InterfaceDynamic

AUTHOR(S):
John Rose

OVERVIEW
Create source-code syntaxes for using new JVM features from JSR 292. These are in three parts, (1) method handle invocation, (2) invokedynamic instructions, and (3) exotic (non-Java) identifiers. Those JVM features are defined, as bytecode constructs, elsewhere as part of the JVM specification (and specifically JSR 292).

(Note: Only part (1) of this specification is scheduled for JDK 7.)

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 is java.dyn.MethodHandle and the method is marked as sig nature-polymorphic: 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. method handle invocation
Method handles (class java.dyn.MethodHandle) provide “plumbing” for connecting bits of functional behavior in a Java program, especially in support of the invokedynamic instruction. There are library routines for creating and adapting them, as specified by JSR 292. It is necessary to provide a way to invoke a method handle as an explicit target from Java code.

Since a method handle invocation can have any argument types and return value, method handles also need special treatment for invocation. The type MethodHandle contains signature-polymorphic methods of certain fixed names. A signature-polymorphic method accepts any number and type of arguments, and (with an optional cast) returns any desired result type. Moreover, unlike a normal varargs method, the argument and return types are represented exactly in the linkage information associated with invocation.

Here is a first example:

```java
MethodHandle mh = ...;
Object x = mh.invokeExact();  // type () -> Object, not (Object[]) -> Object
```

(Note: The names of the signature-polymorphic methods, as defined by JSR 292, are invokeExact and invokeGeneric, but that is not a normative part of this specification. They will appear in the examples. Even the choice of name MethodHandle is not normative to this specification, but rather to the JSR 292 API. The marking of signature-polymorphic names is managed via an annotation, as will be seen below.)

2. dynamic invocation
The non-instantiated 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. The return type is determined by a context-dependent target typing convention, as described below.

In effect, java.dyn.InvokeDynamic appears to have an infinite number of static signature-polymorphic methods, of every possible name. More details are given below, but here is a first example:

```java
Object x = InvokeDynamic.getMeSomething();  // type (Object) -> Object
```

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. The details of the bootstrap protocol are irrelevant to language support, except that there is a syntax for specifying bootstrap methods (section 1.6).

(Note: The choice of name InvokeDynamic is not normative to this specification, but rather to the JSR 292 API. This will be clarified in the specification below. But the name will continue to appear in examples.)

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:

```java
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.

*Comment added 4/2010:* There are many degrees of syntactic freedom for designing this new type of expression. Here are some of the alternatives. The first is the present proposal, and the others are roads not taken:

```
/*A1*/ (Object) InvokeDynamic.greet("hello", "world", 123); //recommended
/*A2*/ InvokeDynamic.<Object>greet("hello", "world", 123); //JSR 292 EDR
/*B*/ InvokeDynamic.invoke("greet",
    new Class<?>[] { Object.class, String.class, String.Class, int.class },
    new Object[] { "hello", "world", 123 }); //reflective formulation
/*C*/ $invokedynamic Object greet("hello", "world", 123); //new keyword
/*D*/ $invokedynamic Object greet("hello", "world", 123); //explicit signature
/*E*/ $asm { push "hello"; push "world"; push 123; invokedynamic greet:Object(String,String,int); pop result; }
//asm statement
```

Cases A1 and A2 use static types and names to fill in the required name and signature information for the bytecode. Case B is a different extreme: Every natural degree of freedom is expressed by a reflective expression. Case C uses a keyword rather than a "magic class", though there is ample precedent in the Java language for special treatment of known magic classes (like Integer or String).

It is not clear with the reflective formulations what limitations (if any) are to be imposed on the reflective subexpressions. What happens if the method name or type signature is specified by a non-constant expression? Must the array expressions always be explicit subexpressions, or may they be bound as constants to temporary names? Is the statically bound invokedynamic instruction simply an optimization of a more general reflective facility, and if so, how can a user reliably get this optimization (if at all)? Must operands like 123 be autoboxed, even though the invokedynamic instruction does not box its primitive arguments? Modeling an invokedynamic instruction reflectively makes it compatible with the existing Java language; the features of autoboxing and varargs (not used here) can even make the expressions of tolerable brevity, but there it is very difficult to mesh such expressions with the static typing guarantees inherent to the invokedynamic instruction.

Cases B and D make the invokedynamic instruction signature explicit. The instruction is polymorphic across all VM-level signatures, so somehow each dynamic invocation expression must specify (implicitly or explicitly) the signature to use. Moreover, the arguments and return value must agree with that signature. (The JVM verifier ensures this for invokedynamic instructions just as for other invoke instructions.) Given that there must be matching arguments at the call site, the argument type information is already present, and need not be redundantly specified via an explicit signature. This is the root cause for some of the problems with B and D: Not only are they verbose because they specify the argument types two different ways (explicitly and implicitly in the argument types), but they are also buggy and complex because those redundant specifications might fail to coincide.

At minimum, only the intended return type needs to be specified explicitly. In the current proposal this is done via a context-dependent target typing convention. (The JSR 292 EDR used an optional type parameter.) If there is no explicit marking, the return type defaults (helpfully) to Object. In case C, it is done by a type name following the new keyword. In case A2, the type parameter has a peculiar feature that, as a literal part of the invokedynamic signature, it can be a primitive type or the non-type void, in addition to any reference type.

The last case E presupposes a way to inline bytecode assembly code into a Java program. If there were such a thing, perhaps that would be the most natural way to introduce dynamic call sites. However, there is no such thing proposed for the Java language, and designing such a thing would be a much larger undertaking than the present one.
All of these observations (except those about the variable method name) apply also to the signature-polymorphic methods of MethodHandle.

**EXAMPLES:**

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

