Going meta to Panama & Valhalla

John Rose, JVM Architect

JVM Language Summit, Santa Clara, July 2018

http://cr.openjdk.java.net/~jrose/pres/201807-GoingMeta.pdf

ORACLE

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.

Meta…

How I learned to relax and generate my code on the fly…

OR

Many things worth doing are worth doing automatically.

ORACLE

Demo first: Panama native library workflow

- New command:
  - `jextract` scans a header file into metadata JAR
- JARs to watch for:
  - **`stdio.jar`, `python.jar`**: C API's extracted as Java interfaces
- Types to watch for:
  - **`Library`**: a loaded native artifact
  - **`Pointer`**: cursor pointing into a native memory block
  - **`Scope`**: holder for native memory blocks and other resources
- Operations to watch for:
  - **`Libraries.bind`**: implement a set of extracted APIs

How to get Panama (1)

```
Projects$ hg clone http://hg.openjdk.java.net/panama/dev panama
http://hg.openjdk.java.net/panama/dev
pulling from http://hg.openjdk.java.net/panama/dev
…

Projects$ cd panama

panama$ hg config paths.default
http://hg.openjdk.java.net/panama/dev

panama$ hg log -b foreign -l1 --style compact
51692[tip]:51602,51687   9b3cc75708ea   2018-07-25 22:10 +0200   mcimadamore
  Automatic merge with nicl

panama$ hg co foreign
315 files updated, 0 files merged, 150 files removed, 0 files unresolved
```

How to get Panama (2)

```
panama$ vi README.foreign

panama$ CLANG=clang+llvm-6.0.0-x86_64-apple-darwin
panama$ open http://releases.llvm.org/6.0.0/$CLANG.tar.xz
panama$ (cd ~/Downloads; tar -xJf - < $CLANG.tar.xz)

panama$ (cd ~/Downloads/$CLANG; file lib/libclang.dylib)
lib/libclang.dylib: Mach-O 64-bit dynamically linked shared library x86_64

panama$ sh ./configure --with-sdk-name=MacOSX10.6 \
                 –with-boot-jdk=$HOME/env/JAVA11_HOME \
                 –with-libclang=$HOME/Downloads/$CLANG
configure: Configuration created at Wed Jul 25 16:26:13 PDT 2018.
…

panama$ make
Building target 'default (exploded-image)' in configuration 'macosx-x86_64-
normal-server-release'
…
```

How to get Panama (3)

```
panama$ ln -s build/macosx-x86_64-normal-server-release/jdk  JH

panama$ JH/bin/java -version
openjdk version "12-internal" 2019-03-19
OpenJDK Runtime Environment (build 12-internal+0-adhoc.jrose.panama)
OpenJDK 64-Bit Server VM (build 12-internal+0-adhoc.jrose.panama, mixed mode)

panama$ file JH/bin/jextract
JH/bin/jextract: Mach-O 64-bit executable x86_64

panama$ # make yourself an IDE project & open your favorite IDE

panama$ sh bin/idea.sh java.base

panama$ open -a 'IntelliJ IDEA 2018.2 EAP' IDE/java.foreign
```

How to kick the tires on Panama (1)

```
panama$ #ln -s build/macosx-x86_64-normal-server-release/jdk  JH

panama$ WRKARD='--exclude-symbols __.*|zopen /usr/include/i386/_types.h'
panama$ JH/bin/jextract -t nat /usr/include/stdio.h -o stdio.jar $WRKARD

panama$ zipinfo stdio.jar nat/stdio.class
-rw----     2.0 fat    21790 bl defN 18-Jul-30 17:06 nat/stdio.class

panama$ cat imports.jshell
import java.foreign.*
import java.foreign.memory.*
import java.foreign.layout.*
…

panama$ JH/bin/jshell JAVASE imports.jshell –class-path stdio.jar

jshell> <T> T bind(Class<T> api) {
                  return Libraries.bind(MethodHandles.lookup(), api); }

jshell> var stdio = bind(nat.stdio.class)
stdio ==> nat.stdio$Impl/0x0000000800168040@19d37183
```

