jdk.dynalink is here!

Attila Szegedi, Fauna Inc.

Part I: Overview

Dynalink in the JDK

- In JDK 9 as `jdk.dynalink` package and module.
- Nashorn uses it.
- Available for integration with other language runtimes and tools.

What Changed?

- Dynalink was presented at conferences before.
- The JDK version is very similar, but the API was simplified and clarified:
  - class hierarchies were flattened
  - some speculative functionality was removed
  - new type safe representation of operations

What is Dynalink?

- A library for dynamic linking of high-level operations on objects ("read a property", "write a property", "invoke a function" etc).
- Useful for implementing programming languages that have:
    - at least some expressions with dynamic types, or
    - object models that differ from the JVM static class model, or
    - type conversions above and beyond the JLS ones.
- Operations are call sites and will be linked to target method handles at run time based on actual types of the evaluated expressions.
    - These can change between invocations, necessitating relinking the call site multiple times for new types.
    - Dynalink handles all that and more.

Why is Dynalink useful?

- When expressions are statically typed and the object model matches that of JVM, you can just use `getfield`, `invokevirtual`, etc. instructions and let the JVM link them.

- Otherwise, use `invokedynamic` and let Dynalink link them.

Nashorn Example

```
public class A {
    public String color;
    public A(String color) { this.color = color; }
}

public class B {
    private String color;
    public B(String color) { this.color = color; }
    public String getColor() { return color; }
}

public class C {
    private int color;
    public C(int color) { this.color = color; }
    public int getColor() { return color; }
}
```

Nashorn Example

```
var objs = [
   new (Java.type("A"))("red"),
   new (Java.type("B"))("green"),
   new (Java.type("C"))(0x0000ff)
]
```

```
for each(var obj in objs) {
     print(obj.color);
 }
```

In bytecode, the `.color` accessor becomes:

```
aload 4 // obj
invokedynamic "color" (Object;)Object; flags(GET_PROPERTY)
```

Operation Encoding

- Dynalink used to prescribe string encoding of operations in call site name (“dyn:getProp:color”). Not anymore.
- It now uses type safe `Operation` objects.
- Decoding the operation from call site is left to the language runtime’s bootstrap method.
  - Nashorn now uses only 3 bits in its bootstrap method’s `int` static argument to represent its operations.

Operation Encoding Example

What used to be:

```
"dyn:getProp|getElem|getMethod:color"
```

is now:

```
new NamedOperation(
  new NamespaceOperation(
    StandardOperation.GET,
    StandardNamespace.PROPERTY,
    StandardNamespace.ELEMENT,
    StandardNamespace.METHOD),
 "color")
```

Operation Encoding Example

What used to be:

```
"dyn:getProp|getElem|getMethod:color"
```

is now with fluent API:

```
StandardOperation.GET
  .withNamespaces(
    StandardNamespace.PROPERTY,
    StandardNamespace.ELEMENT,
    StandardNamespace.METHOD)
  .named("color")
```

Operation Encoding Example

What used to be:

```
"dyn:getProp|getElem|getMethod:color"
```

is now with fluent API and static imports:

```
import static jdk.dynalink.StandardOperation.*;
import static jdk.dynalink.StandardNamespace.*;

  GET
    .withNamespaces(PROPERTY, ELEMENT, METHOD)
    .named("color")
```

Nashorn Operation Encoding

```
public final class NashornCallSiteDescriptor extends CallSiteDescriptor {
    public static final int GET_PROPERTY        = 0; /** Property getter: obj.prop */
    public static final int GET_ELEMENT         = 1; /** Element getter: obj[index] */
    public static final int GET_METHOD_PROPERTY = 2; /** Property getter, invoked: obj.prop() */
    public static final int GET_METHOD_ELEMENT  = 3; /** Element getter, invoked: obj[index]() */
    public static final int SET_PROPERTY        = 4; /** Property setter: obj.prop = value */
    public static final int SET_ELEMENT         = 5; /** Element setter: obj[index] = value */
    public static final int CALL                = 6; /** Call: fn(args...) */
    public static final int NEW                 = 7; /** New: new Constructor(args...) */

    // Correspond to the operation indices above.
    private static final Operation[] OPERATIONS = new Operation[] {
        GET.withNamespaces(PROPERTY, ELEMENT, METHOD),
        GET.withNamespaces(ELEMENT, PROPERTY, METHOD),
        GET.withNamespaces(METHOD, PROPERTY, ELEMENT),
        GET.withNamespaces(METHOD, ELEMENT, PROPERTY),
        SET.withNamespaces(PROPERTY, ELEMENT),
        SET.withNamespaces(ELEMENT, PROPERTY),
        StandardOperation.CALL,
        StandardOperation.NEW
    };
```

Dynalink and Bytecode

- Dynalink does not mandate bytecode-generating language runtimes.

