



- Home
- Newsletter
- Java Specialist Club
- Java Training
- Conference Venue Hire
- Java Resources
- Contact

The Java Specialists' Newsletter  
 ▶ Issue 161 ▶ 2008-05-23 ▶ Category: **Language** ▶ Java version: Sun JDK 6

16 [Subscribe Free](#) [RSS Feed](#) [Print](#)

### Of Hacking Enums and Modifying "final static" Fields

by Dr. Heinz M. Kabutz

#### Abstract:

The developers of the Java language tried their best to stop us from constructing our own enum instances. However, for testing purposes, it can be useful to temporarily add new enum instances to the system. In this newsletter we show how we can do this using the classes in sun.reflect. In addition, we use a similar technique to modify static final fields, which we need to do if we want the switch statements to still work with our new enums.

Welcome to the 161st issue of **The Java(tm) Specialists' Newsletter**. The last 10 days in San Francisco were memorable. I got to meet some super people. The most interesting was Alex Ruiz, who is developing a funky testing methodology with his wife, using **fluent interfaces**. Pair programming at its best.

#### 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.

### Of Hacking Enums and Modifying "final static" Fields

In this newsletter we examine how it is possible to create enum instances in the Sun JDK, by using the reflection classes from the sun.reflect package. This will obviously only work for Sun's JDK. If you need to do this on another JVM, you're on your own.

This all started with an email from Ken Dobson of Edinburgh, which pointed me in the direction of the sun.reflect.ConstructorAccessor, which he claimed could be used to construct enum instances. My **previous approach (newsletter #141)** did not work in Java 6.

I was curious why Ken wanted to construct enums. Here is how he wanted to use it:

```
public enum HumanState {
    HAPPY, SAD
}

public class Human {
    public void sing(HumanState state) {
        switch (state) {
            case HAPPY:
                singHappySong();
                break;
            case SAD:
                singDirge();
                break;
            default:
                new IllegalStateException("Invalid State: " + state);
        }
    }

    private void singHappySong() {
        System.out.println("when you're happy and you know it ...");
    }

    private void singDirge() {
        System.out.println("Don't cry for me Argentina, ...");
    }
}
```

The above code needs a unit test. Did you spot the mistake? If you did not, go over the code again with a fine comb to try to find it. When I first saw this, I did not spot the mistake either.

When we make bugs like this, the first thing we should do is produce a unit test that shows it. However, in this case we cannot cause the **default** case to happen, because the HumanState only has the HAPPY and SAD enums.

Ken's discovery allowed us to make an instance of an enum by using the ConstructorAccessor class from the

#### 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](#)

sun.reflect package. It would involve something like:

```
Constructor cstr = clazz.getDeclaredConstructor(
    String.class, int.class
);
ReflectionFactory reflection =
    ReflectionFactory.getReflectionFactory();
Enum e =
    reflection.newConstructorAccessor(cstr).newInstance("BLA", 3);
```

However, if we just do that, we end up with an `ArrayIndexOutOfBoundsException`, which makes sense when we see how the Java compiler converts the switch statement into byte code. Taking the above Human class, here is what is the decompiled code looks like (thanks to [Pavel Kouznetsov's JAD](#)):

```
public class Human {
    public void sing(HumanState state) {
        static class _cls1 {
            static final int $SwitchMap$HumanState[] =
                new int[HumanState.values().length];
            static {
                try {
                    $SwitchMap$HumanState[HumanState.HAPPY.ordinal()] = 1;
                } catch (NoSuchFieldError ex) {}
                try {
                    $SwitchMap$HumanState[HumanState.SAD.ordinal()] = 2;
                } catch (NoSuchFieldError ex) {}
            }
        }

        switch(_cls1.$SwitchMap$HumanState[state.ordinal()]) {
            case 1:
                singHappySong();
                break;
            case 2:
                singDirge();
                break;
            default:
                new IllegalStateException("Invalid State: " + state);
                break;
        }
    }
    private void singHappySong() {
        System.out.println("When you're happy and you know it ...");
    }
    private void singDirge() {
        System.out.println("Don't cry for me Argentina, ...");
    }
}
```

You can see immediately why we would get an `ArrayIndexOutOfBoundsException`, thanks to the inner class `_cls1`.

My first attempt at fixing this problem did not result in a decent solution. I tried to modify the `$VALUES` array inside the `HumanState` enum. However, I just bounced off Java's protective code. You **can modify final fields**, as long as they are non-static. This restriction seemed artificial to me, so I set off on a quest to discover the holy grail of static final fields. Again, it was hidden in the chamber of `sun.reflect`.

