

Java Reflection

Java looking at Java

- One of the unusual capabilities of Java is that a program can examine itself.
 - You can determine the class of an object.
 - You can find out all about a class: its access modifiers, superclass, fields, constructors, and methods.
 - You can find out what is there in an interface.
 - Even if you don't know the names of things when you write the program, you can:
 - Create an instance of a class
 - Get and set instance variables
 - Invoke a method on an object
 - Create and manipulate arrays

What is reflection for?

- In “normal” programs you don’t need reflection.
- You *do* need reflection if you are working with programs that process programs.
- Typical examples:
 - A class browser
 - A debugger
 - A GUI builder
 - An IDE, such Netbeans or eclipse

Using Reflection to Analyze the Capabilities of Classes

- The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class, respectively.
- All three classes have a method called `getName()` that returns the name of the item. The `Field` class has a method `getType()` that returns an object, again of type `Class`, that describes the field type.
- The `Method` and `Constructor` classes have methods to report the types of the parameters, and the `Method` class also reports the return type.

Con't

- All three of these classes also have a method called `getModifiers()` that returns an integer, with various bits turned on and off, that describes the modifiers used, such as public and static.
- Use methods like `isPublic()`, `isPrivate()`, or `isFinal()` in the `Modifier` class to tell whether a method or constructor was public, private, or final.
- All you have to do is have the appropriate method in the `Modifier` class work on the integer that `getModifiers()` returns.
 - You can also use the `Modifier.toString()` method to print the modifiers.

Con't

- The `getFields()`, `getMethods()`, and `getConstructors()` methods of the `Class` class return arrays of the public fields, methods, and constructors that the class supports.
 - This includes public members of superclasses.
- The `getDeclaredFields()`, `getDeclaredMethods()`, and `getDeclaredConstructors()` methods of the `Class` class return arrays consisting of all fields, operations, and constructors that are declared in the class.
 - This includes private and protected members, but not members of superclasses.

The Class class

- While your program is running, the Java runtime system always maintains what is called runtime type identification on all objects.
- This information keeps track of the class to which each object belongs.
- Runtime type information is used by the virtual machine to select the correct methods to execute.
- You can also access this information by working with a special Java class.
- The class that holds this information is called, somewhat confusingly, Class.
- The getClass() method in the Object class returns an instance of Class type.

The Class class

- To find out about a class, first get its **Class** object
 - If you have an object **obj**, you can get its class object with **Class c = obj.getClass();**
 - You can get the class object for the superclass of a Class **c** with **Class sup = c.getSuperclass();**
 - If you know the name of a class (say, **Button**) at compile time, you can get its class object with **Class c = Button.class;**
 - If you know the name of a class at run time (in a String variable **str**), you can get its class object with **Class c = class.forName(str);**

Getting the class name

- If you have a class object `c`, you can get the name of the class with `c.getName()`

- `getName` returns the fully qualified name; that is,

```
Class c = Button.class;  
String s = c.getName();  
System.out.println(s);
```

will print

```
java.awt.Button
```

- Class `Class` and its methods are in `java.lang`, which is always imported and available.
- `Fclass.java`
- `Fobject.java`

Getting all the superclasses

- `getSuperclass()` returns a `Class` object (or null if you call it on `Object`, which has no superclass)
- The following code is from the Sun tutorial:

```
static void printSuperclasses(Object o) {  
    Class subclass = o.getClass();  
    Class superclass = subclass.getSuperclass();  
    while (superclass != null) {  
        String className = superclass.getName();  
        System.out.println(className);  
        subclass = superclass;  
        superclass = subclass.getSuperclass();  
    }  
}
```

- `Fsupercls.java`

Getting the class modifiers I

- The modifiers (e.g., public, final, abstract etc.) of a **Class** object is encoded in an int and can be queried by the method **getModifiers()**.
- To decode the **int** result, we need methods of the **Modifier** class, which is in **java.lang.reflect**, so:

```
import java.lang.reflect.*;
```
- Then we can do things like:

```
if (Modifier.isPublic(m))  
    System.out.println("public");
```

