11:25

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!**

Background

- **Heinz Kabutz**
  - **Lives in Greece on the Island of Crete**
  - **Java Programmer**
  - **The Java Specialists' Newsletter**
    - **50 000 readers in 121 countries**
    - **http://www.javaspecialists.eu**
  - **Java Champion**

Two Events in Crete

- **Java Specialists Symposium Crete 29 Aug - 1 Sep**
  - **"Open Spaces" conference**
  - **Title "Making Java Fun Again"**
  - **Free entry, but seats are limited**
- **Java Specialists Master Course Crete 5-8 Sep**
  - **Advanced Java Course for Java experts**
  - **€2500 per seat**
  - **You may also attend this remotely**

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**

Popular interview question:
"Do you know reflection?"

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

"This interview is over. Thanks for applying and good luck for your future."

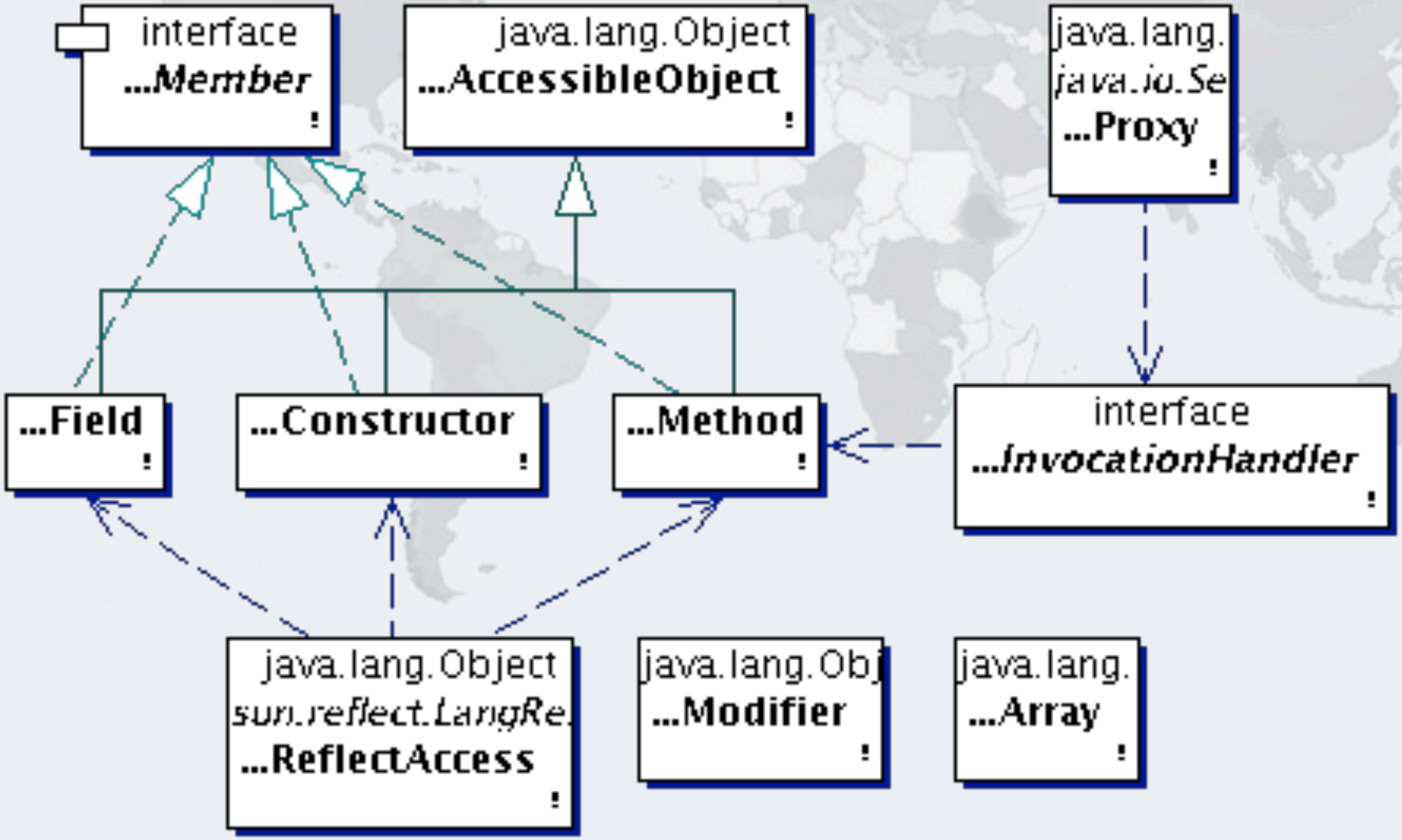
Benefits of Reflection

- **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

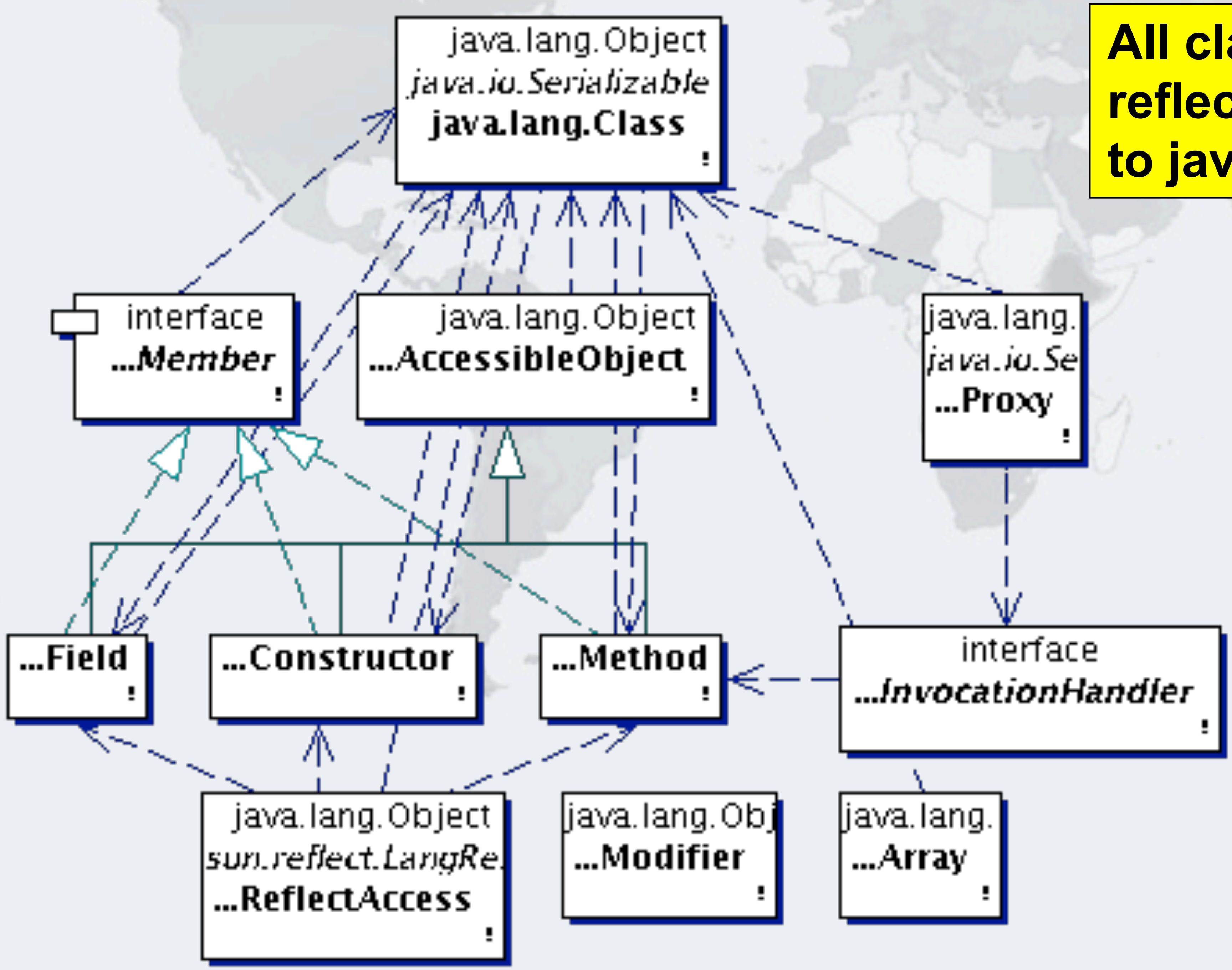
Dangers of Reflection

- 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**

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**

```
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, 2001-03-21

- **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
```

Newsletter 102, 2005-01-31

- **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<Integer, String>();
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**

```
Map<Integer, String> meaningOfLife =
    new HashMap<Integer, String>();
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
    - 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
  - 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

Reified Primitive Types?

- How will ArrayList<int> be implemented?
  - If with Integers, we will use 24 bytes per Integer instead of 4 for an int
    - Rather use primitive specific collection classes

Instrumentation-Based Memory Counting

- Returns an implementation-specific *estimate* of object size
  - Only a shallow size, for deep sizes we still need reflection

```
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);
  }
}
```

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
```

Finding Out Who Called You #2

- **JVM independent using Exception Stack Traces**
  - **Does not tell you parameters, 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)**
  - **Instead of the typical**

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

  - **We can do this**

```
public class LoggerFactory {
  public static Logger create() {
    Throwable t = new Throwable();
    StackTraceElement caller = t.getStackTrace()[1];
    return Logger.getLogger(caller.getClassName());
  }
}