```java
void test(MethodHandle mh) {
    mh.invokeExact("world", 123);
    // previous line generates invokevirtual MethodHandle.invokeExact(String,int) -> void
    InvokeDynamic.greet("hello", "world", 123);
    // previous line generates invokedynamic greet(Object,String,int) -> void
    // enclosing class, method, or declaration must declare 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.

```java
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",
                MethodType.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:
    @BootstrapMethod(value=MOP.class, name="linkCallSite")
    private static Object method2(Object x) {
        // native to the JVM and the JIT
        return InvokeDynamic.#"myscript:ready?"(x);
    }
}
```
class MOP {
    // shared logic for resolving call sites
    public static CallSite linkCallSite(Class caller, String name, MethodType type) {
        MethodHandle target = resolveCallSite(caller, name, type);
        return new CallSite(target);
    }
}

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 {
    @BootstrapMethod(value=Hello.class, name="bootstrapDynamic")
    public static void main(String... av) throws Throwable {
        if (av.length == 0) av = new String[] { "world" };  
        greeter(av[0] + " (from a statically linked call site)");
        for (String whom : av) {
            greeter.invokeExact(whom);  // strongly typed direct call
            // previous line generates invokevirtual MethodHandle.invokeExact(String) -> void
            Object x = whom;
            @BootstrapMethod(value=Hello.class, name="bootstrapDynamic")
            Object y = InvokeDynamic.hail(x);  // weakly typed invokedynamic
            // previous line generates invokedynamic hail(Object) -> Object
        }
    }

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

    // Set up a class-local bootstrap method.
    private static CallSite bootstrapDynamic(Class caller, String name, MethodType type) {
        assert(type.parameterCount() == 1 && name.equals("hail"));  // in lieu of MOP
        System.out.println("set target to adapt "+greeter);
        return new CallSite(greeter.asType(type));
    }
}

ADVANCED EXAMPLE:
(See before-and-after MOP example above.)

DETAILS

SPECIFICATION:
1.1 A signature-polymorphic method is a method which is declared with the annotation @java.dyn.MethodHandle.PolymorphicSignature.

1.2 Every signature-polymorphic method must be declared with the following properties:
   • It must be native.
   • It must take a single varargs parameter of the form Object....
   • It must produce a return value of type Object.
   • It must be contained within the java.dyn package.

Direct primitive casts are specifically allowed on the result of a signature-polymorphic method invocation, even if the result is a reference type such as Object. (This last point may have been unclear in some versions of the JLS; see Sun bug 6979683.)

(Note: Because of these requirements, a signature-polymorphic method is able to accept any number and type of actual arguments, and can, with a cast, produce a value of any type.)

Here is an example:
When a call to a signature-polymorphic method is compiled, the associated linkage information for its arguments is not an array of `Object` (as for other similar vararg methods) but rather the erasure of the static types of all the arguments.

In an argument position of a method invocation on a signature-polymorphic method, a null literal has type `java.lang.Void`, unless cast to a reference type.

The linkage information for the return type is derived from a context-dependent target typing convention. The return type for a signature-polymorphic method invocation is determined as follows:

- If the method invocation expression is an expression statement, the method is `void`.
- Otherwise, if the method invocation expression is the immediate operand of a cast, the return type is the erasure of the cast type.
- Otherwise, the return type is the method’s nominal return type, `Object`.

Programmers are encouraged to use explicit casts unless it is clear that a signature-polymorphic call will be used as a plain `Object` expression.

