Architecture of software systems

Course 2: Virtual machines, byte-code, decompiler, code protection, classloaders, reflection

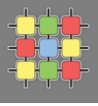
David Šišlák
david.sislak@fel.cvut.cz

Introduction - Programming languages

- from source-execution flow perspective:
  - scripting – no pre-compilation
  - with compilation
    - into native-code
    - into byte-code
- from execution perspective:
  - interpreted (script or byte-code/threaded code)
  - machine-code execution
  - hybrid – both interpreted and machine-code execution

- BUT
  - ahead-of-time (AOT) compiler – byte-code into native-code compiler
    for JAVA, .NET and others

Introduction - Programming languages



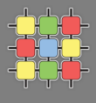
- Interpreted vs. machine-code
  - (+) platform independence – architecture (RISC/CISC, bits), OS
  - (+) reflection – observe, modify own structure at run-time
  - (+) dynamic typing – at run-time (JAVA has no dynamic typing !)
  - (+) small size
  - (+) dynamic scoping – (Perl)
  - (-) slower execution – interpreted mode, JIT latencies

```
var x := 5;
var y := "37";
var z := x + y;
```

```
$x = 0;
sub f { return $x; }
sub g { my $x = 1; return f(); }
print g()."\n";
```

```
$x = 0;
sub f { return $x; }
sub g { local $x = 1; return f(); }
print g()."\n";
```