### Setting "final static" Fields

Several things are needed in order to set a **final static** field. First off, we need to get the `Field` object using normal reflection. If we passed this to the `FieldAccessor`, we will just bounce off the security code, since we are dealing with a static final field. *Secondly, we change the modifiers field value inside the `Field` object instance to not be final.* Thirdly, we pass the doctored field to the `FieldAccessor` in the `sun.reflect` package and use this to set it.

Here is my `ReflectionHelper` class, which we can use to set **final static** fields via reflection:

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

public class ReflectionHelper {
    private static final String MODIFIERS_FIELD = "modifiers";

    private static final ReflectionFactory reflection =
        ReflectionFactory.getReflectionFactory();

    public static void setStaticFinalField(
        Field field, Object value)
        throws NoSuchFieldException, IllegalAccessException {
        // we mark the field to be public
        field.setAccessible(true);
        // next we change the modifier in the Field instance to
        // not be final anymore, thus tricking reflection into
        // letting us modify the static final field
        Field modifiersField =
            Field.class.getDeclaredField(MODIFIERS_FIELD);
        modifiersField.setAccessible(true);
        int modifiers = modifiersField.getInt(field);
        // blank out the final bit in the modifiers int
        modifiers &= ~Modifier.FINAL;
        modifiersField.setInt(field, modifiers);
        FieldAccessor fa = reflection.newFieldAccessor(
            field, false
        );
        fa.set(null, value);
    }
}
```

With this `ReflectionHelper`, I could thus set the `$VALUES` array inside the enum to contain my new enum. This worked, except that I had to do this before the `Human` class was loaded for the first time. This would introduce a racing condition into our test cases. By themselves each test would work, but collectively they could fail. Not a good

scenario!

## Rewiring Enum Switches

The next idea was to rewire the actual switch statement's `$$SwitchMap$HumanState` field. It would be fairly easy to find this field inside the anonymous inner class. All you need is the prefix `$$SwitchMap$` followed by the enum class name. If the enum is switched several times in one class, then the inner class is only created once.

One of the other solutions that I wrote yesterday did a check on whether our switch statement was dealing with all the possible cases. This would be useful in discovering bugs when a new type is introduced into the system. I discarded that particular solution, but you should be able to easily recreate that based on the `EnumBuster` that I will show you later.

## The Memento Design Pattern

I recently rewrote my [Design Patterns Course](#) (warning, the website might not have the up-to-date structure up yet - please enquire for more information), to take into account the changes in Java, to throw away some outdated patterns and to introduce some that I had excluded previously. One of the "new" patterns was the Memento, often used with undo functionality. I thought it would be a good pattern to use to undo the damage done to the enum in our great efforts to test our impossible case.

Publishing a Specialists' newsletter gives me certain liberties. I do not have to explain every line that I write. So, without further ado, here is my `EnumBuster` class, which allows you to make enums, add them to the existing values[], delete enums from the array, whilst at the same time maintaining the switch statement of any class that you specify.

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

public class EnumBuster<E extends Enum<E>> {
    private static final Class[] EMPTY_CLASS_ARRAY =
        new Class[0];
    private static final Object[] EMPTY_OBJECT_ARRAY =
        new Object[0];

    private static final String VALUES_FIELD = "$VALUES";
    private static final String ORDINAL_FIELD = "ordinal";

    private final ReflectionFactory reflection =
        ReflectionFactory.getReflectionFactory();

    private final Class<E> clazz;
    private final Collection<Field> switchFields;

    private final Deque<Memento> undoStack =
        new LinkedList<Memento>();

    /**
     * Construct an EnumBuster for the given enum class and keep
     * the switch statements of the classes specified in
     * switchUsers in sync with the enum values.
     */
    public EnumBuster(Class<E> clazz, Class... switchUsers) {
        try {
