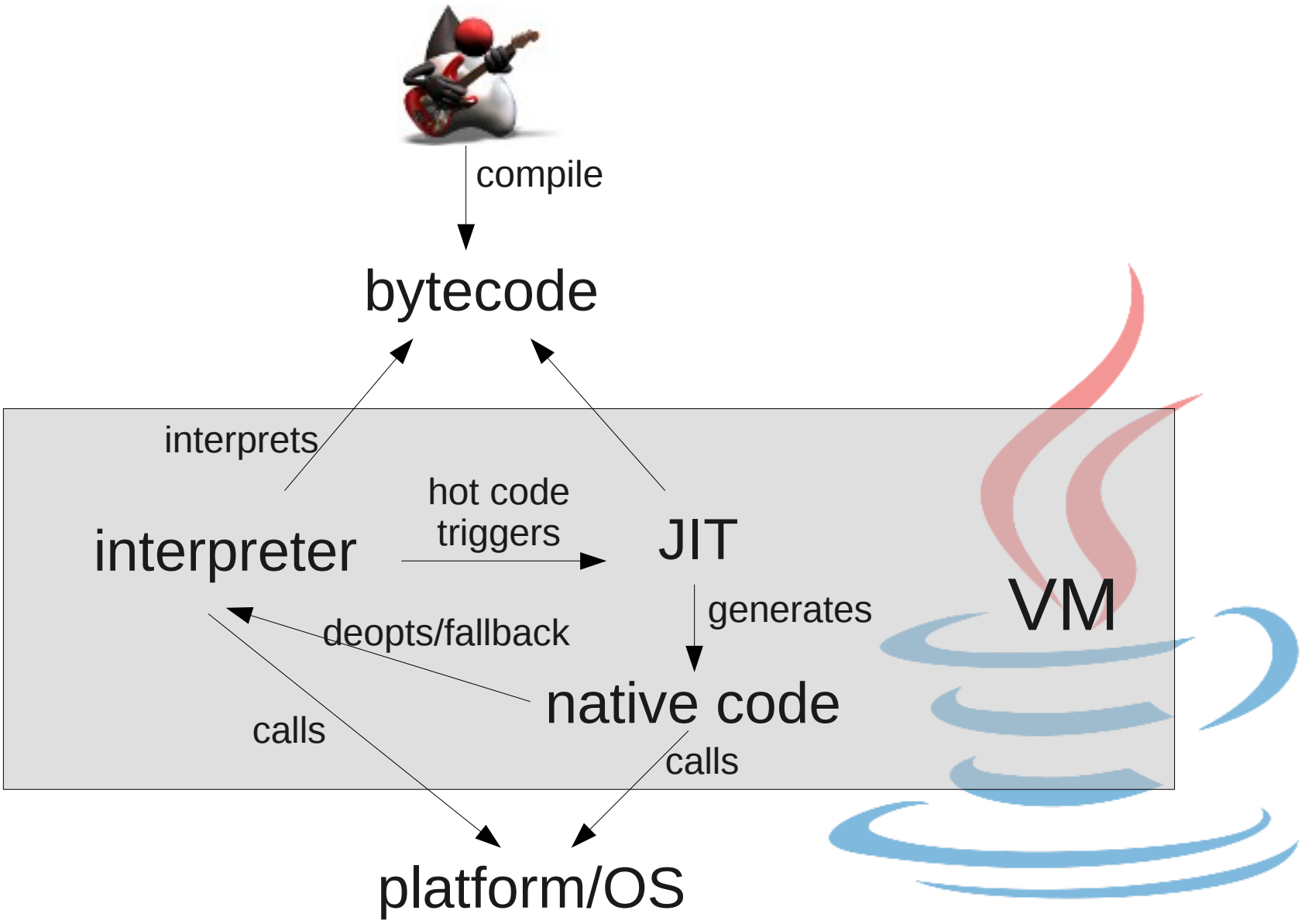
JSR 292 / PHP.reboot

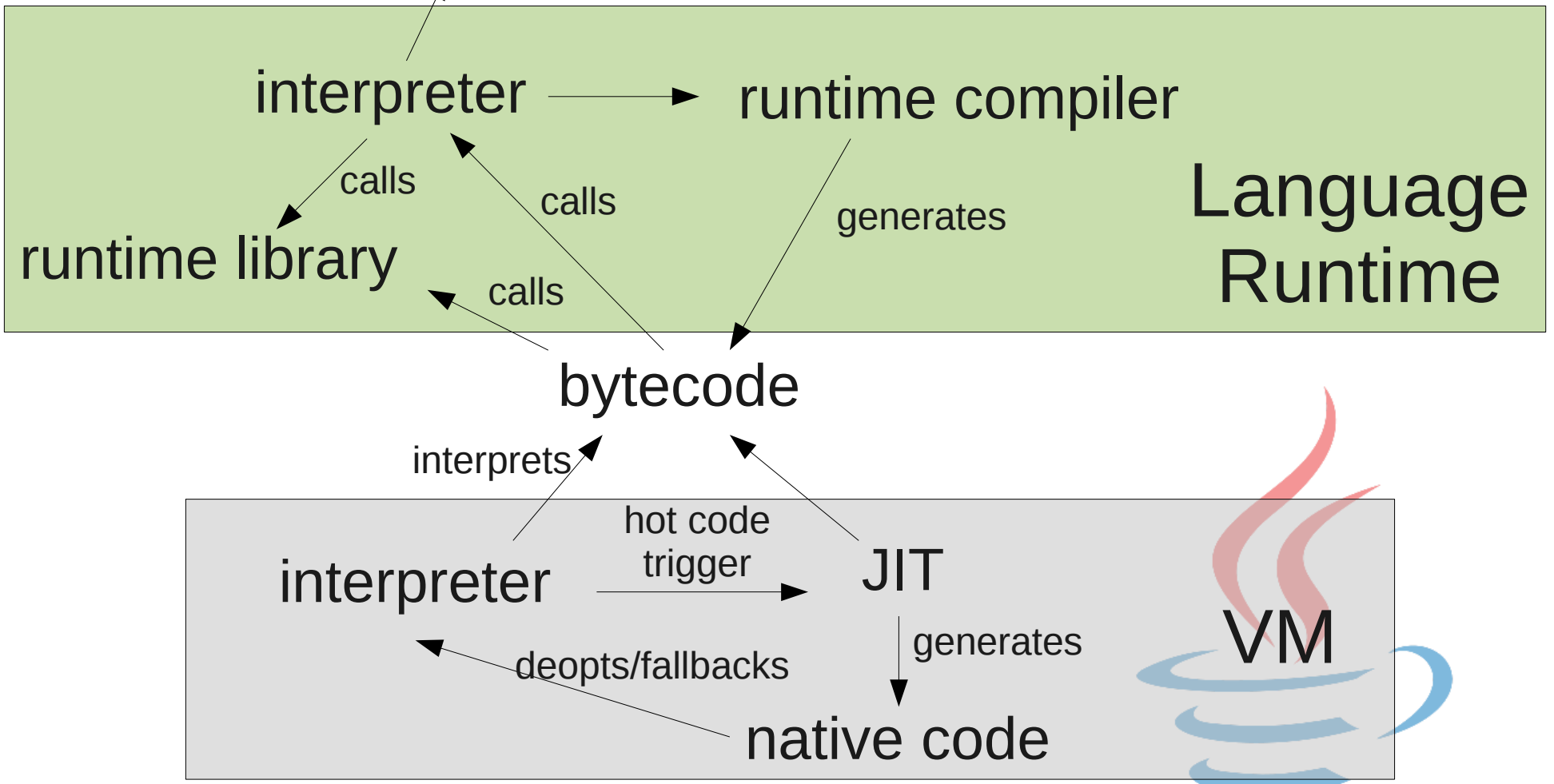
Rémi Forax

Université Paris-Est Marne-la-Vallée

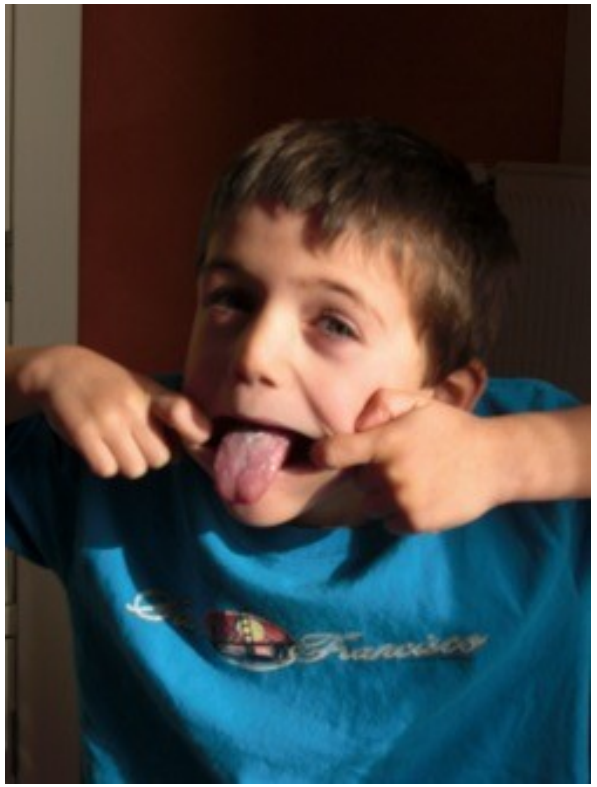
Classic JVM


compile
bytecode
interprets
hot code
triggers
interpreter
JIT
generates
VM
deopts/fallback
native code
calls
calls
platform/OS

Dynamic Language Runtime



Pain features of dynamic languages



- no declared type
  - operators are overloaded
  - overflow