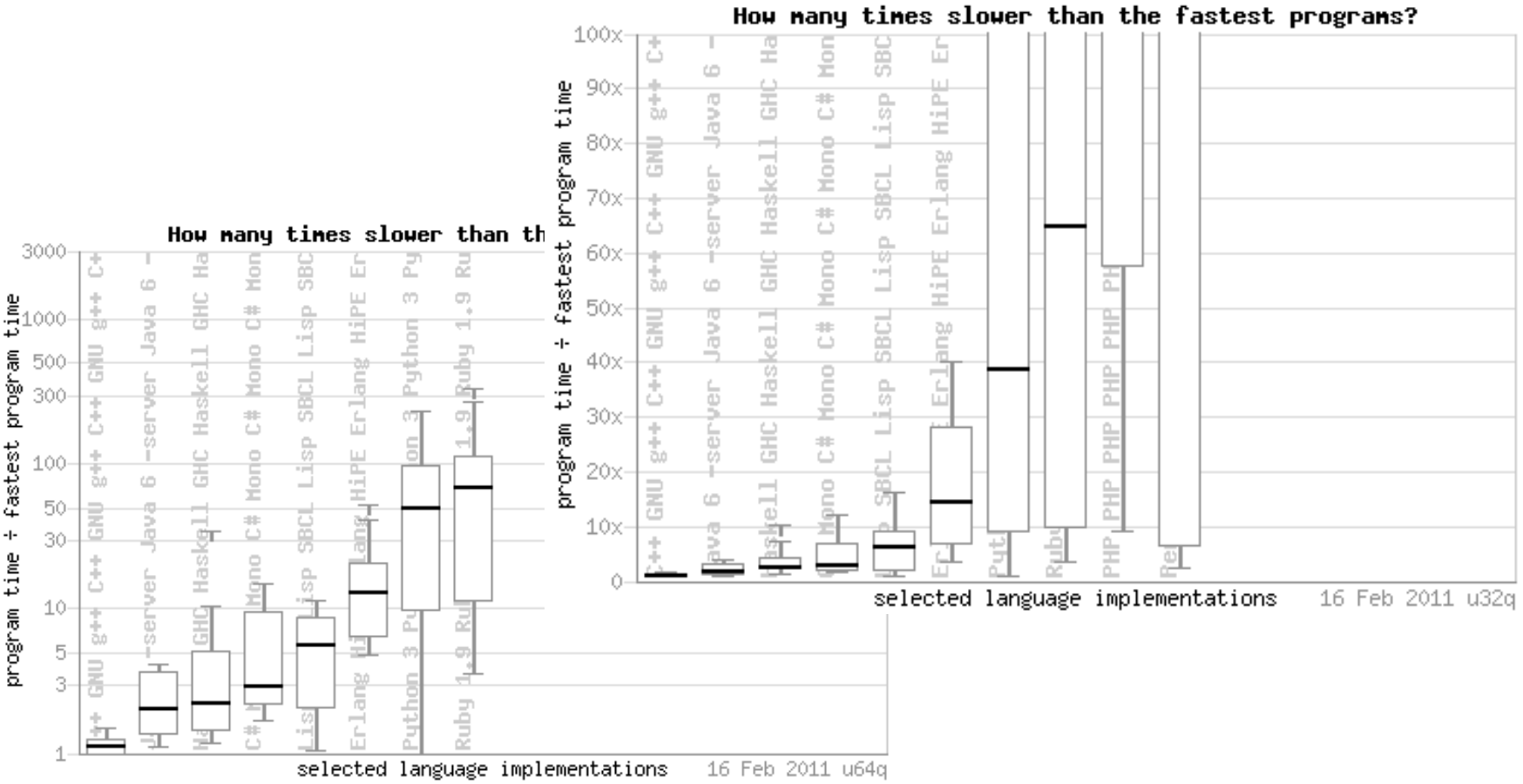
Introduction - Programming languages



» The Computer Language Benchmarks Game

(source: http://shootout.alioth.debian.org/)

» 24 langs, 13 different domains (binary-trees, fast, n-body, etc.)



Introduction - JAVA

- first release 1995 by Sun Microsystems (now Oracle)
- many different implementations (GNU, IBM, etc.)

- standard source-execution model:
  - create: source (.java)
  - compile: byte-code (.class)
  - run-time: Java Virtual Machine (JVM) in Java Runtime Environment (JRE)
    - interpreter: byte-code
    - Just-in-Time (JIT) compiler: native-code

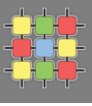
Java virtual machine and byte-code

- Why understand byte-code?
  - helps the same way as assembler helps C programmer
  - tuning your programs
- **stack-oriented** - stack machine model for passing parameters

$$(2 + 3) \times 11 + 1$$

Input	2	3	add	11	mul	1	add
Stack	2	3 2	5	11 5	55	1 55	56

Java virtual machine and byte-code

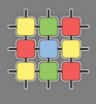


- opcode (1 byte + various parameters):
  - load and store (aload_0, istore, aconst_null, …)
  - arithmetic and logic (ladd, fcmpl, …)
  - type conversion (i2b, d2i, …)
  - object manipulation (new, putfield, getfield, …)
  - stack management (swap, dup2, …)
  - control transfer (ifeq, goto, …)
  - method invocation (invokespecial, areturn, …)
  - exceptions and monitor concurrency (athrow, monitorenter, …)
- prefix/suffix – i, l, s, b, c, f, d and a (reference)
- variables as registers – e.g. istore_1 (variable 0 is **this**)

