ProjectCoinProposal

AUTHOR(S):

John Rose

OVERVIEW

Create source-code syntaxes for using new JVM features from JSR 292. These are invokedynamic instructions, method handle invocation, certain relaxed conversions, and exotic identifiers.

(Change Notes: Part 4, the proposed relaxed conversion and invocation rules for `Dynamic`, has been moved to a separate proposal. The marker type for dynamic invocations has been renamed to `InvokeDynamic`, and made a non-instantiated class with no public members.)

BACKGROUND:

At the JVM level, an invokedynamic instruction is used to call methods which have linkage and dispatch semantics defined by non-Java languages. Again, a JVM-level invokevirtual instruction has slightly altered linkage rules when the target class method is `java.dyn.MethodHandle.invoke`: The change is that any type signature is acceptable, and the JVM will make a type-safe method call, regardless of the signature. In addition, the JVM already accepts (since Java 5) any of a large set of strings as field, method, and class identifiers, and many languages will use such identifiers beyond the limits of the Java identifier syntax.

FEATURE SUMMARY:

We will make small, localized modifications the Java language to make it easy to with with these JVM features (old and new). This will allow Java code to interoperate with and/or implement libraries in non-Java languages. The changes are as follows:

1. dynamic invocation

The static-only class `java.dyn.InvokeDynamic` may be used with the static call syntax to form an invokedynamic call site. The method name may be any Java identifier (including an exotic one; see point 3). The arguments may be of any number and type. A type argument can optionally supply the return type, in the existing <T> syntax. In effect, `java.dyn.InvokeDynamic` appears to have an infinite number of static methods, of every possible name and signature. More details are given below, but here is a first example:

```
Object x = InvokeDynamic.getMeSomething();
```

(As defined by JSR 292, an invokedynamic call site is linked to a target method under the control of an application-defined bootstrap method. The linkage state is determined by a method handle with the same type descriptor as the call site itself.)

2. method handle invocation

Method handles (class `java.dyn.MethodHandle`) provide the "plumbing" behind any invokedynamic instruction. There are library routines for creating and adapting them, as specified by JSR 292. In addition, it is necessary to provide a way of invoking a method handle as an explicit target, rather than implicitly as the linked target of an `invokedynamic` instruction. Since a method handle invocation (like invokedynamic itself) can have any argument types and return value, method handles also need a special extension for invocation. As for type `InvokeDynamic`, the type `MethodHandle` accepts an infinite variety of non-static calls to the method named `invoke`. The argument and return types for the descriptor presented to the JVM are determined as for invokedynamic calls. Here is a first example:

```
MethodHandle mh = ...;
mh.invoke();
```

3. exotic identifiers

The grammar for Java identifiers is extended to include "exotic identifiers", whose spellings can be any sequence of characters, as long as they avoid certain minor restrictions imposed by the JVM. An exotic identifier is introduced by a hash mark, which is immediately followed by a string literal. No special treatment is given to the identifier, other than ensuring that its spelling contains exactly the character sequence denoted by the string literal. Details are given below; here is an example:

```
int #"strange variable name" = 42;
System.out.println(#"strange variable name");  // prints 42
```

MAJOR ADVANTAGE:

These changes allow full access to invokedynamic and related new JVM features from JSR 292. This allows Java to interoperate with new JVM languages. It also enables Java to serve well as an language implementation or systems programming language.

MAJOR BENEFIT:

Much greater ease of creating, for the JVM, with javac, new programming languages and language runtimes.

(The potential opportunity cost is that language implementors who presently use Java as a systems programming language will be forced to stay down at the bytecode assembly level, making them slower to adopt the JVM for their work.)

MAJOR DISADVANTAGE:

The JLS gets more complicated.

ALTERNATIVES:

The only viable alternative is assembly coding parts of the system which must interoperate with new languages. This will discourage the creation of common runtimes and libraries, and greatly reduce the synergy between languages.

