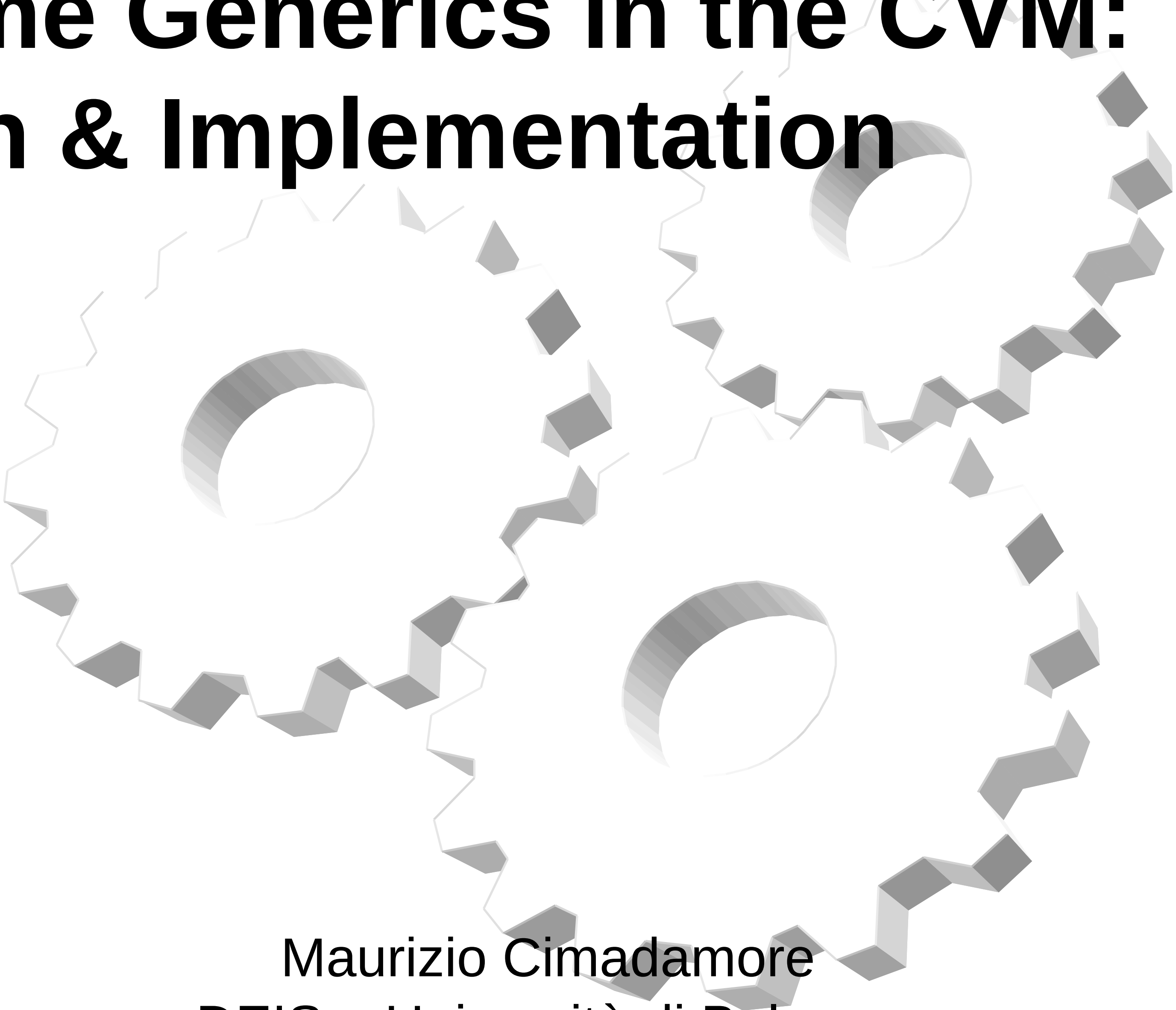
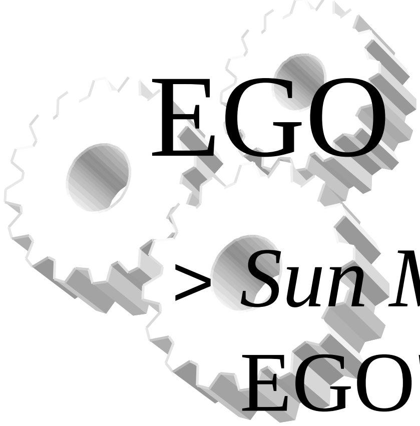
Runtime Generics in the CVM: Design & Implementation



Maurizio Cimadamore
DEIS – Università di Bologna
mcimadamore@deis.unibo.it

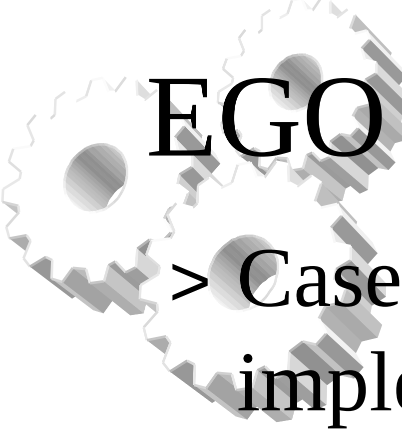
Outline

- EGO *inside* the JVM
  - The goal
  - Main architecture & reference implementation
- Runtime generics in the CVM
  - *Bytecode* generic extension
  - Runtime *type system* extension
  - *Interpreter* loop generic extension
- Conclusion
  - first impressions
  - future works



