Reflection Madness

Dr Heinz M. Kabutz

Javaspecialists.eu
java training

The Java Painkiller

- Reflection is like Opium
  - A bit too strong for every day use
  - But can relieve serious pain
  - Please do not become a Reflection Addict!

Heinz Kabutz

- **Author of The Java Specialists' Newsletter**
  - **Articles about advanced core Java programming**
- **http://www.javaspecialists.eu**

Java™ Champions

Introduction to Reflection

Introduction To Reflection

- **Java Reflection has been with us since Java 1.1**
  - **We can find out what type an object is and what it can do**
  - **We can call methods, set fields and make new instances**

Job interview: *"Do you know reflection?"*

"Yes, I do. You can use it to modify private final fields and call methods dynamically."

"This interview is over."

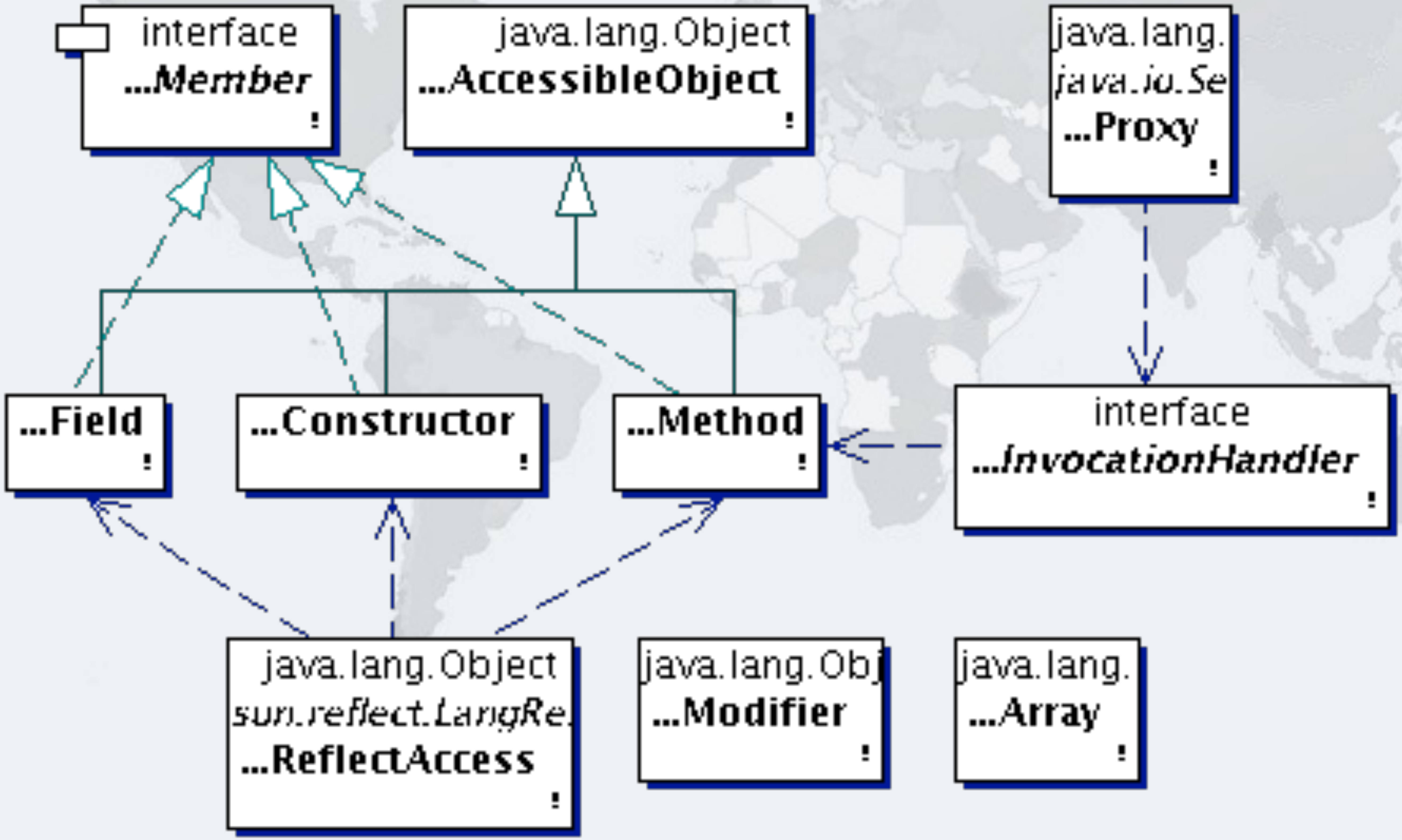
Benefits Of Reflection

- **Flexibility**
  - – Choose at runtime which methods to call
- **Raw Power**
  - – Background work such as reading private data
- **Magic Solutions**
  - – Do things you should not be able to do
    - • Sometimes binds you to JVM implementation