- add/remove fields
- add/remove/replace methods
  - *monkey patching*

real dynamic code is uncommon

>> type inference & typechecking

monkey patching & overflow are rare

>> bailout guard + deoptimization

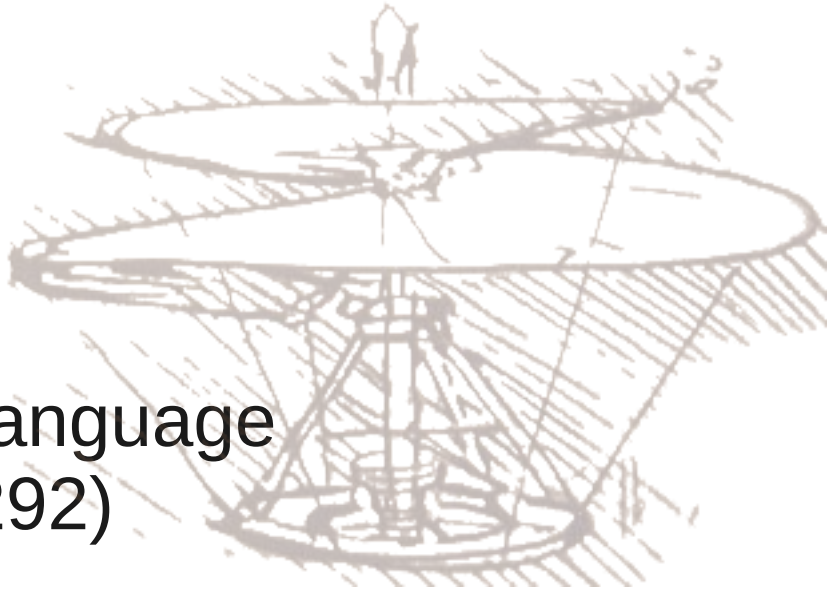
operator overload & duck typing

>> inlining cache/tree

Hotspot JVM >> Multi-Language VM

OpenJDK - Da Vinci Project

setup in 2007



First step

Support dynamically typed language
on the Java Platform (JSR 292)