- CallSite objects usually come from bytecode invokedynamic instructions, but can also be arbitrarily created by Java code (so-called “free floating” call sites).

- Dynalink handles CallSite objects no matter where they came from.

- A Dynalink-based interpreter would need to maintain its own CallSite objects associated with its interpreted program representations (e.g. AST).

Simplest Dynalink Use

- Simplest Dynalink use is for a language with:
  - its own dynamic expressions, but
    - no object model of its own, nor
    - type conversions of its own.
- Think something BeanShell-like.

Simplest Dynalink Use

```
import java.lang.invoke.*;
import jdk.dynalink.*;
import jdk.dynalink.support.*;

class MyLanguageRuntime {
  private static final DynamicLinker dynamicLinker =
    new DynamicLinkerFactory().createLinker();

  public static CallSite bootstrap(MethodHandles.Lookup lookup, String name, MethodType type) {
    return dynamicLinker.link(
      new SimpleRelinkableCallSite(
        new CallSiteDescriptor(lookup, decodeOperation(name), type)));
  }

  private static Operation decodeOperation(String name) {
    ...
  }
}
```

Simplest Dynalink Use

```
import java.lang.invoke.*;
import jdk.dynalink.*;
import jdk.dynalink.support.*;

class MyLanguageRuntime {
  private static final DynamicLinker dynamicLinker =
    new DynamicLinkerFactory().createLinker();

  public static CallSite bootstrap(MethodHandles.Lookup lookup, String name, MethodType type) {
    return dynamicLinker.link(
      new SimpleRelinkableCallSite(
        new CallSiteDescriptor(lookup, decodeOperation(name), type)));
  }

  private static Operation decodeOperation(String name) {
    ...
  }
}
```

- Bootstrap methods delegate to a **DynamicLinker**'s **link**() method.
- DynamicLinker is created through a **DynamicLinkerFactory**.

Simplest Dynalink Use

```
import java.lang.invoke.*;
import jdk.dynalink.*;
import jdk.dynalink.support.*;

class MyLanguageRuntime {
  private static final DynamicLinker dynamicLinker =
    new DynamicLinkerFactory().createLinker();

  public static CallSite bootstrap(MethodHandles.Lookup lookup, String name, MethodType type) {
    return dynamicLinker.link(
      new SimpleRelinkableCallSite(
        new CallSiteDescriptor(lookup, decodeOperation(name), type)));
  }

  private static Operation decodeOperation(String name) {
    ...
  }
}
```

- DynamicLinker links CallSite objects that implement Dynalink **RelinkableCallSite** interface.
- DynamicLinker comes with two relinkable call site implementations, **SimpleRelinkableCallSite** and **ChainedCallSite**, both subclasses of j.l.invoke.MutableCallSite. You can also roll your own.

Simplest Dynalink Use

```
import java.lang.invoke.*;
import jdk.dynalink.*;
import jdk.dynalink.support.*;

class MyLanguageRuntime {
  private static final DynamicLinker dynamicLinker =
    new DynamicLinkerFactory().createLinker();

  public static CallSite bootstrap(MethodHandles.Lookup lookup, String name, MethodType type) {
    return dynamicLinker.link(
      new SimpleRelinkableCallSite(
        new CallSiteDescriptor(lookup, decodeOperation(name), type)));
  }

  private static Operation decodeOperation(String name) {
    ...
  }
}
```

- Relinkable call sites carry **CallSiteDescriptor**s, **immutable** triples of Lookup, Dynalink **Operation**, and MethodType.
- Dynalink internally passes descriptors to linkers and never exposes the call sites so their setTarget can't be invoked.
- Language runtimes can subclass CallSiteDescriptor to carry additional static bootstrap arguments.

Simplest Dynalink Use

```
import java.lang.invoke.*;
import jdk.dynalink.*;
import jdk.dynalink.support.*;

class MyLanguageRuntime {
  private static final DynamicLinker dynamicLinker =
    new DynamicLinkerFactory().createLinker();

  public static CallSite bootstrap(MethodHandles.Lookup lookup, String name, MethodType type) {
    return dynamicLinker.link(
      new SimpleRelinkableCallSite(
        new CallSiteDescriptor(lookup, decodeOperation(name), type)));
  }

  private static Operation decodeOperation(String name) {
    ...
  }
}
```

- Dynalink defines a **StandardOperation** enum as well as **NamespaceOperation** and **NamedOperation** classes.