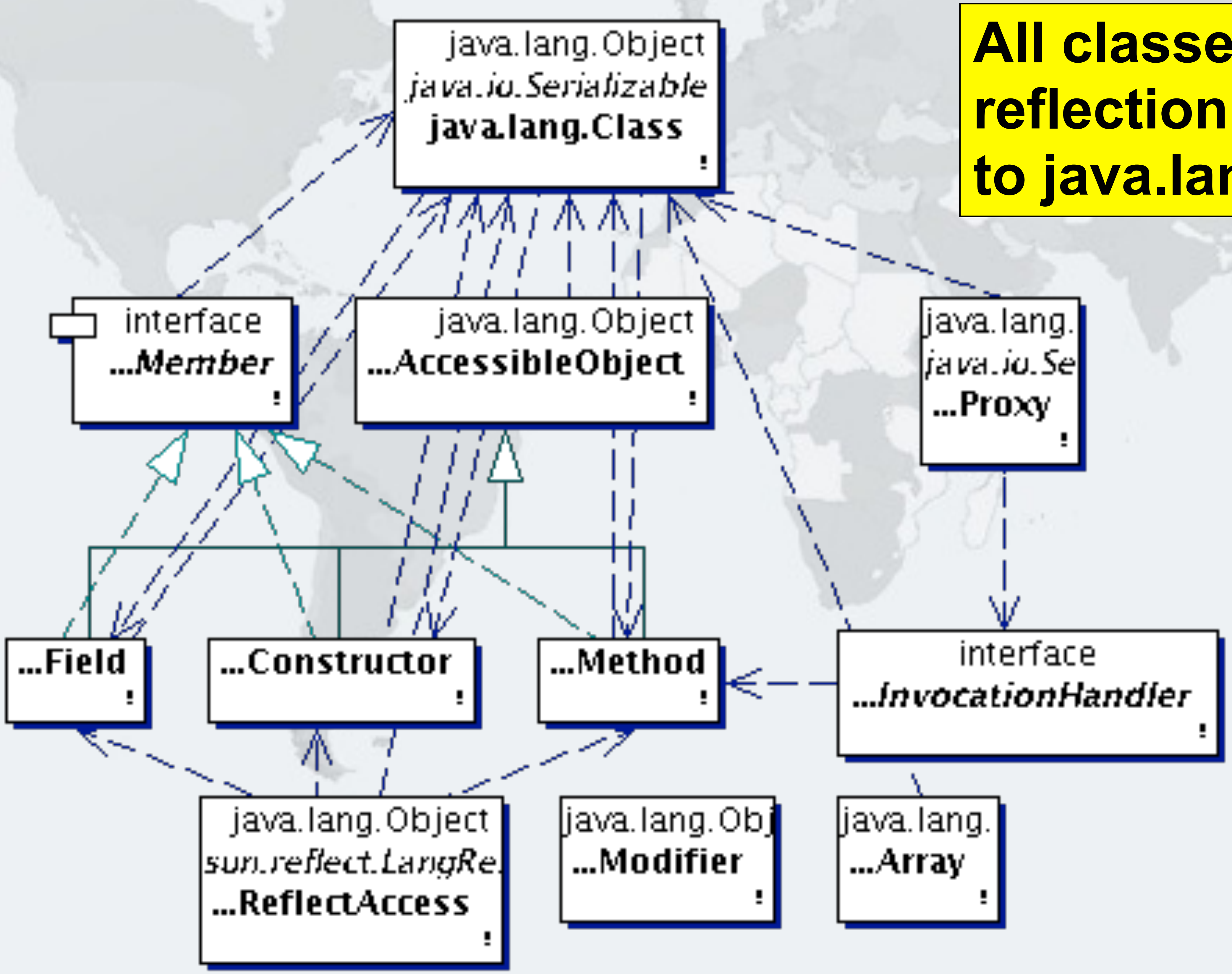
Dangers Of Reflection

- **Static Code Tools**
- **Complex Code**
- **Static compiling does not find typical errors**
  - **For example, code is written in XML and converted dynamically to Java objects**
- **Runtime Performance**
- **Limited Applicability**
  - **Does not always work in Sandbox**

Overview - Reflection Package



With Class Class Drawn In



All classes in reflection refer to java.lang.Class

Working With Class Objects

- Once we have the class object, we can find out information about what its objects can do:
  - What is the superclass?
  - What interfaces does it implement?
  - What accessible methods and fields does it have?
    - Include methods from parent classes
  - What are *all* the methods and fields defined in the class, including private and inaccessible?
  - What are the inner classes defined?
  - What constructors are available?
  - We can cast objects

More Interesting - What Can't We Do?

