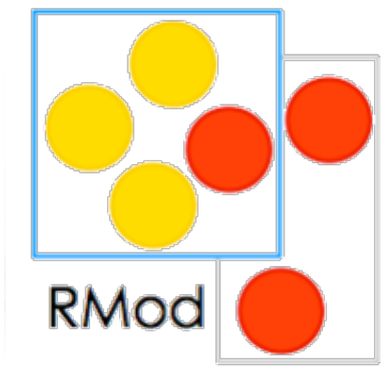
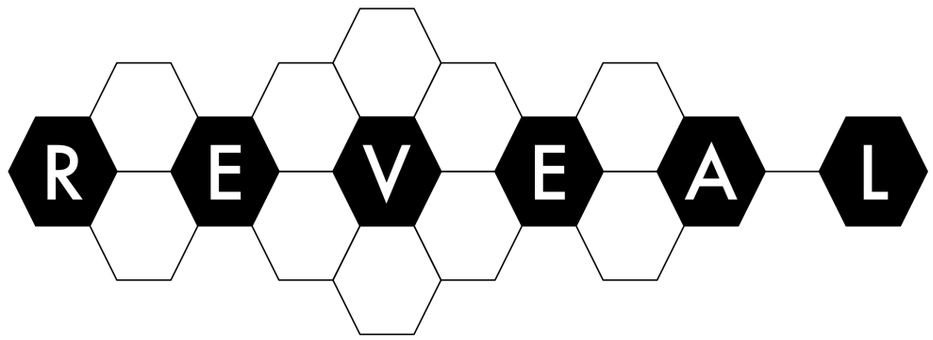
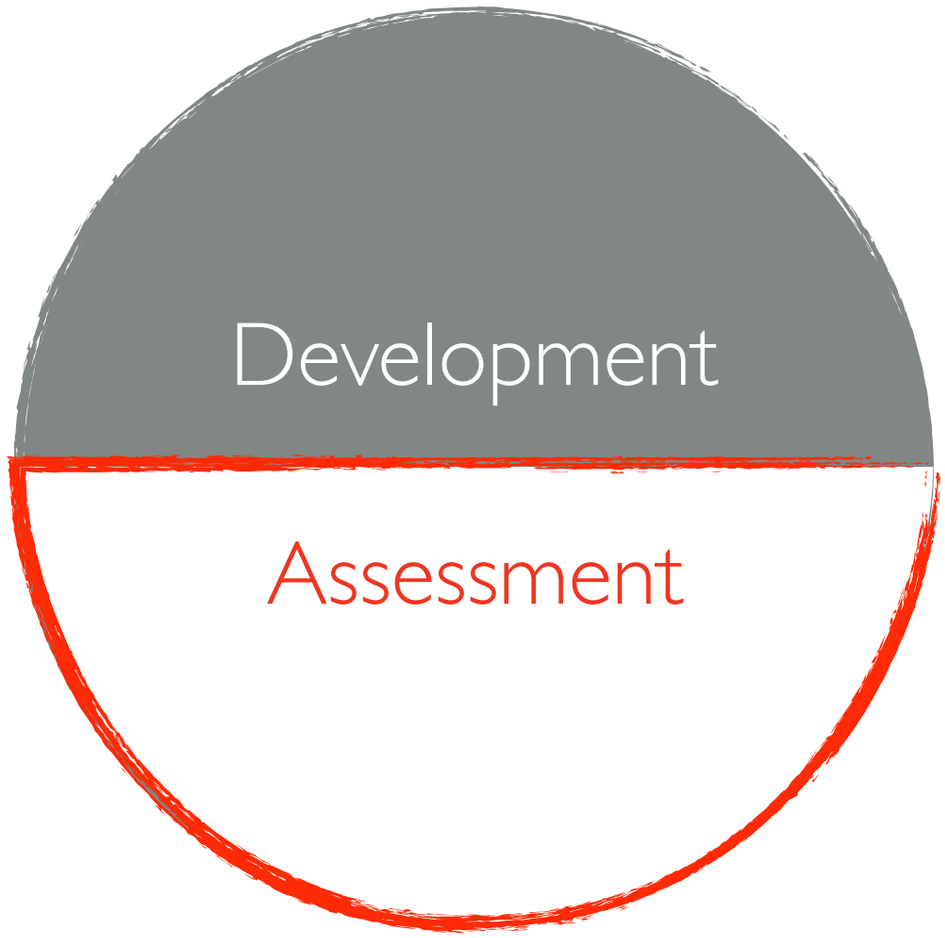
Generic Name Resolution with Metamodels

Yuriy Tymchuk, Benjamin Arezki, Anne Etien,
Nicolas Anquetil and Stéphane Ducasse