Getting the class modifiers II

- **Modifier** contains these methods (among others):
 - public static boolean isAbstract(*int*)
 - public static boolean isFinal(*int*)
 - public static boolean isInterface(*int*)
 - public static boolean isPrivate(*int*)
 - public static boolean isProtected(*int*)
 - public static boolean isPublic(*int*)
 - public static String toString(*int*)
 - This will return a string such as
"public final synchronized strictfp"
- **ClassModifier.java**

Getting interfaces

- A class can implement zero or more interfaces
- `getInterfaces()` returns an *array* of **Class** objects
- Ex:

```
static void printInterfaceNames(Object o) {  
    Class c = o.getClass();  
    Class[] theInterfaces = c.getInterfaces();  
    for (Class inf: interfaces) {  
        System.out.println(inf.getName());    }  
}
```

- Note the convenience of enhanced for-loop
- **Finterface.java**

Examining classes and interfaces

- The class `Class` represents both classes and interfaces.
- To determine if a given `Class` object `c` is an interface, use `c.isInterface()`
 - `ExamineInterface.java`
- To find out more about a class object, use:
 - `getModifiers()`
 - `getFields()` `// "fields" == "instance variables"`
 - `getConstructors()`
 - `getMethods()`
 - `isArray()`

Getting Fields

- `public Field[] getFields()` throws `SecurityException`
 - Returns an array of *public* Fields (including inherited fields).
 - The length of the array may be zero
 - The fields are not returned in any particular order
 - Both locally defined and inherited instance variables are returned, but *not* static variables.
- `public Field getField(String name)`
throws `NoSuchFieldException`, `SecurityException`
 - Returns the named *public* Field
 - If no immediate field is found, the superclasses and interfaces are searched recursively

Using Fields, I

- If *f* is a **Field** object, then
 - *f.getName()* returns the simple name of the field
 - *f.getType()* returns the type (**Class**) of the field
 - *f.getModifiers()* returns the **Modifiers** of the field
 - *f.toString()* returns a String containing access modifiers, the type, and the fully qualified field name
 - Example: `public java.lang.String Person.name`
 - *f.getDeclaringClass()* returns the **Class** in which this field is declared
 - note: `getFields()` may return superclass fields.
- **Ffield.java**

Using Fields, II

- The fields of a particular object *obj* may be accessed with:
 - `boolean f.getBoolean(obj)`, `int f.getInt(obj)`, `double f.getDouble(obj)`, etc., return the value of the field, assuming it is that type or can be widened to that type
 - `Object f.get(obj)` returns the value of the field, assuming it is an Object
 - `void f.set(obj, value)`, `void f.setBoolean(obj, bool)`, `void f.setInt(obj, i)`, `void f.setDouble(obj, d)`, etc. set the value of a field
 - `SettingField.java`

Getting Constructors of a class

- If `c` is a Class, then
 - `c.getConstructors()` : `Constructor[]` return an array of all public constructors of class `c`.
- `c.getConstructor(Class ... paramTypes)` returns a constructor whose parameter types match those given `paramTypes`.
- Ex:
- `String.class.getConstructors().length`
- `> 15;`
- `String.class.getConstrucor(char[].class, int.class, int.class).toString()`
- `> String(char[], int,int).`
- `Fconstructor.java`

Constructors

- If `c` is a **Constructor** object, then
 - `c.getName()` returns the name of the constructor, as a String (this is the same as the name of the class)
 - `c.getDeclaringClass()` returns the **Class** in which this constructor is declared
 - `c.getModifiers()` returns the **Modifiers** of the constructor
 - `c.getParameterTypes()` returns an array of **Class** objects, in declaration order
 - `c.newInstance(Object... initargs)` creates and returns a new instance of class `c`
 - Arguments that should be primitives are automatically unwrapped as needed.