- **With standard reflection, we cannot find**
  - **Names of parameters for a method**
    - **Java 8 allows under certain conditions**
  - **Anonymous classes declared in methods and classes**
  - **Generic type parameters of an object**
    - **At runtime ArrayList<String> and ArrayList<Integer> are the same class**
- **We can find some of them with non-reflection tricks**

Accessing Members

- From the class, we can get fields, methods and constructors
  - getField(name), getDeclaredField
  - getMethod(name, parameters...), getDeclaredMethod
  - getConstructor(parameters...), getDeclaredConstructor
- Private members require setAccessible(true)

Modifying Private State

Private Members

- Can be made "accessible"
  - member.setAccessible(true)
  - Requires security manager support

```java
public class StringDestroyer {
  public static void main(String... args)
      throws IllegalAccessException, NoSuchFieldException {
    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    value.set("hello!", "cheers".toCharArray());
    System.out.println("hello!");
  }
}
```

```
cheers
```

Newsletter #014, March '01

- **String is a special case**
  - **Shared object between classes if the same static content**

```
System.out.println("hello!");
StringDestroyer.main(null);
System.out.println("hello!".equals("cheers"));
```

```
hello!
cheers
true
```

String History Lesson

- Java 1.0 - 1.2
  - String contained char[], offset, count
- Java 1.3 - 1.6
  - Added a cached hash code
  - String became a shared, mutable, but thread-safe class
- Java 1.7
  - Got rid of offset and length and added hash32
- Java 1.8
  - Got rid of hash32 again

Newsletter #102, January '05

- **Integers can also be mangled**
  - **Java typically caches auto-boxed Integers -128 to 127**
  - **We can modify these with reflection**

```
Field value = Integer.class.getDeclaredField("value");
value.setAccessible(true);
value.set(42, 43);
```

Destroying Integer Integrity

- **Integers are more vulnerable than Strings**

```
Field value = Integer.class.getDeclaredField("value");
value.setAccessible(true);
value.set(42, 43);

System.out.printf("Six times Seven = %d%n", 6 * 7);
```

```
Six times Seven = 43
```

Meaning Of Life

- **Hitchhiker's Guide to the Galaxy**
  - **Modifying a field related to hashCode is a *very* bad idea**

```
Field value = Integer.class.getDeclaredField("value");
value.setAccessible(true);
value.set(42, 43);


Map<Integer, String> meaningOfLife = new HashMap<>();
meaningOfLife.put(42, "The Meaning of Life");

System.out.println(meaningOfLife.get(42));
System.out.println(meaningOfLife.get(43));
```

```
The Meaning of Life
The Meaning of Life
```

Meaning Of Life

- **Hitchhiker's Guide to the Galaxy**
  - **Now we modify field after using it as a hash value**
  - **Newsletter # 031 from Nov '01**

```
Map<Integer, String> meaningOfLife = new HashMap<>();
meaningOfLife.put(42, "The Meaning of Life");

Field value = Integer.class.getDeclaredField("value");
value.setAccessible(true);
value.set(42, 43);

System.out.println(meaningOfLife.get(42));
System.out.println(meaningOfLife.get(43));
```

```
null
null
```

Size of Objects

Determining Object Size

- Object Size is not defined in Java
  - Differs per platform (Newsletters #029 Aug '01 and #078 Sep '03)
    - Java 1.0 - 1.3: Each field took at least 4 bytes
    - 32-bit: Pointer is 4 bytes, minimum object size 8 bytes
    - 64-bit: Pointer is 8 bytes, minimum object size 16 bytes
    - All platforms we looked at increase memory usage in 8 byte chunks
  - Can be measured with the Instrumentation API
    - Newsletter #142 - March '07
  - We traverse object graph using reflection and IdentityHashMap to avoid duplicates
    - You might need to define your own endpoints