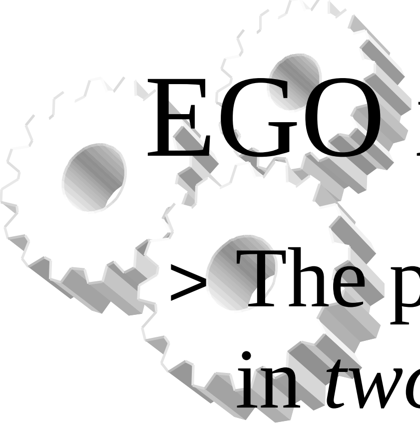
EGO inside JVM (1/2)

- *Sun Microsystems* expressed interest in having EGO's type passing approach implemented *inside* the JVM
- This approach ensures the benefits of having *full* support for generic types at runtime...
- ...*without* the runtime overhead (yet low) introduced by EGO's translational technique!
- We'll have a look at how the architecture of a JVM can be *generified* following the EGO's translation scheme.



EGO inside JVM (2/2)

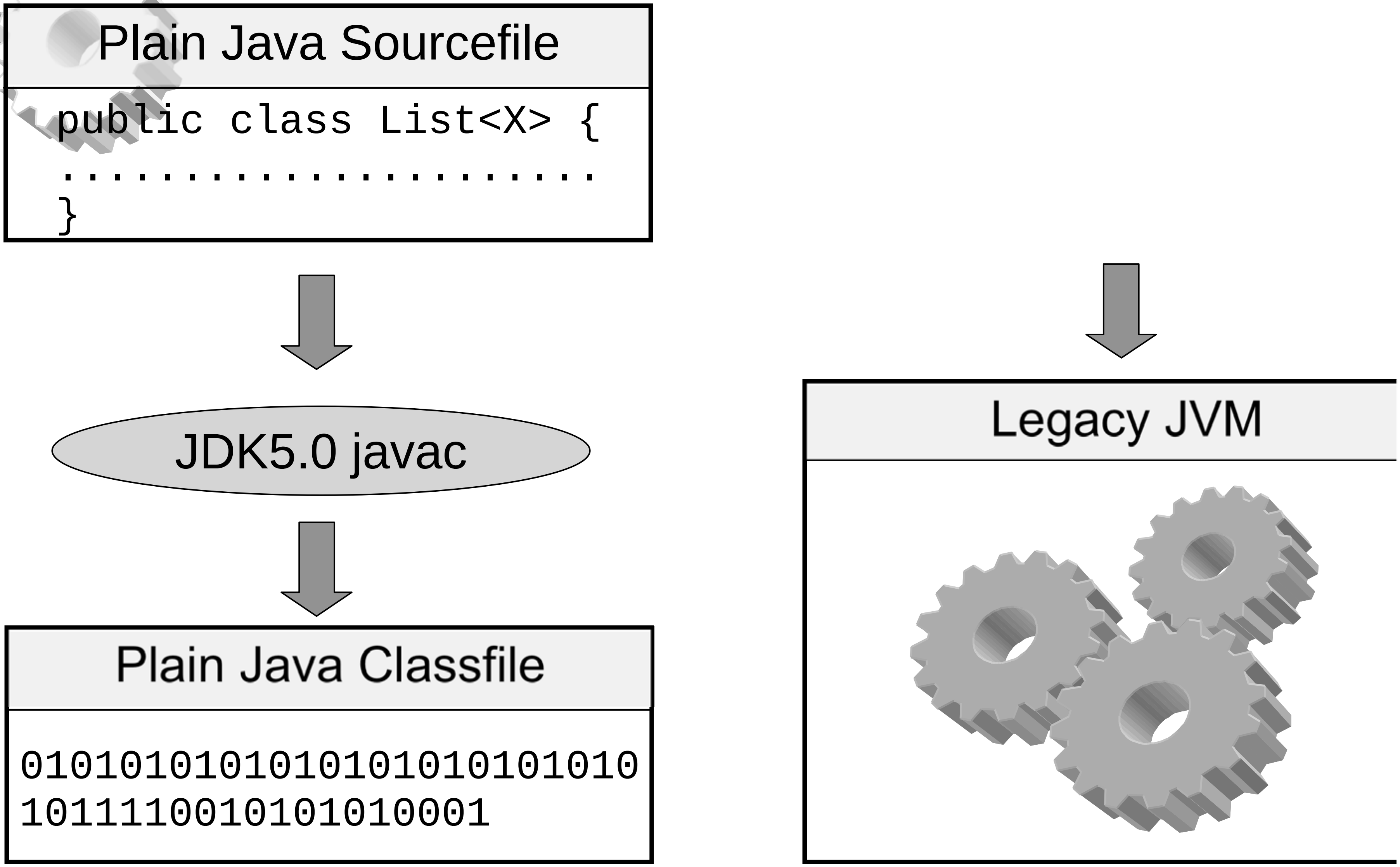
- Case study: *J2ME* platform reference implementation (*CVM*)
  - This is a *true* JVM implementation without all bells & whistles of a full fledged JVM!
- *Features* vs. *Complexity*:
  - CVM has all the *core* features of a standard JVM...
  - ...some features *missing* (es. *JIT compiler*, etc.)
- CVM is a system written in the "*old*" C language
  - About *50000* lines of code
  - *low level* of abstraction



EGO inside JVM - Architecture (1/3)