  // in Application
  private final static Logger logger =
      LoggerFactory.create();
```

Alternative approaches

- **Security Manager to find caller**
- **sun.reflect.Reflection.getCallerClass(1)**

The Delegator

Automatic Delegator

- **Use Case**
  - **We want to count all the bytes flowing across all the sockets in our Java virtual machine**
    - **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?
- Here is what we want to do:

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

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

  - This should automatically call the correct methods in the wrapped object

Impossible?

- **With CallerID, we can get close**
  - **If there is a clash, we specify explicitly what method to call**
  - **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

```
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 correct 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);
```

  - **Method writeFields() uses basic reflection**

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

- **Obviously only works on fields of common superclass**

Complete Code

- **Newsletter #168**
  - **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**

```java
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"

3.1. 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**

```
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);
  }
}
```

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**
    - **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);
```

Generated Enum Switch

- **Decompiled with Pavel Kouznetsov's JAD**
  - **The clazz variable represents the enum's superclass**

```
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;
  }
}
```

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**

Testing Human Class

```
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()));
}
```

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());
```

Singletons?

- **Classic approach is private constructor**
  - **More robust: throw exception if constructed twice**
- **With Unsafe and ReflectionFactory we can construct objects without calling constructor!**

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**

Our MovieCharacter Class

```
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, final String name, 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

```java
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

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**

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
- Advanced Java Courses available
  - http://www.javaspecialists.eu/courses
  - See you in Crete in September 2011 :-)

Reflection Madness

Dr Heinz M. Kabutz

heinz@javaspecialists.eu

I would love to hear from you!