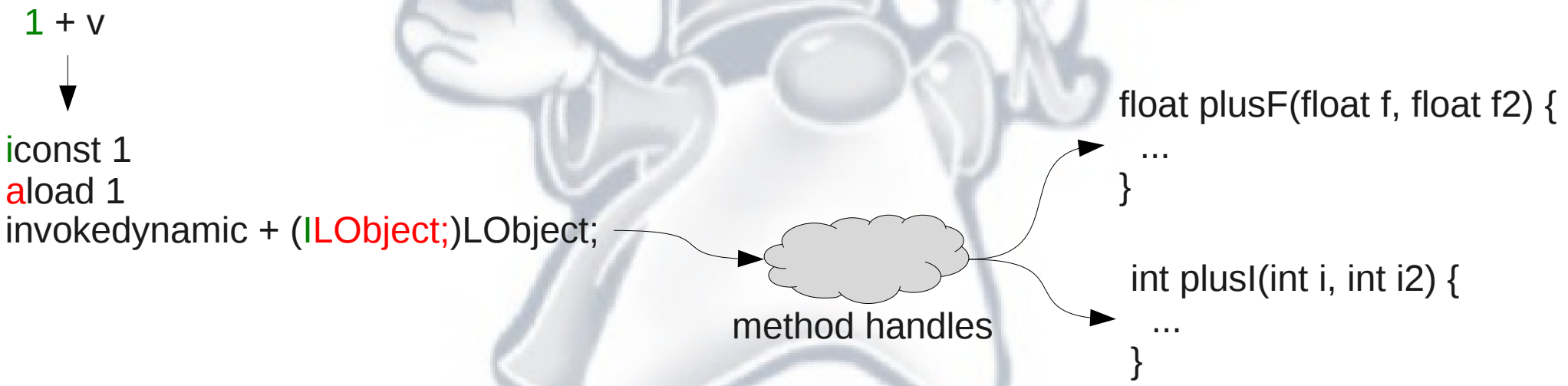
Next steps

Tailcalls, coroutine, continuation, stack inspection,
interface extensions, immediate wrapper type, etc.

Overview of JSR 292

Uncouple call site and callee

- invokedynamic
- method handle



+ API for creating/managing method handles

Method Handle

Single type: java.dyn.MethodHandle

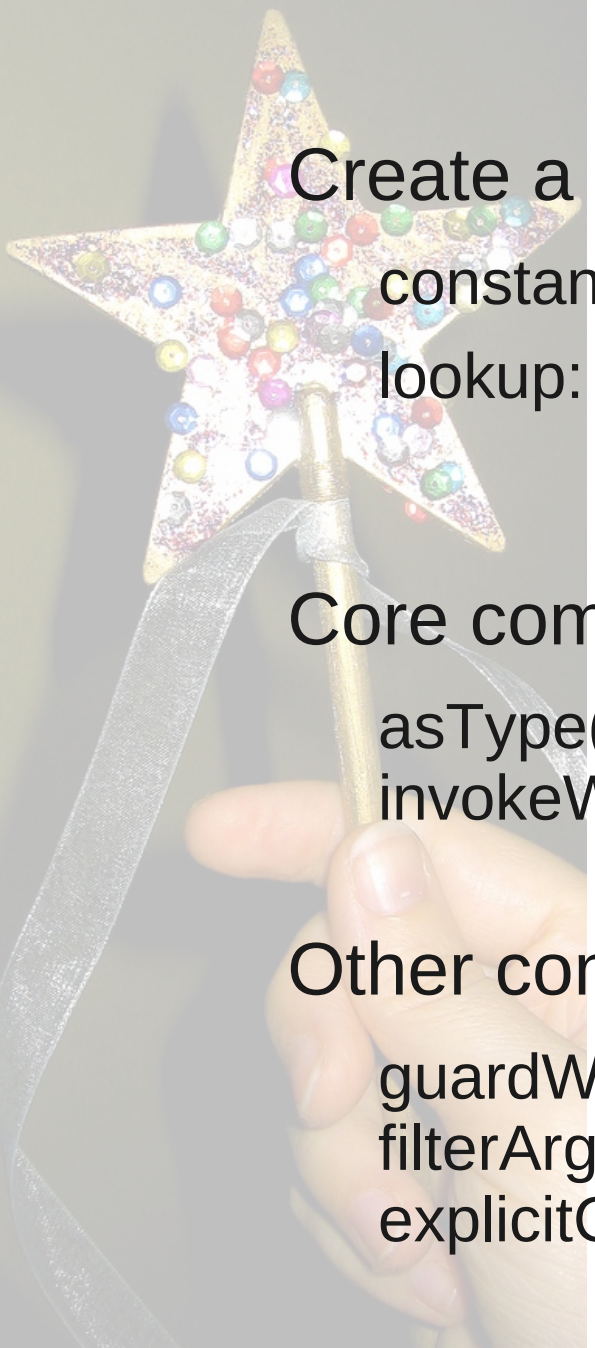
Type safe function pointer

embodies its type (MethodType)

Callable with a polymorphic signature method

(int)mh.invokeExact(2, 3.0)

Managing method handles



Create a Method Handle

constant: String#charAt(int)