Software Assessment



Software Assessment

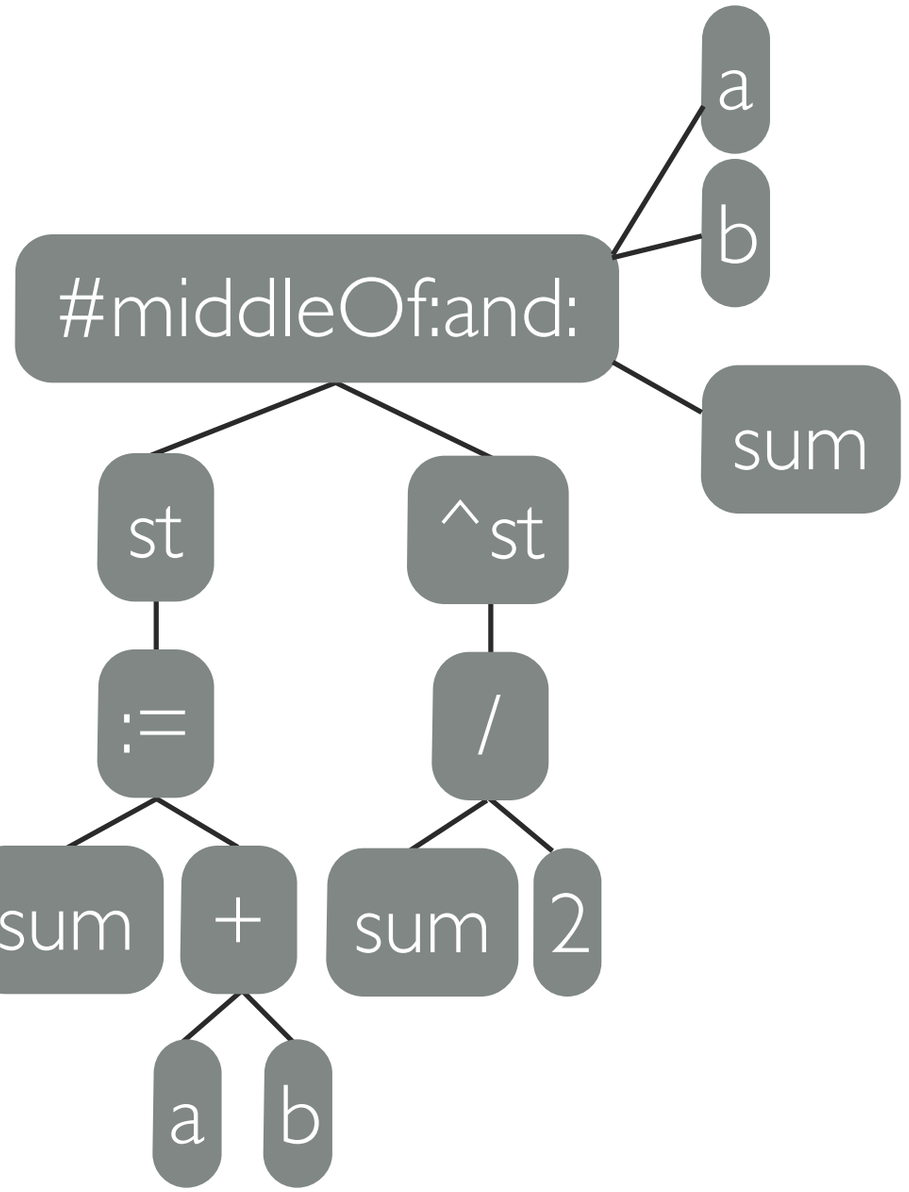
```
    ENTER_V8;
    EXCEPTION_PREAMBLE();
    i::Handle<i::Object> num =
        i::Execution::ToInt32(obj, &has_pending_exception);
    EXCEPTION_BAILOUT_CHECK(0);
    if (num->IsSmi()) {
      return i::Smi::cast(*num)->value();
    } else {
      return static_cast<int32_t>(num->Number());
    }
  }
}


bool Value::Equals(Handle<Value> that) const {
  if (IsDeadCheck("v8::Value::Equals()")
      || EmptyCheck("v8::Value::Equals()", this)
      || EmptyCheck("v8::Value::Equals()", that)) {
    return false;
  }
  LOG_API("Equals");
  ENTER_V8;
  i::Handle<i::Object> obj = Utils::OpenHandle(this);
  i::Handle<i::Object> other = Utils::OpenHandle(*that);
  i::Object** args[1] = { other.location() };
  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> result =
      CallV8HeapFunction("EQUALS", obj, 1, args, &has_pending_exception);
  EXCEPTION_BAILOUT_CHECK(false);
  return *result == i::Smi::FromInt(i::EQUAL);
}


bool Value::StrictEquals(Handle<Value> that) const {
  if (IsDeadCheck("v8::Value::StrictEquals()")
      || EmptyCheck("v8::Value::StrictEquals()", this)
      || EmptyCheck("v8::Value::StrictEquals()", that)) {
    return false;
  }
  LOG_API("StrictEquals");
  i::Handle<i::Object> obj = Utils::OpenHandle(this);
  i::Handle<i::Object> other = Utils::OpenHandle(*that);
  // Must check HeapNumber first, since NaN !== NaN.
  if (obj->IsHeapNumber()) {
    if (!other->IsNumber()) return false;
    double x = obj->Number();
    double y = other->Number();
    // Must check explicitly for NaN:s on Windows, but -0 works fine.
    return x == y && !isnan(x) && !isnan(y);
  } else if (*obj == *other) {  // Also covers Booleans.
    return true;
  } else if (obj->IsSmi()) {
    return other->IsNumber() && obj->Number() == other->Number();
  } else if (obj->IsString()) {
    return other->IsString() &&
      i::String::cast(*obj)->Equals(i::String::cast(*other));
  } else if (obj->IsUndefined() || obj->IsUndetectableObject()) {
    return other->IsUndefined() || other->IsUndetectableObject();
  } else {
    return false;
  }
}


uint32_t Value::Uint32Value() const {
  if (IsDeadCheck("v8::Value::Uint32Value()")) return 0;
  LOG_API("Uint32Value");
  i::Handle<i::Object> obj = Utils::OpenHandle(this);
  if (obj->IsSmi()) {
    return i::Smi::cast(*obj)->value();
  } else {
    ENTER_V8;
    EXCEPTION_PREAMBLE();
    i::Handle<i::Object> num =
        i::Execution::ToUint32(obj, &has_pending_exception);
    EXCEPTION_BAILOUT_CHECK(0);
    if (num->IsSmi()) {
      return i::Smi::cast(*num)->value();
    } else {
      return static_cast<uint32_t>(num->Number());
    }
  }
}
```

```
bool v8::Object::Set(v8::Handle<Value> key, v8::Handle<Value> value,
                     v8::PropertyAttribute attribs) {
  ON_BAILOUT("v8::Object::Set()", return false);
  ENTER_V8;
  HandleScope scope;
  i::Handle<i::Object> self = Utils::OpenHandle(this);
  i::Handle<i::Object> key_obj = Utils::OpenHandle(*key);
  i::Handle<i::Object> value_obj = Utils::OpenHandle(*value);
  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> obj = i::SetProperty(
      self,
      key_obj,
      value_obj,
      static_cast<PropertyAttributes>(attribs));
  has_pending_exception = obj.is_null();
  EXCEPTION_BAILOUT_CHECK(false);
  return true;
}


bool v8::Object::Set(uint32_t index, v8::Handle<Value> value) {
  ON_BAILOUT("v8::Object::Set()", return false);
  ENTER_V8;
  HandleScope scope;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);
  i::Handle<i::Object> value_obj = Utils::OpenHandle(*value);
  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> obj = i::SetElement(
      self,
      index,
      value_obj);
  has_pending_exception = obj.is_null();
  EXCEPTION_BAILOUT_CHECK(false);
  return true;
}


bool v8::Object::ForceSet(v8::Handle<Value> key,
                          v8::Handle<Value> value,
                          v8::PropertyAttribute attribs) {
  ON_BAILOUT("v8::Object::ForceSet()", return false);
  ENTER_V8;
  HandleScope scope;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);
  i::Handle<i::Object> key_obj = Utils::OpenHandle(*key);
  i::Handle<i::Object> value_obj = Utils::OpenHandle(*value);
  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> obj = i::ForceSetProperty(
      self,
      key_obj,
      value_obj,
      static_cast<PropertyAttributes>(attribs));
  has_pending_exception = obj.is_null();
  EXCEPTION_BAILOUT_CHECK(false);
  return true;
}


bool v8::Object::ForceDelete(v8::Handle<Value> key) {
  ON_BAILOUT("v8::Object::ForceDelete()", return false);
  ENTER_V8;
  HandleScope scope;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);
  i::Handle<i::Object> key_obj = Utils::OpenHandle(*key);

  // When turning on access checks for a global object deoptimize all functions
  // as optimized code does not always handle access checks.
  i::Deoptimizer::DeoptimizeGlobalObject(*self);

  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> obj = i::ForceDeleteProperty(self, key_obj);
  has_pending_exception = obj.is_null();
  EXCEPTION_BAILOUT_CHECK(false);
  return obj->IsTrue();
}


Local<Value> v8::Object::Get(v8::Handle<Value> key) {
  ON_BAILOUT("v8::Object::Get()", return Local<v8::Value>());
  ENTER_V8;
  i::Handle<i::Object> self = Utils::OpenHandle(this);
  i::Handle<i::Object> key_obj = Utils::OpenHandle(*key);
  EXCEPTION_PREAMBLE();
```