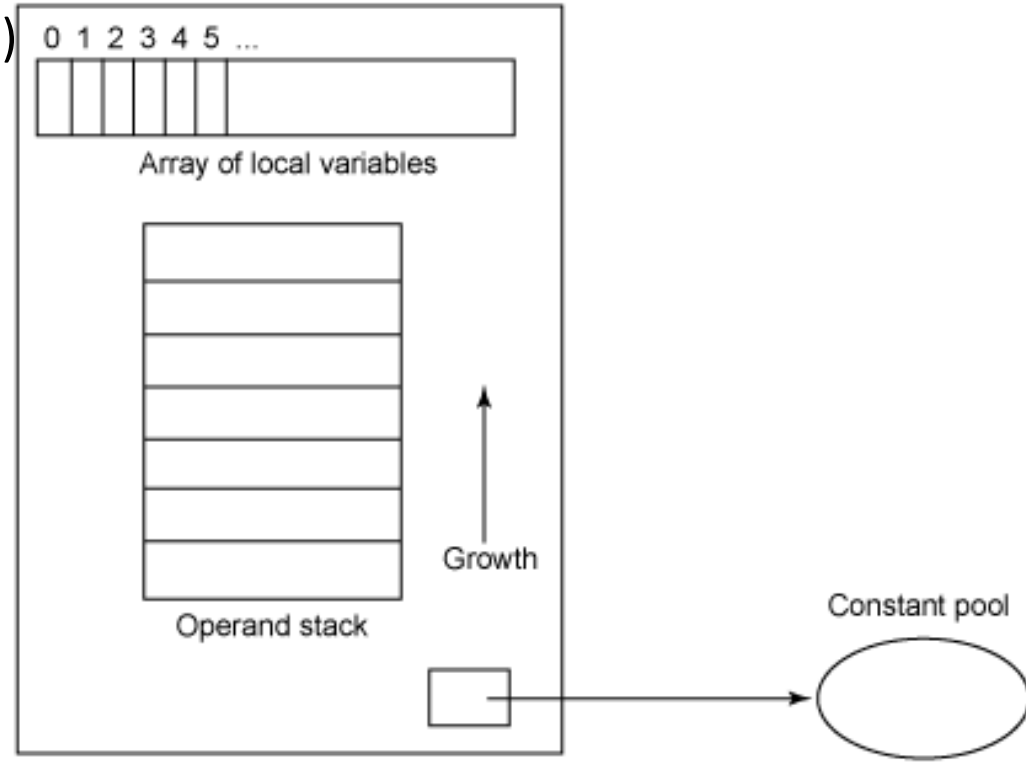
```
add eax, edx
mov ecx, eax
```

VS.

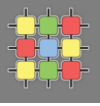
```
0 iload_1
1 iload_2
2 iadd
3 istore_3
```



- **frame**
  - each thread has stack with frames (heap located, not contiguous)
    StackOverflowError, OutOfMemoryError
  - frame is created each time method is invoked (after it is destroyed)
  - size determined at compile-time (in class file)
  - variables (long, double in two)
    - {this}
    - {method parameters}
    - {local variables}
  - operand stack (any type)
    - LIFO
  - reference to run-time
    constant pool (class/interf)
  - method + class is associated

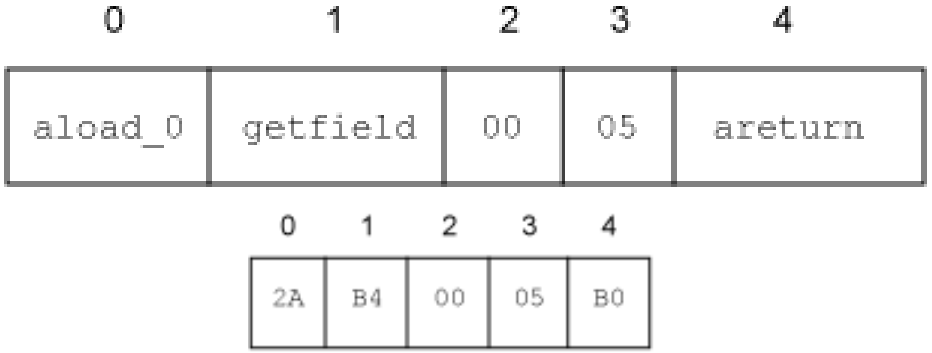


Java virtual machine and byte-code

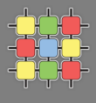


```
public String employeeName()
{
return name;
}

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn
```

- getfield
  - takes 1 ref from stack
  - build an index into runtime pool of class by reference
- areturn
  - takes 1 ref from stack
  - push onto the stack of calling method



Java virtual machine and byte-code



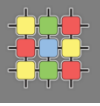
```
public Employee(String strName, int num)
{
name = strName;
idNumber = num;
storeData(strName, num);
}

Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object()>
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void storeData(java.lang.String, int)>
20 return
```