EXAMPLES:

See above and below (in the specification) for one-line examples demonstrating each aspect of the syntax and type rules.

```
void test(MethodHandle mh) {
    mh.invoke("world", 123);
    // previous line generates invokevirtual MethodHandle.invoke(String,int)Object
    InvokeDynamic.greet("hello", "world", 123);
    // previous line generates invokedynamic greet(Object,String,int)Object
    // enclosing class must register a bootstrap method (handle) to link InvokeDynamic.greet
}
```

BEFORE/AFTER

There are no concise before/after examples for these language features per se, because without the new syntax, dynamic language implementors must resort to assembly code.

But, here is a mocked up example that shows how call site caches can be created before and after JSR 292. This is for no particular language; call it MyScript. Note the use of the proposed features to form and manage dynamic call sites.

```
class Foo {
  // compiled method for def ready? = lambda (x) { print "Hello, " + x }
  private static Object method56(Object x) {
    System.out.println("Hello, "+x);
    return null;
  }
  // function pointer, old style:
  public static Method1 bind_ready__63() { /*ready?*/
    return new Method1() {
      // there is a new classfile per expression reference
      public Object apply(Object arg) { return method56(arg); }
    }
  }
  // function pointer, new style:
  public static MethodHandle #"bind:ready?"() {
    // it all happens in one classfile
    return MethodHandles.findStatic(Foo.class, "method56",
                MethodTypes.makeGeneric(1));
  }
  // Note: the language runtime uses Java reflection to help it link.
}
```

```
class Bar {
  // compiled method for lambda (x) { x . ready? }
  // call-site cache, old style:
  private static Method1 csc42 = null;
  private static Object method2(Object x) {
    Method1 tem = csc42;
    // complex machinery with little hope of optimization
    if (tem == null)
      csc42 = tem = MOP.resolveCallSite(Foo.class, "ready?", x);
    return tem.apply(x);
  }
  // call-site cache, new style:
  private static Object method2(Object x) {
    // native to the JVM and the JIT
    return InvokeDynamic.#"myscript:ready?"(x);
  }
  static { Linkage.registerBootstrapMethod("bootstrapInvokeDynamic"); }
  // Note:  The Linkage API is subject to change.
  private static Object bootstrapInvokeDynamic(java.dyn.CallSite site, Object... args) {
    return MOP.executeCallSite(site, args);
  }
}
```

```
class MOP {
  // shared logic for resolving call sites
  public static executeCallSite(java.dyn.CallSite site, Object... args) {
    MethodHandle target = site.target();
    if (target == null) {
      target = resolveCallSite(site.callerClass(), site.name(), args);
      site.setTarget(target);  // next time it will be a direct call
    }
    return MethodHandles.invoke(target, args);
  }
}
```

SIMPLE EXAMPLE:

This example greets the world using (a) normal static linkage, (b) direct method handle invocation, and (c) a lazily linked call site (invokedynamic). The output from the "bootstrap" routine appears only once, after which the linked call site runs by directly calling the target method, with no reflection.

```
import java.dyn.*;

public class Hello {
    public static void main(String... av) {
        if (av.length == 0)  av = new String[] { "world" };
        greeter(av[0] + " (from a statically linked call site)");
        for (String whom : av) {
            greeter.<void>invoke(whom);  // strongly typed direct call
            // previous line generates invokevirtual MethodHandle.invoke(String)void
            InvokeDynamic.hail(whom);      // weakly typed invokedynamic
            // previous line generates invokedynamic MethodHandle.invoke(String)Object
        }
    }

    static void greeter(String x) { System.out.println("Hello, "+x); }
    // intentionally pun between the method and its reified handle:
    static MethodHandle greeter
        = MethodHandles.findStatic(Hello.class, "greeter",
                                   MethodType.make(void.class, String.class));

    // Set up a class-local bootstrap method.
    static { Linkage.registerBootstrapMethod("bootstrapInvokeDynamic"); }
    // Note:  The Linkage API is subject to change.
    private static Object bootstrapInvokeDynamic(CallSite site, Object... args) {
        assert(args.length == 1 && site.name() == "hail");  // in lieu of MOP
        System.out.println("set target to adapt "+greeter);
        MethodHandle target = MethodHandles.convertArguments(greeter, site.type());
        site.setTarget(target);
        System.out.println("calling the slow path; this should be the last time!");
        return MethodHandles.invoke(target, args);
    }
}
```

ADVANCED EXAMPLE:

(See before-and-after MOP example above.)