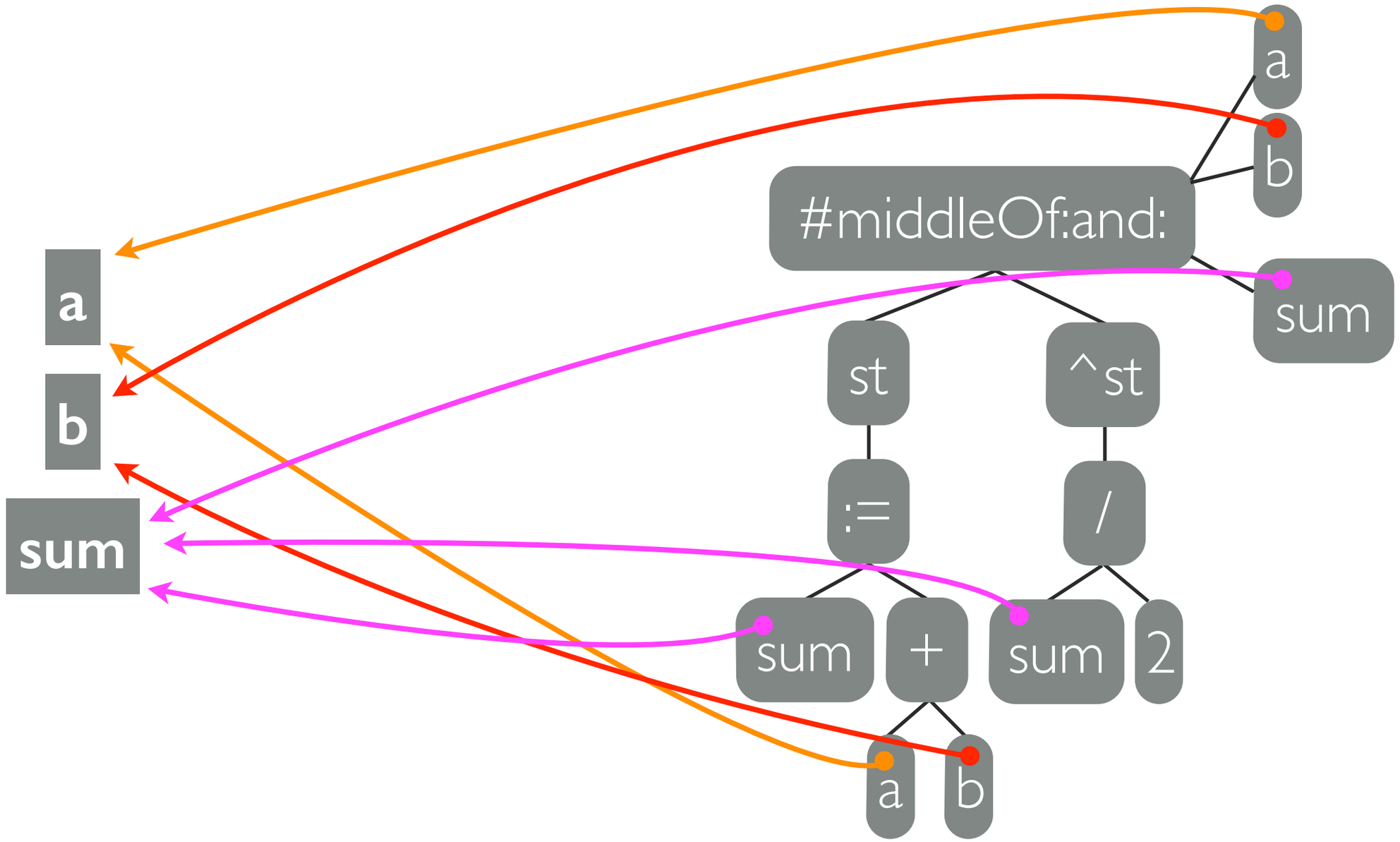
```
  i::Handle<i::Object> result = i::GetProperty(self, key_obj);
  has_pending_exception = result.is_null();
  EXCEPTION_BAILOUT_CHECK(Local<Value>());
  return Utils::ToLocal(result);
}


Local<Value> v8::Object::Get(uint32_t index) {
  ON_BAILOUT("v8::Object::Get()", return Local<v8::Value>());
  ENTER_V8;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);
  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> result = i::GetElement(self, index);
  has_pending_exception = result.is_null();
  EXCEPTION_BAILOUT_CHECK(Local<Value>());
  return Utils::ToLocal(result);
}


Local<Value> v8::Object::GetPrototype() {
  ON_BAILOUT("v8::Object::GetPrototype()", return Local<v8::Value>());
  ENTER_V8;
  i::Handle<i::Object> self = Utils::OpenHandle(this);
  i::Handle<i::Object> result = i::GetPrototype(self);
  return Utils::ToLocal(result);
}


bool v8::Object::SetPrototype(Handle<Value> value) {
  ON_BAILOUT("v8::Object::SetPrototype()", return false);
  ENTER_V8;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);
  i::Handle<i::Object> value_obj = Utils::OpenHandle(*value);
  EXCEPTION_PREAMBLE();
  i::Handle<i::Object> result = i::SetPrototype(self, value_obj);
  has_pending_exception = result.is_null();
  EXCEPTION_BAILOUT_CHECK(false);
  return true;
}


Local<Object> v8::Object::FindInstanceInPrototypeChain(
    v8::Handle<FunctionTemplate> tmpl) {
  ON_BAILOUT("v8::Object::FindInstanceInPrototypeChain()",
             return Local<v8::Object>());
  ENTER_V8;
  i::JSObject* object = *Utils::OpenHandle(this);
  i::FunctionTemplateInfo* tmpl_info = *Utils::OpenHandle(*tmpl);
  while (!object->IsInstanceOf(tmpl_info)) {
    i::Object* prototype = object->GetPrototype();
    if (!prototype->IsJSObject()) return Local<Object>();
    object = i::JSObject::cast(prototype);
  }
  return Utils::ToLocal(i::Handle<i::JSObject>(object));
}


Local<Array> v8::Object::GetPropertyNames() {
  ON_BAILOUT("v8::Object::GetPropertyNames()", return Local<v8::Array>());
  ENTER_V8;
  v8::HandleScope scope;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);
  i::Handle<i::FixedArray> value =
      i::GetKeysInFixedArrayFor(self, i::INCLUDE_PROTOS);
  // Because we use caching to speed up enumeration it is important
  // to never change the result of the basic enumeration function so
  // we clone the result.
  i::Handle<i::FixedArray> elms = i::Factory::CopyFixedArray(value);
  i::Handle<i::JSArray> result = i::Factory::NewJSArrayWithElements(elms);
  return scope.Close(Utils::ToLocal(result));
}


Local<String> v8::Object::ObjectProtoToString() {
  ON_BAILOUT("v8::Object::ObjectProtoToString()", return Local<v8::String>());
  ENTER_V8;
  i::Handle<i::JSObject> self = Utils::OpenHandle(this);

  i::Handle<i::Object> name(self->class_name());

  // Native implementation of Object.prototype.toString (v8natives.js):
  //   var c = %ClassOf(this);
  //   if (c === 'Arguments') c  = 'Object';
  //   return "[object " + c + "]";
```