- The process of extending the CVM is structured in *two* independent steps; Let `C` be a class:
  - *Bytecode reification*: we have to store into `C`'s classfile all generic types information exploited by all `C`'s *type-dependent* operations (e.g. *casts*);
  - *CVM reification*: we have to extend the CVM's *runtime type system* to support runtime generics
- We also need to satisfy the following constraints:
  - 100% full *backward compatibility* of the new bytecode with legacy JVMs
  - low impact on *performances*

EGO inside JVM - Architecture (2/3)



Plain Java Sourcefile

```
public class List<X> {
........................
}
```

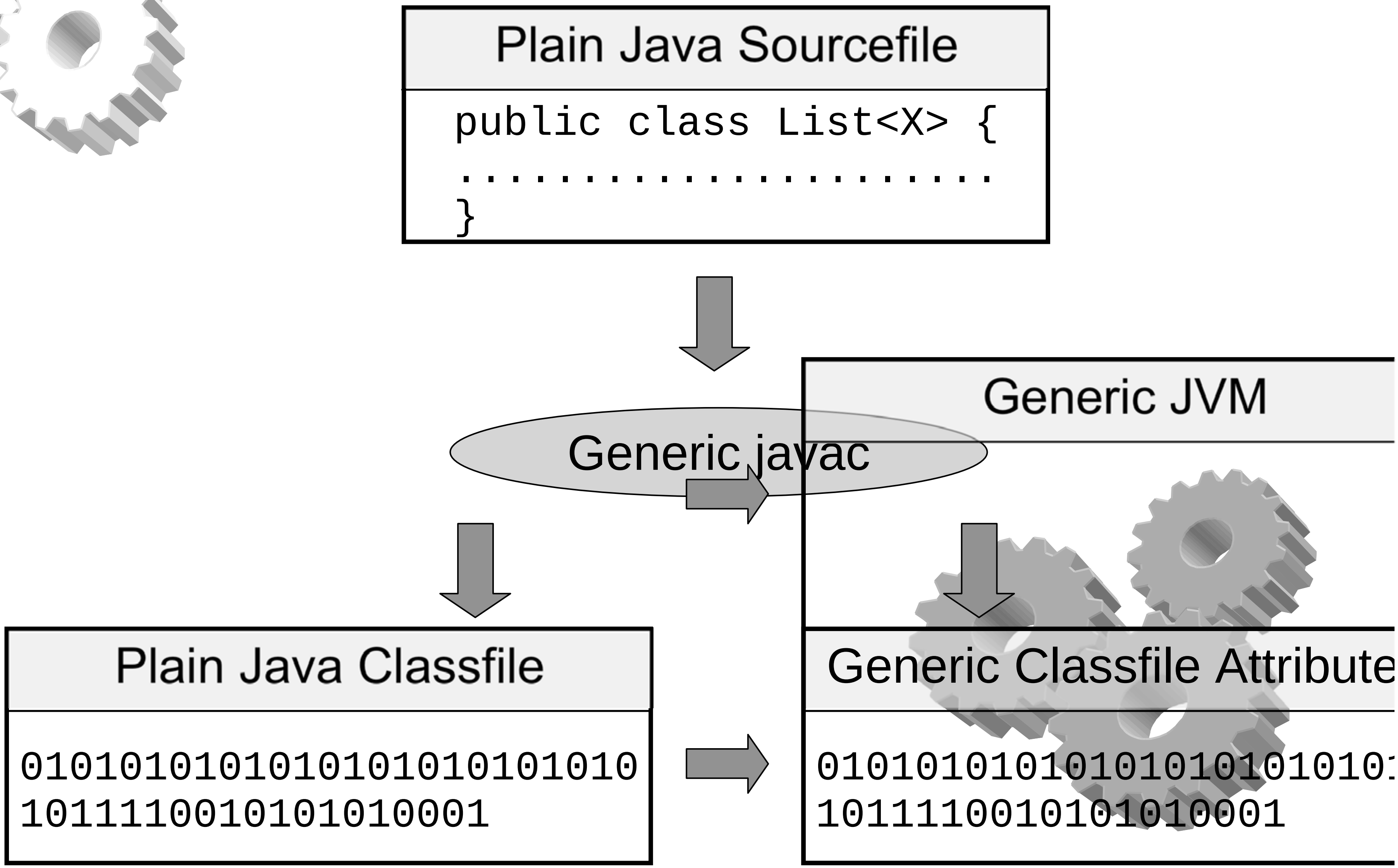
JDK5.0 javac

Plain Java Classfile

```
01010101010101010101010
1011110010101010001
```

Legacy JVM

EGO inside JVM - Architecture (3/3)



Plain Java Sourcefile

```
public class List<X> {
........................
}
```

Generic javac

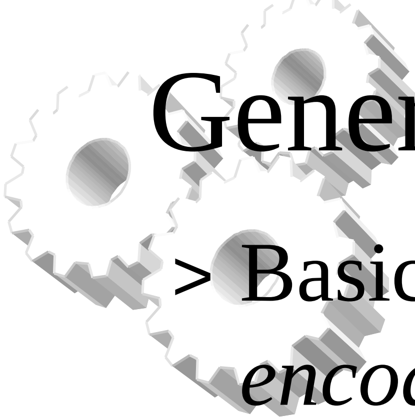
Plain Java Classfile

```
01010101010101010101010
1011110010101010001
```