How to kick the tires on Panama (2)

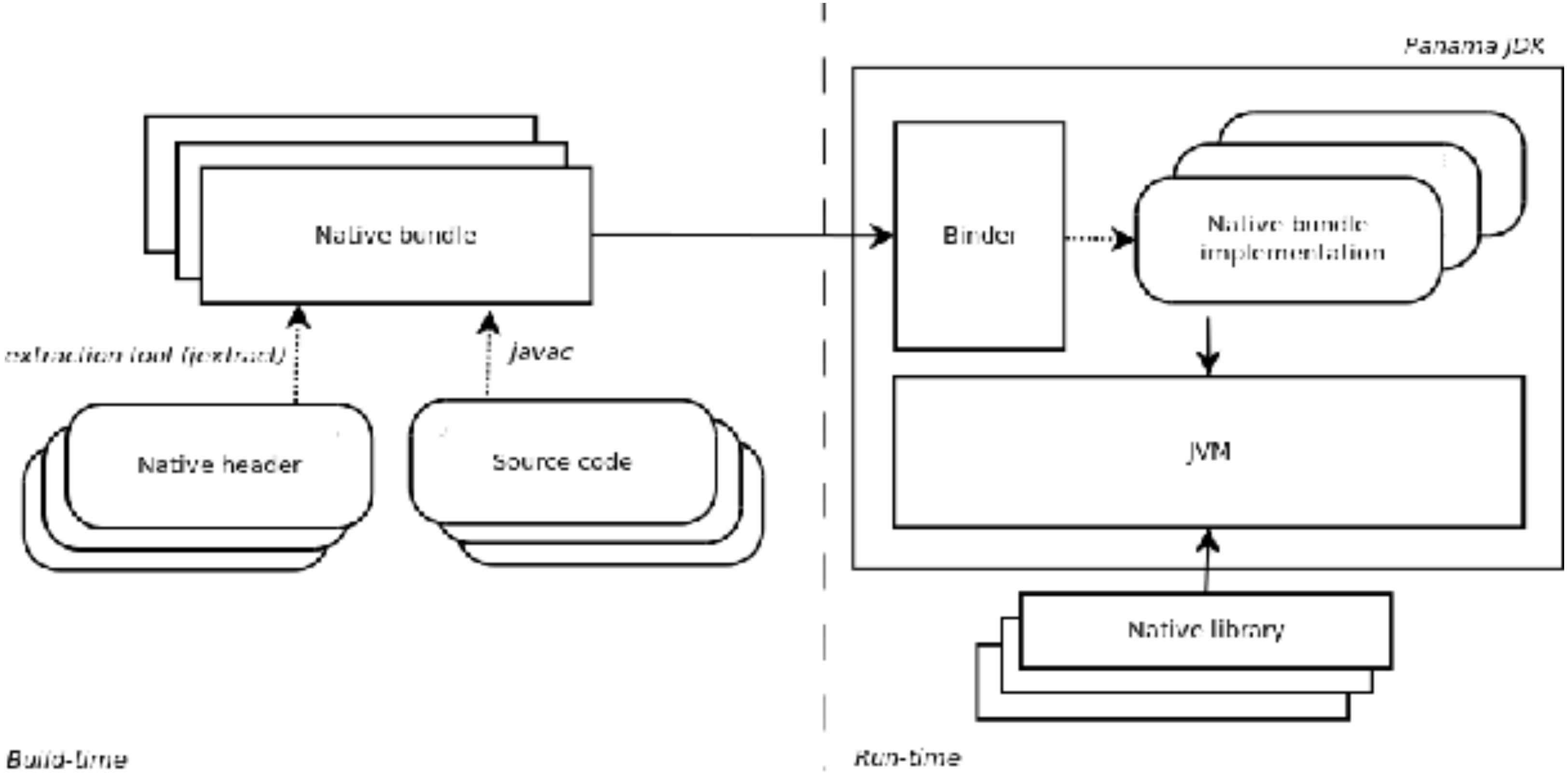
```
jshell> var globals = Scope.newNativeScope()
globals ==> jdk.internal.foreign.ScopeImpl$NativeScope@3bd40a57

jshell> Pointer<Byte> s(String s) { return globals.toCString(s); }
jshell> String s(Pointer<Byte> s) { return Pointer.toString(s); }

jshell> var buf = globals.allocate(NativeTypes.INT8, 100)
jshell> /var buf
|    Pointer<Byte> buf = { … }

jshell> var fmt = s("hello %s from #%d"); var world = s("world")
jshell> stdio.sprintf(buf, fmt, world, 42); s(buf)
$57 ==> "hello world from #42"

jshell> buf.offset(8).set((byte)0); s(buf)
$58 ==> “hello, w”
```