Modelling source code

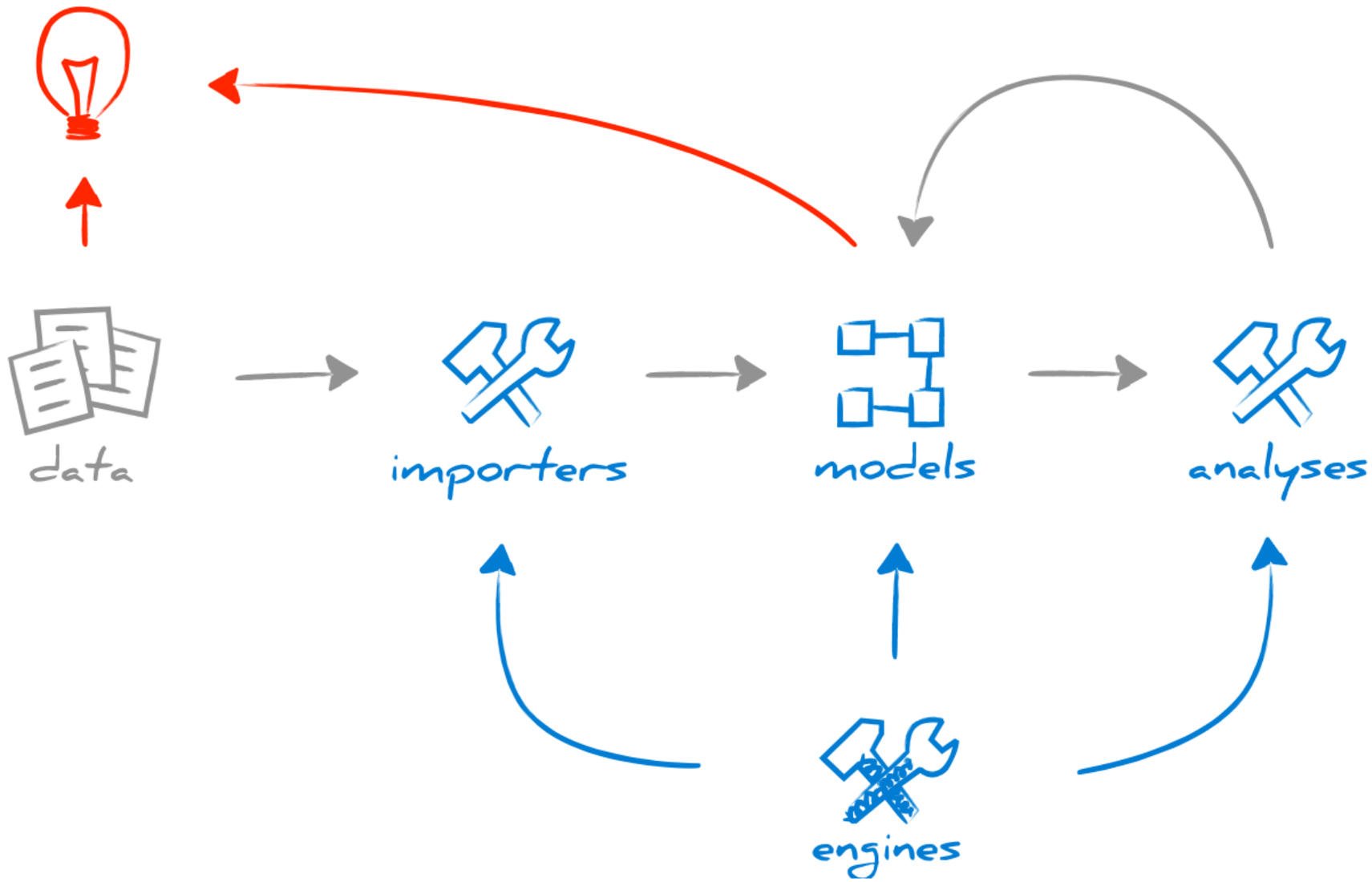
```
middleOf: a and: b
    | sum |
    sum := a + b.
    ^ sum / 2
```