Example

- Constructor `c = String.class.getConstructor(char[].class, int.class, int.class).toString()`

➤ `String(char[], int,int).`

- `String s = c.newInstance(
new char[] { 'a','b','c','d' }, 1, 2);`

Methods

- `public Method[] getMethods()` throws `SecurityException`
 - Returns an array of `Method` objects.
 - These are the *public member* methods of the class or interface, including inherited methods.
 - The methods are returned in no particular order.
- `public Method getMethod(String name, Class... parameterTypes)` throws `NoSuchMethodException`, `SecurityException`
- `Fmethod.java`

Method methods, I

- `getDeclaringClass()`
 - Returns the **Class** object representing the class or interface that declares the method represented by this **Method** object
- `getName()`
 - Returns the name of the method represented by this **Method** object, as a **String**
- `getModifiers()`
 - Returns the Java language modifiers for the method represented by this **Method** object, as an integer
- `getParameterTypes()`
 - Returns an array of **Class** objects that represent the formal parameter types, in declaration order, of the method represented by this **Method** object

Method methods, II

- `getReturnType()`
 - Returns a `Class` object that represents the formal return type of the method represented by this `Method` object
- `toString()`
 - Returns a `String` describing this `Method`
 - `MethodInfo.java`
- `public Object invoke(Object obj, Object... args)`
 - Invokes the underlying method represented by this `Method` object, on the specified object with the specified parameters.
 - Individual parameters are automatically unwrapped to match primitive formal parameters.

Examples of invoke()

- `"abcdefg".length()`
> 7
- `Method lengthMethod = String.class.getMethod("length");`
`lengthMethod.invoke("abcdefg")`
> 7
- `"abcdefg".substring(2, 5)`
> cde
- `Method substringMethod = String.class.getMethod ("substring",`
`int.class, int.class);`
- `substringMethod.invoke("abcdefg", 2, 5)`
> cde
- `CallingMethod.java`

Arrays I

- To determine whether an object `obj` is an array,
 - Get its class `c` with `Class c = obj.getClass();`
 - Test with `c.isArray()`
 - `IdentifyArray.java`
- To find the type of components of the array,
 - `c.getComponentType()`
 - Returns `null` if `c` is not the class of an array
- Ex:
 - `int[].class.isArray() == true ;`
 - `int[].class.getComponentType() == int.class`
 - `ComponentType.java`

Arrays II

- The `Array` class in `java.lang.reflect` provides *static* methods for working with arrays.
- To create an array,
- `Array.newInstance(Class componentType, int size)`
 - This returns, as an `Object`, the newly created array
 - You can cast it to the desired type if you like
 - The `componentType` may itself be an array
 - This would create a multiple-dimensioned array
 - The limit on the number of dimensions is usually 255
- `Array.newInstance(Class componentType, int... sizes)`
 - This returns, as an `Object`, the newly created multidimensional array (with `sizes.length` dimensions)

Examples

- The following two objects are of the same type:
 - `new String[10]`
 - `Array.newInstance(String.class, 10)`
- The following two objects are of the same type:
 - `new String[10][20]`
 - `Array.newInstance(String.class, 10, 20)`

Arrays III

- To get the value of array elements,
 - `Array.get(Object array, int index)` returns an `Object`
 - `Array.getBoolean(Object array, int index)` returns a `boolean`
 - `Array.getByte(Object array, int index)` returns a `byte`
 - etc.
- To store values into an array,
 - `Array.set(Object array, int index, Object value)`
 - `Array.setInt(Object array, int index, int i)`
 - `Array.setFloat(Object array, int index, float f)`
 - etc.
 - `Array1.java`

Examples

- `a = new int[] {1,2,3,4};`
- `Array.getInt(a, 2) // → 3`
- `Array.setInt(a, 3, 5) // a = {1,2,3, 5 }.`

- `s = new String[] { "ab", "bc", "cd" };`
- `Array.get(s, 1) // → "bc"`
- `Array.set(s, 1, "xxx") // s[1] = "xxx"`

Getting non-public members of a class

- All `getXXX()` methods of `Class` mentioned above return only **public** members of the target (as well as ancestor) classes, but they cannot return non-public members.
- There are another set of `getDeclaredXXX()` methods in `Class` that will return all (**even private or static**) members of target class but no inherited members are included.
- `getDeclaredConstructors()`,
`defDeclaredConstrucor(Class...)`
- `getDeclaredFields()`,
- `getDeclaredField(String)`
- `getDeclaredmethods()`,
- `getDeclaredMethod(String, Class...)`
- **ReflectionExample.java**

Concluding comments

- Many of these methods throw exceptions not described here
 - For details, see the Java API
- Reflection isn't used in “normal” programs, but when you need it, it's indispensable
- Studying the java reflection package gives you a chance to review the basics of java class structure.