How to kick the tires on Panama (3)

```
panama$ jextract -l=python2.7 -rpath=/System/Library/Frameworks/
Python.framework/Versions/2.7/lib/ /usr/include/stdio.h /usr/include/
stdlib.h /usr/include/python2.7/Python.h -t org.python -o python.jar
…
panama$ JH/bin/jshell imports.jshell --class-path python.jar

jshell> var Python = bind(org.python.Python.class)
Python ==> org.python.Python$Impl/0x0000000800142c40@6cc7b4de

jshell> var pythonrun = bind(org.python.pythonrun.class)
pythonrun ==> org.python.pythonrun$Impl/0x0000000800158040@3d299e3

jshell> pythonrun.Py_Initialize()
jshell> pythonrun.Py_AtExit(() -> System.out.println("Good bye!”))
jshell> pythonrun.PyRun_SimpleStringFlags(s("exit()"),Pointer.nullPointer())
Good bye!
… (jshell subprocess exits) …
```

What did we just see?

`http://cr.openjdk.java.net/~mcimadamore/panama/panama-binder-v3.html`



Wrapperless native programming

- Native resources:
  - `jextract -l=python2.7 -rpath=…/Python.framework/…`
  - the JVM itself
  - no JNI wrappers, no extra native methods for Python or stdio
- We saw methods in stdio; where did their bytecode come from?
  - ***not*** statically generated by jextract (interfaces only in stdio.jar)
  - the binder Libraries.bind implemented the native APIs
  - …using Unsafe-style native call generators (as trusted code)
  - …and Unsafe API points to peek-n-poke inside native structs

The “binder pattern”

Application specific schema (foo.h)

⇒ extract metadata (jextract) ⇒

JAR of annotated interfaces foo.jar

⇒ run binder (bind(foo.class)) ⇒

implemented interfaces plus loaded resources (var foo)

⇒ Java programmer ⇒

use implemented API points on the interfaces (foo.natfunc())

Advantages of the binder pattern

- Static configuration is metadata only, not executable.
  - Says “what we need”, not “how to get it”
  - Metadata JAR has multiple uses, not just for execution.
- The runtime has the final word on implementation
  - Can take into account exact hardware and class-path configuration
  - Can be trusted to deploy unsafe implementation techniques
  - Can be configured with more or less “debug mode” or “perf mode”
  - The binder is a JIT API creator
- General pattern, applies to C, C++, IDLs, parser generators, etc.

Drawbacks of the binder pattern

- Runtime generation of code might make apps. slow to start
  - (It cuts both ways: Meta-JARs are tiny compared to regular JARs.)
- No source code to single-step through
- Configuration failures are detected late

- It’s a work in progress.
- Idea: A bound API could be a compile-time or jlink-time constant!

meta-programs manage meta-data *about* programs

- The output of a meta-program is some part of a “normal” program
- javac and `URLClassLoader` are familiar meta-programs
- The JVM is a great honking monster metaprogram
- Over time, we are refactoring the JVM to allow more metaprogramming
  - key refactorings take the form of “hooks” like `invokedynamic`
  - bootstrap methods execute at “meta” level and then return control
- Are we done adding bootstraps? Probably not…