Resolving symbols

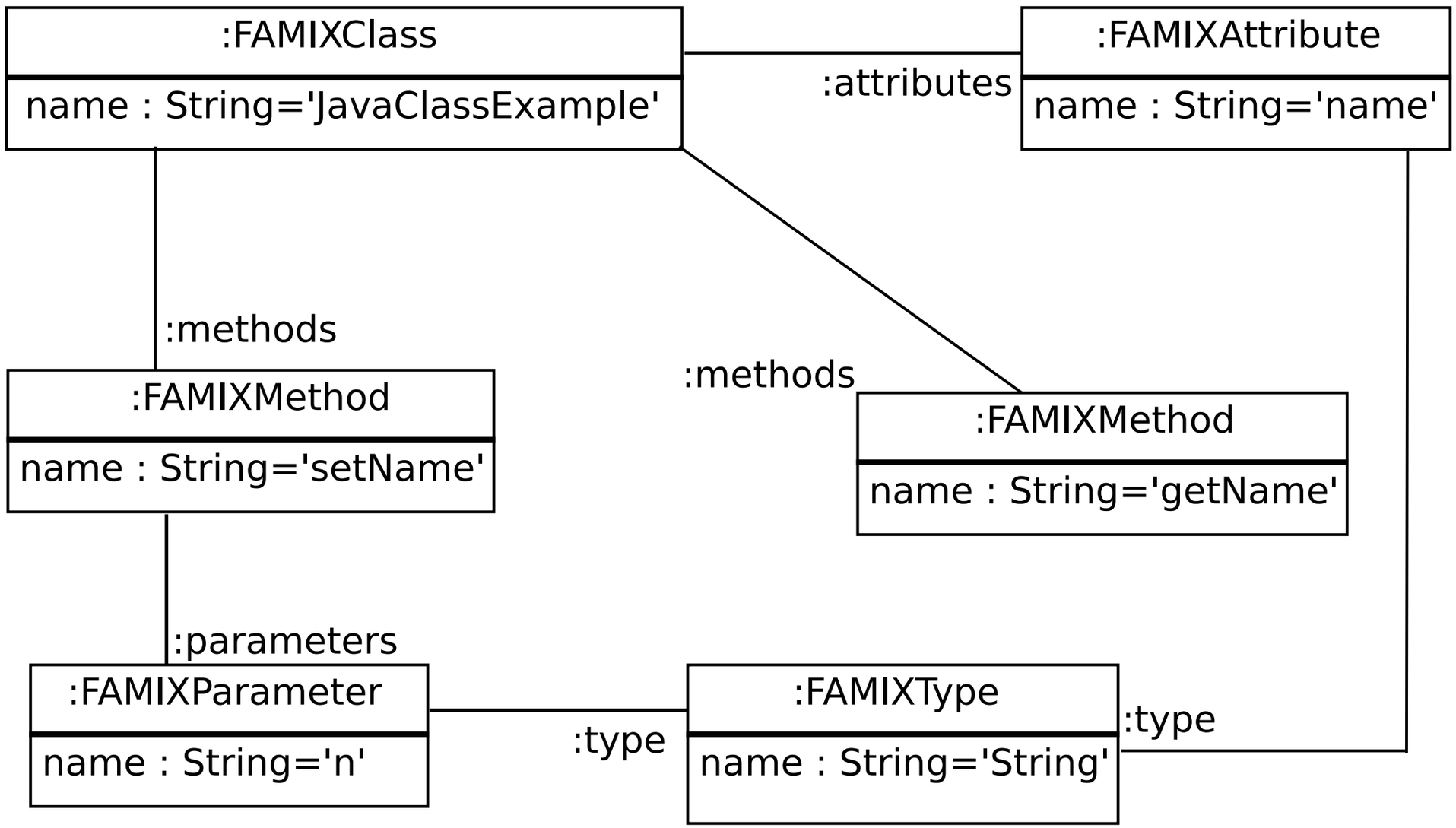


Moose Toolchain

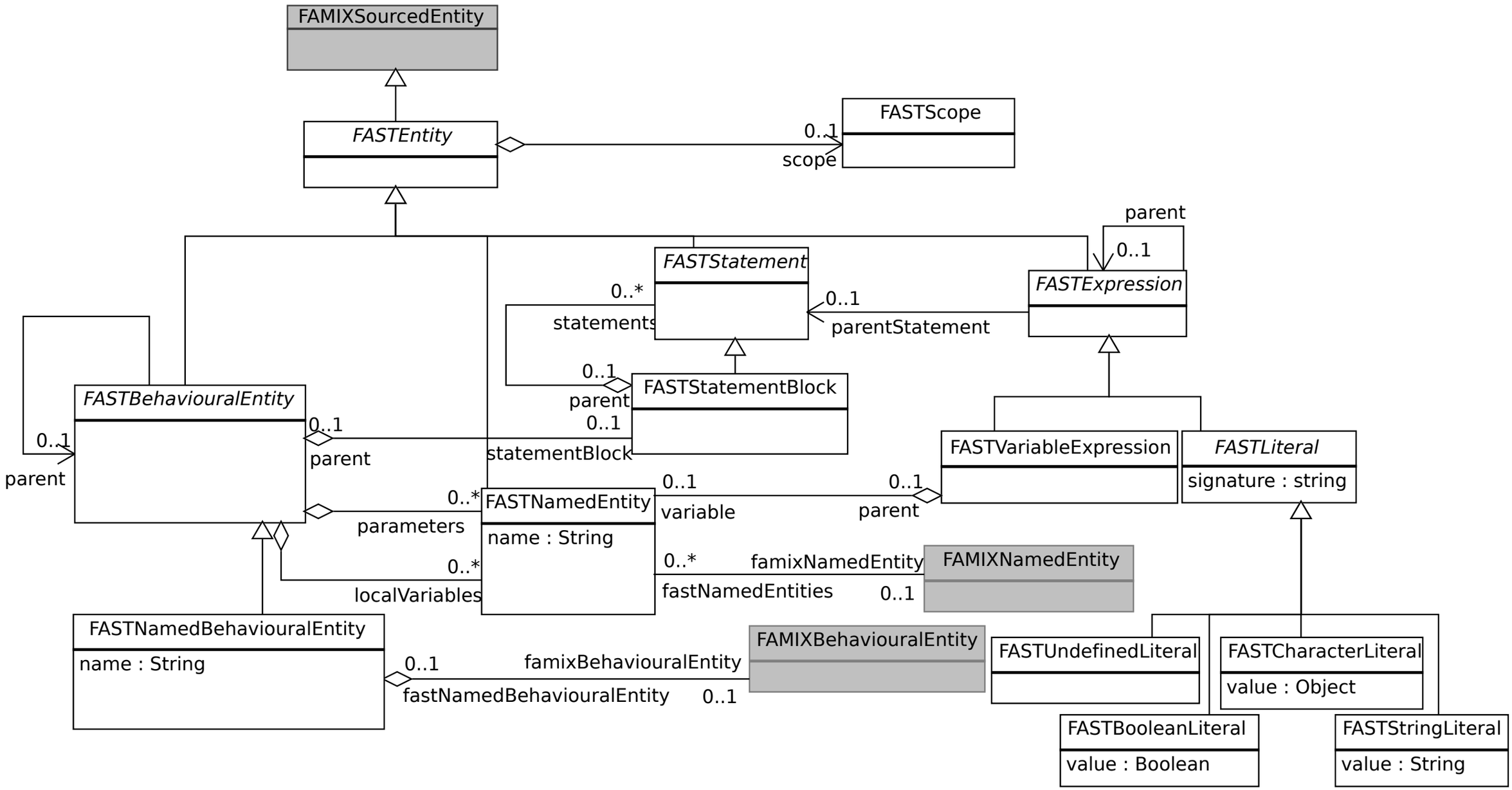


www.moosetechnology.org