DETAILS

SPECIFICATION:

1.1 The type `java.dyn.InvokeDynamic` shall be defined as an empty class with no supertypes (other than Object) and no constructors (other than a private one, as required by the language). This static-only pattern is like that of `java.util.Collections`, except that there are no statically defined methods at all. It is an error if `InvokeDynamic` is defined as an interface, or if it has any supertypes, or if it has any explicit member definitions (other than a private constructor). Although it has the usual implicit supertype of `Object`, it inherits no static methods from its supertype (which, indeed, has none to bequeath).

```
package java.dyn;
public class InvokeDynamic /*must be empty*/ {
  private InvokeDynamic() { }
  /*must be empty except for private constructor*/
}
```

(Note that `InvokeDynamic` is not useful as a reference type. Like `java.util.Collections`, it is a non-instantiated class.)

1.2 The type name `InvokeDynamic` may be qualified with any method name whatever, and invoked on any number and type of arguments. The call always resolves to an implicitly defined static method. Instead of an `invokestatic` call, however, the compiler generates an `invokedynamic` call site with the given name and a descriptor (symbolic type signature) derived from the erasure of the static types of all the arguments. In this way, an `invokedyn amic` instruction can be written in Java to use any of the full range of calling sequences (i.e., descriptors) supported by the JVM. Neither the JVM instruction nor the Java syntax is limited in its use of argument types.

```
InvokeDynamic.anyNameWhatever();   // no argument types
InvokeDynamic.anotherName("foo", 42);  // argument types (String, int)
```

1.3 Any call to a static method in `InvokeDynamic` accepts an optional type parameter which specifies the return type of the call site's descriptor. The type parameter may any type whatever, including void or a primitive type. If it is omitted it defaults to the type `Object` itself.

```
Object x = InvokeDynamic.myGetCurrentThing();  // type () -> Object
InvokeDynamic.<void>myPutCurrentThing(x);  // type (Object) -> void
int y = InvokeDynamic.<int>myHashCode(x);  // type (Object) -> int
boolean z = InvokeDynamic.<boolean>myEquals(x, y);  // type (Object, int) -> boolean
```

(Rationale: Although it is strange for non-references to appear in the syntactic position of type arguments, this design is far simpler than inventing a completely new syntax for specifying the return type of the call site, as some early prototypes did.)

1.4 For the purposes of determining the descriptor of the `invokedynamic` instruction, null arguments (unless casted to some other type) are deemed to be of reference type `java.lang.Null`, if this type exists. The type `java.lang.Null` will appear only to the bootstrap method, and will serve notice that the call site contains an untyped null reference, rather than an explicitly typed reference.

```
InvokeDynamic.myPrintLine(null);  // type (Null) -> Object
InvokeDynamic.<void>foo((String)null, null);  // type (String, Null) -> void
```

1.5 As the JVM executes, checked exceptions may be produced by any `invokedynamic` call. However, there is (currently) no way to statically infer which exceptions may be thrown at any given call site. Therefore, it is valid both to catch and not to catch checked exceptions at any dynamic call site.

```
InvokeDynamic.write(out, "foo");  // might throw a checked exception
try { InvokeDynamic.foo(); } catch (IOException ee) { }  // must be accepted
try { "foo".hashCode(); } catch (IOException ee) { }  // a compile-time error
```

(Note: This means that `invokedynamic` calls are a potential loophole for checked exceptions. As Java programs are already constructed to cope with unchecked exceptions and error, the possibility of a missed catch of a checked exception is not considered to be a hazard. In fact, dynamic languages cannot be supported without some such relaxation of static exception checking.)

2.1 The class java.dyn.MethodHandle shall be defined (external to this specification) without any method named `invoke`. It is an error if it or any of its supertypes define a method named `invoke`. (In any case, such supertypes are a fixed part of the Java APIs and/or implementations. JSR 292 happens to define a method named "type" on method handles.)

```
package java.dyn;
public class MethodHandle { ... }
```

2.2 Any reference of type MethodHandle may be qualified with the method name `invoke` and invoked on any number and type of arguments. Only the method named `invoke` is treated this new way. All other expressions involving MethodHandle are unchanged in meaning, including selection of members other than `invoke`, casting, and the `instanceof` operator.

```
MethodHandle mh = ...;
mh.invoke("foo", 42);  // argument types (String, int)
MethodType mtype = mh.type();  // no new rules here; see JSR 292 javadocs
mh.neverBeforeSeenName();  // no new rules; must raise an error
```

In effect, java.dyn.MethodHandle appears to have an infinite number of non-static methods named `invoke`, of every possible signature.