ORACLE

class file structure: the constant pool

```
ClassFile
  cp_info
  1: CONSTANT_Utf8 “Foo”
  2: CONSTANT_Class #1
  …
  41: CONSTANT_InvokeDynamic …
  …
```

class file structure: class/field/method schema

```
ClassFile

cp_info
1: CONSTANT_Utf8 “Foo”
2: CONSTANT_Class #1
…
41: CONSTANT_InvokeDynamic …
…

this_class = #2 “Foo”
super_class = #9 “Object”
interfaces[0] = #12 “Iterable”

field_info
name = “x”, desc = “LBar;”

method_info
name = “f”, desc = “()V”
```

class file structure: bytecode

```
ClassFile

cp_info
1: CONSTANT_Utf8 “Foo”
2: CONSTANT_Class #1
…
41: CONSTANT_InvokeDynamic …
…

this_class = #2 “Foo”
super_class = #9 “Object”
interfaces[0] = #12 “Iterable”

field_info
name = “x”, desc = “LBar;”

method_info
name = “f”, desc = “()V”

method_info
name = “g”, desc = “(I)I”

Code_attribute

0: iload 0 …
```

class file structure: indy

```
ClassFile
```

```
cp_info
1: CONSTANT_Utf8 “Foo”
2: CONSTANT_Class #1
…
41: CONSTANT_InvokeDynamic …
…
```

```
this_class = #2 “Foo”
super_class = #9 “Object”
interfaces[0] = #12 “Iterable”
```

```
field_info
name = “x”, desc = “LBar;”
```

```
method_info
name = “f”, desc = “()V”
```

```
method_info
name = “g”, desc = “(I)I”
```

```
Code_attribute

0: iload 0
3: invokedynamic #41
8: ldc #42
11: iadd
12: ireturn
```

indy

class file structure: indy, condy

ClassFile

```
cp_info
1: CONSTANT_Utf8 "Foo"
2: CONSTANT_Class #1
…
41: CONSTANT_InvokeDynamic …
42: CONSTANT_Dynamic …
…
```

condy

```
this_class = #2 "Foo"
super_class = #9 "Object"
interfaces[0] = #12 "Iterable"
```

```
field_info
name = "x", desc = "LBar;"
```

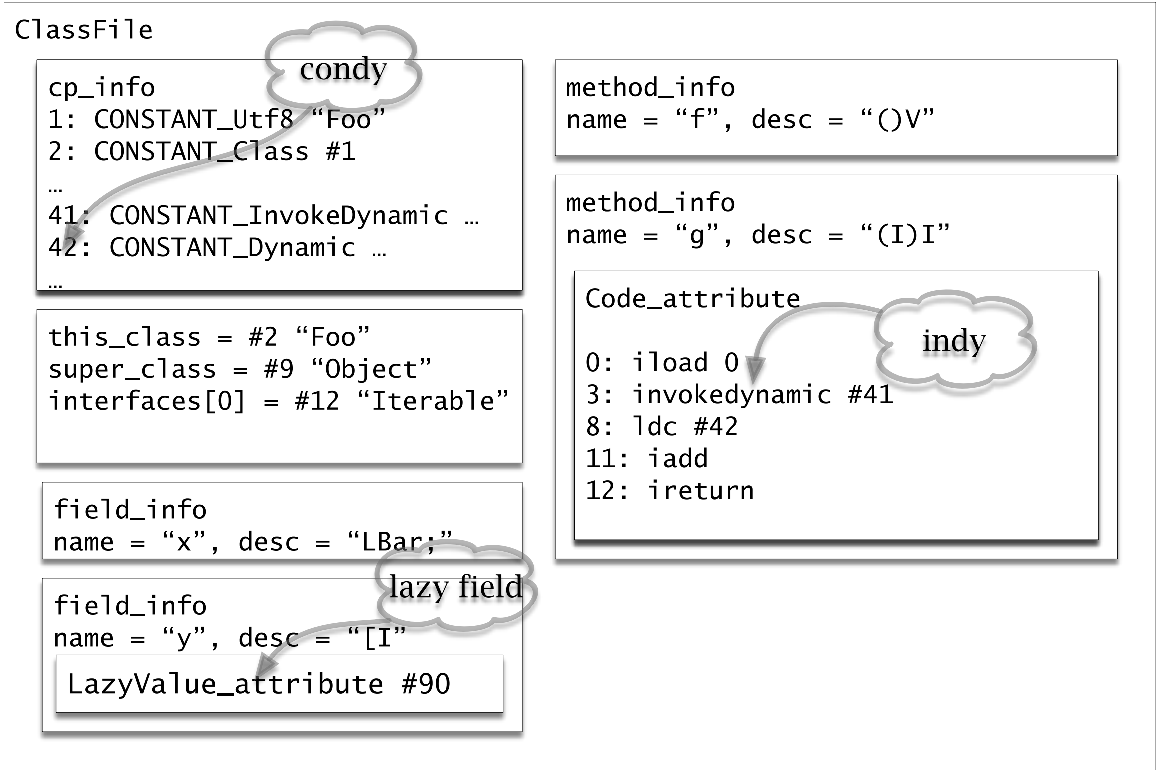
```
method_info
name = "f", desc = "()V"
```

```
method_info
name = "g", desc = "(I)I"
```