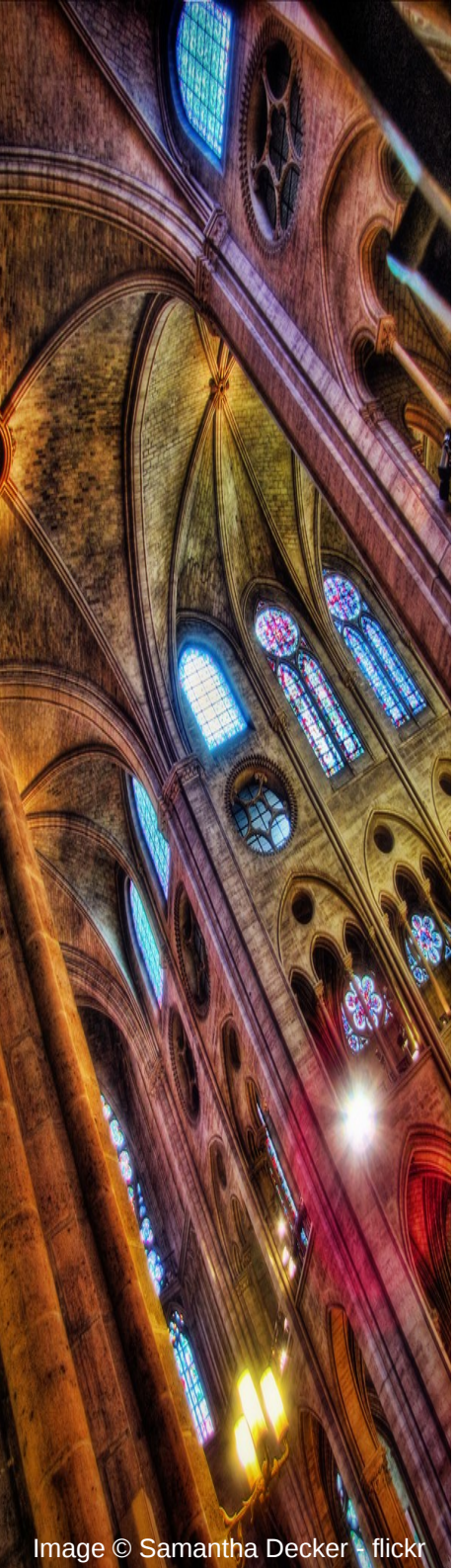
lookup: MHs.lookup().findVirtual(String.class,
"charAt", methodType(char.class, int.class))

Core combinators

asType(), bindTo(), asCollector()/asSpreader(), invokeWithArguments()

Other combinators

guardWithTest(), dropArguments/insertArguments(), filterArguments()/filterReturnValue(), explicitCastArguments(), etc.



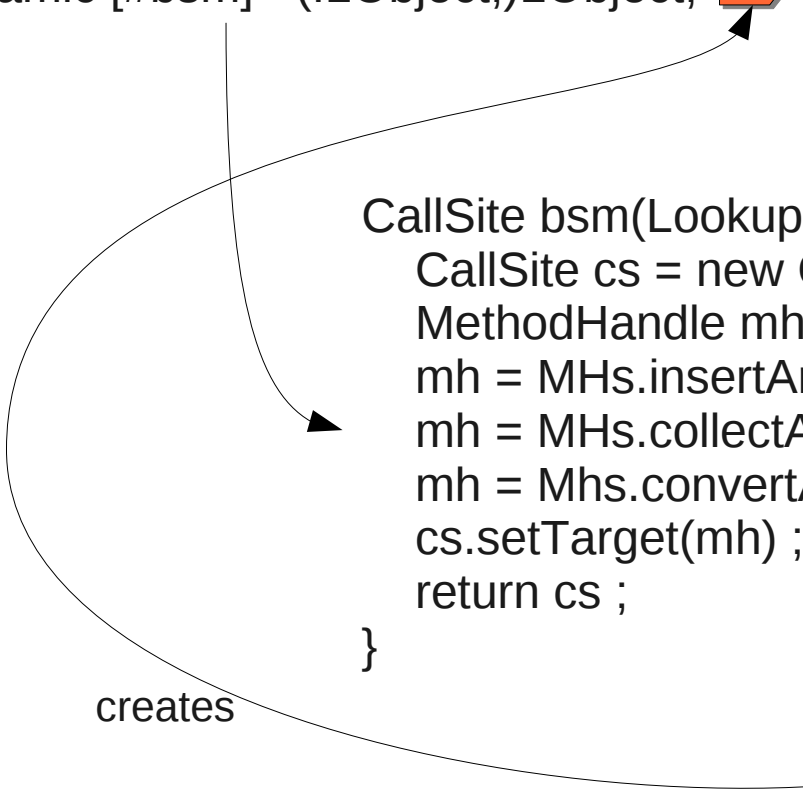
Invokedynamic

- First call
  - a bootstrap method does the linkage
    - Each invokedynamic has its own BSM
    - Creates a reified CallSite
      - CallSite contains the target method handle
- Next calls
  - Use the target method handle
- CallSite.setTarget()
  - Ask a relinking

How to implement an inlining cache ?

The BSM installs a fallback method

```
iconst 1
aload 1
invokedynamic [#bsm] +(ILObject;)LObject;
```



```
CallSite bsm(Lookup lookup, String name, MethodType type) {
    CallSite cs = new CallSite(type);
    MethodHandle mh = #fallback(CallSite, Object[]);
    mh = MHs.insertArguments(mh, 0, cs);
    mh = MHs.collectArguments(mh, type.pCount(), type.generic());
    mh = Mhs.convertArguments(mh, type);
    cs.setTarget(mh) ;
    return cs ;
}
```

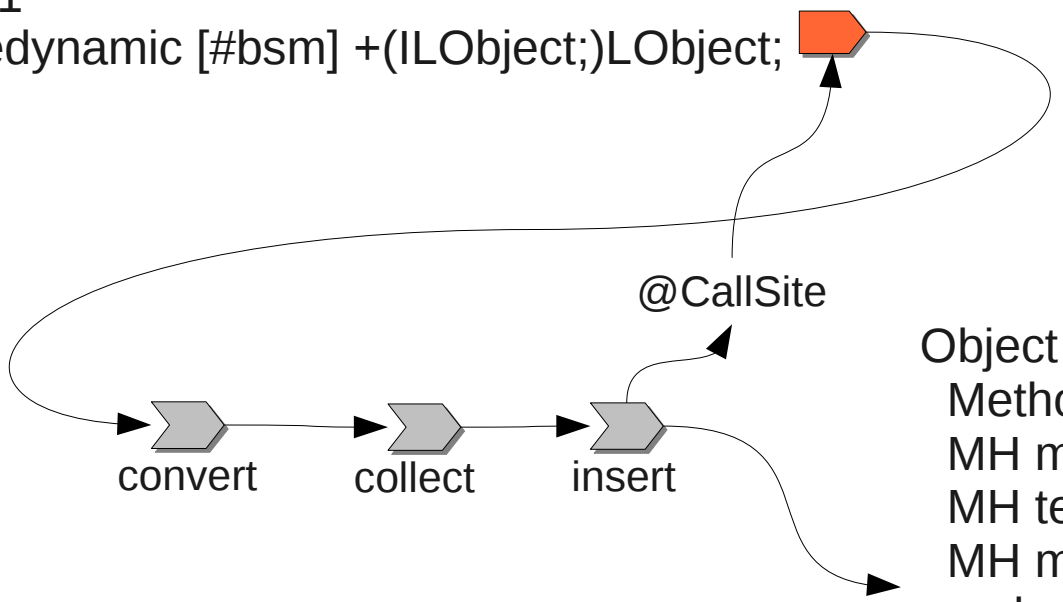
creates

```
Object fallback(CallSite cs, Object[] args) { ... }
```