Generic JVM

Generic Classfile Attribute

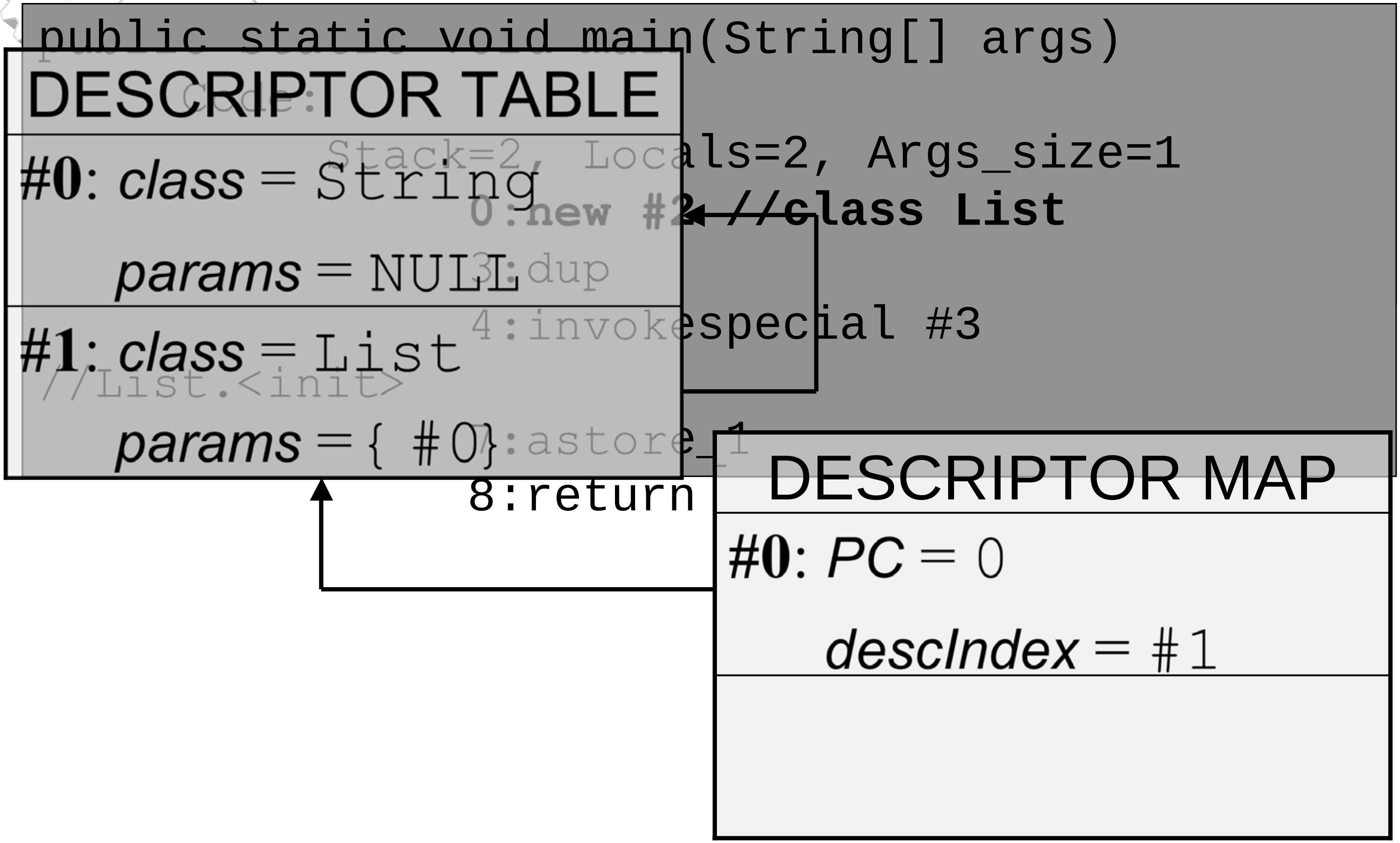
```
0101010101010101010101010
1011110010101010001
```



Generic Bytecode (1/3)

- Basic Idea: generic types’ erased signatures are *encoded* into special data structures called *type descriptors*
- The *DescriptorTable* (DT) generic *attribute* defines all the type descriptors exploited in the type dependent operations of a given class `C`;
- For a given method `m` in `C`, its *DescriptorMap* (DM) generic attribute defines the mapping beetween type dependent operations in `m` and type descriptors in the DT.

Generic Bytecode (2/3)



```
public static void main(String[] args)
  Code:
   Stack=2, Locals=2, Args_size=1
   0:new #2 //class List
   3:dup
   4:invokespecial #3
//List.<init>
   7:astore_1
   8:return
```

