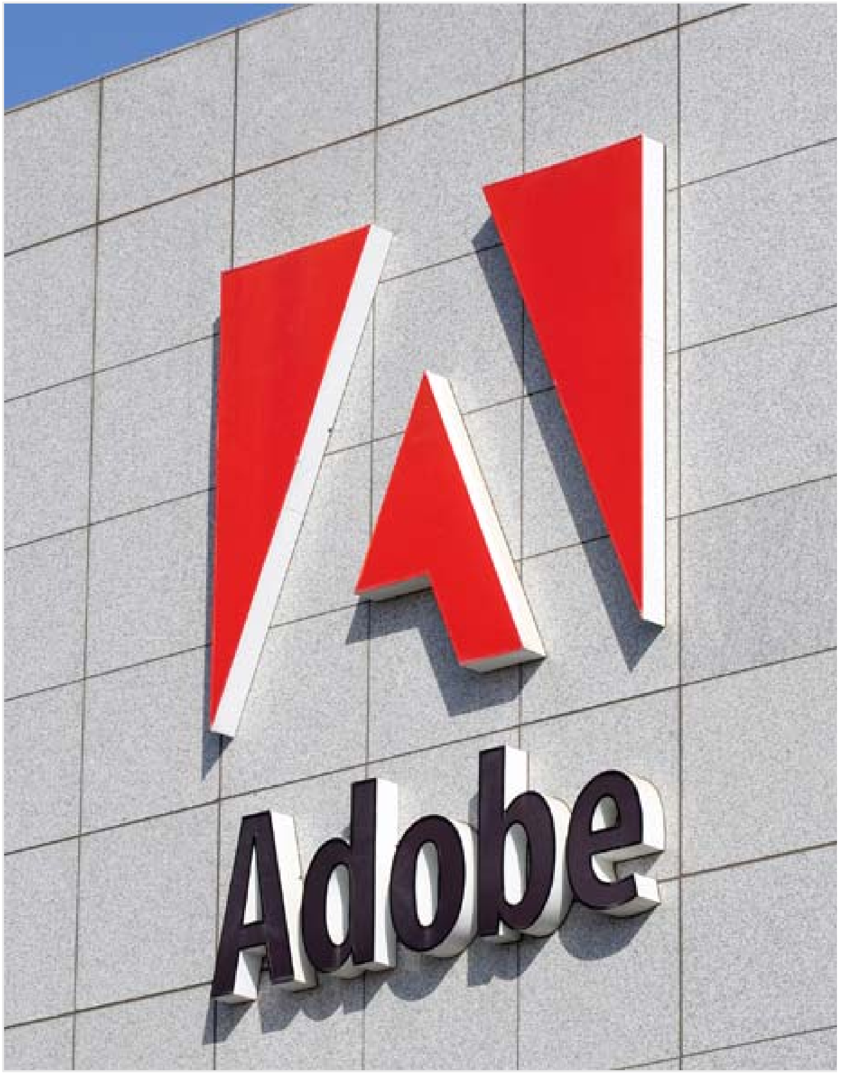
ActionScript 3.0 and AVM2: Performance Tuning

Gary Grossman

Adobe Systems



Agenda

- Two goals:
  - Cover some techniques that can help performance
  - Pop the hood and talk about how the new VM works

Classes and Type Annotations

Adobe

Runtime natively supports classes

```
class A
{
  var a:Number = 3.14;
  var b:String = “a string”;
  var c:int = -1;
  public function A()
  {
    trace(“Constructor”);
  }
  public function method()
  {
    trace(“A.method”);
  }
}
```

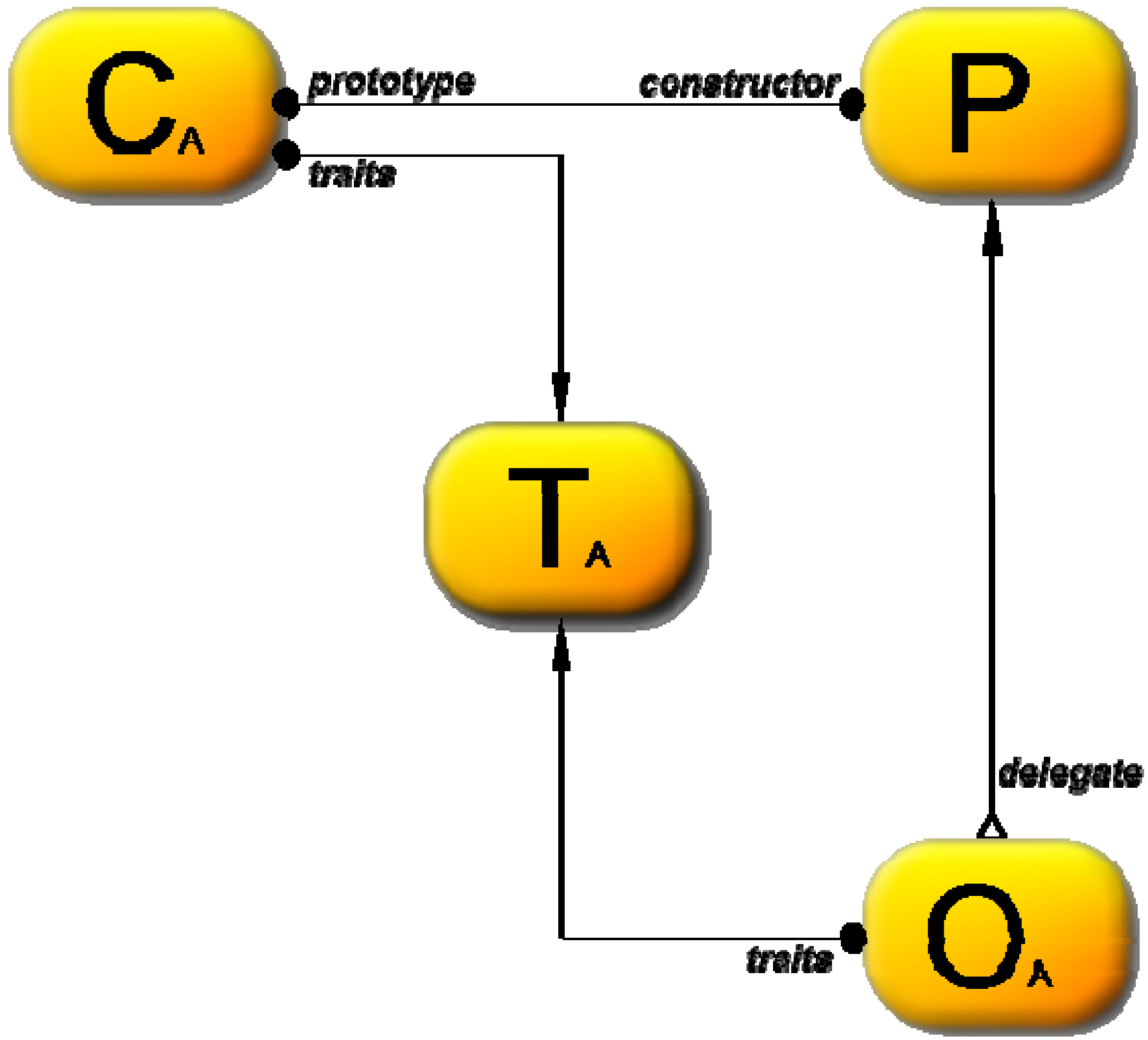
What that compiles to in AS2...

```
_global.A = function ()
{
  this.a = 3.14;
  this.b = “a string”;
  this.c = -1;
  trace(“Constructor”);
}
_global.A.prototype.method = function ()
{
  trace(“A.method”);
}
```