            this.clazz = clazz;
            switchFields = findRelatedSwitchFields(switchUsers);
        } catch (Exception e) {
            throw new IllegalArgumentException(
                "Could not create the class", e);
        }
    }

    /**
     * Make a new enum instance, without adding it to the values
     * array and using the default ordinal of 0.
     */
    public E make(String value) {
        return make(value, 0,
            EMPTY_CLASS_ARRAY, EMPTY_OBJECT_ARRAY);
    }

    /**
     * Make a new enum instance with the given ordinal.
     */
    public E make(String value, int ordinal) {
        return make(value, ordinal,
            EMPTY_CLASS_ARRAY, EMPTY_OBJECT_ARRAY);
    }

    /**
     * Make a new enum instance with the given value, ordinal and
     * additional parameters. The additionalTypes is used to match
     * the constructor accurately.
     */
    public E make(String value, int ordinal,
        Class[] additionalTypes, Object[] additional) {
        try {
            undoStack.push(new Memento());
            ConstructorAccessor ca = findConstructorAccessor(
                additionalTypes, clazz);
            return constructEnum(clazz, ca, value,
                ordinal, additional);
        } catch (Exception e) {
            throw new IllegalArgumentException(
                "Could not create enum", e);
        }
    }
}
```

```

/**
 * This method adds the given enum into the array
 * inside the enum class. If the enum already
 * contains that particular value, then the value
 * is overwritten with our enum. Otherwise it is
 * added at the end of the array.
 *
 * In addition, if there is a constant field in the
 * enum class pointing to an enum with our value,
 * then we replace that with our enum instance.
 *
 * The ordinal is either set to the existing position
 * or to the last value.
 *
 * warning: This should probably never be called,
 * since it can cause permanent changes to the enum
 * values. Use only in extreme conditions.
 *
 * @param e the enum to add
 */
public void addByValue(E e) {
    try {
        undoStack.push(new Memento());
        Field valuesField = findValuesField();

        // we get the current Enum[]
        E[] values = values();
        for (int i = 0; i < values.length; i++) {
            E value = values[i];
            if (value.name().equals(e.name())) {
                setOrdinal(e, value.ordinal());
                values[i] = e;
                replaceConstant(e);
                return;
            }
        }

        // we did not find it in the existing array, thus
        // append it to the array
        E[] newValues =
            Arrays.copyOf(values, values.length + 1);
        newValues[newValues.length - 1] = e;
        ReflectionHelper.setStaticFinalField(
            valuesField, newValues);

        int ordinal = newValues.length - 1;
        setOrdinal(e, ordinal);
        addSwitchCase();
    } catch (Exception ex) {
        throw new IllegalArgumentException(
            "Could not set the enum", ex);
    }
}

/**
 * We delete the enum from the values array and set the
 * constant pointer to null.
 *
 * @param e the enum to delete from the type.
 * @return true if the enum was found and deleted;
 *         false otherwise
 */
public boolean deleteByValue(E e) {
    if (e == null) throw new NullPointerException();
    try {
        undoStack.push(new Memento());
        // we get the current E[]
        E[] values = values();
        for (int i = 0; i < values.length; i++) {
            E value = values[i];
            if (value.name().equals(e.name())) {
                E[] newValues =
                    Arrays.copyOf(values, values.length - 1);
                System.arraycopy(values, i + 1, newValues, i,
                    values.length - i - 1);
                for (int j = i; j < newValues.length; j++) {
                    setOrdinal(newValues[j], j);
                }
                Field valuesField = findValuesField();
                ReflectionHelper.setStaticFinalField(
                    valuesField, newValues);
                removeSwitchCase(i);
                blankOutConstant(e);
                return true;
            }
        }
    } catch (Exception ex) {
        throw new IllegalArgumentException(
            "Could not set the enum", ex);
    }
    return false;
}

/**
 * Undo the state right back to the beginning when the
 * EnumBuster was created.
 */
public void restore() {
    while (undo()) {
        //
    }
}

/**
 * Undo the previous operation.
 */
public boolean undo() {
    try {
        Memento memento = undoStack.poll();

