



- [Home](#)
- [Newsletter](#)
- [Java Specialist Club](#)
- [Java Training](#)
- [Conference Venue Hire](#)
- [Java Resources](#)
- [Contact](#)

The Java Specialists' Newsletter
 ➤ Issue 117 ➤ 2005-12-02 ➤ Category: [Language](#) ➤ Java version: All

0 [✉ Subscribe Free](#) [📡 RSS Feed](#) [Print](#)

Reflectively Calling Inner Class Methods

by Dr. Heinz M. Kabutz

Abstract:

Sometimes frameworks use reflection to call methods. Depending how they find the correct method to call, we may end up with `IllegalAccessExceptions`. The naive approach of `clazz.getMethod(name)` is not correct when we send instances of non-public classes.

Welcome to the 117th edition of **The Java(tm) Specialists' Newsletter**. I am sitting outside in our garden in Cape Town, South Africa, watching the Helmeted Guinea Fowls (*Numida meleagris* according to BEA's Anton de Swardt) scurrying around on the road outside. These remarkable birds often sit in our pine trees, making their strange noise, which in the African night can be heard at a great distance. **Here is a short clip of these birds** making their noise somewhere in Namibia. **Here is a picture** that I took with my phone. The quality is not that great, but you can clearly see the helmet on the top of its head.

Upcoming Java Master Courses:

- Duesseldorf, Germany, Aug 22
- Chania, Crete, Sep 6
- Cape Town, South Africa, Sep 12

In-house courses if these dates or locations do not suit you. Note that the course in Crete may also be attended remotely via webinar.

Reflectively Calling Inner Class Methods

A year ago we had an open source workshop in Cape Town. One of our **The Java(tm) Specialists' Newsletter** readers, Marcus Sundman, flew all the way down from Finland to attend it and contribute to the discussions.

I extend an open invitation to any of our readers who happen to come to Cape Town, to visit and maybe have a barbeque together. Marcus joined us for that, and amazed us with all sorts of puzzles that we struggled to solve.

A few days ago, Marcus sent me a Java puzzle. I wrote back what I imagined the answer would be, and promised to publish the question if I **got it wrong**. So here it is...

Here is a slightly simplified version of what Marcus sent me:

```

package com.cretesoft.tjsn.issue117;

public class Greeter {
    public void hello() {
        System.out.println("Hello from Greeter");
    }
}

package com.cretesoft.tjsn.issue117;

public class Exec {
    public static void run(Greeter target) {
        System.out.println();
        System.out.print("method call> ");
        target.hello();

        System.out.print("base class > ");
        run(target, Greeter.class, "hello");

        System.out.print("obj class > ");
        run(target, target.getClass(), "hello");
    }

    // this calls the method using reflection
    static void run(Greeter target, Class cls, String method) {
        try {
            cls.getMethod(method, null).invoke(target, null);
        } catch (Exception x) {
            System.out.println(x);
        }
    }
}
    
```

Newsletter Links

- [Book Review](#)
- [Concurrency](#)
- [Exceptions](#)
- [GUI](#)
- [Inspirational](#)
- [Language](#)
- [Performance](#)
- [Software Engineering](#)
- [Tips and Tricks](#)



Java Courses

- [Java Master](#)
- [Java Foundation](#)
- [Java 5 Tiger](#)
- [Design Patterns](#)

[➤ find out more](#)

What is the Java Specialists Club?

Venue for Hire

Looking for a super conference room for your next company event?



Crete is your perfect destination.

[Click here for more details](#)

```

}

package com.cretesoft.tjsn.issue117;

public class InsideJob {
    public static void run() {
        Exec.run(new Greeter() {
            public void hello() {
                System.out.println("Hello from InsideJob");
            }
        });
    }
}

import com.cretesoft.tjsn.issue117.*;

public class Main {
    public static void main(String[] args) {
        InsideJob.run();
        Exec.run(new Greeter() {
            public void hello() {
                System.out.println("Hello from Main");
            }
        });
    }
}

```

I read through the code, and expected to see the following output:

```

method call> Hello from InsideJob
base class > Hello from InsideJob
obj class > Hello from InsideJob

method call> Hello from Main
base class > Hello from Main
obj class > Hello from Main

```

Infact, when I tried it out, I decided to simplify matters by putting all the classes in the same package, in which case you do see that output. However, when you have Main in a separate package to the rest, you get:

```

method call> Hello from InsideJob
base class > Hello from InsideJob
obj class > Hello from InsideJob

method call> Hello from Main
base class > Hello from Main
obj class > java.lang.IllegalAccessException: class
com.cretesoft.tjsn.issue117.Exec can not access
a member of class Main$1 with modifiers "public"

```

I tried this in JDK 1.1.8, 1.2.2, 1.3.1, 1.4.2 and 1.5.0, all with the same effect. I was amazed.