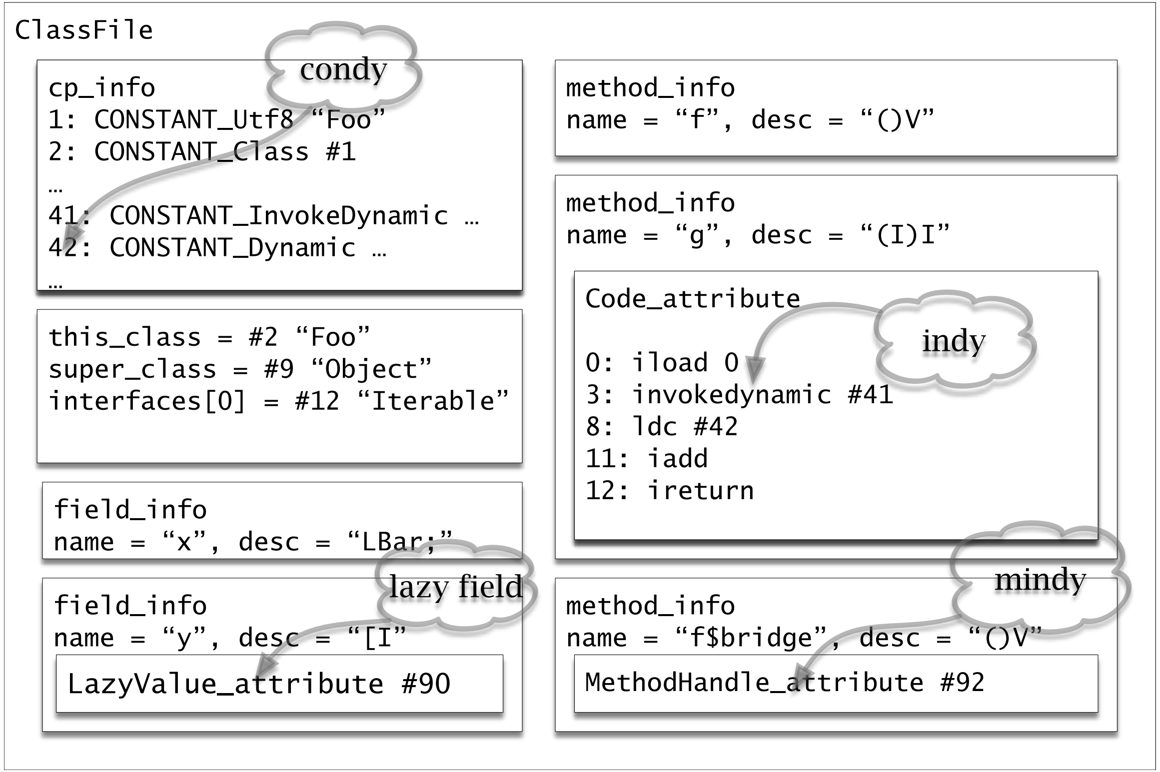
```
Code_attribute

0: iload 0
3: invokedynamic #41
8: ldc #42
11: iadd
12: ireturn
```

indy

class file structure: field impl. (“findy” ?)