DESCRIPTOR TABLE

#0: $class = \texttt{String}$

$params = \mathrm{NULL}$

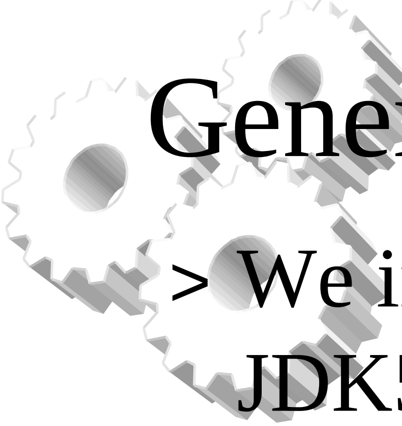
#1: $class = \texttt{List}$

$params = \{\#0\}$

DESCRIPTOR MAP

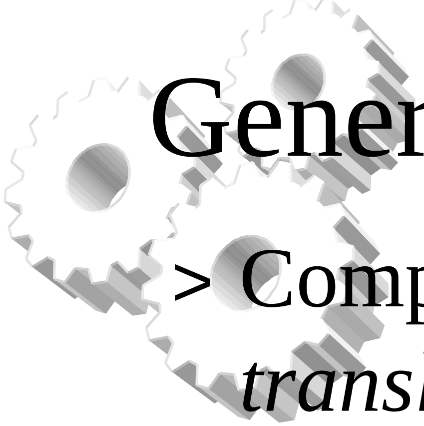
#0: $PC = 0$

$descIndex = \#1$



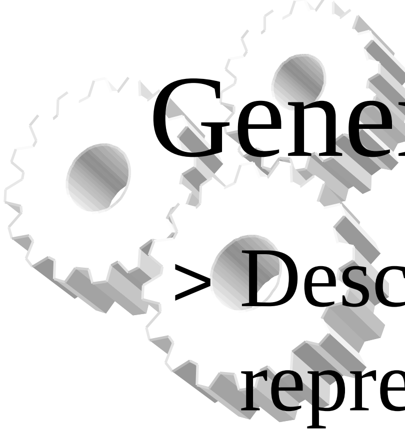
Generic Bytecode (3/3)

- We implemented an extended version of the JDK5.0 *javac* compiler which produces as output the additional generic attributes *DescriptorTable* and *DescriptorMap*:
- Some results...
  - The impact on classfile size is *not significant* (though it *increases* with the number of type dependent operations in a class' methods)
  - *100% fully backward compatible* since non-standard classfile attributes are *discarded* by legacy JVMs!



Generic CVM (1/2)

- Compile-time generated descriptors have to be *translated* into proper *runtime data structures* which can be exploited by type-dependent ops
- When executing a method `m` of a given class `C`, the interpreter has to build proper *runtime type descriptors* by looking into:
  - The `m`'s *DescriptorMap* attribute (if `i` is the *PC* of a type-dependent instruction then `m`'s DM has an entry *{*`i`, `d`*}* where `d` points to a valid DT slot.
  - The *C*'s *DescriptorTable*; its `d`-th slot stores the type descriptor to be exploited when executing `i`;



Generic CVM (2/2)

- Descriptors are stored *directly* into the runtime representation of a Java object (whose layout is sligthly changed)
    - This happens each time a generic “new” is executed (remember, *each* generic opcode refers a type descriptor in the current DT)
- This way the interpreter can access *exact* runtime type information on a given instance `obj` when executing type-dependent opcodes such as:
    - cast (`checkcast` opcode)
    - instanceof (`instanceof` opcode)

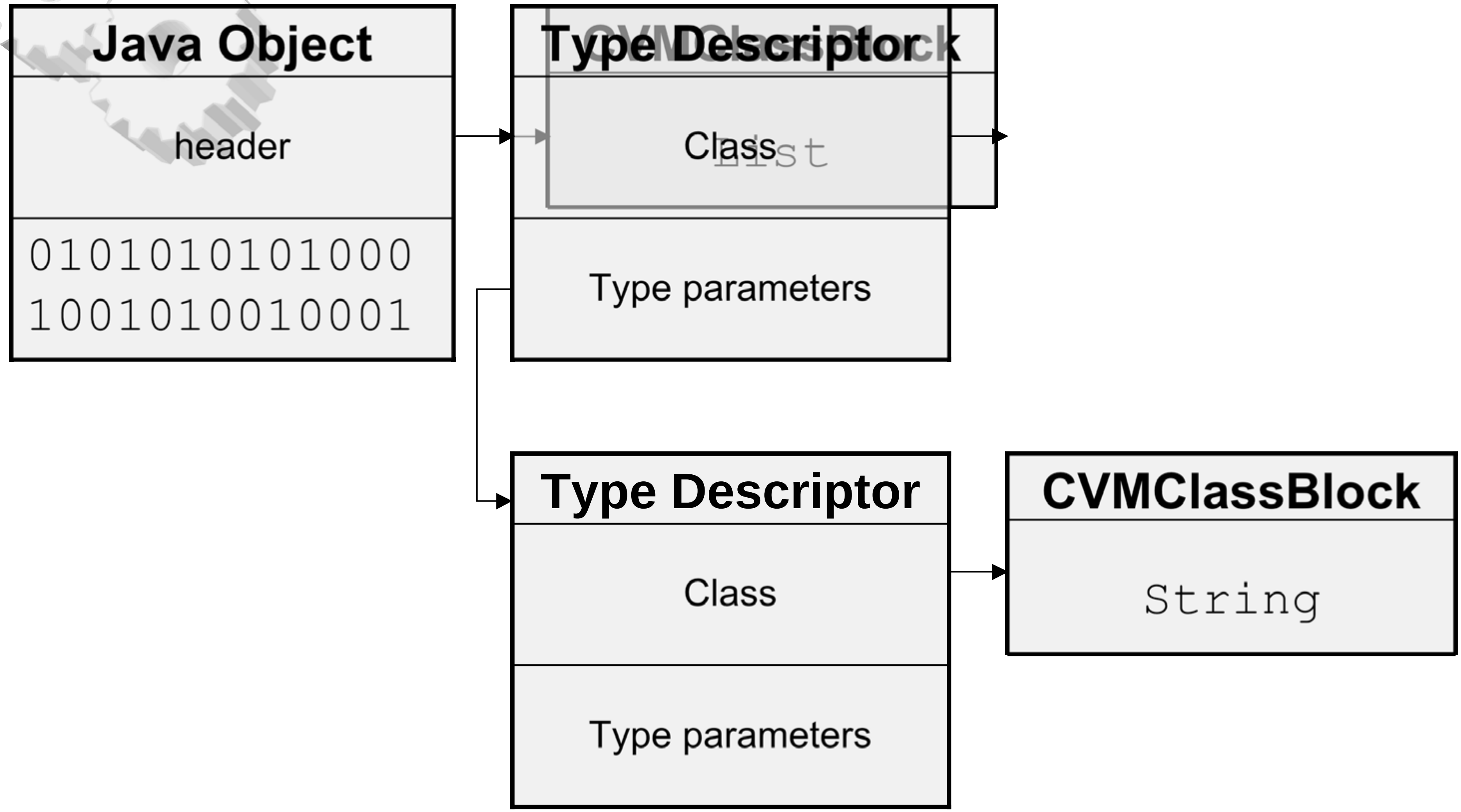
Generic CVM – Object layout (1/4)