Reflection-Based Memory Counting

- **Find all connected objects and measure size**
  - **Count each object only once (IdentityHashMap)**
  - **Skip shared objects (Strings, Boxed Primitives, Classes, Enums, etc.)**
- **Result is scary**
  - **In "C", "Heinz" was 6 bytes**
    - **String "Heinz" uses 80 bytes on a 64-bit JVM**
      - **Unless it is an "interned" String, then zero**
  - **Empty HashMap uses 216 bytes**
  - **List of 100 boolean values set to true**
      - **LinkedList uses 6472 bytes**
      - **ArrayList uses 3520 bytes**
  - **BitSet uses 72 bytes**

String Deduplication

- Java 1.8.0_20, with G1, char[]s in Strings shared
  - Happens automatically
  - Can save substantial memory on some systems
  - -XX:+UseG1GC -XX:+UseStringDeduplication
  - Show effect with -XX:+PrintStringDeduplicationStatistics

Reified Primitive Types?

- **Java 7 was going to support ArrayList<int>**
  - **Fortunately this was dropped**
  - **Each int would use 24 bytes!**
    - **Rather use primitive specific collection classes**

Instrumentation-Based Memory Counting

- **Implementation-specific *estimate* of object size**

```
public class MemoryCounterAgent {
  private static Instrumentation inst;

  /** Initializes agent */
  public static void premain(
      String agentArgs, Instrumentation inst) {
    MemoryCounterAgent.inst = inst;
  }

  /** Returns object size. */
  public static long sizeOf(Object obj) {
    return instrumentation.getObjectSize(obj);
  }
}
```

  - **Only a shallow size, for deep sizes we still need reflection**

Application of MemoryCounter

- Educational Tool
  - – Explains why Java needs 100 TB of RAM just to boot up
- Debugging
  - – One customer used it to discover size of user sessions
    - Need to define custom end-points in object graph
- Ongoing Monitoring
  - – Not that useful, too much overhead

Java Caller ID

Finding Out Who Called You

- **With Sun's JVM, we have sun.reflect.Reflection**
  - **Used in Class.forName(String)**

```
public class CallerID {
  public static Class<?> whoAmI() {
    return sun.reflect.Reflection.getCallerClass(2);
  }
}

public class CallerIDTest {
  public static void main(String... args) {
    System.out.println(CallerID.whoAmI());
  }
}
```

class CallerIDTest

@CallerSensitive

- **Java 8 disabled the call to Reflection.getCallerClass(int)**
- **For finding out direct caller, use**

```
Class<?> clazz = MethodHandles.lookup().lookupClass();
```

- **However, it doesn't give us classes deeper in stack**

Finding Out Who Called You #2

- **JVM independent using Exception Stack Traces**
  - **Does not tell you parameter types, only method name**

```
public class CallerID {
  public static String whoAmI() {
    Throwable t = new Throwable();
    StackTraceElement directCaller = t.getStackTrace()[1];
    return directCaller.getClassName() + "." +
        directCaller.getMethodName() + "()";
  }
}
```

class CallerIDTest.main()

Application of CallerID

- **Creating Loggers (Newsletter #137 - Dec '06)**
  - **Loggers often created with copy & paste**

```
public class Application {
  private final static Logger logger =
      Logger.getLogger(Application.class.getName());
}
```

Avoiding Copy & Paste Bugs

- **We can do this instead**

```java
public class LoggerFactory {
  public static Logger create() {
    Throwable t = new Throwable();
    StackTraceElement caller = t.getStackTrace()[1];
    return Logger.getLogger(caller.getClassName());
  }
}
```

```java
public class Application {
  private final static Logger logger =
      LoggerFactory.create();
}
```

Cost of Throwable.getStackTrace?

- **Creating a new Throwable is expeeeennnsssiiivvee**
  - **The fillInStackTrace() method is a native method call**
  - **However, the actual stack trace objects are empty**
    - **During debugging, if you want to see the actual stack trace of an exception, watch the getStackTrace() method (idea by Maik Jäkel)**
    - **The getStackTrace() method is even more expensive!**
- **However, you need call it only once per logger**

Finding Out Who Called You #3

- **JVM independent using Security Manager**

```
public class CallerID {
  public static Class<?> whoAmI() {
    MySecMgr sm = new MySecMgr();
    return sm.getClassContext()[2];
  }
  private static class MySecMgr extends SecurityManager {
    public Class[] getClassContext() {
      return super.getClassContext();
    }
  }
}
```

class CallerIDTest

Application of CallerID for JUnit

- **Make running unit tests from main()**

```
public class UnitTestRunner {
  public static void run() {
    MySecMgr sm = new MySecMgr();
    Class<?> clazz = sm.getClassContext()[2];
    System.out.println("Running unit tests for " + clazz);
    TestRunner.run(new JUnit4TestAdapter(clazz));
  }

  private static class MySecMgr extends SecurityManager {
    public Class[] getClassContext() {
      return super.getClassContext();
    }
  }
}
```