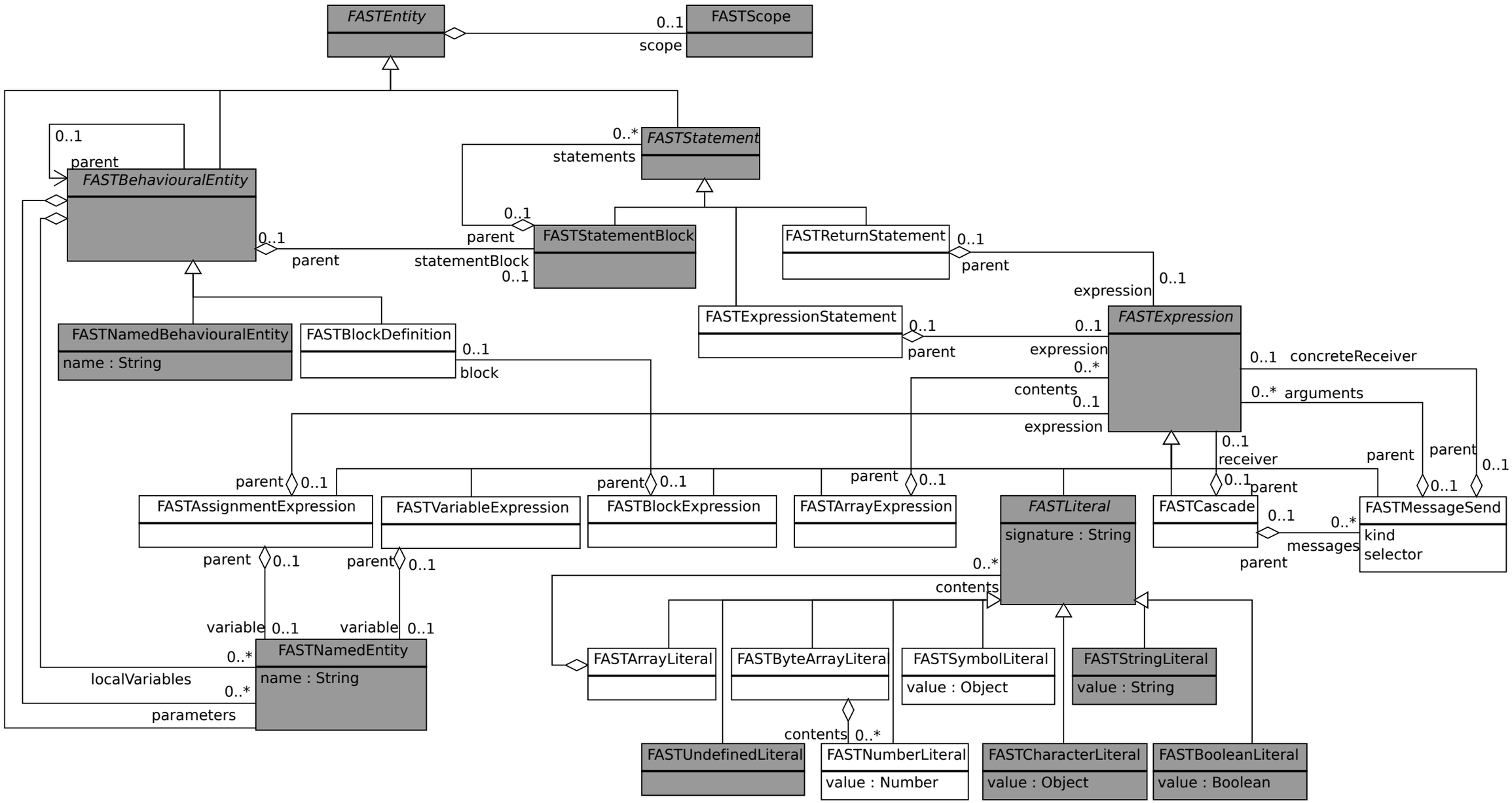
FAMIX Metamodel



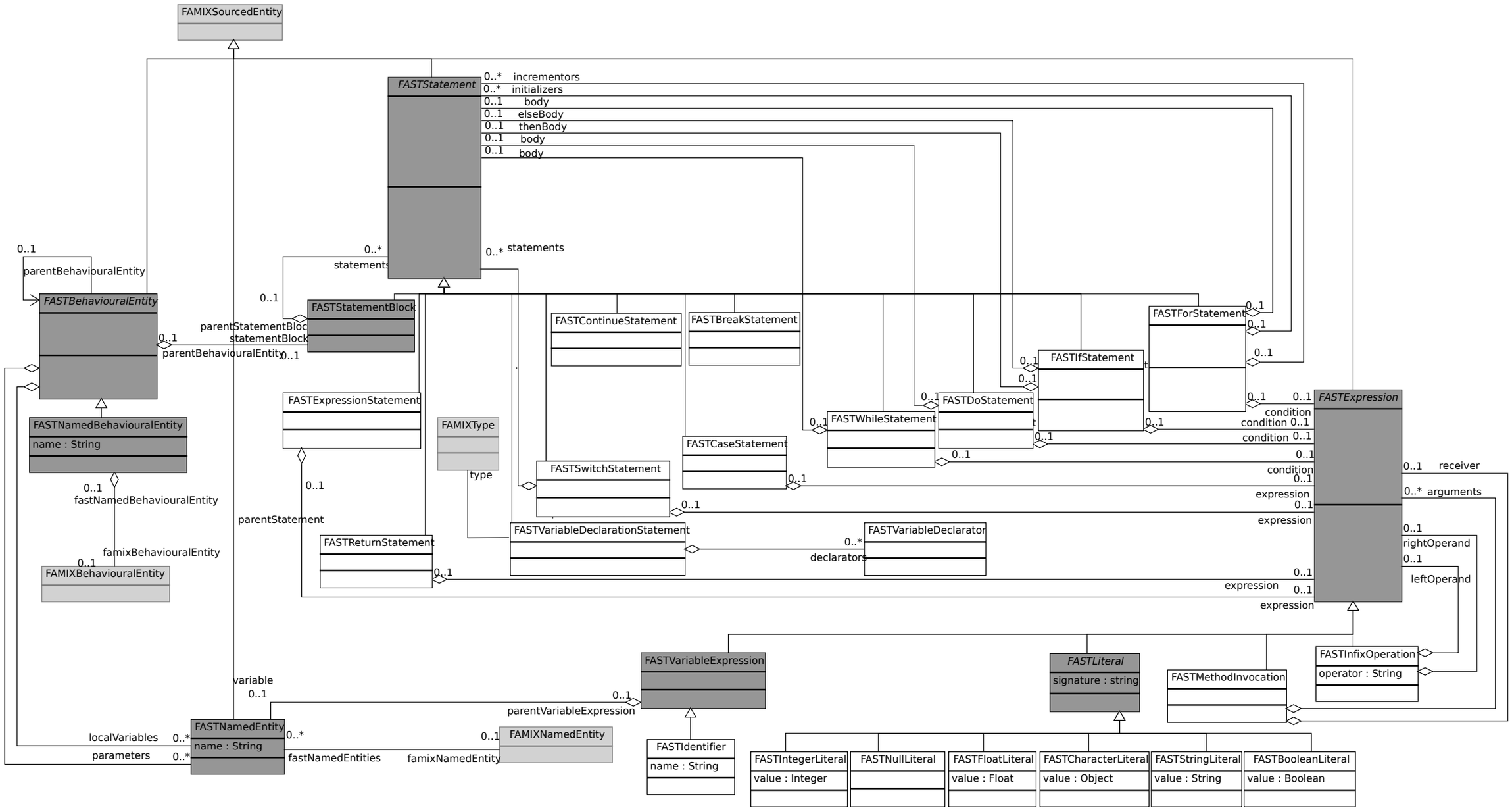
Core metamodel



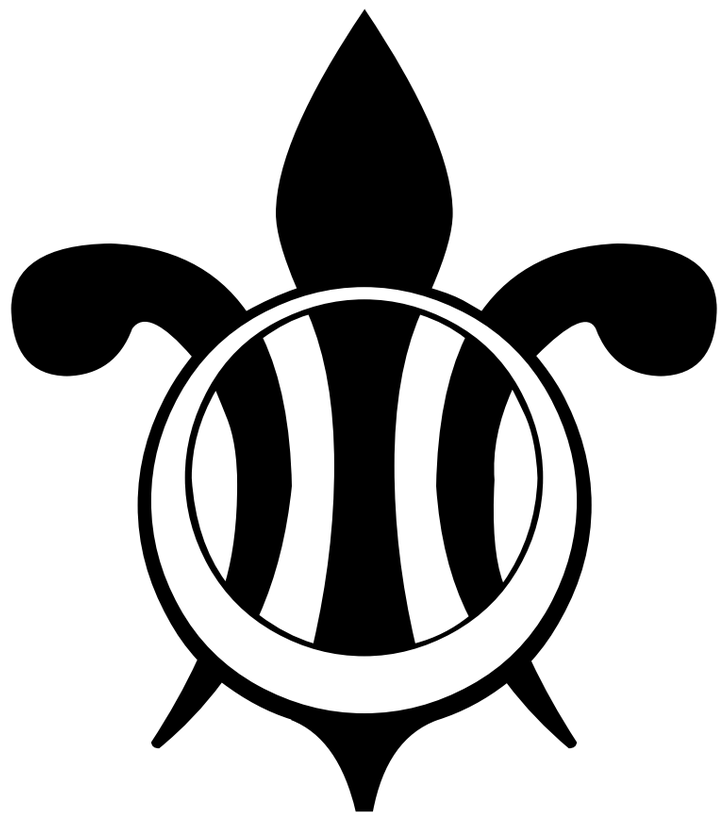