(In fact, JSR 292 specifies that each individual method handle has a unique type signature, and may be invoked only under that specific type. This type is checked on every method handle call. JSR 292 guarantees runtime type safety by requiring that an exception be thrown if a method handle caller and callee do not agree exactly on the argument and return types. The details of this check are not part of this specification, but rather of the MethodHandle API.)

2.3 A method handle call on `invoke` accepts an optional type parameter to specify the return type. As with `InvokeDynamic`, the type parameter to `MethodHandle.invoke` may be any type, including void or a primitive type. If it is omitted it defaults to `Object`.

```
MethodHandle mh1, mh2, mh3, mh4; ...
Object x = mh1.invoke();  // type () -> Object
mh2.<void>invoke(x);  // type (Object) -> void
int y = mh3.<int>invoke(x);  // type (Object) -> int
boolean z = mh4.<boolean>invoke(x, y);  // type (Object, int) -> boolean
```

2.4 As with `InvokeDynamic`, otherwise untyped null argument values are treated as if they were of type `Void`.

```
mh1.invoke(null);  // type (Void) -> Object
mh2.<void>invoke((String)null, null);  // type (String, Void) -> void
```

2.5 As the JVM executes, checked exceptions may be produced by any virtual call. However, there is (currently) no way to statically infer which exceptions may be thrown at any given method handle invocation. Therefore, it is valid both to catch and not to catch checked exceptions at any method handle invocation.

```
writeMH.invoke(out, "foo");  // might throw a checked exception
try { mh.invoke(); } catch (IOException ee) { }  // must be accepted
try { "foo".hashCode(); } catch (IOException ee) { }  // a compile-time error
```

(Note: As with `InvokeDynamic`, such method handle invocations are a potential loophole for checked exceptions. The same comments apply in this case as in 1.5 above.)

2.6 As usual, if a null value typed as a method handle is qualified with the method name `invoke`, the expression must terminate abnormally with a `NullPointerException`.

```
MethodHandle nmh = null;
nmh.invoke();       // must produce a NullPointerException
```

2.7 If a class extends `MethodHandle`, it does not inherit any of the implicitly defined `invoke` methods. To invoke a subclass of a method handle, it must first be cast to `MethodHandle` per se. (Non-inheritance of `invoke` methods prevents `MethodHandle` from disturbing static scoping of any type other than `MethodHandle` itself. Note that public method handle subclasses are not necessarily a feature of JSR 292.)

```
class MySpecialMethodHandle extends MethodHandle { ... }
MySpecialMethodHandle dmh = mh;
mh.invoke<void>();  // must be rejected: special MH.invoke not visible here
```

2.8 The bytecode emitted for any call to `MethodHandle.invoke` is an `invokevirtual` instruction, exactly as if a public virtual method of the desired descriptor were already present in `java.dyn.MethodHandle`.

```
mh.<void>invoke(1);  // produces an invokevirtual instruction
class MethodHandle { ... public abstract void invoke(int x); ... }  // hypothetical overloading of 'invoke'
mh.invoke(1);        // would produce an identical invokevirtual, if that overloading could exist
```

3.1 The two-character sequence '#' "" (hash and string-quote, or ASCII code points 35 and 24) introduces a new token similar in structure to a Java string literal. The token is in fact an identifier (JLS 3.8), which may be used for all the same syntactic purposes as ordinary identifiers are used for.

```
int #"strange variable name" = 42;
System.out.println(#"strange variable name");  // prints 42
```