Atoms

- Atoms are the most primitive value in the AS1/AS2 system – a single dynamically typed value
- Atoms still exist in AS3, but only when type is unknown

AS3 Object Model: Traits



How traits represent objects: a sample class

```
class Shape
{
  var id:int;
  var name:String;
}
class Circle extends Shape {
  var radius:Number;
  var color:uint;

  public function Circle(radius:Number)
  {
    this.radius = radius;
  }

  public function area():Number
  {
    return Math.PI*radius*radius;
  }
}
```


Adobe

How traits describe objects

```
class Shape
{
  var id:int;
  var name:String;
}

class Circle extends Shape {
  var radius:Number;
  var color:uint;

  public function Circle(radius:Number)
  {
    this.radius = radius;
  }

  public function area():Number
  {
    return Math.PI*radius*radius;
  }
}
```

traits for class Shape

base class: Object **final: false** **dynamic: false**

methods

name	method id	type	return type	params
$construct	0	final	Void	(none)

properties

name	type	offset
id	int	12
name	String	16

How traits describe objects

```
class Shape
{
  var id:int;
  var name:String;
}

class Circle extends Shape {
  var radius:Number;
  var color:uint;

  public function Circle(radius:Number)
  {
    this.radius = radius;
  }

  public function area():Number
  {
    return Math.PI*radius*radius;
  }
}
```

traits for class Circle

base class: Shape **final: false** **dynamic: false**

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20

How traits describe objects

traits for class Circle

base class: Shape **final: false** **dynamic: false**

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20

instance of class Circle

0	vtable
4	traits
8	__proto__
12	id
16	name
20	color
24	radius

How traits describe objects

traits for class Circle

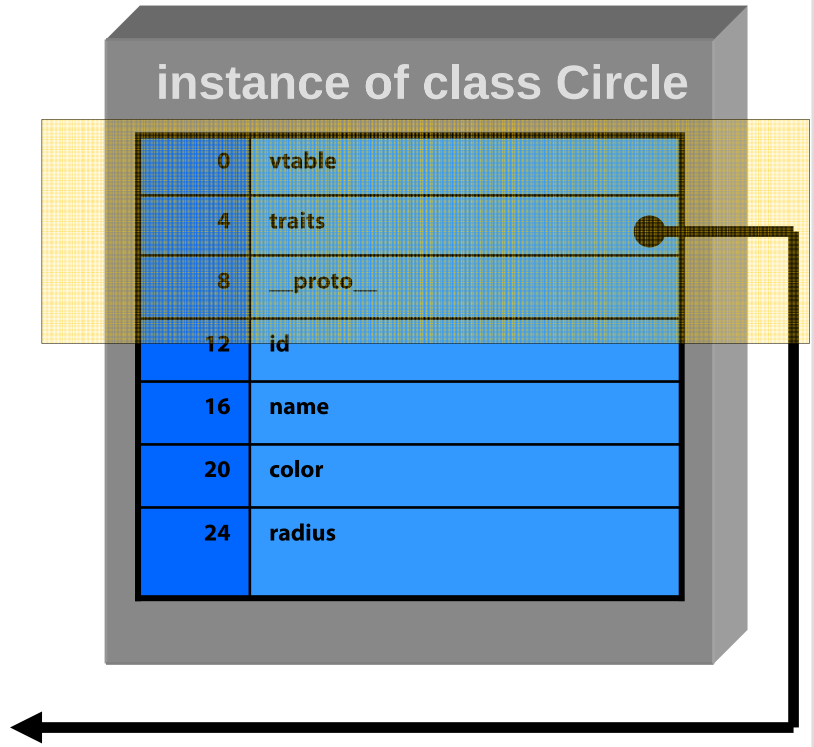
base class: Shape **final: false** **dynamic: false**