EXAMPLE 1

(see different size and speed issue)

- javac xxx (see class file structure)
- *java xxx*
- javap -c -l -s xxx

Java virtual machine and byte-code



- Summary of Example1
  - byte-code is not better than your source code
    - invariants in loop are not removed
  - no optimizations like
    - loop unrolling
    - algebraic simplification
    - strength reduction

- debug information (affect the size of bytecode)
  - -g
  - -g:none

Java virtual machine and byte-code

- method area shared among all threads
  - class definitions
    - run-time constant pool
    - field and method data
    - byte-code for methods and constructors
    - initialization methods (**<clinit>**, **<init>**)

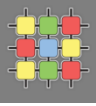
- native method stacks
  - implementation of native methods

EXAMPLE 2

(more complex example + decompiler)

- e.g. jad
- javap -c -l -s -private Test
- jad -p Test.class
- try: -g:none

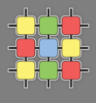
Protection techniques



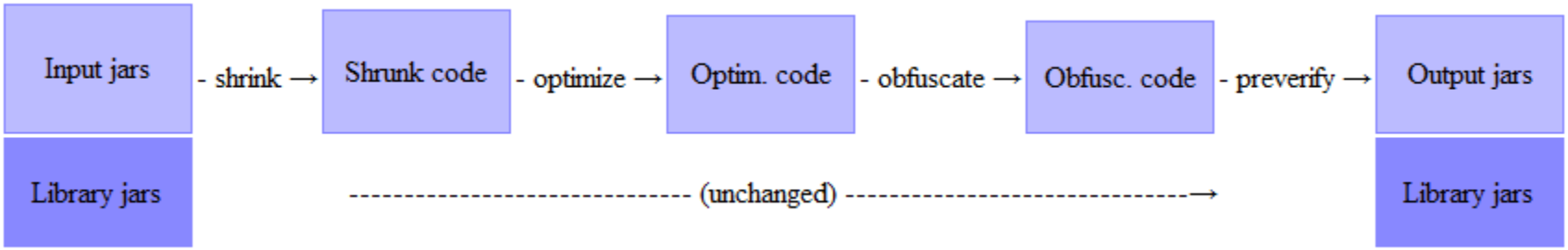
- Protect your code
  - no physical access to the program – e.g. client-server model
  - encryption of code - if not in hardware, can be intercepted
  - native codes instead of byte-code
  - code obfuscation

- Obfuscator
  - the same functional result
  - difficult to understand
  - obfuscations
    - layout – identifier names, class definitions
    - data – local <-> global, encoding, aggregation (e.g. arrays), ordering
    - control – aggregation, ordering, control flow
    - preventative transformations – extra instructions, mix commands

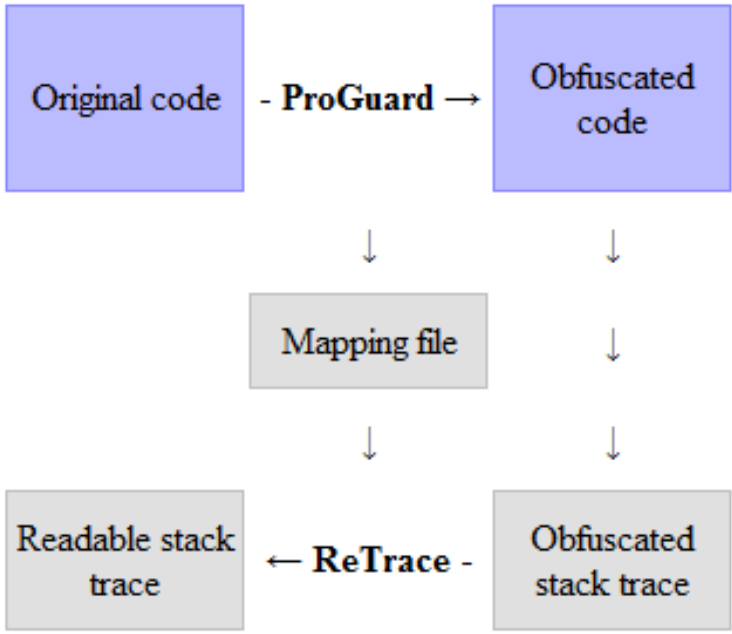
JAVA obfuscators

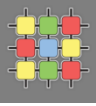


- many various open-source obfuscators available
- one example **(changes in byte-code)**