Normal Unit Test With JUnit 4

- **Cannot be directly invoked from the command line**

```
import org.junit.*;
import static org.junit.Assert.assertEquals;

public class MyTest {
  @Test
  public void testHello() {
    assertEquals("HELLO", "hello".toUpperCase());
  }
}
```

Augmented Unit Test With JUnit 4

- Context aware method UnitTestRunner.run()

```
import org.junit.*;
import static org.junit.Assert.assertEquals;

public class MyTest {
  @Test
  public void testHello() {
    assertEquals("HELLO", "hello".toUpperCase());
  }

  public static void main(String... args) {
    UnitTestRunner.run();
  }
}
```

Tests Automagically Run

- **Context aware method UnitTestRunner.run()**

```
Running unit tests for class MyTest
.
Time: 0.048

OK (1 test)
```

The Delegator

Automatic Delegator

- **MI5 want to see bytes flowing across my sockets**
  - **Java provides plugin methods to specify SocketImpl**

```
public class MonitoringSocketFactory
    implements SocketImplFactory {
  public SocketImpl createSocketImpl() {
    return new MonitoringSocketImpl();
  }
}

SocketImplFactory socketImplFactory =
    new MonitoringSocketFactory();
Socket.setSocketImplFactory(socketImplFactory);
ServerSocket.setSocketFactory(socketImplFactory);
```

  - **Only catch, default SocketImpl classes are package access**

Delegating To Inaccessible Methods

- All methods in SocketImpl are protected
- We cannot call them directly, only with reflection
  - But how do we know which method to call?
- We want to write

```
public void close() throws IOException {
  delegator.invoke();
}
```

```
public void listen(int backlog) throws IOException {
  delegator.invoke(backlog);
}
```

  - Should automatically call correct methods in wrapped object

Impossible?

- **With Stack Trace CallerID, we can get close**
  - **If there is a clash, we specify method explicitly**
  - **First, we find the method that we are currently in**

```
private String extractMethodName() {
  Throwable t = new Throwable();
  return t.getStackTrace()[2].getMethodName();
}
```

Finding The Correct Method By Parameters

- **Simple search**
  - **Find method with same name and number of parameters**
    - **Check that each of the objects are assignable**
  - **If not exactly *one* method is found, throw an exception**

```
private Method findMethod(String methodName, Object[] args) {
  Class<?> clazz = superclass;
  if (args.length == 0)
    return clazz.getDeclaredMethod(methodName);
  Method match = null;
  next:
  for (Method method : clazz.getDeclaredMethods()) {
    if (method.getName().equals(methodName)) {
      Class<?>[] classes = method.getParameterTypes();
      if (classes.length == args.length) {
        for (int i = 0; i < classes.length; i++) {
          Class<?> argType = classes[i];
          argType = convertPrimitiveClass(argType);
          if (!argType.isInstance(args[i])) continue next;
        }
        if (match == null) match = method;
        else throw new DelegationException("Duplicate");
      }
    }
  }
  if (match != null) return match;
  throw new DelegationException("Not found: " + methodName);
}
```

Manual Override

- **Delegator allows you to specify method name and parameter types for exact match**

```
public void connect(InetAddress address, int port)
    throws IOException {
  delegator
      .delegateTo("connect", InetAddress.class, int.class)
      .invoke(address, port);
}
```

Invoking The Method

- **Generics "automagically" casts to return type**

```
public final <T> T invoke(Object... args) {
  try {
    String methodName = extractMethodName();
    Method method = findMethod(methodName, args);
    @SuppressWarnings("unchecked")
    T t = (T) invoke0(method, args);
    return t;
  } catch (NoSuchMethodException e) {
    throw new DelegationException(e);
  }
}
```

When Generics Fail

- **Workaround: Autoboxing causes issues when we convert automatically**

```
public int getPort() {
  Integer result = delegator.invoke();
  return result;
}
```

- **Workaround: Inlining return type makes it impossible to guess what type it is**

```
public InputStream getInputStream() throws IOException {
  InputStream real = delegator.invoke();
  return new DebuggingInputStream(real, monitor);
}
```