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20



Adobe

How traits describe objects

traits for class Circle

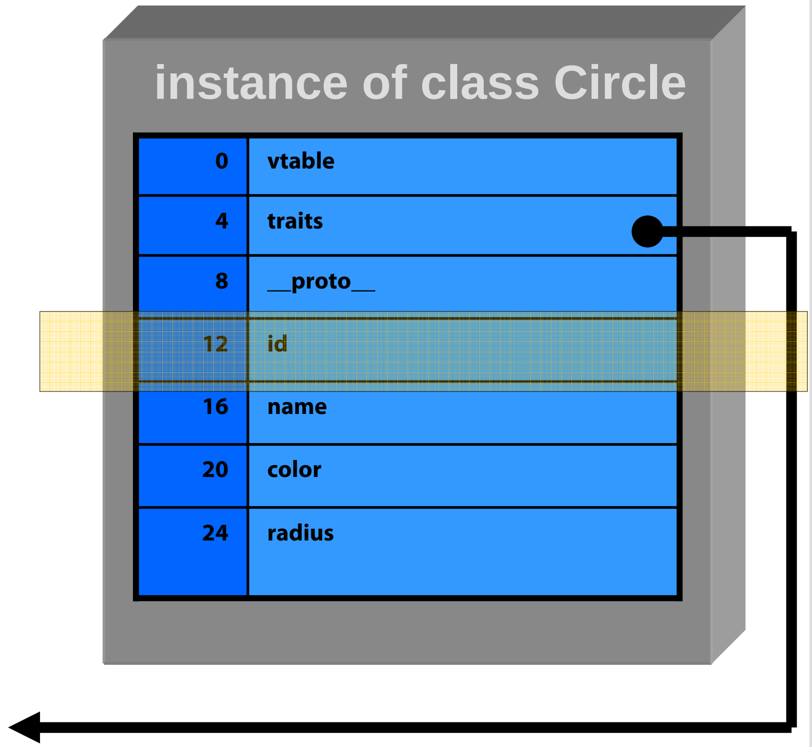
base class: Shape   final: false   dynamic: false

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20



Adobe

How traits describe objects

traits for class Circle

base class: Shape **final: false** **dynamic: false**

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20

instance of class Circle

0	vtable
4	traits
8	__proto__
12	id
16	name
20	color
24	radius

Adobe

How traits describe objects

traits for class Circle

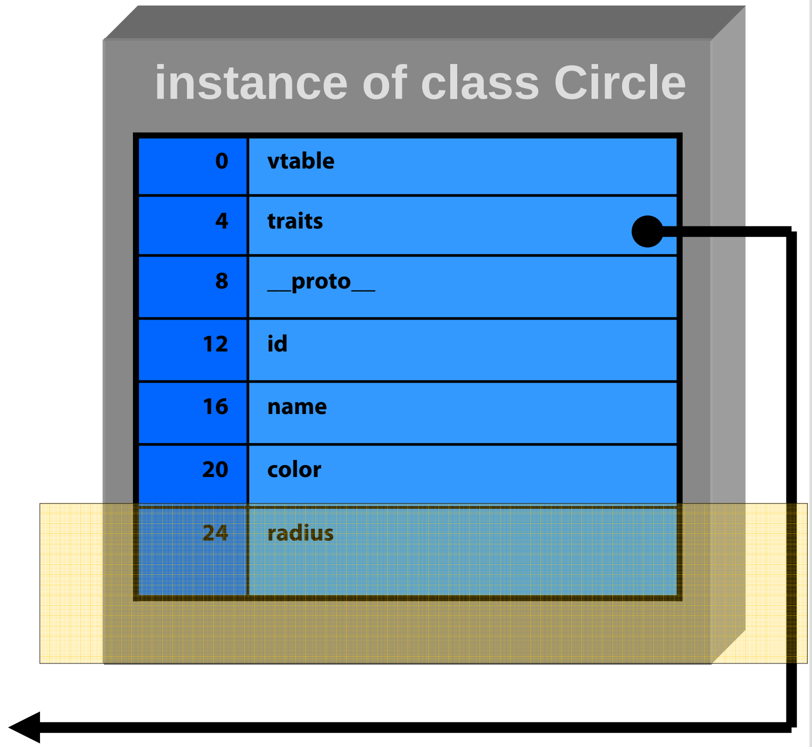
base class: Shape   final: false   dynamic: false

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20



Adobe

How traits describe objects

traits for class Circle

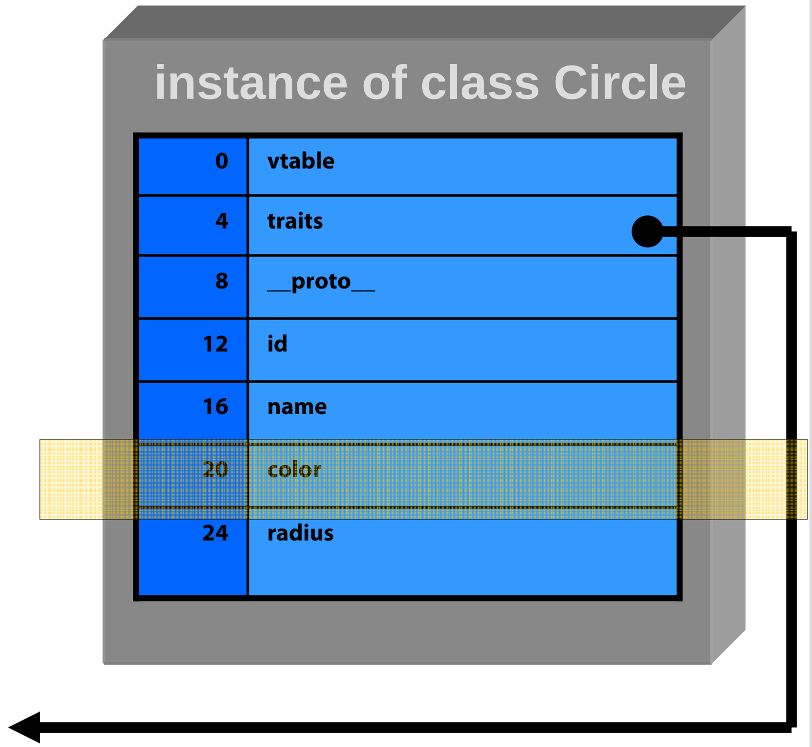
base class: Shape   final: false   dynamic: false