- ProGuard
  - define entry points !
  - shrinker- compact code, remove dead code
  - optimizer - faster code
    - private, static, final
    - inline
  - obfuscator
    - renaming, layout, etc.
    - automatically detect reflection
  - preverifier – check for Java 6 loading



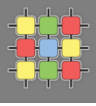


EXAMPLE 2 cont.

(obfuscator + decompiler)

- jar -ce Test Test.class >Test.jar (see jar structure)
- *java –jar Test.jar*
- java –jar proguard.jar @applications.pro (see .pro file)
- jad –p Test.class

Java virtual machine



- can implement also other languages than JAVA
  - Erlang -> Erjang
  - JavaScript -> Rhino
  - Python -> Jython
  - Ruby -> Jruby
  - etc.
- java virtual machine VS. java processor

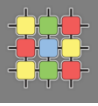
- byte-code is verified before executed:
  - branches (jump) are always to valid locations – only within method
  - any instruction operates on a fixed stack location (helps JIT for registers)
  - data is always initialized and references are always type-safe
  - access to private, package is controlled

- heap – area for dynamic memory allocation (all objects)

Java virtual machine

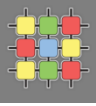
- Just-in-time (JIT)
  - converts byte-code into native code in run-time
  - different version for client, server run (-client, -server)
  - client – three phase, background compilation
    - high-level intermediate representation (HIR) – platform independent
    - low-level intermediate representation (LIR) – platform specific
    - final phase
      - window-based optimization over small set of instr. of LIR
      - inlining of functions without exception handlers or synchroniz.
      - generate machine code from LIR
  - server – high-end fully optimizing compiler
    - dead code elimination, loop invariant hoisting, common sub-expression elimination, constant propagation
    - full inlining, full deoptimization

Java virtual machine



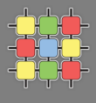
- adaptive optimization
  - balance trade-off between JIT and interpreting instructions
  - monitors frequently executed parts “hot spots” including data on caller-callee relationship for virtual method invocation
  - makes dynamic recompilation based on current execution profile
  - inline expansion to remove context switching
  - optimize branches
  - can make risky assumption (e.g. skip code) ->
    - unwind to valid state
    - deoptimize previously JITed code even if code is already executed

Java virtual machine



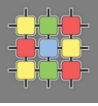
- key startup parameters related with JIT and optimization
  - server (defualt for 64-bit edition)
  - XX:+CITime (prints time spent in JIT compiler)
  - XX:+PrintCompilation
  - XX:CompileThreshold=XXXX (number of invocation, branches)
  - XX:+PrintClassLoading

Java virtual machine – start-up



- create initial class
  - must be present in bootstrap class loader
- links the initial class
  - cause loading, linking and invocation of other classes
- initialize it (class vs. instance initialization !)
- start public void main (String[])

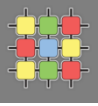
Loading and Linking



- Loading
  - finding binary form of class or interface (e.g. computing on the fly)
  - form **Class**
  - implemented by **ClassLoader**
    - can cache binary representations
    - prefetch them based on expected usage
    - load group of related classes together

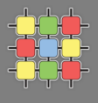
- Linking
  - binary form into run-time state in JVM
  - verification – structural check (correct opcodes, branches, …)
  - preparation
    - create static fields, fill default values (no initializers !)
    - precompute additional data structures (e.g. method table)
  - resolution of symbolic references – validation, direct reference