class file structure: method impl. (mindy)



ORACLE

class file structure: class impl.

```
ClassFile

cp_info
1: CONSTANT_Utf8 “Foo”
2: CONSTANT_Class #1
…
41: CONSTANT_InvokeDynamic …
42: CONSTANT_Dynamic …
…

this_class = #2 “Foo”
super_class = #9 “Object”
interfaces[0] = #12 “Iterable”
interfaces[1] = #43 “List<int>”

field_info
name = “x”, desc = “LBar;”

field_info
name = “y”, desc = “[I”
  LazyValue_attribute #90

method_info
name = “f”, desc = “()V”

method_info
name = “g”, desc = “(I)I”
  Code_attribute
  0: iload 0
  3: invokedynamic #41
  8: ldc #42
  11: iadd
  12: ireturn

method_info
name = “f$bridge”, desc = “()V”
  MethodHandle_attribute #92
```

condy

cindy

lazy field

indy

mindy

ORACLE

meta programming with bootstrap methods

- Key question: What's in a name?
  - The JVM names classes, supers, fields, methods, descriptor types
  - Names are interpreted in the course of constant pool resolution
  - Can we add our own BSMs to enhance such resolution?
- invokedynamic on an arbitrary name
- ldc or getstatic on a lazy constant
- method resolution can use a BSM: "mindy" aka. "bridge-o-matic"
- class resolution uses `ClassLoader` API but can also use a BSM
- "Bootstrap methods everywhere!"

mindy (bridge-o-matic)

- Given: JVM needs to resolve a method (from `invokevirtual`, say)
- Step 1: JVM performs hardwired method lookup, and fails.
- Step 2: JVM notices one or more related classes with mindy BSMs.
- Step 3: JVM goes meta, hands request to each BSM in turn.
- Step 4: A BSM returns a valid MH for the method descriptor
- Step 5: JVM saves the resolution result, and uses it.
- Result: User request links to just-in-time special delivery code.
  - The code is generated using accurate information; no ICCE.

mindy (bridge-o-matic): today and tomorrow

- We are using automatic method generation today for values & records
- Idea: Pre-generate the method; body is a single `invokedynamic`
- Followup #1: No bytecodes, swap out `Code` for `MethodHandle` attribute
- Followup #2: Inherit specialized methods from the super
  - No methods at all in the class-file, just sugary supers
- Followup #3: ***import*** specialized methods from supplier classes.
  - The supplier would be privately declared inside the class.
  - This decouples inheritance from reuse, a very powerful move.
  - (Did you ever notice that `AbstractList` smells a little bit bad?)

ORACLE

cindy (class-o-matic)

- Given: JVM needs to resolve a class (from `invokestatic`, say)
- Step 1: JVM performs hardwired class lookup, and fails.
- Step 2: JVM notices one or more related classes with cindy BSMs.
- Step 3: JVM goes meta, hands request to each BSM in turn.
- Step 4: A BSM returns a meta-class object for the class descriptor
- Step 5: JVM saves the resolution result, and uses it.
- Result: User request links to just-in-time special delivery code.

- Hang on! What's that "meta-class object"? Did we spin bytecodes?

going meta on classes

- A `MethodHandle` is reflected new behavior, not just a plain method
- So, what’s a reflected new type?

  ```
  jl.Class : jlr.Method  ::  ??? : jli.MethodHandle
  ```
- Idea: A type is a bundle of the same “stuff” as in a classfile
  - So a new reflected type is a bundle of reflected “classfile stuff”
- Operations on a reflected type (not just a `jl.Class`):
  - Resolve a static field, method, or constructor.
  - Test an object for membership in the type, or cast it.
  - Ask about related types, supers, etc.   (Also make an array?)
- If we can do “mindy”, then “cindy” becomes a bunch of “mindy” points.