A signature-polymorphic class is a class which is declared with the annotation `@java.dyn.MethodHandle.PolymorphicSignature`.

Every signature-polymorphic class must be declared with the following properties:

- It must be final.
- It must have no supertype other than `java.lang.Object`.
- It must declare a private constructor (thereby suppressing an implicit non-private constructor).
- It must not explicitly declare any public members.
- It must be contained within the `java.dyn` package.

For example:

```java
package java.dyn;

interface PolymorphicSignature { // non-public, used only as directed in this proposal
}

public class MethodHandle {
    @interface PolymorphicSignature { }

    public native @PolymorphicSignature Object invokeExact(Object... args) throws Throwable; // example
    public native @PolymorphicSignature Object invokeGeneric(Object... args) throws Throwable; // example
}
```

A signature-polymorphic class implicitly declares one signature-polymorphic method for each method name that is legal on the JVM. The implicitly declared method will have the following properties:

- It will be signature-polymorphic (native, varargs, returning `Object`).
- It will be public and static.
- It will throw `java.lang.Throwable`.

A class annotated as signature-polymorphic may serve as a qualifier to any method name whatever. Even if this method name appears as a member of `Object`, it will be looked up in the signature-polymorphic class.

If the class is `InvokeDynamic`, the invocation mode is `dynamic`. This means that instead of an `invokestatic` call, the compiler generates an `invokedynamic` call site with the given name and a descriptor (symbolic type signature) derived from the signature-polymorphic class of the call.

Implications for `invokedynamic`.

(Note: This section is redundant with specifications already in the JLS.)
In this way, an `invokedynamic` 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.

```java
InvokeDynamic.anyNameWhatever(); // type () -> void
InvokeDynamic.anotherName("foo", 42); // type (String, int) -> void
Object x = InvokeDynamic.myGetCurrentThing(); // type () -> Object
invokeDynamic.myPutCurrentThing(x); // type (Object) -> void
int y = (int) invokeDynamic.myHashCode(x); // type (Object) -> int
boolean z = (boolean) invokeDynamic.myEquals(x, y); // type (Object, int) -> boolean
String v = (String) invokeDynamic.myToString(); // type () -> String
Object w = InvokeDynamic.#"it's complicated"(0); // type (int) -> Object
```

(Rationale: This design uses syntaxes which are already correct, even if signature polymorphism is neglected, assuming a suitable static method exists with `object` return type and varargs parameters.)

As noted above, a null literal has type `java.lang.Void`. This type 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. For example:

```java
Object junk = InvokeDynamic.myPrintLine(null); // type (Void) -> Object
InvokeDynamic.foo((String)null, null); // type (String, Void) -> void
```

As the JVM executes, checked or unchecked 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. This is why these calls throw the most general type, `Throwable`. For example:

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

In practice, this means that methods containing `InvokeDynamic` calls will generally throw `Throwable` to their callers, or else include some complicated catch-all logic around their dynamic call sites. This limits but does not destroy the usefulness of `InvokeDynamic` calls.

2.7 Every invocation of a signature-polymorphic method of `InvokeDynamic` must be lexically enclosed in a declaration which is annotated with `@java.dyn.BootstrapMethod.`

Every `invokedynamic` instruction executed by JVM requires its own bootstrap method, which reflectively controls the initial linkage of that instruction. The bootstrap method is declared in a constant pool entry associated with each `invokedynamic` instruction.

It is common for dynamic code to use a single bootstrap method in common for group of call sites. There is a syntax which allows an entire class declaration, method declaration, or variable declaration to be annotated with a bootstrap method to be used for all `invokedynamic` expressions occurring in that declaration. This syntax is derived from annotations. (Note: This may be changed to a keyword-based syntax. It may also be changed to use method references, if they are introduced into the language.)

```java
@BootstrapMethod(value=Bar.class, name="baz")
class {
  static {
    InvokeDynamic.thisUsesBaz(); // this invocation bootstraps using Bar.baz
    InvokeDynamic.soDoesThis(); // so does this
  }
  @BootstrapMethod(value=Bar2.class, name="baz2")
  void method1() { InvokeDynamic.thisUsesBaz2(); } // BSM = Bar2.baz2
  @BootstrapMethod(CallSite3.class)
  void method2() { InvokeDynamic.thisUsesCallSite3(); } // BSM = new CallSite3
}
```

A bootstrap method declaration contains a class name and an optional method name. If the method name is present, the bootstrap method is a reference to a static method in the named class with three arguments. If there is no method name, the bootstrap method is a reference to an (unnamed) factory method which calls the constructor on the named class of three arguments.