Standard Operations

Expression	Operation	Expected arguments
`obj[x]`	`StandardOperation.GET` `  .withNamespace(StandardNamespace.ELEMENT)`	`(obj, x)`

`obj["foo"]`	`StandardOperation.GET`
	`  .withNamespace(StandardNamespace.ELEMENT)`
	`  .named("foo")`
		`(obj)`

`obj.foo`	`StandardOperation.GET`
	`  .withNamespace(StandardNamespace.PROPERTY)`
	`  .named("foo")`
		`(obj)`

`obj[x] = y`	`StandardOperation.SET` `  .withNamespace(StandardNamespace.ELEMENT)`	`(obj, x, y)`

`obj[“foo"] = y`	`StandardOperation.SET`
	`  .withNamespace(StandardNamespace.ELEMENT)`
	`  .named("foo")`
		`(obj, y)`

`obj.foo = y`	`StandardOperation.SET`
	`  .withNamespace(StandardNamespace.PROPERTY)`
	`  .named("foo")`
		`(obj, y)`

Standard Operations

Expression	Operation	Expected arguments

`obj.fn(x, y)`	1. `StandardOperation.GET`
	`.withNamespace(StandardNamespace.METHOD)`
	`.named("foo")`
	2. `StandardOperation.CALL`
		1. `(obj)`
		2. `(fn, obj, x, y)`

`new Ctor(x, y)`	`StandardOperation.NEW`	`(Ctor, x, y)`

Bring Your Own Linker

- If your language has :
  - its own object model, or
  - its own type conversions
- … then you need to define your own linker.

Bring Your Own Linker

```
import java.lang.invoke.*;
import jdk.dynalink.*;
import jdk.dynalink.support.*;

class MyLanguageRuntime {
  private static final DynamicLinker dynamicLinker;
  static {
    DynamicLinkerFactory factory = new DynamicLinkerFactory();
    factory.setPrioritizedLinker(new MyLanguageLinker());
    dynamicLinker = factory.createLinker();
  }

  public static CallSite bootstrap(MethodHandles.Lookup lookup, String name, MethodType type) {
    return dynamicLinker.link(
      new SimpleRelinkableCallSite(
        new CallSiteDescriptor(lookup, decodeOperation(name), type)));
  }

  private static Operation decodeOperation(String name) {
    ...
  }
}
```

- You need to pass your own linker to the factory.
- You can have more than one. Nashorn has nine!

Linker Examples in Nashorn

- `jdk9/nashorn/samples/dynalink` has some linkers that extend Nashorn's functionality.

- You can just use
  `javac *.java`
  and then
  `jjs any_example.js`
  from the same directory ('cause current dir is on classpath).

- Let's inspect an example linker that adds a ".stream" property to Java arrays.

Linker Examples in Nashorn

```
public final class ArrayStreamLinkerExporter extends GuardingDynamicLinkerExporter {
    public List<GuardingDynamicLinker> get() {
        final ArrayList<GuardingDynamicLinker> linkers = new ArrayList<>();
        linkers.add(new TypeBasedGuardingDynamicLinker() {
            @Override
            public boolean canLinkType(final Class<?> type) {
                return type == Object[].class || type == int[].class ||
                       type == long[].class || type == double[].class;
            }

            @Override
            public GuardedInvocation getGuardedInvocation(LinkRequest request,
                final LinkerServices linkerServices) throws Exception {
                final Object self = request.getReceiver();
                if (self == null || !canLinkType(self.getClass())) {
                    return null;
                }

                final CallSiteDescriptor desc = request.getCallSiteDescriptor();
                final Operation op = desc.getOperation();
                final Object name = NamedOperation.getName(op);
                final boolean getProp = NamespaceOperation.contains(
                        NamedOperation.getBaseOperation(op),
                        StandardOperation.GET, StandardNamespace.PROPERTY);
                if (getProp && "stream".equals(name)) {
                    return new GuardedInvocation(ARRAY_TO_STREAM,
                        Guards.isOfClass(self.getClass(), GUARD_TYPE));
                }

                return null;
            }
        });
        return linkers;
    }

    public static Object arrayToStream(Object array) {
        if (array instanceof int[]) {
            return IntStream.of((int[])array);
        } else if (array instanceof long[]) {
            return LongStream.of((long[])array);
        } else if (array instanceof double[]) {
            . . .
       }
    }

    private static final MethodType GUARD_TYPE = MethodType.methodType(Boolean.TYPE, Object.class);
    private static final MethodHandle ARRAY_TO_STREAM = Lookup.PUBLIC.findStatic(ArrayStreamLinkerExporter.class,
        "arrayToStream", MethodType.methodType(Object.class, Object.class));
}
```