other JVM work required: Species

- The JVM works very hard to efficiently implement class instances.
- It can help us implement abstract reflected types.
  - Even if they are not just dynamically spun classfiles.
- When is a type not a class? When it is a template species.
  - A class which can be specialized has one or more “holes”.
  - It is therefore a template which must be filled out.
  - Each specialization (“species”) fills the holes in a different way.
  - `List<int>` and `List<Point>` are species of template `List<·>`
- Conjecture: the JVM can spin up species more quickly than classes.

template classes in the JVM

http://cr.openjdk.java.net/~jrose/values/template-classes.html

- Idea: Punch some holes in a constant pool.
- Mark them (with metadata) describing how they must be filled.
  - Many holes will be types, as with reified generics
  - Conjecture: Other holes could be behaviors or arities or signatures
- Every time a template is filled in, a new species is created.
  - Implementor responsibility: Choose names and/or registries.
  - Filling in a template is like running a `ClassLoader`
- Generic methods can be localized templates inside a template class.
- Appropriate BSMs can perform ad hoc template expansion
  - For example: `FooList<T>` is comparable if `T` is comparable

what’s in a name, take #2

- Objects are nifty; we can model behaviors and types with them.
- When an object models meta-thing, it’s a meta-object.
- BSMs and binders are good ways to answer meta-questions.

- But what’s the ***name*** of a meta-object? Can a name be “meta”?
- Idea: Expand the JVM’s naming of classes and descriptors.
  - Add annotation-like decorations to existing type names.
  - Give useful rules for how to process annotated types.
  - Annotations trigger BSM invocation at link-time (resolution).
  - The verifier treats annotations conservatively (no BSM upcalls).

type operators in the JVM

https://bugs.openjdk.java.net/browse/JDK-8204937

- A type name can be followed by one or more type operators.
- Type operators have a syntax distinct from class names or descriptors.
- An appropriate BSM protocol (TBD) dispatches resolution requests.
- The verifier ensures
- Key semantic rule: A type operator ***always narrows*** its type operand.
  - Thus `Ljava/util/Map;/[ID]` is a subtype of `Ljava/util/Map;`
  - The prefix always names a less constrained type.
  - The undecorated type is the ***carrier type*** known to the JVM

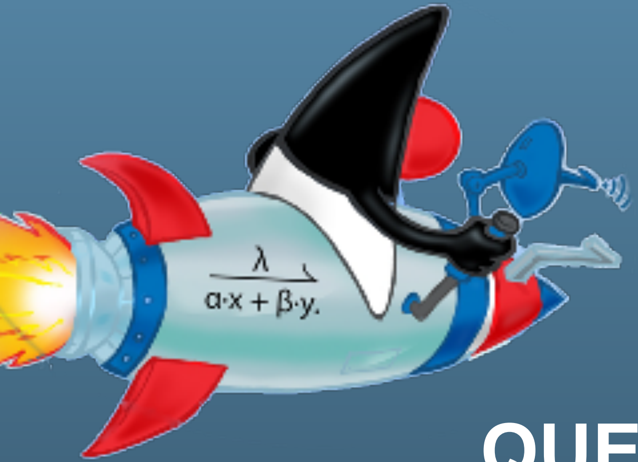
ORACLE

hypothetical type operator examples

- `Ljava/util/Map;/[ID]` denotes `Map<int,double>`
- `Ljava/util/List;/[$wild;]` denotes the wildcard species of `List`
- `I/$interval[$ge;0;]` denotes an int whose value is non-negative
- `Ljava/lang/String;/$N;` denotes `the` "`N`" variant of `String`
- `(Ljava/lang/String;)Ljava/lang/String;/$N;`
  … describes a method wrapping a String as an N-String
- `L/$union[LFoo;LBar;]` denotes a union of `Foo` and `Bar`
- `LFoo;/$and[LBar;]` denotes a `Foo` which must also be a `Bar`
- `L/$nullable[LPoint;]`
  … denotes a nullable reference further constrained to `Point`

meta-programs everywhere

- API bundles (Panama binder)
- method bundles (value type hashCode, datum toString)
- linkable instructions (indy, ldc/condy)
- template classes (reified generics)
- names (type operators)



QUESTIONS?

The previous is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.