A dynamic call site is bootstrapped according to the innermost annotated declaration within which it lexically occurs.

The declared method or constructor must be accessible at the location of any `InvokeDynamic` expression to which the declaration applies. It must also be unambiguously applicable to three arguments of type `Class, String, and MethodType`. In the case of a static method, the selected method must return a reference type.

The annotation itself is not retained in the class file, and does not affect the generation of any bytecode instruction. It does, however, affect information in the constant pool which configures the initial linkage of the instruction.

2.8 Implications for method handles.
Because of the above restrictions on signature-polymorphic methods, these methods can take any type and number of arguments, and (with a single cast) can return a value of any type. In effect, `java.dyn.MethodHandle` appears to have an infinite number of non-static signature-polymorphic methods of fixed names, such as `invokeExact` and `invokeGeneric`, and of every possible (erased) signature. However, this appearance is indistinguishable from the simpler varargs-based definitions described above.

```
MethodHandle mh = ...;
  mh.invokeExact("foo", 42); // type (String, int) -> void (not Object)
  int x = (int) mh.invokeGeneric(); // type () -> int (not Integer or Object)
MethodType mtype = mh.type(); // no new rules here; see JSR 292 javadocs
  mh.neverBeforeSeenName(); // no new rules; must raise an error
```

(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.)

Here are some examples of signature-polymorphic calls on method handles:

```
MethodHandle mh1, mh2, mh3, mh4, mh5, mh6; ...
  Object x = mh1.invokeGeneric(); // type () -> Object
  mh2.invokeExact(x); // type (Object) -> void
  int y = (int) mh3.invokeExact(x); // type (Object) -> int
  boolean z = (boolean) mh4.invokeGeneric(x, y); // type (Object, int) -> boolean
  String v = (String) mh5.invokeExact(); // type () -> String
  Object w = mh6.invokeGeneric(0); // type (int) -> Object
```

Here are some examples of null processing:

```
  Object junk = mh1.invokeExact(null); // type (Void) -> Object
  mh2.invokeGeneric((String)null, null); // type (String, Void) -> void
```

As the JVM executes, checked or unchecked 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, signature polymorphic methods defined by JSR 292 throw the most general type, `Throwable`. Here are some examples:

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

(Note: This means that methods containing method handle invocations will generally throw `Throwable`, or else include some complicated catch-all logic around their method handle call sites. This limits but does not destroy the usefulness of method handle calls.)

As usual, if a null value typed as a method handle qualifies a signature-polymorphic method, the expression must terminate abnormally with a `NullPointerException`.

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

The bytecode emitted for any call to a signature-polymorphic method of `MethodHandle` will be an `invokevirtual` instruction, exactly as if a public virtual method of the desired descriptor were already present in `java.dyn.MethodHandle`:

```
  mh.invokeExact(1); // produces an invokevirtual instruction
  class MethodHandle { ... public abstract void invokeExact(int x); ... } // hypothetical overloading of 'invoke'
  mh.invokeGeneric(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 kind of 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. Such a token is called an "exotic identifier".

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

- **ExoticIdentifierCharacters**:
  - **ExoticIdentifierCharacter**:
    - **StringCharacter** but not **DangerousCharacter**
    - **ExoticEscapeChar**
  - **DangerousCharacter**: one of `/ . ; < > [ ]`
  - **ExoticEscapeChar**: one of `! # $ % & ( ) * + , - : = ? @ ^ _ ` { | } ~`

```java
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 keyword or token.

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

// another take on java.lang.Integer:
```java
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 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).

```java
int #"\t" = 5; // a two-character identifier
System.out.println(#"\u0009"); // prints 5
```

Even if the spelling of an exotic identifier is identical to the spelling of a reserved word, keyword, or other non-identifier token, the exotic identifier is treated syntactically as an identifier.

```java
import java.util.*; // imports all classes in java.util
import my.chars.#"**"; // imports one class bytecoded as "my/chars/**"
```

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 denote the same identifier, and may be freely mixed.

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

As implied by the grammar above, an exotic identifier may not be empty. That is, there must be at least one character between the opening and closing quotes.

```java
int #""; // must be rejected
```
3.3 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.

3.3.1 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 [.

3.4 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 ! "$ % & ( ) * + , - : = ? @ ^ _ ` ~ and no others.

3.5 As with string and character tokens, a string character escape containing an octal or hexadecimal code may denote any character whatever, including dangerous characters or other punctuation.

3.6 Further discussion (non-normative)
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.

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.