- A Java Object is basically a bunch of fields (which can be 16, 32 or 64-bit values) along with a 64-bit *header*:
  - The object's header *determines its runtime type* (it stores a reference to the object's `CVMClassBlock` structure)
- Our aim is to *link* generic instances with exact runtime type information of type descriptors
  - We should *replace* the `CVMClassBlock` reference in the object's header with something else…

Generic CVM – Object layout (2/4)

- Given an instance `obj`, we decided to *change* the layout of its header as described below:
  - If `obj` is a generic instance, then `obj`’s header will point to a class descriptor carrying exact `obj`’s runtime type information;
  - If `obj` is a legacy instance (non generic) then o’s header will still be pointing to its `CVMClassBlock`
- This way we *minimize* the impact of the generic extension on the existing code!

Generic CVM – Object layout (3/4)



Java Object

header

0101010101000
1001010010001

Type Descriptor

Class

Type parameters

CVMClassBlock

List

Type Descriptor

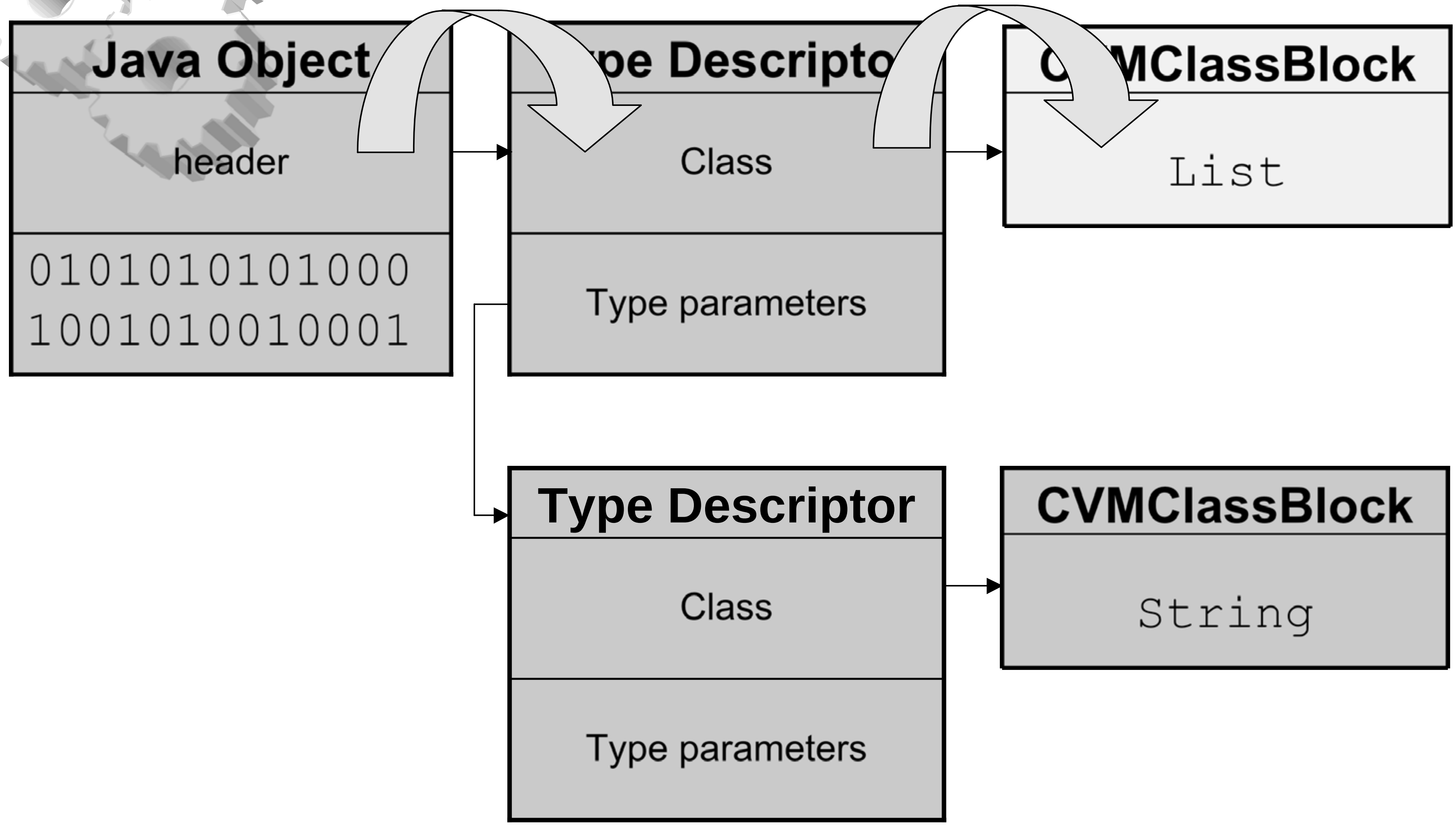
Class

Type parameters

CVMClassBlock

String

Generic CVM – Object layout (4/4)