Fixing Broken Encapsulation

- **Socket implementations modify parent fields directly**
  - **Before and after calling methods, we copy field values over**

```
writeFields(superclass, source, delegate);
method.setAccessible(true);
Object result = method.invoke(delegate, args);
writeFields(superclass, delegate, source);
```

Fixing Broken Encapsulation

- **Method writeFields() uses basic reflection**
  - **Obviously only works on fields of common superclass**

```
private void writeFields(Class clazz, Object from, Object to) {
  for (Field field : clazz.getDeclaredFields()) {
    field.setAccessible(true);
    field.set(to, field.get(from));
  }
}
```

Complete Code

- Newsletter #168 - Jan '09
  - Includes primitive type mapper
  - Allows you to delegate to another object
    - Without hardcoding all the methods
- Warning:
  - Calling delegated methods via reflection is *much* slower

Application Of Delegator

- **Wrapping of SocketImpl object**

```
public class MonitoringSocketImpl extends SocketImpl {
  private final Delegator delegator;

  public InputStream getInputStream() throws IOException {
    InputStream real = delegator.invoke();
    return new SocketMonitoringInputStream(getSocket(), real);
  }

  public OutputStream getOutputStream() throws IOException {
    OutputStream real = delegator.invoke();
    return new SocketMonitoringOutputStream(getSocket(), real);
  }

  public void create(boolean stream) throws IOException {
    delegator.invoke(stream);
  }

  public void connect(String host, int port) throws IOException {
    delegator.invoke(host, port);
  }
  // etc.
}
```

Alternative To Reflection

- **Various other options exist:**
  - **Modify SocketImpl directly and put into boot class path**
  - **Use Aspect Oriented Programming to replace call**
    - **Needs to modify all classes that call Socket.getInputStream() and Socket.getOutputStream()**

Of "Final" Fields

Manipulating Objects – Final Fields

- Final fields cannot be reassigned
- If they are bound at compile time, they will get inlined
- However, reflection may allow us to rebind them with some versions of Java
  - Can introduce dangerous concurrency bugs
  - Final fields are considered constant and can be inlined at runtime by HotSpot compilers
  - Only ever do this for debugging or testing purposes

Setting "Final" Field

- Can be set since Java 1.5
  - char[] value is actually "final"
    - We could still modify *contents* of array

```
public class StringDestroyer {
  public static void main(String... args)
      throws IllegalAccessException, NoSuchFieldException {
    Field value = String.class.getDeclaredField("value");
    value.setAccessible(true);
    value.set("hello!", "cheers".toCharArray());
    System.out.println("hello!");
  }
}
```

```
cheers
```

Setting "Static Final" Fields

- **Should not be possible, according to Lang Spec**
- **However, here is how you can do it (Sun JVM):**

1. **Find the field using normal reflection**
2. **Find the "modifiers" field of the Field object**
3. **Change the "modifiers" field to not be "final"**
    1. **modifiers &= ~Modifier.FINAL;**
4. **Get the FieldAccessor from the sun.reflect.ReflectionFactory**
5. **Use the FieldAccessor to set the final static field**

ReflectionHelper Class

- **Now we can set static final fields**
  - **Newsletter #161 in May '08**

```
import sun.reflect.*;  import java.lang.reflect.*;
public class ReflectionHelper {
  private static final ReflectionFactory reflection =
      ReflectionFactory.getReflectionFactory();

  public static void setStaticFinalField(Field field, Object value)
      throws NoSuchFieldException, IllegalAccessException {
    field.setAccessible(true);
    Field modifiersField = Field.class.getDeclaredField("modifiers");
    modifiersField.setAccessible(true);
    int modifiers = modifiersField.getInt(field);
    modifiers &= ~Modifier.FINAL;
    modifiersField.setInt(field, modifiers);
    FieldAccessor fa = reflection.newFieldAccessor(field, false);
    fa.set(null, value);
  }
}
```

Example of ReflectionHelper

```java
public class StaticFieldTest {
  private final static Object obj = "Hello world!";

  public static void main(String... args)
      throws NoSuchFieldException, IllegalAccessException {
    ReflectionHelper.setStaticFinalField(
        StaticFieldTest.class.getDeclaredField("obj"),
        "Goodbye cruel world!"
    );
    System.out.println("obj = " + obj);
  }
}
```

Goodbye cruel world!

Application Of Setting Final Fields

- **Create new enum values dynamically for testing**

```
public enum HumanState { HAPPY, SAD }

public class Human {
  public void sing(HumanState state) {
    switch (state) {
      case HAPPY: singHappySong(); break;
      case SAD:   singDirge();     break;
      default:
 throw 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, ...");
  }
}
```

Any problems?