To summarise the problem: If you find the method using reflection on the subclass, make sure that it is not package access and living in another package to yourself, otherwise you will get an error when you try to call it.

This does not make any sense to me, and in my opinion should be classed as a bug in the Java Programming Language.

At runtime, there is no difference between inner classes that have private, protected or package access. They all get compiled to a package access class, and the compiler add links to the outer class' object if necessary. You can verify that by decompiling the inner class with JAD using the `-noinner` setting.

The Exec class would therefore also not work if we used a plain package access class, such as MyGreeter:

```

import com.cretesoft.tjsn.issue117.Greeter;

class MyGreeter extends Greeter {
    public void hello() {
        System.out.println("Hello from MyGreeter");
    }
}

```

This is rather confusing, because we can call the method "hello" on the Greeter object, and that works. We can find the method reflectively on the Greeter class, and call it on the derived object, and that works as well. But if we find the method "hello" on the derived object, and then call it, we get an `IllegalAccessException`.

I did not know this, so I've learnt something new, thanks Marcus :-)

Now the question: does it matter? If yes, how do I fix it?

In Marcus' case, it does matter, because of the framework that he is using. It effectively means that he cannot use anonymous inner classes, which by default are package access classes. He would have to write the Main class like this:

```

import com.cretesoft.tjsn.issue117.*;

public class Main2 {
    public static void main(String[] args) {
        InsideJob.run();
        Exec.run(new MyGreeter());
    }
    public static class MyGreeter extends Greeter {
        public void hello() {
            System.out.println("Hello from Main");
        }
    }
}

```

```
}

```

Now the output is as we would expect. So, no more anonymous inner classes!

Finding the Correct Method to Call

Here is another possibility, if you have the option of changing the framework code. Instead of picking the method at the lowest level, we recursively go up the hierarchy tree until we find a class that implements it.

```
package com.cretesoft.tjsn.issue117;
import java.lang.reflect.Method;
public class Exec2 {
    public static void run(Greeter target) {
        System.out.println();

        System.out.print("method call> ");
        target.hello();

        System.out.print("base class > ");
        run(target, Greeter.class, "hello");

        System.out.print("obj class > ");
        run(target, target.getClass(), "hello");
    }

    // this calls the method using reflection
    static void run(Greeter target, Class cls, String method) {
        try {
            findHighestMethod(cls, method).invoke(target, null);
        } catch (Exception x) {
            System.out.println(x);
        }
    }

    // recurse up hierarchy, looking for highest method
    private static Method findHighestMethod(Class cls,
                                           String method) {
        if (cls.getSuperclass() != null) {
            Method parentMethod = findHighestMethod(
                cls.getSuperclass(), method);
            if (parentMethod != null) return parentMethod;
        }
        Method[] methods = cls.getMethods();
        for (int i = 0; i < methods.length; i++) {
            // we ignore parameter types for now - you need to add this
            if (methods[i].getName().equals(method)) {
                return methods[i];
            }
        }
        return null;
    }
}

```

That works - but only if there is a superclass with that method definition. It won't work if we are implementing an interface. In that case, we need to change the method findHighestMethod to also include interfaces. Disclaimer: I am not completely sure that this will work in all cases.

```
// recurse up hierarchy, looking for highest method
private static Method findHighestMethod(Class cls,
                                       String method) {
    Class[] ifaces = cls.getInterfaces();
    for (int i = 0; i < ifaces.length; i++) {
        Method ifaceMethod = findHighestMethod(ifaces[i], method);
        if (ifaceMethod != null) return ifaceMethod;
    }
    if (cls.getSuperclass() != null) {
        Method parentMethod = findHighestMethod(
            cls.getSuperclass(), method);
        if (parentMethod != null) return parentMethod;
    }
    Method[] methods = cls.getMethods();
    for (int i = 0; i < methods.length; i++) {
        // we ignore parameter types for now - you need to add this
        if (methods[i].getName().equals(method)) {
            return methods[i];
        }
    }
    return null;
}

```

Thanks Marcus for that interesting information. I was not aware of this subtlety of reflection.

Kind regards

Heinz

[▶ Language Articles](#) [▶ Related Java Course](#) [▶ Discuss at The Java Specialist Club](#)