Linker Examples in Nashorn

```
public final class ArrayStreamLinkerExporter extends GuardingDynamicLinkerExporter {
    public List<GuardingDynamicLinker> get() {
        final ArrayList<GuardingDynamicLinker> linkers = new ArrayList<>();
        linkers.add(new TypeBasedGuardingDynamicLinker() {
            @Override
            public boolean canLinkType(final Class<?> type) {
                return type == Object[].class || type == int[].class ||
                       type == long[].class || type == double[].class;
            }

            @Override
            public GuardedInvocation getGuardedInvocation(LinkRequest request,
                final LinkerServices linkerServices) throws Exception {
                final Object self = request.getReceiver();
                if (self == null || !canLinkType(self.getClass())) {
                    return null;
                }

                final CallSiteDescriptor desc = request.getCallSiteDescriptor();
                final Operation op = desc.getOperation();
                final Object name = NamedOperation.getName(op);
                final boolean getProp = NamespaceOperation.contains(
                        NamedOperation.getBaseOperation(op),
                        StandardOperation.GET, StandardNamespace.PROPERTY);
                if (getProp && "stream".equals(name)) {
                    return new GuardedInvocation(ARRAY_TO_STREAM,
                        Guards.isOfClass(self.getClass(), GUARD_TYPE));
                }

                return null;
            }
        });
        return linkers;
    }

    public static Object arrayToStream(Object array) {
        if (array instanceof int[]) {
            return IntStream.of((int[])array);
        } else if (array instanceof long[]) {
            return LongStream.of((long[])array);
        } else if (array instanceof double[]) {
            . . .
       }
    }

    private static final MethodType GUARD_TYPE = MethodType.methodType(Boolean.TYPE, Object.class);
    private static final MethodHandle ARRAY_TO_STREAM = Lookup.PUBLIC.findStatic(ArrayStreamLinkerExporter.class,
        "arrayToStream", MethodType.methodType(Object.class, Object.class));
}
```

- **GuardingDynamicLinkerExporter**s are automatically loaded by Dynalink dynamic linker factories when declared in `META-INF/services`.
- Exporters are Suppliers for a List of **GuardingDynamicLinkers**.
- A **TypeBasedGuardingDynamicLinker** declares it can link operations based on the Java type (class) of the target.
- A guarding dynamic linker receives a **LinkRequest** and produces a **GuardedInvocation**.
- A guarded invocation is a tuple of an invocation MethodHandle for the operation and a guard MethodHandle that defines the condition of validity of the invocation (a class test etc.).