Smalltalk metamodel



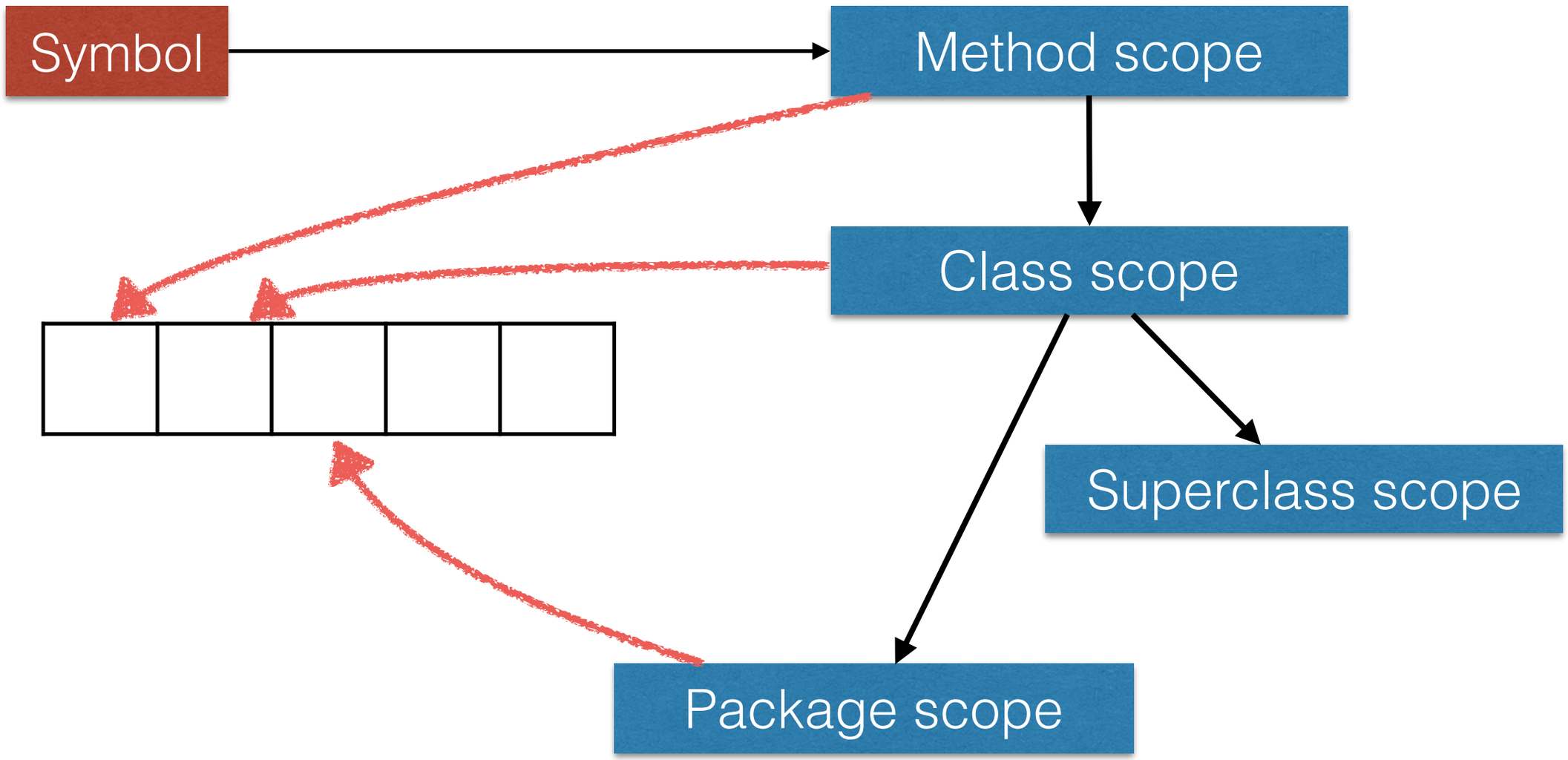
Java (sub)Metamodel



Resolving a symbol



Lookup

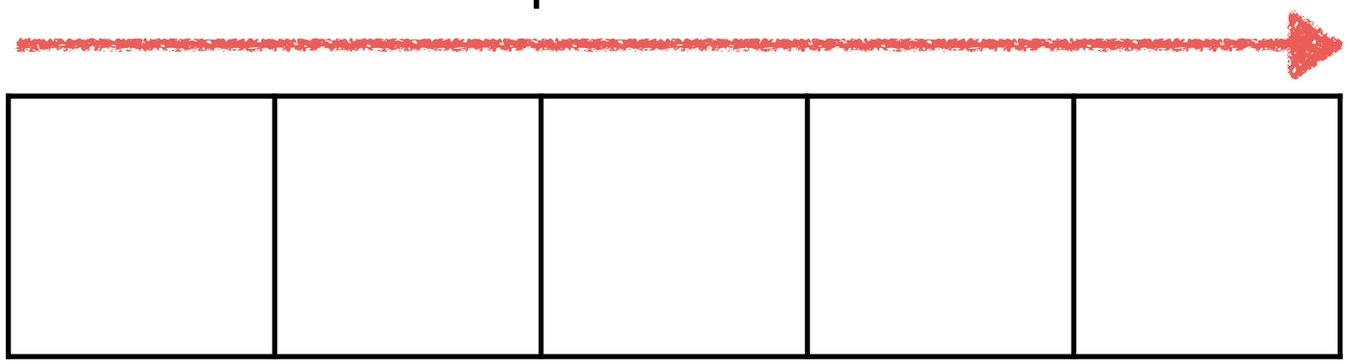

Symbol
Method scope
Class scope
Superclass scope
Package scope

Select

Symbol

private? protected?

public?



Case studies

Pharo (smalltalk)

Java (subset)

Cobol