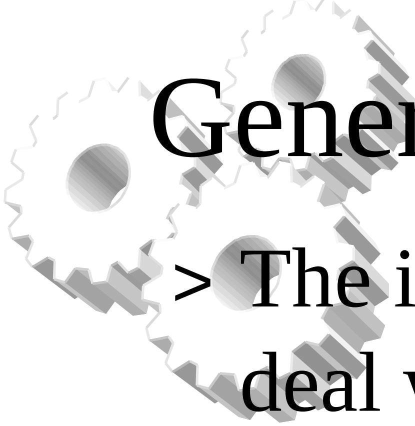


Generic CVM – Interpreter loop (1/4)

- The Java interpreter is basically a *loop* which executes each opcode of a given method `m`;
- When an opcode has to be executed:
  - first we have a *quickening* process that consist in *symbolic name resolution* (this phase could trigger the *loading* of other classes)
  - Once an opcode is quickened, *it's ready to be executed by the interpreter* (since we are sure that every symbolic reference has already been *resolved*)

Generic CVM – Interpreter loop (2/4)

- Assume we are quickening a `new` opcode
  - The `new` opcode specifies a Constant Pool (CP) entry as its unique operand (`new CP_IDX`)
- If `CP_IDX` refers to a not-yet resolved CP entry
  - The entry `CP_IDX` of the current CP is resolved (and the corresponding class loaded if necessary)
  - The opcode is changed to `new_quick CP_IDX`
- This ensures that the resolution process happens *only once* for each opcode of a given method `m`.
  - What if `CP_IDX` refers to a generic type?

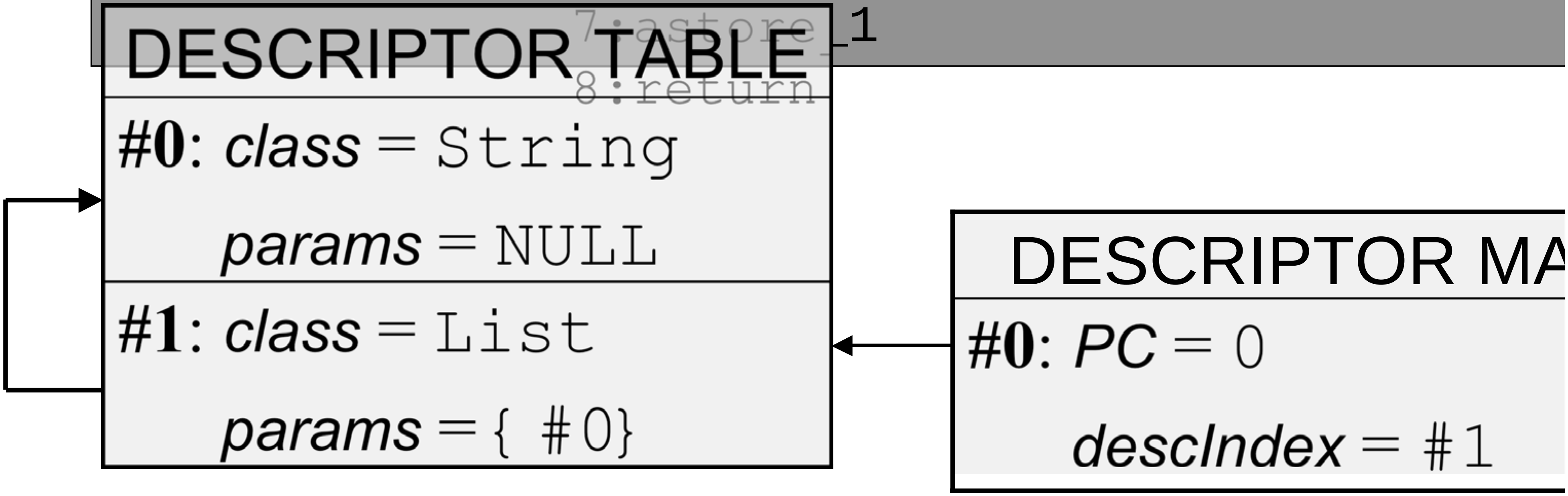


Generic CVM – Interpreter loop (3/4)

- The interpreter loop of the generic CVM has to deal with *generic instance creation* as well...
  - Let's look at our opcode `new CP_IDX`
- If the current method’s *DescriptorMap* attribute contains an entry for the above opcode (let `desc_idx` be the value of that entry):
  - The `desc_idx`-th descriptor in the current *DescriptorTable* is resolved;
  - The above opcode now is changed as follows: `new_generic desc_idx`

Generic CVM – Interpreter loop (4/4)

```
public static void main(String[] args)
     Code:
          Stack=2, Locals=2, Args_size=1
               0:new_generic #1 //class List
               3:dup
               4:invokespecial #3
//List.<init>
               7:astore_1
               8:return
```