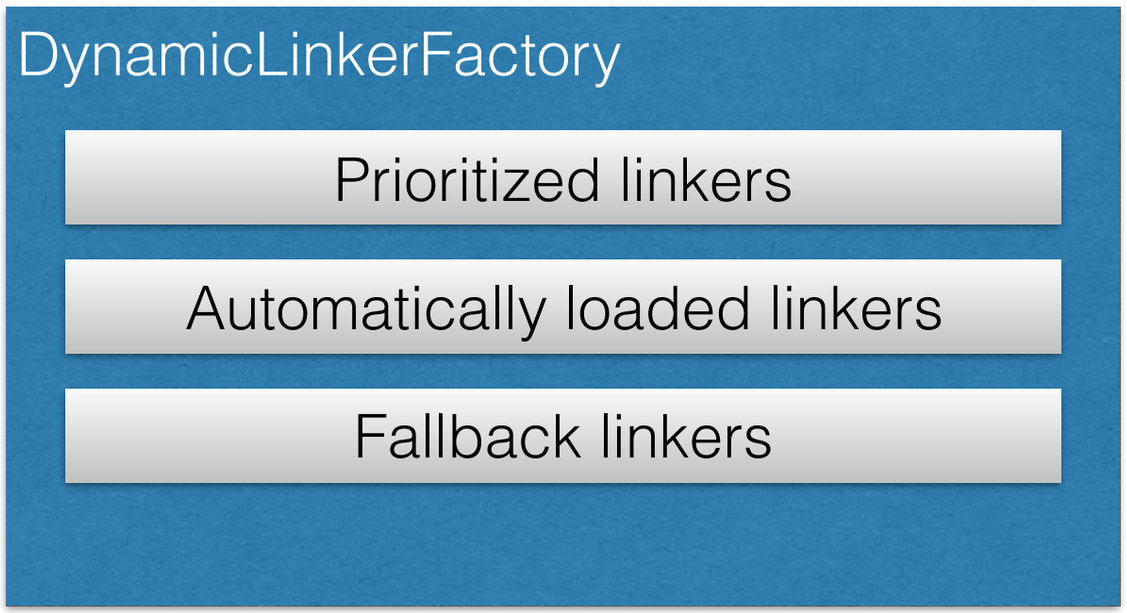
Linker Examples in Nashorn

```
public final class ArrayStreamLinkerExporter extends GuardingDynamicLinkerExporter {
    public List<GuardingDynamicLinker> get() {
        final ArrayList<GuardingDynamicLinker> linkers = new ArrayList<>();
        linkers.add(new TypeBasedGuardingDynamicLinker() {
            @Override
            public boolean canLinkType(final Class<?> type) {
                return type == Object[].class || type == int[].class ||
                       type == long[].class || type == double[].class;
            }

            @Override
            public GuardedInvocation getGuardedInvocation(LinkRequest request,
                final LinkerServices linkerServices) throws Exception {
                final Object self = request.getReceiver();
                if (self == null || !canLinkType(self.getClass())) {
                    return null;
                }

                final CallSiteDescriptor desc = request.getCallSiteDescriptor();
                final Operation op = desc.getOperation();
                final Object name = NamedOperation.getName(op);
                final boolean getProp = NamespaceOperation.contains(
                        NamedOperation.getBaseOperation(op),
                        StandardOperation.GET, StandardNamespace.PROPERTY);
                if (getProp && "stream".equals(name)) {
                    return new GuardedInvocation(ARRAY_TO_STREAM,
                        Guards.isOfClass(self.getClass(), GUARD_TYPE));
                }

                return null;
            }
        });
        return linkers;
    }

    public static Object arrayToStream(Object array) {
        if (array instanceof int[]) {
            return IntStream.of((int[])array);
        } else if (array instanceof long[]) {
            return LongStream.of((long[])array);
        } else if (array instanceof double[]) {
            . . .
       }
    }

    private static final MethodType GUARD_TYPE = MethodType.methodType(Boolean.TYPE, Object.class);
    private static final MethodHandle ARRAY_TO_STREAM = Lookup.PUBLIC.findStatic(ArrayStreamLinkerExporter.class,
        "arrayToStream", MethodType.methodType(Object.class, Object.class));
}
```

- **LinkRequest** gives access to receiver, arguments, and the call site descriptor.
- **NamedOperation** and **NamespaceOperation** have utility methods for common operations.
- **Guards** is a utility class for creating widely useful guard method handles.

What Goes Into a DynamicLinker

- When a factory creates a **DynamicLinker**, it'll compose it from various **GuardingDynamicLinker**s:
  - explicitly set prioritized linkers,
  - any linkers from exporters declared in `META-INF/services`,
  - and explicitly set fallback linkers.
- When no fallback is set, **BeansLinker** is used as a default fallback.



BeansLinker

- Default fallback linker, used to link objects that weren't linked by anything else.
- Provides common sense operation semantics for POJOs:
- `setXxx()`, `getXxx()` and `isXxx()` methods are treated as property setters/getters
- public methods can be invoked
- public fields also act as properties, and so on.

BeansLinker Finer Points

- Supports overloaded method invocation based on argument types. It even takes into account your language runtime allowed type conversions.
- Supports variable arity method invocation.
- Caller sensitive methods are linked in a secure manner.
- Behavior for missing method is pluggable allowing language-specific semantics for handling them on POJOs.

Representation of Statics

- Static methods and fields on classes are represented by objects of class **jdk.dynalink.beans.StaticClass**.

- There's one StaticClass instance for every java.lang.Class object.

- BeansLinker recognizes them and links access to static members and invocation as constructor on them accordingly.

- They're distinct from java.lang.Class; j.l.Class is a runtime class object, StaticClass is a namespace for class' static members as well as a stand-in for its constructors.

    - Just like in Java language you understand the distinction between BitSet and BitSet.class

Dynalink Security

- New `RuntimePermission`s are required for some operations:
  - `"dynalink.exportLinkersAutomatically"` for a `GuardingDynamicLinkerExporter` to load automatically from `META-INF/services`
  - `"dynalink.getLookup"` for invoking `CallSiteDescriptor.getLookup()`

Part II: Advanced Topics

Rhinoceros and Snake

- Our colleague Sundar wrote a demo Jython linker.
- As a result, you can use Jython objects in Nashorn.
  - Or in any other language that uses Dynalink.