New "enum" Values

Most Protected Class

- Enums are subclasses of java.lang.Enum
- Almost impossible to create a new instance
  - One hack was to let enum be an anonymous inner class
    - Newsletter #141 - March '07
    - We then subclassed it ourselves
    - This hack was stopped in Java 6
  - We can create a new instance using sun.reflect.Reflection
    - But the enum switch statements are tricky
      - Adding a new enum will cause an ArrayIndexOutOfBoundsException

Creating New Enum Value

- **We use the sun.reflect.ReflectionFactory class**
  - **The clazz variable represents the enum's class**

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

Original Human.Sing()

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

- **Let's see how this is converted into byte code**

An Inner Class Is Generated

- **Decompiled with Pavel Kouznetsov's JAD**

```
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) { }
    }
  }
  ...
```

Generated Enum Switch

```
switch(_cls1.$SwitchMap$HumanState[state.ordinal()]) {
  case 1:
    singHappySong();
    break;
  case 2:
    singDirge();
    break;
  default:
    new IllegalStateException(
      "Invalid State: " + state);
    break;
}
```

Modifying Enum "Switch" Statements

- **Follow this procedure:**
  1. **Specify which classes contain enum switch statements**
  2. **For each class, find all fields that follow the pattern $SwitchMap$enum_name**
  3. **Make fields (int[]) larger by one slot**
  4. **Set field values to new int[]**

Memento Design Pattern

- Every time we make a change, first copy the state
  - Allows us to undo previous change
  - Useful for testing purposes
- EnumBuster class contains stack of undo mementos

```
EnumBuster<HumanState> buster =
  new EnumBuster<>(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()));
}
```

Test Output

- **When we run it, we should see the following**

```
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)
```

  - **Note that when the test run is complete, all the classes have been changed back to what they were before**

Constructing without Constructor

Serialization Basics

- **When we serialize an object, fields are read with reflection and written to stream**
- **When we deserialize it again, an object is *constructed without calling the constructor***
  - **We can use the same mechanism to create objects**

Basic Class

- **Whenever this object is instantiated, a message is printed to console**
  - **Furthermore, i is always 42**

```
public class MyClass {
  private int i = 42;

  public MyClass(int i) {
    System.out.println("Constructor called");
  }

  public String toString() {
    return "MyClass i=" + i;
  }
}
```

Serialization Mechanism

- **Serialization can make objects without calling constructor**
  - **We can use the same mechanism (JVM specific)**

```
ReflectionFactory rf =
  ReflectionFactory.getReflectionFactory();
Constructor objDef = Object.class.getDeclaredConstructor();
Constructor intConstr = rf.newConstructorForSerialization(
    MyClass.class, objDef
);

MyClass mc = (MyClass) intConstr.newInstance();
System.out.println("mc = " + mc.toString());
System.out.println(mc.getClass());
```

```
mc = MyClass i=0
class MyClass
```

Unsafe

- **Alternatively, we can use sun.misc.Unsafe**
  - **Again, JVM specific**

```
Object o = Unsafe.getUnsafe().allocateInstance(
    MyClass.class);
System.out.println("o = " + o.toString());
System.out.println(o.getClass());
```

Or Just Make New Constructor

```
public class MagicConstructorMaker {
  public static <T> Constructor<T> make(Class<T> clazz)
      throws NoSuchMethodException, IllegalAccessException,
      InvocationTargetException, InstantiationException {
    Constructor<?> constr =
      Constructor.class.getDeclaredConstructor(
        Class.class,    // Class<T> declaringClass
        Class[].class,  // Class<?>[] parameterTypes
        Class[].class,  // Class<?>[] checkedExceptions
        int.class,      // int modifiers
        int.class,      // int slot
        String.class,   // String signature
        byte[].class,   // byte[] annotations
        byte[].class);  // byte[] parameterAnnotations
    constr.setAccessible(true);
...
```

And Create New Constructor Object

- **This even works on Android Dalvik**
  - **However, Java 8 core dumps with wrong slot #**

```
  int slot = clazz.getDeclaredConstructors().length + 1;
  return (Constructor<T>) constr.newInstance(
    clazz,
    new Class[0],
    new Class[0],
    Modifier.PUBLIC,
    slot,
    "MyMagicConstructor",
    null,
    null);
 }
}
```

Singletons?