```

```

        if (memento == null) return false;
        memento.undo();
        return true;
    } catch (Exception e) {
        throw new IllegalStateException("Could not undo", e);
    }
}

private ConstructorAccessor findConstructorAccessor(
    Class[] additionalParameterTypes,
    Class<E> clazz) throws NoSuchMethodException {
    Class[] parameterTypes =
        new Class[additionalParameterTypes.length + 2];
    parameterTypes[0] = String.class;
    parameterTypes[1] = int.class;
    System.arraycopy(
        additionalParameterTypes, 0,
        parameterTypes, 2,
        additionalParameterTypes.length);
    Constructor<E> cstr = clazz.getDeclaredConstructor(
        parameterTypes
    );
    return reflection.newConstructorAccessor(cstr);
}

private E constructEnum(Class<E> clazz,
    ConstructorAccessor ca,
    String value, int ordinal,
    Object[] additional)
    throws Exception {
    Object[] parms = new Object[additional.length + 2];
    parms[0] = value;
    parms[1] = ordinal;
    System.arraycopy(
        additional, 0, parms, 2, additional.length);
    return clazz.cast(ca.newInstance(parms));
}

/**
 * The only time we ever add a new enum is at the end.
 * Thus all we need to do is expand the switch map arrays
 * by one empty slot.
 */
private void addSwitchCase() {
    try {
        for (Field switchField : switchFields) {
            int[] switches = (int[]) switchField.get(null);
            switches = Arrays.copyOf(switches, switches.length + 1);
            ReflectionHelper.setStaticFinalField(
                switchField, switches
            );
        }
    } catch (Exception e) {
        throw new IllegalArgumentException(
            "Could not fix switch", e);
    }
}

private void replaceConstant(E e)
    throws IllegalAccessException, NoSuchFieldException {
    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {
        if (field.getName().equals(e.name())) {
            ReflectionHelper.setStaticFinalField(
                field, e
            );
        }
    }
}

private void blankOutConstant(E e)
    throws IllegalAccessException, NoSuchFieldException {
    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {
        if (field.getName().equals(e.name())) {
            ReflectionHelper.setStaticFinalField(
                field, null
            );
        }
    }
}

private void setOrdinal(E e, int ordinal)
    throws NoSuchFieldException, IllegalAccessException {
    Field ordinalField = Enum.class.getDeclaredField(
        ORDINAL_FIELD);
    ordinalField.setAccessible(true);
    ordinalField.set(e, ordinal);
}

/**
 * Method to find the values field, set it to be accessible,
 * and return it.
 *
 * @return the values array field for the enum.
 * @throws NoSuchFieldException if the field could not be found
 */
private Field findValuesField()
    throws NoSuchFieldException {
    // first we find the static final array that holds
    // the values in the enum class
    Field valuesField = clazz.getDeclaredField(
        VALUES_FIELD);
    // we mark it to be public
    valuesField.setAccessible(true);
    return valuesField;
}

private Collection<Field> findRelatedSwitchFields(

```

```

    Class[] switchUsers) {
    Collection<Field> result = new ArrayList<Field>();
    try {
        for (Class switchUser : switchUsers) {
            Class[] clazzes = switchUser.getDeclaredClasses();
            for (Class suspect : clazzes) {
                Field[] fields = suspect.getDeclaredFields();
                for (Field field : fields) {
                    if (field.getName().startsWith("$SwitchMap$" +
                        clazz.getSimpleName()) {
                        field.setAccessible(true);
                        result.add(field);
                    }
                }
            }
        }
    } catch (Exception e) {
        throw new IllegalArgumentException(
            "Could not fix switch", e);
    }
    return result;
}

private void removeSwitchCase(int ordinal) {
    try {
        for (Field switchField : switchFields) {
            int[] switches = (int[]) switchField.get(null);
            int[] newSwitches = Arrays.copyOf(
                switches, switches.length - 1);
            System.arraycopy(switches, ordinal + 1, newSwitches,
                ordinal, switches.length - ordinal - 1);
            ReflectionHelper.setStaticFinalField(
                switchField, newSwitches
            );
        }
    } catch (Exception e) {
        throw new IllegalArgumentException(
            "Could not fix switch", e);
    }
}

@SuppressWarnings("unchecked")
private E[] values()
    throws NoSuchFieldException, IllegalAccessException {
    Field valuesField = findValuesField();
    return (E[]) valuesField.get(null);
}

private class Memento {
    private final E[] values;
    private final Map<Field, int[]> savedSwitchFieldValues =
        new HashMap<Field, int[]>();

    private Memento() throws IllegalAccessException {
        try {
            values = values().clone();
            for (Field switchField : switchFields) {
                int[] switchArray = (int[]) switchField.get(null);
                savedSwitchFieldValues.put(switchField,
                    switchArray.clone());
            }
        } catch (Exception e) {
            throw new IllegalArgumentException(
                "Could not create the class", e);
        }
    }

    private void undo() throws
        NoSuchFieldException, IllegalAccessException {
        Field valuesField = findValuesField();
        ReflectionHelper.setStaticFinalField(valuesField, values);

        for (int i = 0; i < values.length; i++) {
            setOrdinal(values[i], i);
        }

        // reset all of the constants defined inside the enum
        Map<String, E> valuesMap =
            new HashMap<String, E>();
        for (E e : values) {
            valuesMap.put(e.name(), e);
        }
        Field[] constantEnumFields = clazz.getDeclaredFields();
        for (Field constantEnumField : constantEnumFields) {
            E en = valuesMap.get(constantEnumField.getName());
            if (en != null) {
                ReflectionHelper.setStaticFinalField(
                    constantEnumField, en
                );
            }
        }

        for (Map.Entry<Field, int[]> entry :
            savedSwitchFieldValues.entrySet()) {
            Field field = entry.getKey();
            int[] mappings = entry.getValue();
            ReflectionHelper.setStaticFinalField(field, mappings);
        }
    }
}
}
}