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20



Adobe

How traits describe objects

traits for class Circle

base class: Shape **final: false** **dynamic: false**

methods

name	method id	type	return type	params
$construct	0	final	Void	radius:Number
area	1	virtual	Number	(none)

properties

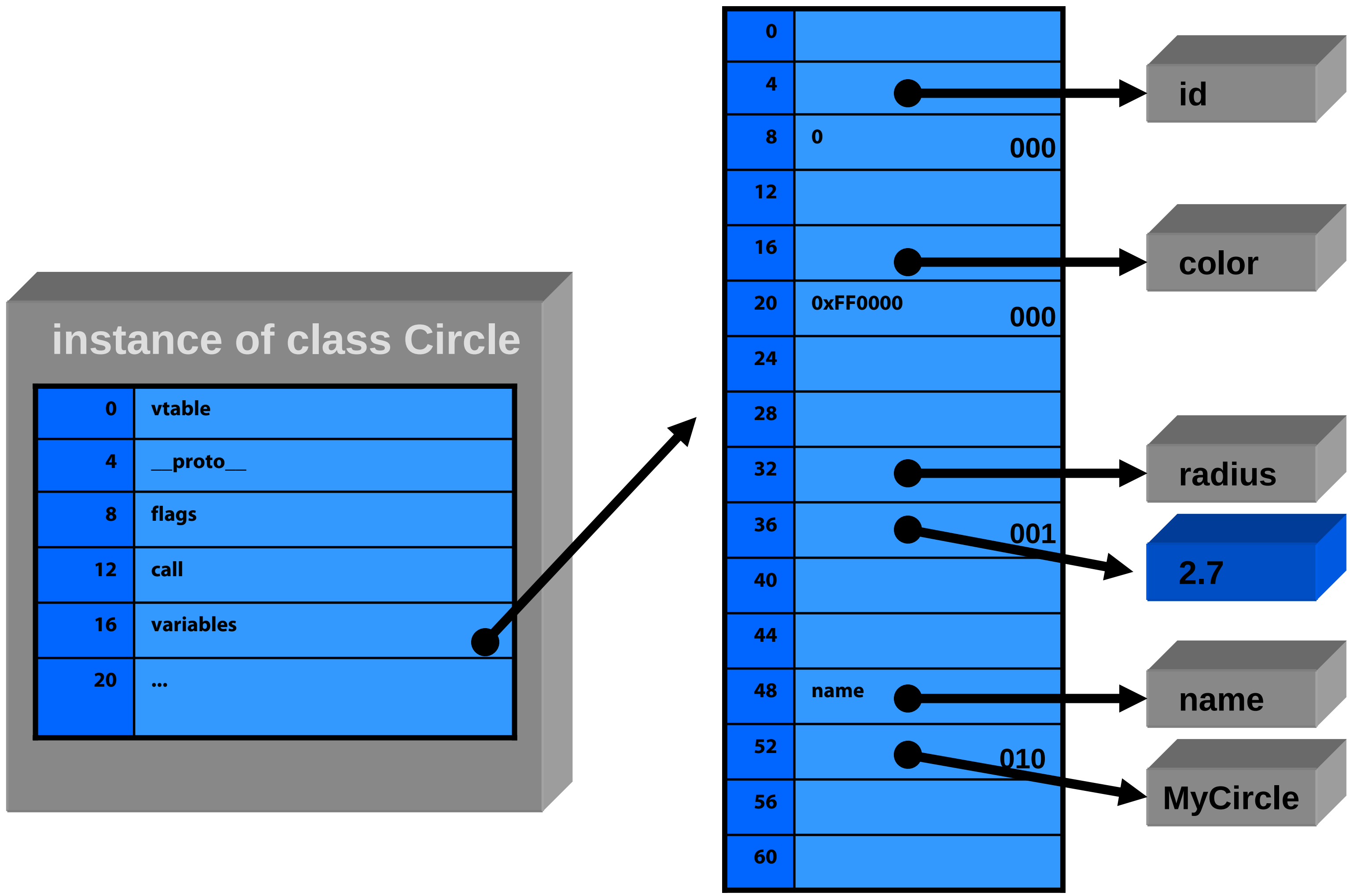
name	type	offset
id	*int*	*12*
name	*String*	*16*
radius	Number	24
color	uint	20