DESCRIPTOR TABLE

#0: *class* = String
params = NULL

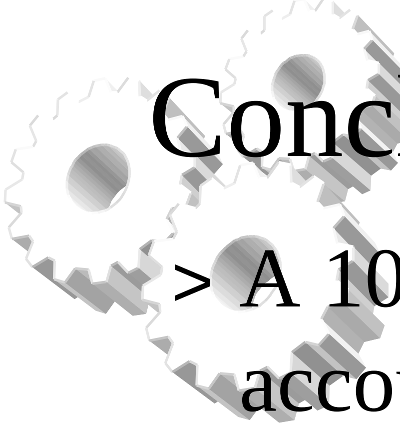
#1: *class* = List
params = { #0}

DESCRIPTOR MA

#0: *PC* = 0
descIndex = #1

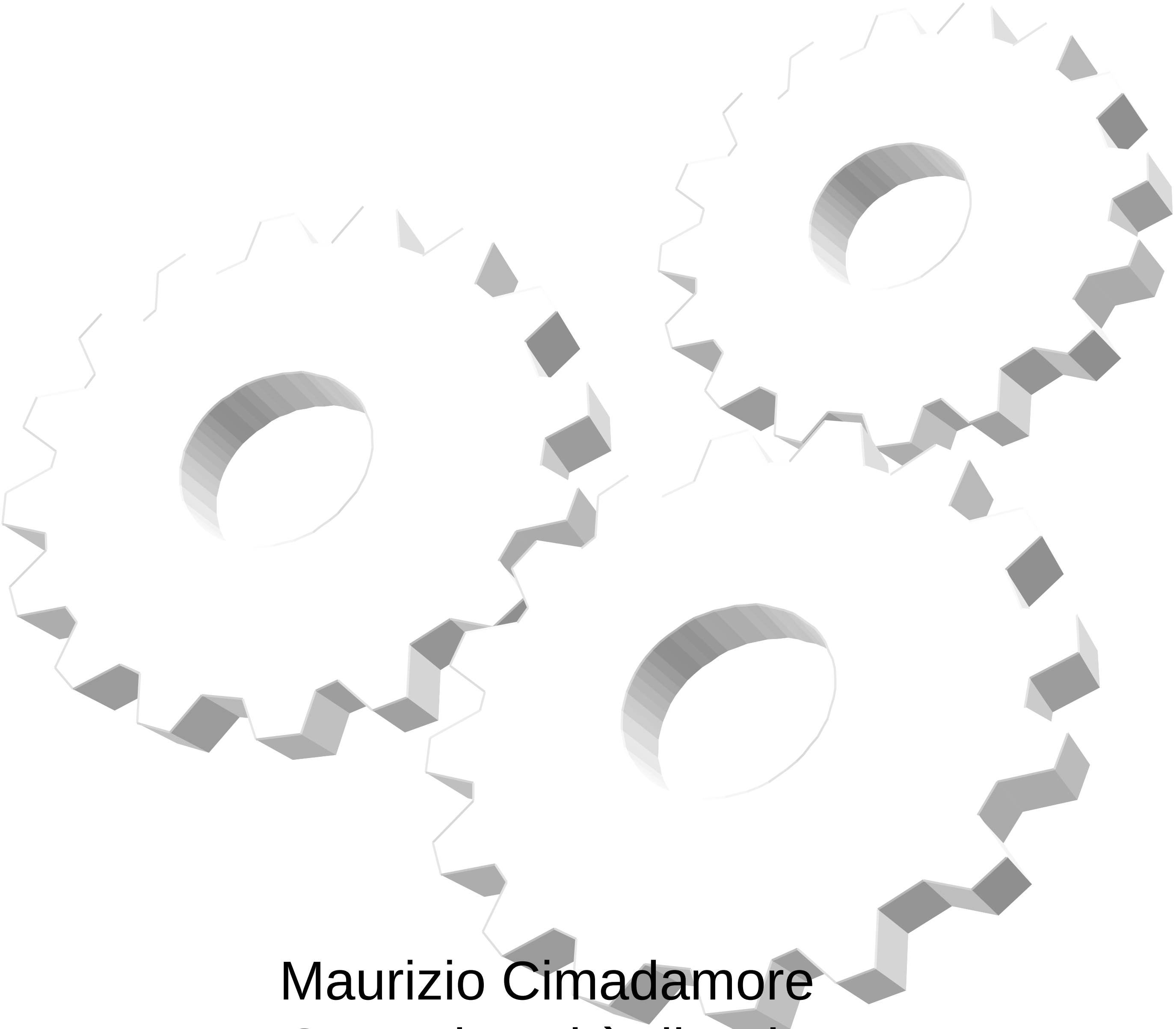
Conclusions (1/2)

- Currently, the following features have been implemented:
    - Generic classes/arrays creation
    - Generic methods calls
    - Execution of type dependent operation envolving generic types (such as *cast*, *instanceof*)
- Some *micro-benchmarks* have shown that generic CVM is almost *as fast* as its non-generic version
- We are planning *system-wide benchmarks* in order to evaluate the cost of generic types support in *real world* case studies



Conclusions (2/2)

- A 100% full generic JVM should take into account aspects like:
  - Generic bytecode *verification*
  - Generic types integration into the *Java Reflection API*
  - *Serialization* of generic objects
  - *JIT*
  - ...
- As you can see, there is still a *lot of work* to be done...
  - **This could be the starting point of your thesis**!



Maurizio Cimadamore
DEIS – Università di Bologna
mcimadamore@deis.unibo.it