This is true whether or not the characters are alphanumeric, or whether they happen (when unquoted) to spell any Java Java keyword or token.

```
int #"+", #"\\", #"42" = 24;
System.out.println(#"42" * 100);  // prints 2400
```

```
// another take on java.lang.Integer:
class #"int" extends Number {
  final int #"int";
  #"int"(int #"int") { this.#"int" = #"int"; }
  static #"int" valueOf(int #"int") { return new #"int"(#"int"); }
  public int intValue() { return #"int"; }
  public long longValue() { return #"int"; }
  public float floatValue() { return #"int"; }
  public double doubleValue() { return #"int"; }
  public String toString() { return String.valueOf(#"int"); }
}
```

3.2 The spelling of the identifier is obtained by collecting all the characters between the string quotes. Every string escape sequence (JLS 3.10.6) is replaced by the characters they refer to. As with other tokens, this character collection occurs after Unicode escape replacement is complete (JLS 3.3).

```
int #"\'\t" = 5;  // a two-character identifier
System.out.println(#"'\u0009");  // prints 5
```

3.3 In particular, if the resulting sequence of characters happens to be a previously valid Java identifier, both normal and exotic forms of the same identifier token may be freely mixed.

```
int #"num" = 42, scale = 100;
System.out.println(num * #"scale");  // prints 4200
```

3.4 An exotic identifier may not be empty. That is, there must be at least one character between the opening and closing quotes.

```
int #"";  // must be rejected
```

3.5 Certain characters are treated specially within exotic identifiers even though they are not specially treated in string or character literals. The following so-called "dangerous characters" are illegal in an exotic identifier unless preceded by a backslash: **`/ . ; < > [ ]`**. If a dangerous character is preceded by a backslash, the backslash is elided and the character is collected anyway. Depending on the ultimate use of the identifier, the program may be eventually rejected with an error. This must happen if and only if the escaped character would otherwise participate in a bytecode name forbidden by the Java 5 JVM specification.

```
class #"foo/Bar" { }  // not a package qualifier, must be rejected
class #"foo.Bar" { }  // not a package qualifier, must be rejected
x.#"<init>"();        // not a method call; must be rejected
x.#"f(Ljava/lang/Long;)"(0);  // not a method descriptor; must be rejected
```

3.5.1 Specifically, the compiler must reject a program containing an exotic identifier with an escaped dangerous character happen if any of these is true: (a) the identifier is used as part or all of the bytecode name of a class or interface, and it contains any of **`/ . ; [`**, or (b) the identifier is used as a part or all of the bytecode name of a method, and it contains any of **`/ . ; < >`**, or (c) the identifier is used as a part or all of the bytecode name of a field, and it contains any of **`/ . ;`**. Note that close bracket **`]`** will always pass through; it is included in these rules simply for symmetry with open bracket **`[`**.

```
class #"java/io"  { }  // must be rejected
class #"java\/io" { }  // must be rejected (perhaps in an assembly phase)

class #"<foo>"    { }  // must be rejected
class #"\<foo\>"  { }  // legal (but probably a bad idea)

void f() { int #"&yen;"  = '\u00A5'; }  // must be rejected
void f() { int #"&yen\;" = '\u00A5'; }  // legal (but probably a bad idea)

class #"]"  { int #"]";  void #"]"()  {} }  // must be rejected
class #"\]" { int #"\]"; void #"\]"() {} }  // legal (but probably a bad idea)
```

These rules support the need for avoiding dangerous characters as a general rule, while permitting occasional expert use of names known to be legal to the JVM. However, there is no provision for uttering the method names "<init>" or "<clinit>". Nor may package prefixes ever be encoded within exotic identifiers.

3.6 Any ASCII punctuation character not otherwise affected by these rules may serve as a so-called "exotic escape character". That is, it may be preceded by a backslash; in this case both it and the backslash is collected (as a pair of characters) into the exotic identifier. Specifically, these characters are {{*! # $ % & ( ) * + , - : = ? @ ^ _ `

Unknown macro: { | }

~*}} and no others.

```
int #"="   = 42;
int #"\="  = 99;
System.out.println(#"=");  // must print 42 not 99
```

These escapes are passed through to the bytecode level for further use by reflective applications, such as a bootstrap linker for `invokedynamic`. Such escapes are necessary at the level of bytecode names in order to encode (mangle) the dangerous characters. By sending both the backslash and the exotic escape character through to the bytecode level, we avoid the problem of multiple escaping (as is seen, for example, with regexp packages).