Class initialization



- class is being initialized in the following cases:
  - instance of class has to be created
  - static method of class is invoked
  - non-constant static field of class is used
  - sub-class is initialized
  - Invocation of reflective methods
  - it is initial class for start-up
- class is not initialized
  - when static final field is initialized with compile-time constant
- **class initialization sequence**
  - super class initialization
  - initialization in declaration order (you cannot use values after, compiled into "<cinit>:()V") :
    - user class static initializers
    - initializers static fields (class + interfaces – by default public static final)

Class initialization



- requires careful synchronization (synchronized on class object)
- **Class object state**
  - verified and prepared
  - being initialized by some thread
  - fully initialized and ready for use
  - in error state – verification failed, initialization failed

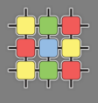
    (throws NoClassDefFoundError)
- **Class instance** initialization
  - memory allocation (fields in class + superclasses) -> OutOfMemoryError
  - all variables are set to default values (0, false, null)
  - prepare args for constructor invocation (follow this(…) and follow super(…))
  - execute instance initializers + field initializers in declaration order (compiled into "<init>:()V”)
  - execute the rest of the body of constructor

Class instance finalization

- protected void finalize() throw Throwable
  - called before object space is reclaimed by GC
  - you should call super.finalize() !!!
  - do not call method explicitly !!!

- class/interface unloading
  - if its class loader is unreachable
  - bootstrap class loader is always reachable -> system classes are never unloaded !

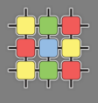
Initializer block



```
public static class InitializerBlocks {

    public InitializerBlocks(){
        System.out.println("1");
    }

    {
        System.out.println("2");
    }

    {
        System.out.println("3");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

Initializer block



```
public static class InitializerBlocks {

    public InitializerBlocks(){
        System.out.println("1");
    }

    {
        System.out.println("2");
    }

    {
        System.out.println("3");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

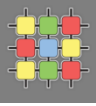
» what is the output?

2

3

1

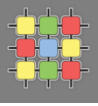
Initializer block - alternative



```
public static class InitializerBlocks {
    private boolean boolField = privInstance();

    private final boolean privInstance() {
        System.out.println("2");
        return true;
    }

    public InitializerBlocks(){
        System.out.println("1");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

Initializer block - alternative



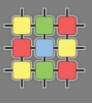
```
public static class InitializerBlocks {
    private boolean boolField = privInstance();

    private final boolean privInstance() {
        System.out.println("2");
        return true;
    }

    public InitializerBlocks(){
        System.out.println("1");
    }

    public static void main(String[] args) {
        new InitializerBlocks();
    }
}
```

» what is the output?

2

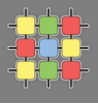
1

Initializer block – static variant



```
public static class InitializerBlocks {
    private static final boolean boolField;

    static {
        // compute value for boolField ...
        boolField = true;
    }

}
```

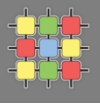
Initializer block – static example



```
public static class A {
    public static final A A = new A();
    private A() { }

    private static final Boolean BOOL = true;

    private final Boolean bool = BOOL;

    public final Boolean bool() { return bool; }

    public static void main(String[] args) {
        System.out.println(A.bool() ?
                "yes" : "no");
    }
}
```

» what is the output?

Initializer block – static example



```
public static class A {
    public static final A A = new A();
    private A() { }

    private static final Boolean BOOL = true;

    private final Boolean bool = BOOL;

    public final Boolean bool() { return bool; }

    public static void main(String[] args) {
        System.out.println(A.bool() ?
                "yes" : "no");
    }
}
```

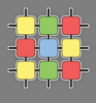
- what is the output?
  - throws NullPointerException – due to recursive class initialization, and auto-unboxing
- correct:

```
public static class A {
    private static final Boolean BOOL = true;

    public static final A A = new A();
    private A() { }

    private final Boolean bool = BOOL;

    public final Boolean bool() { return bool; }

    public static void main(String[] args) {
        System.out.println(A.bool() ?
                "yes" : "no");
    }
}
```

Classloader

- classloader types:
  - **bootstrap class loader**
    - system class loader – searches runtime, inst. extension, class path
  - **user-defined class loader**
    - extraction from encrypted file, verify digital signature
    - loading from non-standard sources (e.g. network)
    - generate on the fly
- each class has
  - defining class loader – finally define Class
  - initiating class loader – initiate class loading (e.g. through other CL)
- class is uniquely identified by pair !
  - fully qualified name
  - defining class loader

Classloader



- referenced classes from X are loaded by its defining CL
- each class is loaded only once if it is not previously unloaded
- method **Class loadClass(String)** - qualified name
  - a cache implemented by **Class findLoadedClass(String)**
  - get raw bytes from class from somewhere
  - if ok, define class from array of bytes using **Class defineClass(…)**
  - if failed, delegate loading to other class loader
    - e.g. **Class findSystemClass(String)**
    - e.g. **getParent().loadClass(String)** – CL which creates the current one
  - if still no class, throw **ClassNotFoundException**
  - if resolve is required call **void resolveClass(Class)** to link class
- **Class Class.forName(String)**, **Class.getSystemClassLoader().loadClass(String)**
- **cl.loadClass(String), c.newInstance(), constructor.newInstance(…)**

Reflection

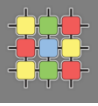
- can examine or modify the run-time behavior
  - create external classes by qualified name (through Class loaders) – do not need to have class during compilation
  - class browser – enumerate members
  - debugger – examine private members

- BUT
  - performance overhead – dynamic resolution, slower
  - security restrictions – security context, e.g. Applet
  - unexpected side-effects – access private fields and methods

Reflection

- retrieve Class object
  - **Class getClass()** – returns instance Class representation
  - **XXX.class** – from type, no instance
    - e.g. “aa”.class
  - **Class Class.forName(String), CL.loadClass(String)**
  - **Class.getSuperClass(), Class.getClasses(), Class.getDeclaredClasses(), Class.getEnclosingClass(), {Field|Method|Constructor}.getDeclaringClass()**
- examine class modifiers and types
  - **Class.getModifiers()**
  - **Class.getTypeParameters()** – get Generic types
  - **Class.getGenericInterfaces()**
  - **Class.getSuperclass()**
  - **Class.getAnnotations()**

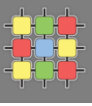
Reflection



» discovering class members

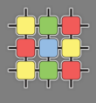
Member	Class API	List of members?	Inherited members?	Private members?
Field	getDeclaredField()	no	no	yes
	getField()	no	yes	no
	getDeclaredFields()	yes	no	yes
	getFields()	yes	yes	no
Method	getDeclaredMethod()	no	no	yes
	getMethod()	no	yes	no
	getDeclaredMethods()	yes	no	yes
	getMethods()	yes	yes	no
Constructor	getDeclaredConstructor()	no	N/A[1]	yes
	getConstructor()	no	N/A[1]	no
	getDeclaredConstructors()	yes	N/A[1]	yes
	getConstructors()	yes	N/A[1]	no

Reflection



- Fields
  - get field types, generic types
  - get field modifiers
  - get and set field value (private if no security manager)
- Methods
  - get method types including attributes
  - get method modifiers
  - invoke method
- Constructors
  - find constructor with specific parameters
  - get constructor modifiers
  - create new class instance
- Arrays (through java.lang.reflect.Array)
  - get array types
  - create new array
  - get/set array components

Reflection



» Method example:

```
Class<T> c = Class.forName("MyClass"); // Class<T> c = MyClass.class;
Method m = c.getMethod(“myMethod”);
Object retVal = m.invoke(object, …);
```

» Field example:

```
Class<T> c = MyClass.class;
Field f = c.getField(“myField”);
Object value = f.get(object);
```