How to implement an inlining cache ?

The fallback prepends a guard

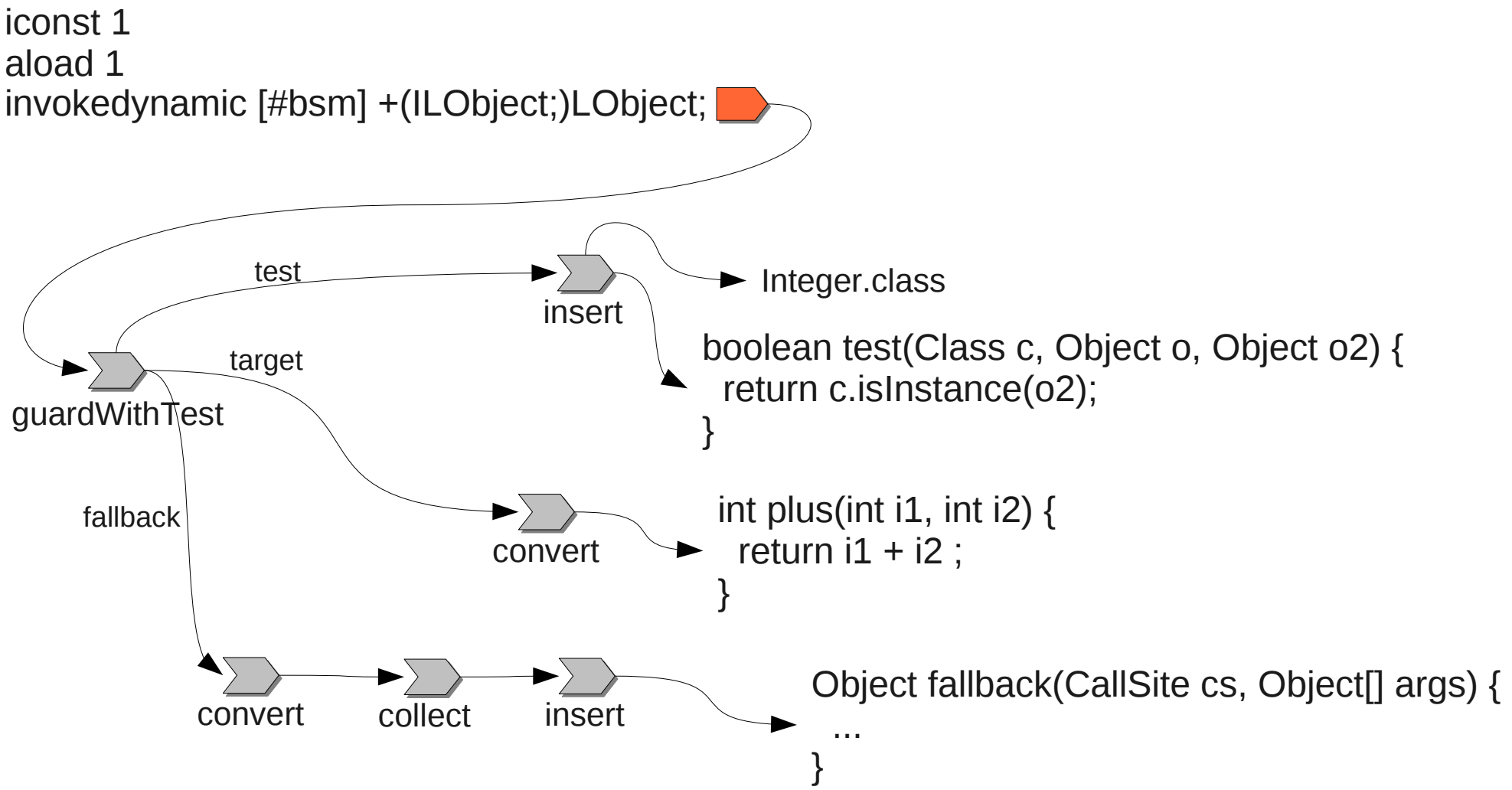
```
iconst 1
aload 1
invokedynamic [#bsm] +(ILObject;)LObject;
```




```
Object fallback(CallSite cs, Object[] args) {
  MethodType type = type(cs, args);
  MH mh = findMH(type, cs.type());
  MH test = findTest(type, mh.type());
  MH mh = MHs.guardWithTest(test,
    mh,
    cs.getTarget());
  cs.setTarget(mh);
  return mh.invokeWithArguments(args);
}
```

How to implement an inlining cache ?

The tree is stable until a new class is discovered



A great book to stick your nose in!