instance of class Circle

0	vtable
4	traits
8	__proto__
12	id
16	name
20	color
24	radius

total 32 bytes / instance

Objects in AVM1



Runtime natively supports strong types

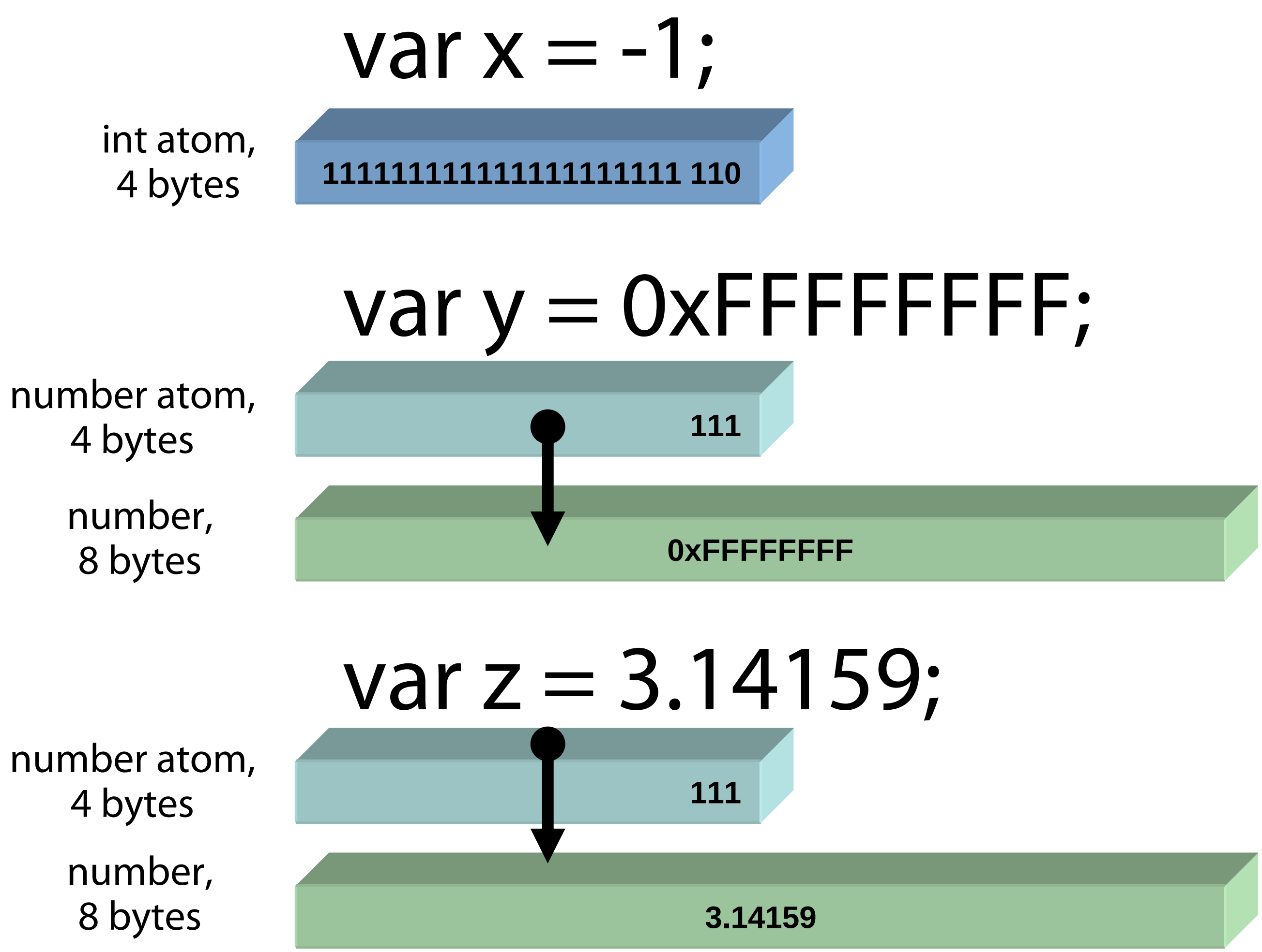
- In ActionScript 2.0:
  - Type annotations were a compiler hint
  - Type information did not reach all the way down to the runtime
  - All values were stored as dynamically typed atoms
  - Type annotations were a “best practice” for developer productivity
- In ActionScript 3.0:
  - Type annotations are employed to efficiently store values as native machine types
  - Type annotations improve performance and reduce memory consumption
  - Type annotations are essential to getting best performance and memory characteristics

The Power of “int”

Numeric Types

- int: 32-bit signed integer
- uint: 32-bit unsigned integer
- Number: 64-bit IEEE 754 double-precision floating-point number

Without Type Annotations



With Type Annotations

var x:int = -1;

4 bytes | -1

var y:uint = 0xFFFFFFFF;

4 bytes | 0xFFFFFFFF

var z:Number = 3.14159;

8 bytes | 3.14159

Adobe

Promotion of Numeric Types

Promotion of Numeric Types

- The semantics of ECMAScript require that ints often be promoted to Number

```
var i:int = 1;

// i+1 here will be a straight
// integer addition
var j:int = i+1

// i+1 here will require
// promotion to Number
print(i+1)
```

Promotion of Numeric Types

- Putting in a coerce to int/uint can help performance, if the compiler cannot infer that int/uint is what you want
- Array access has fast paths for int/uint, so coercion of index can help performance

```
var i:int;
// i*2 gets promoted to Number
for (i=0; i<10000; i++) {
  a[i*2] = 0;
}
// Goes through fast path
for (i=0; i<10000; i++) {
  a[int(i*2+1)] = 1;
}
```

CSE

CSE

- The VM does perform common subexpression elimination
- However, language semantics sometimes get in the way:

```
for (var i:int=0; i<a.length; i++)
{
  processRecord(a[i]);
}
```

- Because “length” might be overridden and have side effects, the VM cannot factor it out of the loop

CSE

- The VM does perform common subexpression elimination
- However, language semantics sometimes get in the way:

```
for (var i:int=0; i<a.length; i++)
{
  processRecord(a[i]);
}
```

- Because “length” might be overridden and have side effects, the VM cannot factor it out of the loop

CSE

- So, some hand CSE is still needed:

```
var n:int = a.length;
for (var i:int=0; i<n; i++)
{
  processRecord(a[i]);
}
```

Method Closures

Method Closures

- Often, developers write event handling code with anonymous function closures:

```
class Form
{
  function setupEvents()
  {
    var f = function(event:Event) {
      trace("my handler");
    }
    grid.addEventListener("click", f);
  }
}
```


Adobe

Method Closures

- Nested functions cause the outer function to create an **activation object**.
- This has some performance and memory impact.

```
class Form
{
  function setupEvents()
  {
    var f = function(event:Event) {
      trace(“my handler”);
    }
    grid.addEventListener(“click”, f);
  }
}
```

Method Closures

- Method closures solve the age-old AS2 problem of “this” changing
- Eliminates need for mx.utils.Delegate class from Flex 1.x

```
import mx.utils.Delegate;
class Form
{
  function setupEvents()
  {
    grid.addEventListener(“click”,
      Delegate.create(this, f)); // No more!
  }
  function f(e)
  {
    trace(“my handler”);
  }
}
```


Adobe

Method Closures

- Method closures are convenient to use, and more efficient, because there won't be an activation object created.

```
class Form
{
  function setupEvents()
  {
    grid.addEventListener("click", f);
  }
  function f(event:Event)
  {
    trace("my handler");
  }
}
```

Activation Objects

```
f:
 newactivation
 setlocal1
 getlocal1
 pushbyte 0
 setslot 0

g:
 pushbyte 0
 setlocal1
```

```
function f()
{
  var x:int = 0;
  …
}
```

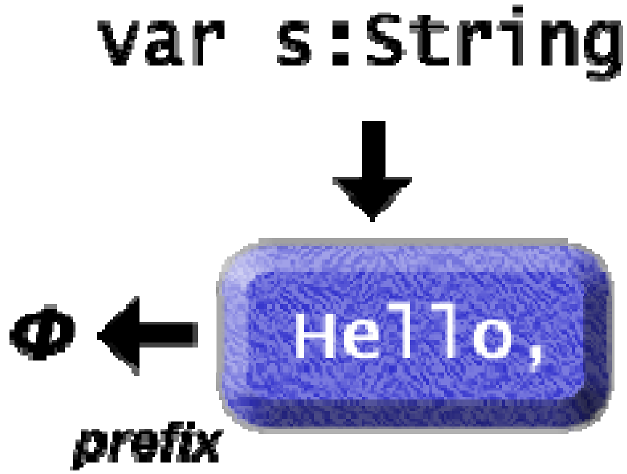
Compound Strings

Compound Strings

- For awhile, we had a class flash.utils.StringBuilder for fast string concatenation
- What happened?
- A: We made the + operator super-fast by implementing compound strings (cords), so StringBuilder was unneeded and removed

