**
- Use **getDeclaredField(String name)** to find a field explicitly declared by the class or interface (i.e. not inherited)
 - Returns fields of all visibilities: public, protected, package, and private

Get Fields

- To find all **public** fields of a class (either declared or inherited), use

Field[] getFields()

E.g.

Field fArray[] = classObject.getFields();

- To find all declared fields of any visibility, use **Field[] getDeclaredFields()**

Field Parts

- A Field object can be queried with:
 - **String getName()**
 - **Class getDeclaringClass()**
 - **Class getType()**
 - **int getModifiers()**
 - The returned int can be decoded with methods in Modifier class

Field Type

- You can find the value of a field reflectively using
 - **Object get(Object obj)**
 - E.g.

```
Object myObj = new String("Hello, world");
Class classObject = myObj.getClass();
System.out.println(classObject.getDeclaredFields()[0]);
System.out.println(classObject.getDeclaredFields()[1]);
Field f = classObject.getDeclaredField("value");
Object value = f.get(myObj);
System.out.println(value);
```

- If the field type is primitive, the returned value is wrapped in the appropriate wrapper object

Field Type (visibility?)

- You can't always access a field despite being able to find out that it exists

```
private final char[] java.lang.String.value
private int java.lang.String.hash
Sep 24, 2019 10:53:27 PM Temp main
SEVERE: null
java.lang.IllegalAccessException: Class Temp can not access a member of class java.lang.String with modifiers "private final"
    at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:102)
    at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(AccessibleObject.java:296)
    at java.lang.reflect.AccessibleObject.checkAccess(AccessibleObject.java:288)
    at java.lang.reflect.Field.get(Field.java:390)
    at Temp.main(Temp.java:35)
```

Fielding it

Field Value

- If you know the type of the primitive, you can access the value directly using methods like
 - **boolean getBoolean(Object obj)**
 - **double getDouble(Object obj)**
 - etc.
- E.g.
int value = f.getInt(myObj);

Field Setting

- Fields can be set reflectively using **void set(Object obj, Object value)**
 - E.g. **f.set(myObj, newValue);**
 - You must wrap primitive values, or use methods like
 - void setBoolean(Object obj, boolean value)
 - void setDouble(Object obj, double value)
 - etc.
 - E.g. **f.setInt(myObj, 37)**

Field modifiers/access

Modifiers

- Any Class, Method, or Field object can be queried using **getModifiers()**
 - Returns an int where particular bits represent one of the 11 modifiers in Java
 - public, protected, private, static, abstract, etc.
 - Can be decoded using static methods in **java.lang.reflect.Modifier**
 - **boolean isPublic(int mod)**
 - **boolean isProtected(int mod)**
 - etc.

Modifiers (cont'd)

- E.g.

```
Object myObj = new String("Hello, world");
Class classObject = myObj.getClass();
Field f = classObject.getDeclaredField("value");
int mod = f.getModifiers();
if (Modifier.isFinal(mod)) {
    System.out.println("is final");
}
```
- Can print out all modifiers using **toString(int mod)**
 - E.g. **System.out.println(Modifier.toString(mod));**

Field Access

- Normally, non-public fields and methods cannot be accessed from outside the class
 - Access checking can be bypassed using void **setAccessible(boolean flag)**
 - Works for all the get and set methods of Field, and the invoke method of Method
 - E.g.
f.setAccessible(true);
Object value = f.get(myObj);

Array we go

Arrays

- **java.lang.reflect.Array** provides static methods to operate reflectively on array objects
 - **Object newInstance(Class componentType, int length)**
 - E.g.
Object myArray = Array.newInstance(int.class, 10);
 - **int getLength(Object array)**
 - E.g.
int length = Array.getLength(anObj);

Arrays Get Entry

- Object **get(Object array, int index)**
 - Returns the element at index, wrapping primitives if necessary
 - E.g. **Object obj = Array.get(myArray, 3);**
 - Also has methods like **getBoolean(...)**, **getDouble()**, etc.
 - E.g. **int i = Array.getInt(myArray, 3);**

Arrays Set Entry

- **void set(Object array, int index, Object value)**
 - Sets the element at index to specified value, unwrapping primitives if necessary
 - Also has methods like **setBoolean(...)**, **setDouble(...)**, etc.
 - E.g. **Array.setInt(myArray, i, iVal)**

Bob the constructor

Constructor

- Constructors for a class are represented with *metaobjects* of the type **java.lang.reflect.Constructor**
 - Constructors can be found at runtime by querying the class object
 -
 - To find a public constructor (either declared or inherited), use **Constructor getConstructor(Class[] parameterTypes)**

Constructor (cont'd)

- E.g.

```
Class classObject = String.class;
for(int i = 0; i < classObject.getDeclaredConstructors().length; i++){
    System.out.println(classObject.getDeclaredConstructors()[i]);
}
Constructor c = classObject.getConstructor(new Class[]{String.class});
```

- If no parameters, use null or zero-length array for the argument
- Throws **NoSuchMethodException** (!) if not found
- Use **getDeclaredConstructor(...)** to find a constructor (of any visibility) explicitly declared by the class

Constructor (cont'd)

- To find all public constructors of a class (either declared or inherited), use **Constructor[] getConstructors()**
 - E.g.
Constructor cArray[] = classObject.getConstructors();
- To find all declared constructors of any visibility, use **Constructor[] getDeclaredConstructors()**

Constructor Parts

- A Constructor object can be queried with:
 - **String getName()**
 - **Class getDeclaringClass()**
 - **Class[] getExceptionTypes()**
 - **Class[] getParameterTypes()**
 - **int getModifiers**

Constructor – New Instance

- Reflective instantiation:
 - Can be done using **newInstance()** on the class object
 - E.g.

```
Object myObj = classObject.newInstance();  
System.out.println(myObj);
```

- Implicitly uses the no-arg constructor

Constructor – New Instance Arguments

- Can be done using a constructor metaobject and the method:
Object newInstance(Object[] initargs)
 - E.g.

```
Class classObject = String.class;  
Constructor c = classObject.getConstructor(new Class[]{String.class});  
Object myObj = c.newInstance(new Object[]{"Hello, world"});  
System.out.println(myObj);
```

Readings

- Forman & Forman Chapters 1, 2
- Java API: **Class**, **Object**, **reflect** package

Onward to ... Java serialization.

Jonathan Hudson
jwhudson@ucalgary.ca
<https://pages.cpsc.ucalgary.ca/~hudsonj/>



UNIVERSITY OF
CALGARY