Rhinoceros and Snake

```
var JythonFactory = Java.type("org.python.jsr223.PyScriptEngineFactory");
var jythonEngine = new JythonFactory().getScriptEngine();

jythonEngine.eval(<<EOF

import math

class Point:
   def __init__(self, x, y):
       self.x = x
       self.y = y

   def distance(self, other = None):
       if other == None:
           other = Point(0.0, 0.0);
       return math.sqrt((self.x - other.x)**2 + (self.y - other.y)**2)

   def show(self):
       print(self.x, self.y)

   def move(self, dx, dy):
       self.x += dx
       self.y += dy

point = Point(4, 5)
list = ["hello", "world"]

dict = { 'foo' : 'bar' }

EOF);

var p = jythonEngine.get("point")

// fetch attribute "x" and "y"
print("p.x = " + p.x)
print("p.y = " + p.y)

// call distance method on the point
print("p's distance from origin = " + p.distance())

// call other methods on point object
p.show()
p.move(1, 1)
p.show()
```

We're reading Python properties and invoking Python methods from Nashorn!

Rhinoceros and Snake

```
jjs -cp jython_linker.jar:jython-standalone-2.7.0.jar jython_sample.js
```

- What's in the JAR? Few classes and a service manifest.

```
JythonLinkerExporter$1.class
JythonLinkerExporter$2.class
JythonLinkerExporter.class
META-INF/services/
META-INF/services/jdk.dynalink.linker.GuardingDynamicLinkerExporter
```

- There's more stuff in the sample JS file.

Other Dynalink Samples

- Sundar prepared a bunch of samples:
- https://blogs.oracle.com/sundararajan/tags/dynalink

- jdk9/nashorn/samples/dynalink in OpenJDK Mercurial

Other Dynalink Samples

- Sample XML DOM linker allows access to child elements and text using underscore prefix.
- Here's code for generating HTML from a RSS feed in Nashorn with sample DOM linker:

```
function getBooksHtml() {
    var doc = parseXML("http://www.gutenberg.org/cache/epub/feeds/today.rss");
    // wrap document root Element as script convenient object
    var rss = doc.documentElement;

    var str = <<HEAD

<html>
<title>${rss._channel._title._}</title>
<body>
<h1>${rss._channel._description._}</h1>
<p>
Published on ${rss._channel._pubDate._}
</p>

HEAD
. . .
```