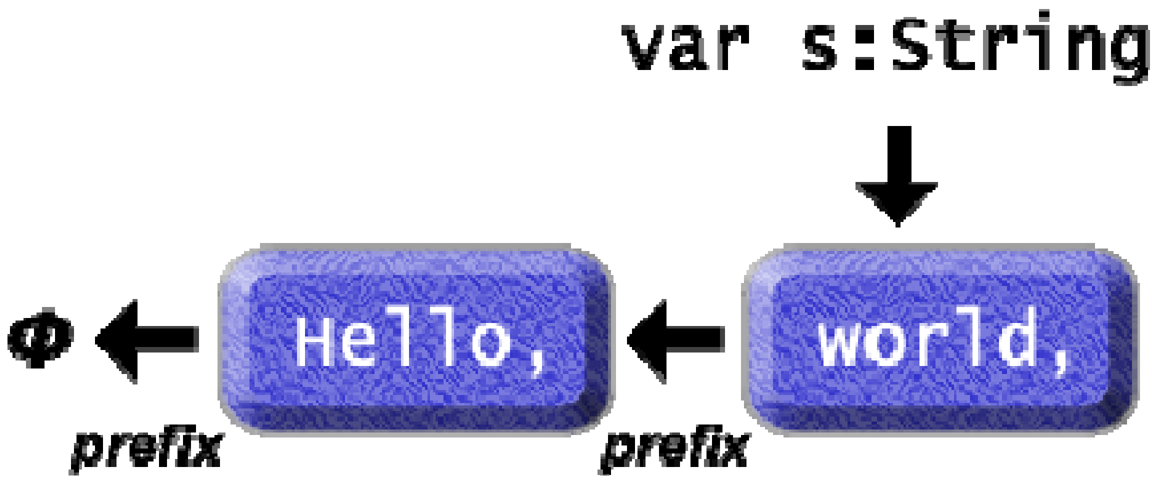
Compound Strings

```
var s:String = “Hello, ”;
s += “world, ”;
s += “from AS3!”;
```



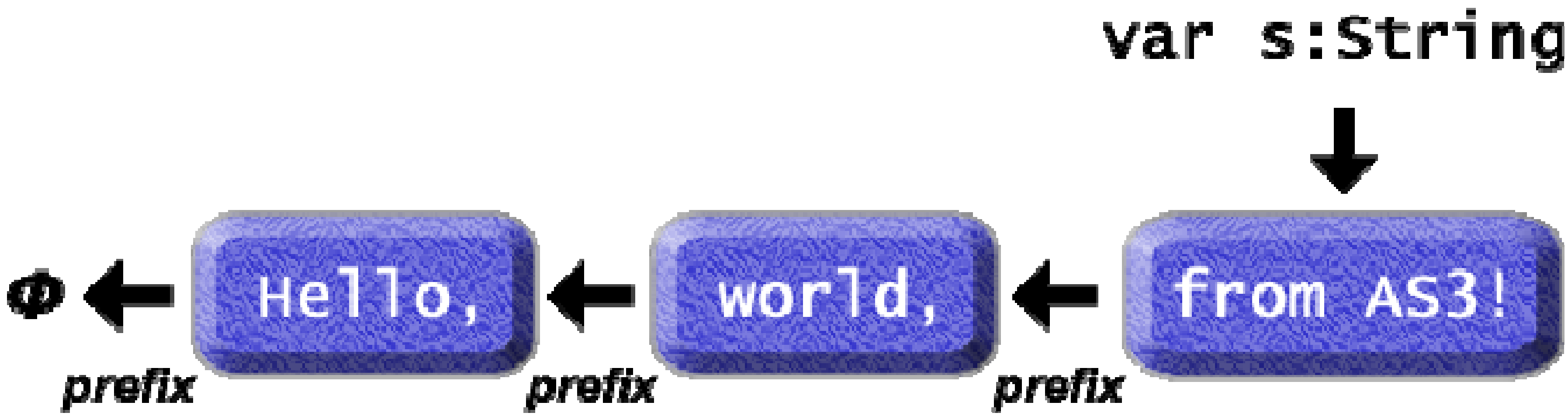
Compound Strings

```
var s:String = “Hello, ”;
s += “world, ”;
s += “from AS3!”;
```



Compound Strings

```
var s:String = “Hello, ”;
s += “world, ”;
s += “from AS3!”;
```



Interpret vs. JIT

Interpret vs. JIT

- We make a simple "hotspot"-like decision about whether to interpret or JIT
- Initialization functions ($init, $cinit) are interpreted
- Everything else is JIT
- Upshot: Don't put performance-intensive code in class initialization:

```
class Sieve
{
  var n:int, sieve:Array=[], c:int, i:int, inc:int;
  set_bit(0, 0, sieve);
  set_bit(1, 0, sieve);
  set_bit(2, 1, sieve);
  for (i = 3; i <= n; i++) set_bit(i, i & 1, sieve);
  c = 3;
  do { i = c * c, inc = c + c; while (i <= n) { set_bit(i, 0, sieve); i += inc; } c += 2;
  while (!get_bit(c, sieve)) c++; } while (c * c <= n); }
  …
```