Inlining Cache & Code Patching

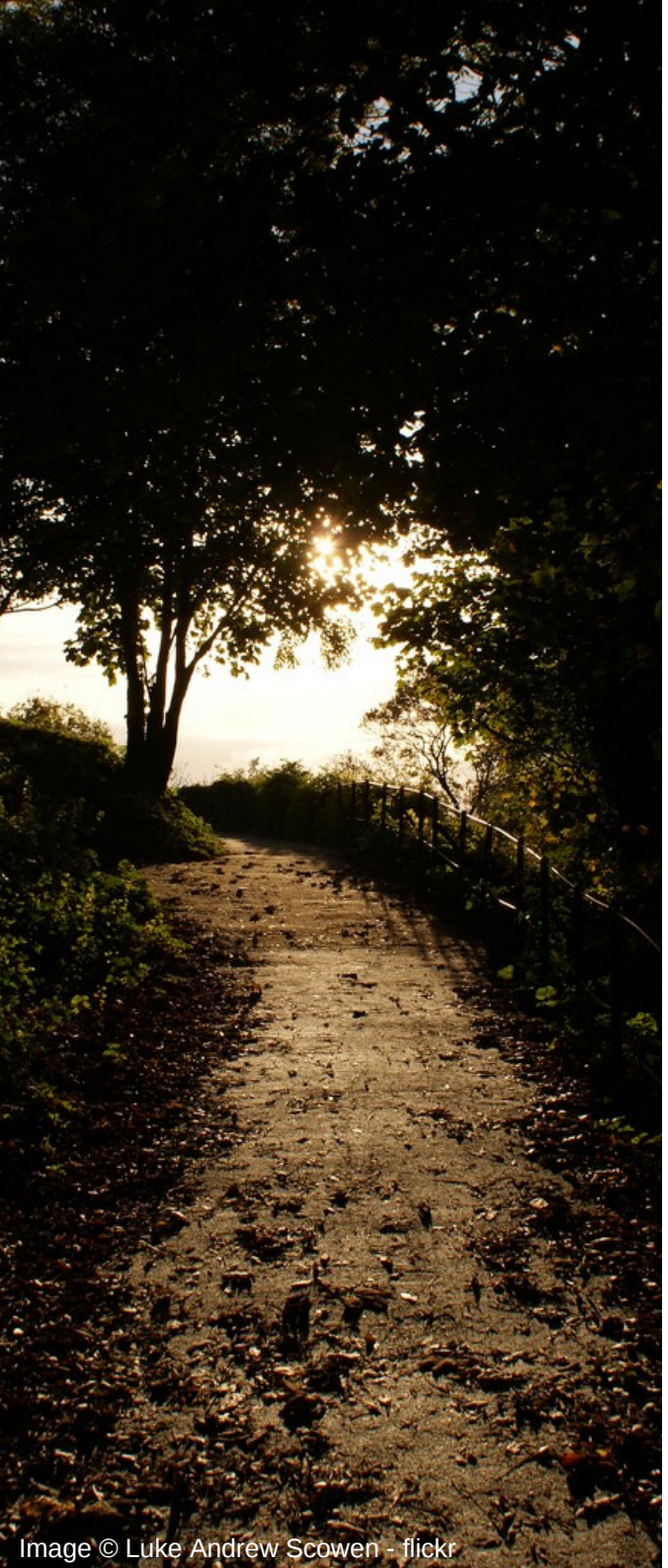
FOR

DUMMIES®

become a friend of the VM

A Reference for the Rest of Us!®

By JSR 292



JSR 292

More than just invokedynamic/MH

- ClassValue, Switcher(invalidation)

Main drawback

- Path from pre-jsr292 to post-jsr292 requires a full rewrite of the runtime

PHP.reboot

In few words ...

Secure!

- No eval, no magic quote

DSLs for XML, JSON, SQL, Xpath, Xquery, etc.

- Sanitize depending on the context

Dynamically typed with gradual type system

Interpreter, ahead of time & runtime compilers

Compatible with 1.6/1.7 VMs

Toolchain

Tatoo – parser generator (LALR)

- Generate ASTs from grammars, change at runtime
- Keyword are *local* by default
- Evaluate AST asap (fast reduce)

JSR 292 + lightweight loading

- Share runtime logic between interpreter and bytecode
- Lambda/closure for free

ASM – fast bytecode generation

- Partial supports of JDK7 classfile format

PHP.reboot runtime

Image © Joe Penniston - flickr

- First interpret then compile
  - update already existing call sites
- Compile hot method and loops body
  - interpreter profiles interpreted code
- In loops:
  - speculatively use profiled runtime types
    - prove with a simple forward typechecker
  - specialize functions
  - generate same code as javac
    - Revert and use invokedynamic if variable is really dynamic
  - record untaken paths to generate escape exit
  - profile field use/alloc-site to generate adhoc object

A simple dynamic example

```
for(i = 0; i < 350; i = i + 1) {
  if (i == 100) {
      i = 102.0
  }
}
```

Type inference without escape

```
trace(LEvalEnv;Ljava/lang/Object;LVar;)Z
l0: aload 1
    sipush 350
    invokedynamic [#op] > (LObject;I)Z
    ifne l1
    aload 1
    bipush 100
    invokestatic Integer.valueOf(I)LInteger;   // boxing
    invokestatic RT.ne(LObject;LObject;)Z
    ifne l2
    ldc 102.0
    invokestatic Double.valueOf(D)LDouble;  // boxing
    astore 1
l2: aload 1
    iconst_1
    invokedynamic [#op] + (LObject;I)LObject;
    astore 1
    goto l0
l1: aload 2       // post instruction, update interpreter variables
    aload 1
    invokevirtual Var.setValue (LObject;)V
    ...
```

Optimization with escape trace

```
trace(LEvalEnv;ILMethodHandle;LVar;)Z
l0: iload 1
    sipush 350
    if_icmpgt l1          // no loop peeling !
    iload 1
    sipush 100
    if_icmpne l2
    aload 2
    aload 0
    iload 1
    invokevirtual invoke(LEvalEnv;I)V
    iconst_0
    ireturn               // return false, trace escape !
l2: iload 1
    iconst_1
    iadd
    istore 1
    goto l0
l1: ...                   // post instruction, update interpreter variables
```

Bailout to interpreter

Need to restore:

- Interpreter's stack of values
  - Typechecker prepares a blank stack + association between parameters position and stack position
- ~~Interpreter's call stack (walk on the AST)~~

Solution >>
Adhoc evaluator that completes the iteration

Loop may be re-optimized later

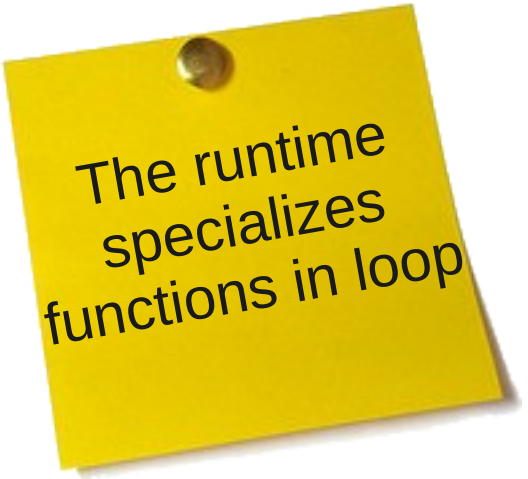
Optimization after escape trace

```
trace(LEvalEnv;DLVar;)V
l0: dload 1
    sipush 350
    i2d                 // no constant propagation !
    dcmpg
    ifge l1
    dload 1
    bipush 100
    i2d
    dcmpg
    ifne l2
    ldc 102.0
    dstore 1
l2: dload 1
    iconst_1
    i2d
    dadd
    dstore 1
    goto l0
l1: ...                 // post instruction, update interpreter variables
```

And with function calls ?

```
function fibo(a) {
  if (a < 2)
    return 1
  return fibo(a - 1) + fibo(a - 2)
}

i = 0
while (i < 200) {
  fibo(i % 7)
  i = i + 1
}
```



Fibo is called often

```
fibo(LEvalEnv;LObject;)LObject;
    aload 1
    iconst_2
    invokedynamic [#op] > (LObject;I)Z
    ifne l0
    iconst_1
    invokestatic Integer.valueOf(I)LInteger;      // boxing
    areturn
l0: aload 0
    aload 1
    iconst_1
    invokedynamic [#op] - (LObject;I)LObject;
    invokedynamic [#call] fibo(LEvalEnv;LObject;)LObject;
    aload 0
    aload 1
    iconst_2
    invokedynamic [#op] - (LObject;I)LObject;
    invokedynamic [#call] fibo(LEvalEnv;LObject;)LObject;
    invokedynamic [#op] + (LObject;LObject;)LObject;
    areturn
```

'while' is traced/compiled

```
trace(LEvalEnv;ILVar;)Z
l0: iload 1
    sipush 200
    if_icmpge l1
    aload 0
    aload 1
    bipush 7
    irem
    invokedynamic [#call] fibo(LEvalEnv;I)LObject;   // fibo is specialized !
    pop
    iload 1
    iconst_1
    iadd
    istore 1
    goto l0
l1: aload 2                  // post instruction, update interpreter variables
    aload 1
    invokestatic Integer.valueOf(I)LInteger;    // boxing
    invokevirtual Var.setValue(LObject;)V
    iconst_1
    ireturn
```

Specialized Fibo

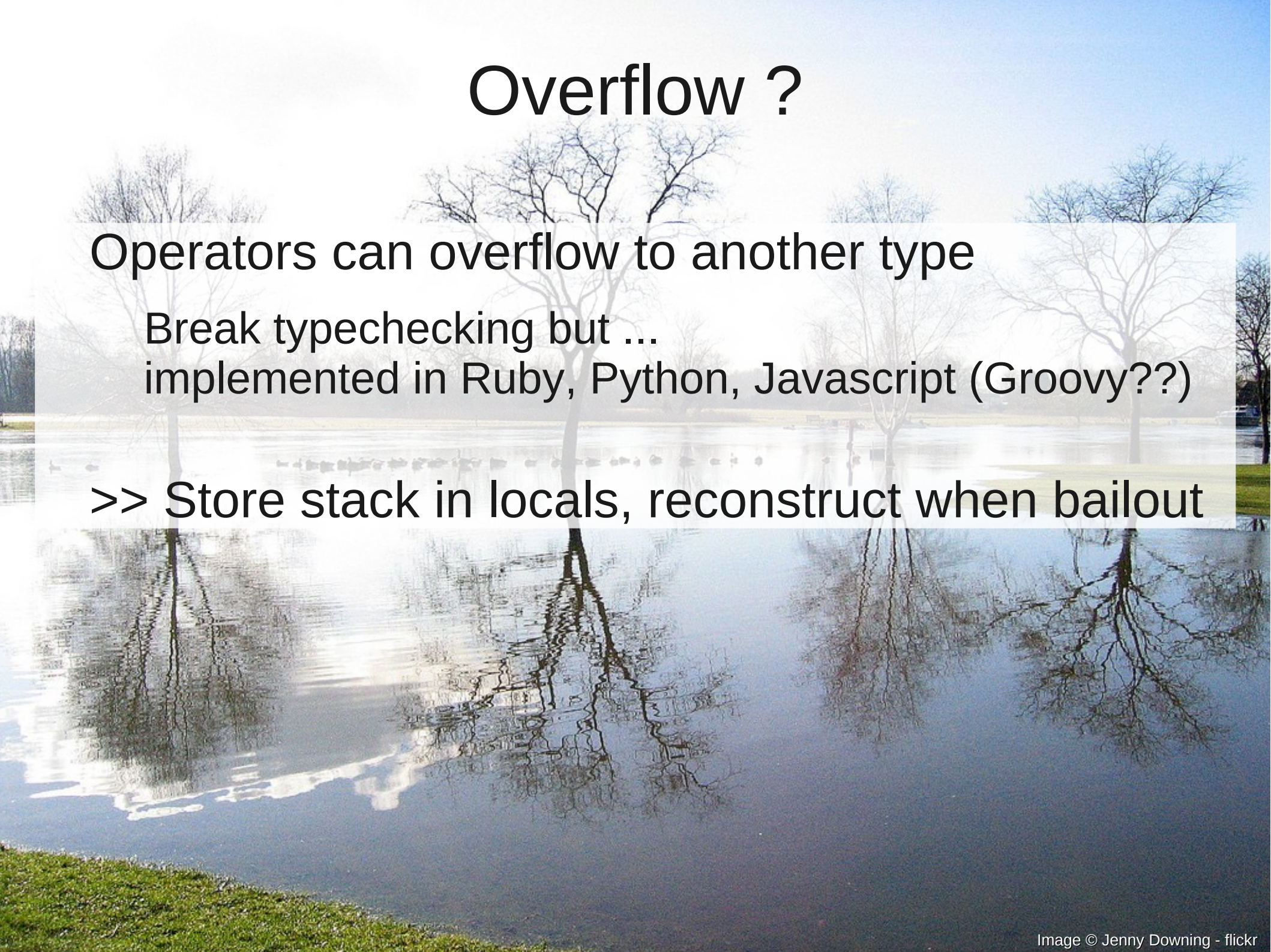
```
fibo(LEvalEnv;I)LObject;
    iload 1
    iconst_2
    if_icmpge l0
    iconst_1
    invokestatic Integer.valueOf(I)LInteger;      // boxing
    areturn
l0: aload 0
    iload 1
    iconst_1
    isub
    invokedynamic [#call] fibo(LEvalEnv;I)LObject;
    aload 0
    iload 1
    iconst_2
    isub
    invokedynamic [#call] fibo(LEvalEnv;I)LObject;
    invokedynamic [#op] + (LObject;LObject;)LObject;
    areturn
```