Although Java has not worked this way in the past, the need for multiple phases of escaping motivates it here and now. Compare this quoting behavior with that of the Unix shells, which perform delayed escaping for similar reasons:

```
$ echo "test: \$NL = '\12'"
test: $NL = '\12'
```

(See http://blogs.oracle.com/jrose/entry/symbolic_freedom_in_the_vm for a proposal that manages bytecode-level mangling of exotic names. This proposal is independent of the present specification.)

3.7 Here is the combined grammar for exotic identifiers, starting with a new clause for Identifier (JLS 3.8):

Identifier: ...
ExoticIdentifier
ExoticIdentifier: # " ExoticIdentifierCharacters "
ExoticIdentifierCharacters:
ExoticIdentifierCharacter ExoticIdentifierCharacters
ExoticIdentifierCharacter:
StringCharacter but not DangerousCharacter
\ DangerousCharacter /* the backslash is elided and the character is collected */
\ ExoticEscapeChar /* both the backslash and the character are collected */
DangerousCharacter: one of / . ; < > [ ]
ExoticEscapeChar: one of ! # $ % & ( ) * + , - : = ? @ ^ _ `

~

```
*3.8* This construct does not conflict with any other existing or proposed use of the hash character.  In
particular, if the hash character were to be defined as a new sort of Java operator, it would not conflict with
this specification.  Even if it were to be a construct which could validly be followed by a normal Java string
literal, any ambiguity between the constructs could be resolved in favor of the operator by inserting
whitespace between the hash and the opening quote of the string literal.

*3.9* Exotic identifiers are occasionally useful for creating dynamically linkable classes or methods whose
names are determined by naming scheme external to Java.  (They may also be used for occasionally avoiding Java
keywords, although a leading underscore will usually do just as well.)  They are most crucially useful for
forming {{invokedynamic}} calls, when the method name must refer to an entity in another language, or must
contain structured information relevant to a metaobject protocol.
```

package my.xml.tags;
class #"\<pre\>"

Unknown macro: { ... }

package my.sql.bindings;
interface Document

Unknown macro: { String title(); Text #"abstract"(); int #"class"(); ... }

Object mySchemeVector = ...;
Object x = InvokeDynamic.#"scheme:vector-ref"(mySchemeVector, 42);

```
h3. COMPILATION:

See JSR 292 for the specification of {{invokedynamic}} instructions.  In brief, they begin with a new opcode, a
{{CONSTANT_NameAndType}} index, and end with two required zero bytes.  Method handle invocation is just an
ordinary invokevirtual instruction, whose class is {{java.dyn.MethodHandle}}, whose name is {{invoke}}, and
whose descriptor is completely arbitrary; this requires no special compilation support beyond putting a
loophole in the method lookup logic.  Exotic identifiers require no compilation support beyond the lexer.
(This assumes Unicode-clean symbol tables all the way to the backend.)  There must be a final validity check in
the class file assembler; this can (and should) be copied from the JVM specification.

h3. TESTING:

Testing will be done the usual way, via unit tests exercising a broad variety of signatures and name spellings,
and by early access customers experimenting with the new facilities.

h3. LIBRARY SUPPORT:

The JVM-level behavior of the type {{java.dyn.MethodHandle}} is defined by JSR 292.  Its language integration
should be defined by an expert group with language expertise.

JSR 292 per se involves extensive libraries for the functionality it defines, but they are not prerequisites to
the features specified here.  Other than exotic identifiers, the features described here have no impact except
when the {{java.dyn}} types exist in the compilation environment.

h3. REFLECTIVE APIS:

The method {{java.lang.Class.getDeclaredMethod}} must be special-cased to always succeed for {{MethodHandle.
invoke}} and for {{InvokeDynamic}} (any method name), regardless of signature.  The JSR 292 JVM has such logic
already, but it must be exposed out through the Class API.

Only single-result reflection lookups need to be changed.  Multiple-method lookups should *not* produce
implicitly defined methods.

The javax.lang.model API (which is used internally by javac) does not need specialization, because the
implicitly defined methods of {{MethodHandle}} and {{InvokeDynamic}} do not ever need to mix with other more
normal methods.  The static (compile-time) model of {{InvokeDynamic}} may not present any enclosed elements,
while that of {{MethodHandle}} must not present any methods named {{invoke}}.

h3. OTHER CHANGES:

Javap needs to disassemble {{invokedynamic}} instructions.

Javap needs to be robust about unusual identifier spellings.  (It already is, mostly.)

There may be bugs in some implementations of javac when processing identifiers not previously seen.  For
example, javac should have Unicode-clean symbol tables all the way to the backend.  As another example, some
spellings (mis-)used internally, like {{*\#"*{}}}{{"}}\*, could cause bugs in some implementations.  For
example:
```