- Classic approach is private constructor
  - More robust: throw exception if constructed twice

```
public class Singleton {
  private final static Singleton instance = new Singleton();

  private Singleton() {
    if (instance != null)
      throw new IllegalStateException("Duplicate singletons!!!");
  }

  public static Singleton getInstance() {
    return instance;
  }
}
```

Singletons?

- **Make new Singleton objects**
  - **My other techniques also work**

```
Singleton.getInstance();
Singleton s = MagicConstructorMaker.make(Singleton.class).newInstance();
System.out.println(s == Singleton.getInstance());
System.out.println(s);
System.out.println(Singleton.getInstance());
```

Application: Constructing Without Constructor

- **Useful when you need to recreate an object**
  - **e.g. Copy an object, de-persist it, etc.**

Externalizable Hack

Standard Serializing Approach

- Class implements Serializable
  - Usually *good enough*
- Next step is to add writeObject() and readObject()
  - Avoids reflection overhead
    - This is usually not measurable
  - Allows custom optimizations
- Class implements Externalizable
  - May be a tiny bit faster than Serializable
  - But, opens security hole

Serializable Vs Externalizable

- Writing of object
  - Serializable
    - Can convert object to bytes and read that - cumbersome
  - Externalizable
    - pass in a bogus ObjectOutput to gather data
- Reading of object
  - Serializable
    - cannot change state of an existing object
  - Externalizable
    - use bogus ObjectInput to modify existing object

```
public class MovieCharacter implements Externalizable {
  private String name;
  private boolean hero;

  public MovieCharacter(String name, boolean hero) {
    this.name = name;
    this.hero = hero;
  }

  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeUTF(name);
    out.writeBoolean(hero);
  }

  public void readExternal(ObjectInput in) throws IOException {
    name = in.readUTF();
    hero = in.readBoolean();
  }

  public String toString() {
    return name + " is " + (hero ? "" : "not ") + "a hero";
  }
}
```

Bogus ObjectInput Created

```
public class HackAttack {
  public static void hackit(MovieCharacter cc, String
      final boolean hero) throws Exception {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(cc);
    oos.close();

    ObjectInputStream ois = new ObjectInputStream(
        new ByteArrayInputStream(baos.toByteArray())
    ) {
      public boolean readBoolean() throws IOException {
        return hero;
      }
      public String readUTF() { return name; }
    };
    cc.readExternal(ois); // no security exception
  }
}
```

Bogus ObjectInput Created

```
public class HackAttackTest {
  public static void main(String... args)
      throws Exception {
    System.setSecurityManager(new SecurityManager());
    MovieCharacter cc = new MovieCharacter("John Hancock", true);
    System.out.println(cc);

    // Field f = MovieCharacter.class.getDeclaredField("name");
    // f.setAccessible(true); // causes SecurityException

    HackAttack.hackit(cc, "John Hancock the drunkard", false);

    // now the private data of the MovieCharacter has changed!
    System.out.println(cc);
  }
}
```

```
John Hancock is a hero
John Hancock the drunkard is not a hero
```

Application: Externalizable Hack

- Be careful with using Externalizable
  - We can change the state of an existing object
- With Serializable, we can create bad objects
  - A lot more effort
  - Should be checked with ObjectInputValidation interface
- Slight performance gain might not be worth it

Soft References and Reflection

Reflection and SoftReferences

- **Reflection information stored as soft refs**
  - **Created lazily on first use**
  - **Can be turned off with**

    -Dsun.reflect.noCaches=true

- **Reflection information costs approximately 24KB per class and takes about 362 µs to generate**

Demo

Effects of not having caches

Effects On Performance

- **Soft References are cleared when system is under memory pressure**
  - **Cache essential reflection information**
  - **Otherwise you get noCaches=true performance**
- **Danger: SoftReferences cause a quadratic degradation of performance during GC**
  - **Don't use them**

Don't Use Softreference

- Don't use SoftReference
- Don't use SoftReference
- Don't use SoftReference
- Don't use SoftReference
- Don't use SoftReference
- Don't use SoftReference
- Don't use SoftReference

Do Not Use SoftReference

Conclusion

- **Reflection allows us some neat tricks in Java**
  - **Great power also means great responsibility**
  - **Don't overdo it, use sparingly**
- **Tons of free articles on JavaSpecialists.EU**
  - **http://www.javaspecialists.eu/archive**

Reflection Madness

Dr Heinz M. Kabutz

heinz@javaspecialists.eu

I would love to hear from you!

The Java Specialists' Newsletter