Custom Linker Advantages

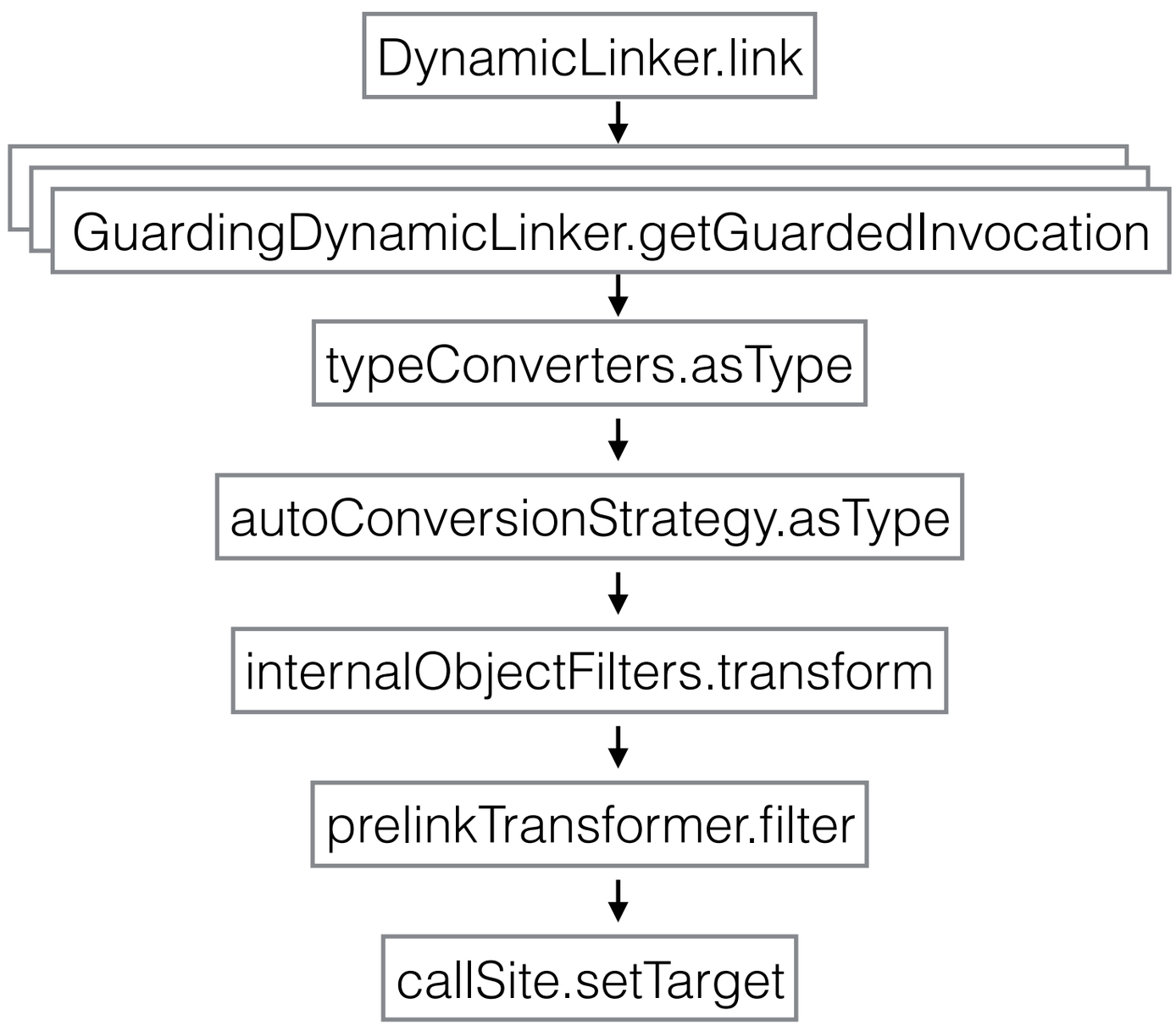
- Great things about custom linkers are:
  - you can effectively create DSLs without the target objects being aware of them; all the logic is in the linker.
  - Efficient: linking is one-time-per-call site cost, after that it's a guard test and direct invoke of a method handle.
  - DOM, JDBC ResultSets, anything goes.