Garbage Collection

MMgc Garbage Collector: Overview

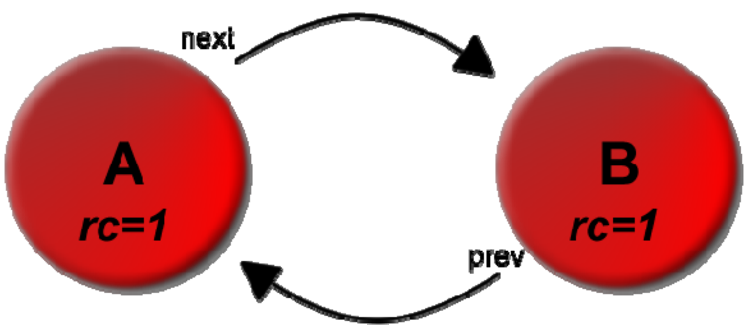
- Reusable C/C++ library
- Used by AVM1, AVM2 and Player's display list
- Not specific to Flash Player
- new/delete (unmanaged memory)
- new w/ optional delete (garbage collection)
- memory debugging aids
- profiling

Adobe

Garbage Collection

- Old school tech mainstreamed by Java
- Key to VM performance
- Our algorithm
  - Deferred Reference Counting (DRC)
  - Backed by incremental conservative mark/sweep collector

```
A = new Object();
B = new Object();
A.next = B;
B.prev = A;
```



Deferred Reference Counting

- All about speed, 20% speedup from 7 to 8
- Only maintain RC for heap to heap references
- Ignore stack and registers (scratch memory)
- Put Zero count items in Zero Count Table (ZCT)
- Scan stack when ZCT is full
- Delete objects in ZCT not found on stack
- Wash and repeat

Incremental Collection

- Marking limited to 30 ms time slices
- Stop start marking
- Smart pointers for minimal dev impact
- Lazy Sweeping
- DRC tied into write barriers (heap to heap)

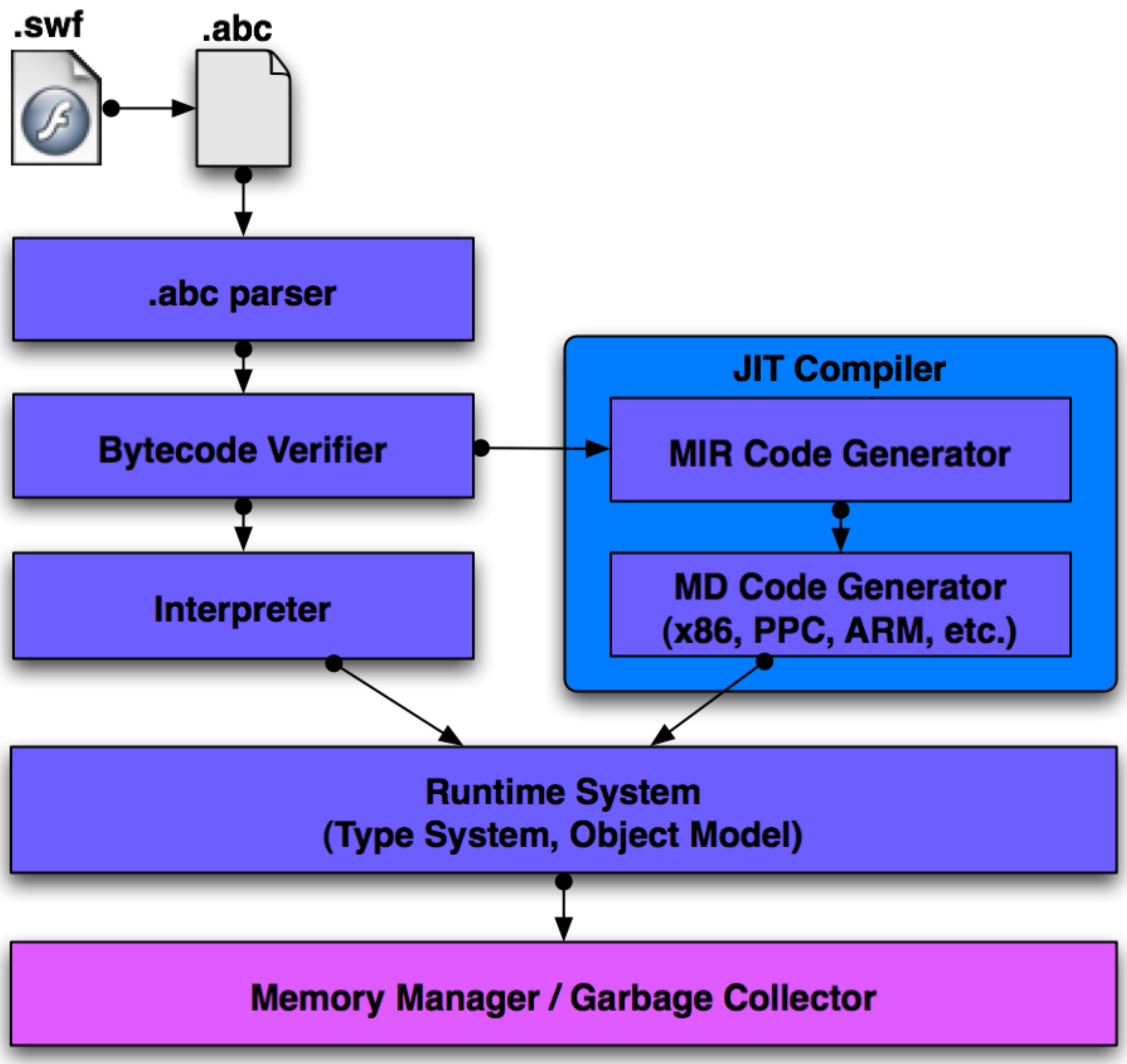
Conservative Collection

- One mark routine for all memory
- False positives are possible but manageable:
  - Clean stack
  - Keep ‘em separated
- No need to write marking routines

A peek inside the JIT

Adobe

AVM2 Architecture



.abc Bytecode: Code Compression

- Constant Data
  - Strings, Numbers, etc
  - Multinames = {ns set}::name
- RTTI
  - Method Descriptors
  - Type Descriptors, a.k.a Traits
- Bytecode
  - Stack Machine notation

Bytecode Verifier

- Structural Integrity
  - Branches must land on valid instructions
  - Can’t fall off end of code
  - Constant references valid
- Type Safety
  - Dataflow Analysis to track types
  - Early Binding
- MIR Code Generation (optional)
  - Generate IR while verifying
  - Single pass to verify + generate IR