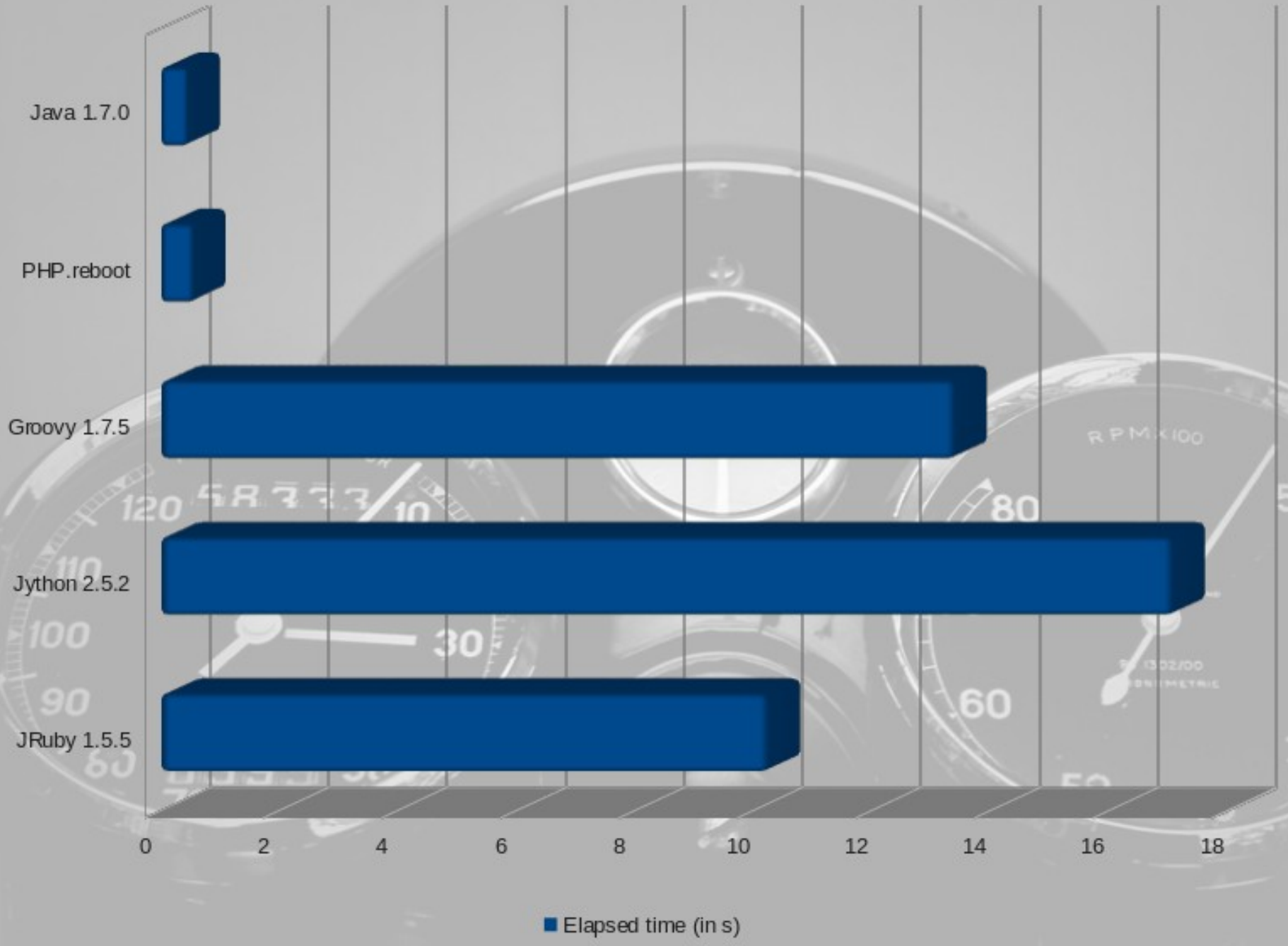
Overflow ?

Operators can overflow to another type

Break typechecking but ...
implemented in Ruby, Python, Javascript (Groovy??)

>> Store stack in locals, reconstruct when bailout





Render Mandelbrot, 1024x1024 tile, 50 iterations

Perspectives

Background compilation

IndyDroid (JSR 292 on Android)

Lambda (JSR 335)