import pkg1.**; // may accidentally import only pkg1.#""**, not pkg1.#"+"
import pkg2.#"*"; // may accidentally import pkg2.#"+"

MIGRATION:

The feature is for new code only.

These language features, along with the related JVM extensions, will make it possible for dynamic language implementations (a) to continue to be coded in Java, but (b) to avoid the performance and complexity overhead of the Core Reflection API.

COMPATIBILITY

BREAKING CHANGES:

None. All changes are associated with previously unused types and/or syntaxes.

EXISTING PROGRAMS:

No special interaction. In earlier class files 186, the code point used by `invokedynamic`, is an illegal opcode, and `java.dyn.InvokeDynamic` and `java.dyn.MethodHandle` are previously unused type names.

REFERENCES

EXISTING BUGS:

- 6754038: writing libraries in Java for non-Java languages requires method handle invocation
- 6746458: writing libraries in Java for non-Java languages requires support for exotic identifiers

URL FOR PROTOTYPE:

General:

- http://hg.openjdk.java.net/mlvm/mlvm/langtools
- http://hg.openjdk.java.net/mlvm/mlvm/langtools/file/tip/nb-javac/

Invokedynamic and method handles:

- http://hg.openjdk.java.net/mlvm/mlvm/langtools/file/tip/meth.txt
- http://hg.openjdk.java.net/mlvm/mlvm/langtools/file/tip/meth.patch

Exotic identifiers:

- http://hg.openjdk.java.net/mlvm/mlvm/langtools/file/tip/quid.txt
- http://hg.openjdk.java.net/mlvm/mlvm/langtools/file/tip/quid.patch

FAQ

(This is not a part of the specification. It captures some reviewer comments and responses.)

Q: Why is this proposal in multiple parts? (Rémi Forax)

A: There are three separable aspects to dynamic language support. The first is simply forming the new kind of dynamic invocation from Java code, while the second is the closely aligned need to form invocations on the composable units of dynamic invocation behavior (method handles). The third aspect is the need to use and define names which are native to languages other than Java. Thesepurposes overlap and synergize in the formation of invokedynamic instructions, because these are likely to use the method name to transmit critical information to the runtime linkage software (typically a metaobject protocol), and because the user code which handles invokedynamic calls is likely to form direct (non-dynamically linked) calls to method handles.

Q: Where did the the interface `Dynamic` go?

An earlier version of this proposal contained a dynamic wildcard type (interface `Dynamic`) also useful for composing dynamic invocation sites. This has been moved into its own separate proposal.

Q: Why is null being inferred as Null instead the compiler raising an error? Let the user cast it to Object, etc. (Rémi Forax)

A: I went back and forth on this point while I was working with the code. At first (a) null was implicitly Object, then (b) it caused an error (as you suggest), then (c) it used a marker type Void. The most correct thing is (d) to use Neal Gafter's marker type Null, so this spec. will interoperate with that type, when it is added. Staying with case (b) ruins the use-case of simulating Java call sites, since null is fundamentally different from any other type; therefore it needs to be reified somehow.

Q: Why didn't you choose to allow catching checked exceptions from InvokeDynamic method calls? (Josh Suereth)

A: That's a very good point. In fact, invokedynamic (like any other invoke instruction) can throw any kind of exception at the JVM level. Since at the Java level it is untyped and therefore not statically checked, it must be possible, though not required, to catch checked exceptions. (Incorporated above in 1.5 and 2.5.)