```plaintext
package my.xml.tags;
class "<pre>" { ... }

package my.sql.bindings;
interface Document { String title(); Text "abstract"(); int "class"(); ... }

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

**COMPILATION:**

See JSR 292 for the specification of invokedynamic instructions. In brief, they begin with a new opcode, a CONSTANT_InvokeDynamic index, and end with two required zero bytes. The CONSTANT_InvokeDynamic constant pool entry contains the index of a CONSTANT_MethodHandle reference (the bootstrap method) and a CONSTANT_NameAndType (which works the same as for other invocation types). In effect, the bootstrap method of an invokedynamic instruction takes the place of a class or interface reference (CONSTANT_Class) in the symbolic reference for another invocation type (CONSTANT_Methodref or CONSTANT_InterfaceMethodref).

Method handle invocation is just an ordinary invokevirtual instruction, whose class is java.dyn.MethodHandle, whose name is the declared name (such as invokeExact or invokeGeneric), and whose descriptor signature is completely arbitrary; this requires no special compilation support beyond undoing the effects of varargs and autoboxing, and applying the contextual target type, when generating the call descriptor type.

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.

**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.

**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.

**REFLECTIVE APIS:**

The method java.lang.Class.getDeclaredMethod must be special-cased to always succeed for signature-polymorphic methods, regardless of signature. The JSR 292 JVM has such logic already, but it must be exposed out through the Class API. Note that this requirement implies that the @PolymorphicSignature annotations are preserved in a RuntimeVisibleAnnotations attribute.

Only single-result reflection lookups need to be changed. Multiple-method lookups should not produce implicitly defined methods or undeclared type signatures.

The java.lang.model API (which is used internally by javac) does not need specialization, because the signature-polymorphic methods do not ever need to mix with other more normal methods. The static (compile-time) model of InvokeDynamic should not present any enclosed elements, while that of MethodHandle should not present any overloading of signature-polymorphic methods, other than native varargs definitions as mandated above.

**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  
These changes are defined in three parts, (1) method handle invocation, (2) dynamic invocation, (3) exotic names. Part (1) requires no actual changes to the language, but is instead a specialized invocation linkage rule. Part (2) part requires minor changes to the language's rules for scoping some qualified method names. Part (3) is a lexical change only, introducing new name spellings.

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.

None of these features should be available in source languages of Java 1.6 or previous. (This affects the treatment of the "-source" flag for javac.)

To prevent accidental interaction with older compilers and JVMs, method handles and invokedynamic must be enabled only when compiling to a new class file version number of Java 1.7 or later. (This affects the treatment of the "-target" flag for javac.)

(Note: Since the invokedynamic and exotic identifier features involve changes to the Java language, they are presently experimental.)

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. These purposes 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 Void 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 would be (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.)

Q: Why didn't you pick a better syntax for exotic identifiers?

A: This question has many variants, because for different questioners "better" might mean more standard or prettier or less confusing or more concise or similar to another language the questioner admires. But the various questions all have one answer. The key requirement is to push some required bytecode name through the Java language, even though it is not a valid Java identifier. Such bytecode names, though perfectly valid in the JVM, are (by definition) second-class citizens in the Java language. It is therefore be a blunder to express them with any syntax which might be more useful for other purposes. This anti-goal trumps the goal of making exotic identifiers look elegant, standard, or non-confusing. After all, the identifier itself is likely to contain highly inelegant, non-standard, and confusing-looking characters. Therefore, we use three characters #** to set apart the exotic bytecode name, instead of one or two, so that exotic identifiers are clearly distinguished from other constructs.

Groovy and Scala have syntaxes for exotic identifiers which using a pair of quotes (of some kind), and this could be made to work for Java also, but the result would be a needless elegance in the appearance of the exotic identifiers, at the cost of greater lexical ambiguity, or (for backquote) the consumption of a character better used for other purposes.

We choose characters which are already in use (double quotes) or likely to be in use soon (hash for closures). We do not attempt to choose some character which we hope nobody is using (backquote, backslash). Such "unused" characters, precisely because they are unused by Java, are sometimes useful for combining Java fragments with other languages, or making experimental extensions to Java. An infrequently used feature like exotic identifiers should not consume such a character.

The supposed confusion of hash with its use by closures is a non-issue in practice. Given the limited number of characters in the ASCII set, programmers are not disturbed by seeing one character used in several ways in a language, as long as the associated lexical rules keep the uses distinct. (When was the last time you were offended that the Java comment characters are also used for multiply and divide?) In this case, the lexical rule is simple, unambiguous, and easy to perceive by eye or by parser: Hash with a quote is an exotic identifier, while hash with something else is something else.