Part III: Even More Advanced Topics

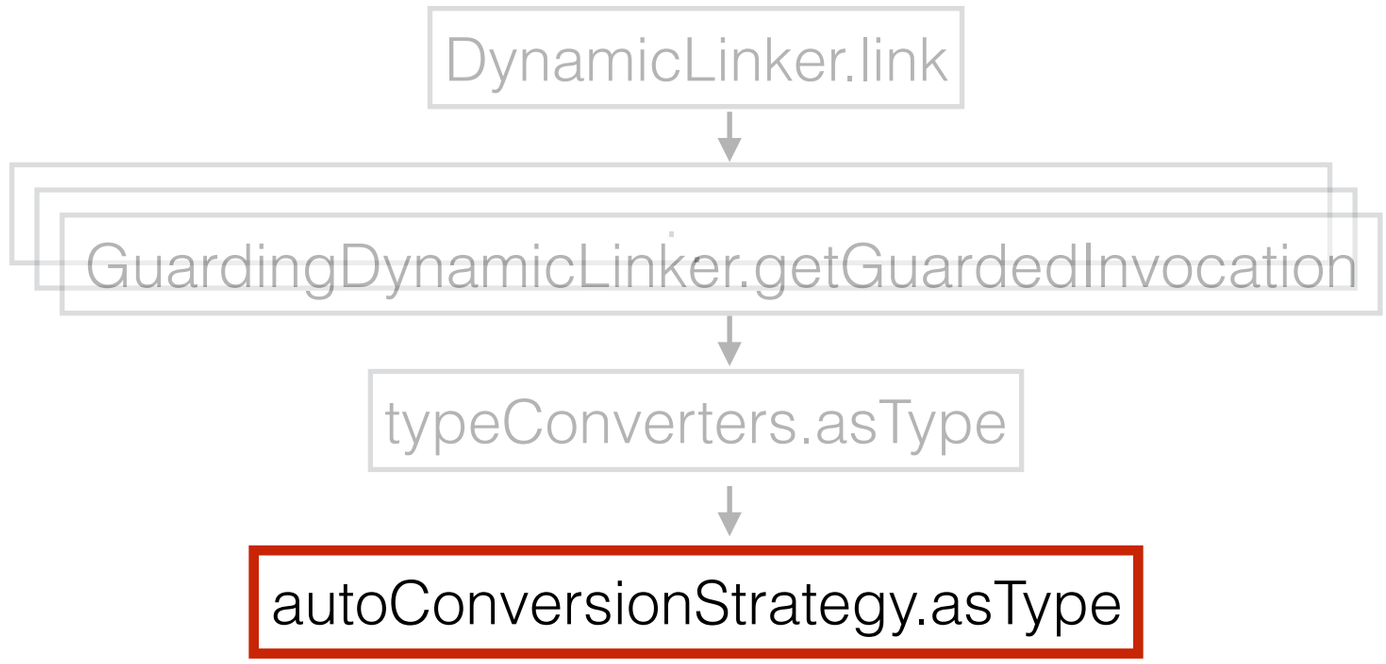
Less Obvious APIs

- DynamicLinkerFactory has methods for some non-obvious behavioral customizations of the DynamicLinker it creates:
  - setAutoConversionStrategy
  - setInternalObjectsFilter
  - setPrelinkTransformer
- What are these oddities for?

Full Linking Flow



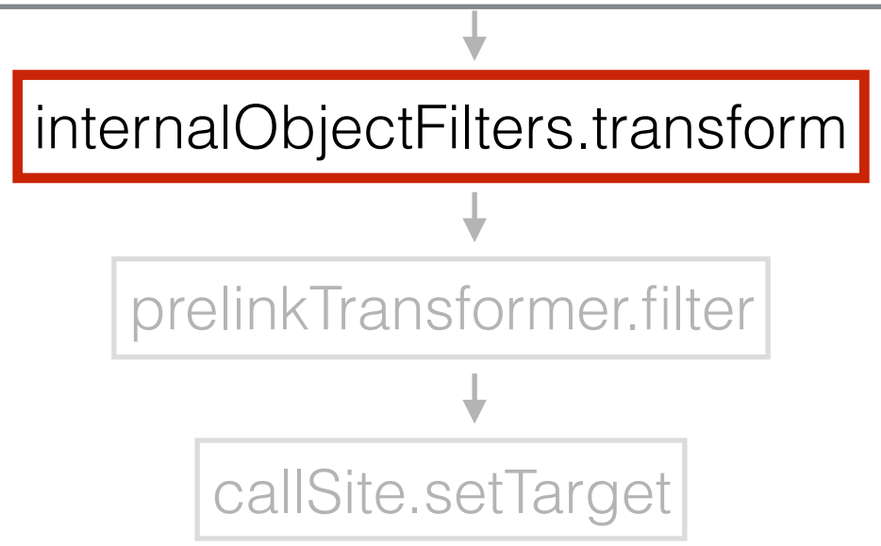
Auto Conversion Strategy



- When call site type and method handle type need to be matched, Dynalink automatically handles all JLS method invocation conversions and all custom type conversions defined by linkers.
- However, some runtimes will need to customize **even** JLS method invocation conversions.
- **Example:** unboxing of null values. To follow JS semantics, Nashorn needs to be able to unbox null to primitive values without a NPE.

Internal Object Filters

- Some language runtime will have internal objects that must not escape the internal scope of the runtime.
- **Example:** Nashorn needs to hide **ConsString** which is a CharSequence of a delayed string concatenations and **ScriptObject** from the outside world.
  - InternalObjectFilters add filters to method handles that collapse ConsString into a j.l.String and wrap ScriptObject in a **JSObject** (public API of Nashorn objects).



Pre-Link Transform

- Pre-link transformer is given the final chance to transform the **GuardedInvocation**'s method handles before they're linked into the call site.
- Nashorn uses a pre-link transformer to add converting filters for optimistic types to method handle return types.
- GuardingDynamicLinkers should refrain from applying non-JLS type conversions on method handles' return types to match them to call site's type and should leave such conversions to the pre-link filter of the runtime that owns the call site.
- That way, a language runtime with optimistic typing can have it work correctly even when linking method handles from a foreign linker into its own call sites.

Resources

- Dynalink is in the JDK 9 builds.
- OpenJDK Mercurial repository has:
  - the source code in jdk9/nashorn/src/jdk.dynalink
  - examples in jdk9/nashorn/samples/dynalink
- JavaDoc:
  http://download.java.net/java/jdk9/docs/jdk/api/dynalink/index.html
- Sundar's Dynalink blog posts:
  https://blogs.oracle.com/sundararajan/tags/dynalink

Questions?