Interpreter

- Stack Machine, no surprises

```
for (;;) {
   switch (*pc) {
          case OP_pushstring: …
          case OP_pop: …
          case OP_callproperty: …
   }
```

- All values are boxed, 32-bit atoms
- Code executes from verified .abc data in SWF

Adobe

MIR: Macromedia Intermediate Representation

- Used in JIT compiler to abstract commonalities between CPU's

Just In Time Code Generation

- MIR Code Generation
  - Concurrent with Verifier
  - Early Binding
  - Constant Folding
  - Copy & Constant Propagation
  - Common Subexpression Elimination (CSE)
  - Dead Code Elimination (DCE)
- MD Code Generation
  - Instruction Selection
  - Register Allocation
  - Dead Code Elimination (DCE)

A Tale of Three Notations

AS3
```
function (x:int):int {
  return x+10
}
```

.abc
```
getlocal 1
pushint 10
add
returnvalue
```

MIR
```
@1 arg +8// argv
@2 load [@1+4]
@3 imm 10
@4 add (@2,@3)
@5 ret @4  // @4:eax
```

x86
```
mov eax,(eap+8)
mov eax,(eax+4)
add eax,10
ret
```

JIT Overview

- Conventional: Write program, compile to platform and then execute.
- Program bound to hardware early raises a number of issues, mainly portability and size.
- JIT idea: write program, but don't 'compile' until code is actually on the target platform.

Balance

- The question - compile or execute?
- First generation
  - JIT spent quite a bit of time compiling.
  - Paid the price in start-up performance.
- Next generation
  - 2 JITs for two environments; ‘client’ and ‘server’
  - Client - better start-up performance for programs like dynamic GUI apps
  - Server – best for apps that can tolerate higher start-up hit

Balance

- Our objectives
  - Fast compile times
  - Limited passes
  - Cautious with memory

- All this and we kept an eye on portability from the onset.

Architecture

- Hybrid execution model – allows us to interpret .abc directly or invoke JIT compiler
- JIT compiler translates bytecodes into native machine code in 2 passes
- Only the back-end of JIT compiler is platform-dependent; needs retargeting for each CPU
- Support for x86, PowerPC, ARM …

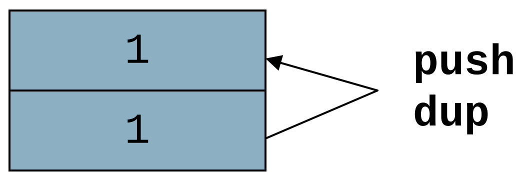
Adobe

MIR

- What?
  - Internal representation of the program that bridges .abc and target instruction set
  - 3-tuple; operation + 2 operands
- Why?
  - Allows us to perform optimizations that otherwise would be quite difficult using a stack based notation
  - Easier to map to underlying hardware

Optimizations

- Translation from .abc
  - Stack manipulation and local moves become no-ops



- Common sub-expression elimination

```
a = x + y          @3 add @1 @2
…                  @4 …
b = x + y          @8 add @1 @2
```

Instruction not generated.
Instead we place ref '@3'

Adobe

Early Binding

- Can take advantage of running state of system.
- Some objects and properties already resolved and bound.
- During verification stage, type information is propagated.
- Allows support for native types.

Field Binding

```
public final class C {
        public var f:int;
}
var o:C = new C();
o.f = 46;
```

```
30:pushbyte 46
       @90     imm   46
                        stack: C?@89 int@90
32:setproperty {public,bind$1}::f
       @93     st    16(@89) <- @90
```

Field Binding

```
public final class C {
        public var f:int;
}
var o:C = new C();
o.f = 46;
```

```
30:pushbyte 46
       @90     imm   46
                        stack: C?@89 int@90
32:setproperty {public,bind$1}::f
       @93     st    16(@89) <- @90
```

o is of type C

Adobe

Field Binding

```
public final class C {
        public var f:int;
}
var o:C = new C();
o.f = 46;
```

46 is an int

```
30:pushbyte 46
       @90     imm   46
                        stack: C?@89 int@90
32:setproperty {public,bind$1}::f
       @93     st    16(@89) <- @90
```


Adobe

Field Binding

```
public final class C {
        public var f:int;
}
var o:C = new C();
o.f = 46;
```

```
30:pushbyte 46
       @90     imm   46
                         stack: C?@89
32:setproperty {public,bind$1}::f
       @93     st    16(@89) <- @90
```

o has field named f that can be resolved

Adobe

Field Binding

```
public final class C {
        public var f:int;
}
var o:C = new C();
o.f = 46;
```

```
30:pushbyte 46
       @90     imm   46
                        stack: C?@89 int@90
32:setproperty {public,bind$1}::f
       @93    st    16(@89) <- @90
```

Location of f resolves to offset on object and type is int so no coerce needed

Adobe

Machine Code (MD) Generation

- In the next and final pass we translate MIR into platform specific instructions.
- Instruction selection (IS)
- Register allocation (RA)
- Register / stack management

Machine Code (MD) Generation

- In the next and final pass we translate MIR into platform specific instructions.
- Instruction selection (IS)
- Register allocation (RA)
- Register / stack management

```
@90 imm   46
                                 active: ecx(89-93)
@93 st    16(@89) <- @90
    03A20153  mov   16(ecx), 46
```


Adobe

Machine Code (MD) Generation

- In the next and final pass we translate MIR into platform specific instructions.
- Instruction selection (IA)
- Re
- Re

ecx contains o pointer and IA32 mov instruction allows immediate (46) as an operand.

```
@90 imm   46
                                          active: ecx(89-93)
@93 st    16(@89) <- @90
    03A20153  mov   16(ecx), 46
```


Adobe

Other IS / RA notables

- A variant of Linear Scan Register Allocation (LSRA)
  - Size/speed requirements made this allocator a good fit
  - Register hinting support
- Location of operands feeds instruction selector
  - Supports optimal use of stack and registers
  - Constants fold directly into instruction

Adobe

Better by Adobe.™

Adobe