```

The class is quite long and probably still has some bugs. I wrote it en route from San Francisco to New York. Here is how we could use it to test our Human class:

```

import junit.framework.TestCase;

public class HumanTest extends TestCase {

```

```

public void testSingingAddingEnum() {
    EnumBuster<HumanState> buster =
        new EnumBuster<HumanState>(HumanState.class,
            Human.class);

    try {
        Human heinz = new Human();
        heinz.sing(HumanState.HAPPY);
        heinz.sing(HumanState.SAD);

        HumanState MELLOW = buster.make("MELLOW");
        buster.addByValue(MELLOW);
        System.out.println(Arrays.toString(HumanState.values()));

        try {
            heinz.sing(MELLOW);
            fail("Should have caused an IllegalStateException");
        }
        catch (IllegalStateException success) { }
    }
    finally {
        System.out.println("Restoring HumanState");
        buster.restore();
        System.out.println(Arrays.toString(HumanState.values()));
    }
}
}

```

This unit test now shows the mistake in our Human.java file, shown earlier. We forgot to add the **throw** keyword!

```

when you're happy and you know it ...
Don't cry for me Argentina, ...
[HAPPY, SAD, MELLOW]
Restoring HumanState
[HAPPY, SAD]

```

```

AssertionFailedError: should have caused an IllegalStateException
    at HumanTest.testSingingAddingEnum(HumanTest.java:23)

```

The EnumBuster class can do more than that. We can use it to delete enums that we don't want. If we specify which classes the switch statements are, then these will be maintained at the same time. Plus, we can undo right back to the initial state. Lots of functionality!

One last test case before I sign off, where we add the test class to the switch classes to maintain.

```

import junit.framework.TestCase;

public class EnumSwitchTest extends TestCase {
    public void testSingingDeletingEnum() {
        EnumBuster<HumanState> buster =
            new EnumBuster<HumanState>(HumanState.class,
                EnumSwitchTest.class);

        try {
            for (HumanState state : HumanState.values()) {
                switch (state) {
                    case HAPPY:
                    case SAD:
                        break;
                    default:
                        fail("Unknown state");
                }
            }

            buster.deleteByValue(HumanState.HAPPY);
            for (HumanState state : HumanState.values()) {
                switch (state) {
                    case SAD:
                        break;
                    case HAPPY:
                    default:
                        fail("Unknown state");
                }
            }

            buster.undo();
            buster.deleteByValue(HumanState.SAD);
            for (HumanState state : HumanState.values()) {
                switch (state) {
                    case HAPPY:
                        break;
                    case SAD:
                    default:
                        fail("Unknown state");
                }
            }

            buster.deleteByValue(HumanState.HAPPY);
            for (HumanState state : HumanState.values()) {
                switch (state) {
                    case HAPPY:
                    case SAD:
                    default:
                        fail("Unknown state");
                }
            }
        }
        finally {
            buster.restore();
        }
    }
}

```

The EnumBuster even maintains the constants, so if you remove an enum from the values(), it will blank out the final static field. If you add it back, it will set it to the new value.

It was thoroughly entertaining to use the ideas by Ken Dobson to play with reflection in a way that I did not know was possible. (Any Sun engineers reading this, please don't plug these holes in future versions of Java!)

Kind regards

Heinz

P.S. I'll be in London for a few days next week on business in the Canary Wharf area. Shout if you have time for a pint.

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

---

© 2010 Heinz Kabutz - All Rights Reserved

[Sitemap](#)

[seo web design](#) Catch22 Marketing

---

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. JavaSpecialists.eu is not connected to Oracle, Inc. and is not sponsored by Oracle, Inc.