WebAssembly as the Basis of Next-Generation Language Runtimes

Ben L. Titzer

SDI Series Talk

2022-02-24

My Background with Language implementation

- 1993-1996: dreams of making own language
- 1998: learned Java
- 2000: first Java virtual machine experience
- 2000-2003: OpenVM: a Java-in-Java research project at Purdue
- 2002-2003: internships at Sun with HotSpot (now OpenJDK)
- 2002-2022: Virgil programming language
- 2004-2006: Avrora, an AVR emulator
- 2006: internship at IBM with J9
- 2007-2010: MaxineVM: a Java-in-Java research project at Sun Labs
- 2013-2019: V8 JavaScript engine, TurboFan optimizing JIT
- 2014-2019: WebAssembly, engine in V8
- 2020-2022 : Wizard engine, Jawa prototype

My Background with Language implementation

- 1993-1996: dreams of making own language
- 1998: learned Java
- 2000: first Java virtual machine experience
- 2000-2003: OpenVM: a Java-in-Java research project at Purdue
- 2002-2003: internships at Sun with HotSpot (now OpenJDK)
- 2002-2022: Virgil programming language
- 2004-2006: Avrora, an AVR emulator
- 2006: internship at IBM with J9
- 2007-2010: MaxineVM: a Java-in-Java research project at Sun Labs
- 2013-2019: V8 JavaScript engine, TurboFan optimizing JIT
- 2014-2019: WebAssembly, engine in V8
- 2020-2022 : Wizard engine, Jawa prototype

WebAssembly Early Days

- 2011: Emscripten/Asm.js: compile C/C++ to JavaScript subset
- 2011: Native Client: compile C/C++ to verifiable (sandboxed) x86 subset
- 2013: Enter V8 collaboration with Spidermonkey
- 2014: early (non-public) prototyping with Luke Wagner
- 2015: Other browser vendors brought in through engineer-to-engineer collab
- 2015: announcement and formation of W3C Community Group
- 2015-2017: collab and open design

WebAssembly Early Goals

- 2011: Emscripten/Asm.js: compile C/C++ to JavaScript subset
- 2011: Native Client: compile C/C++ to verifiable (sandboxed) x86 subset

=>

Improve upon asm.js and Native Client as a target for native code compiled to the web!

What is Wasm?

- A portable, low-level bytecode format
- Optimized for competing with native code
- Fast to decode, parse, and validate
- Formal specification with mechanized proofs of safety
- VMSS 2016 video “A little on V8 and WebAssembly” https://www.youtube.com/watch?v=BRNxM8szTPA
- CurryOn 2018 video “WebAssembly, Past, Present, and Future” https://www.youtube.com/watch?v=WTxPqDBo8CI
- PLISS 2019 video “A little (more) on V8 and WebAssembly” https://www.youtube.com/watch?v=-JmB9MuTc38

WebAssembly 2017-2022

- Lots of performance, engine optimizations
- More applications on the web!
- Blockchains, crypto, digital contracts, edge computing
- Features: multi-value, reference types, bulk memory
- In-flight designs:
    - Exceptions
    - Tail calls
    - Function references
    - Garbage collection
    - Interface types
    - Stack switching

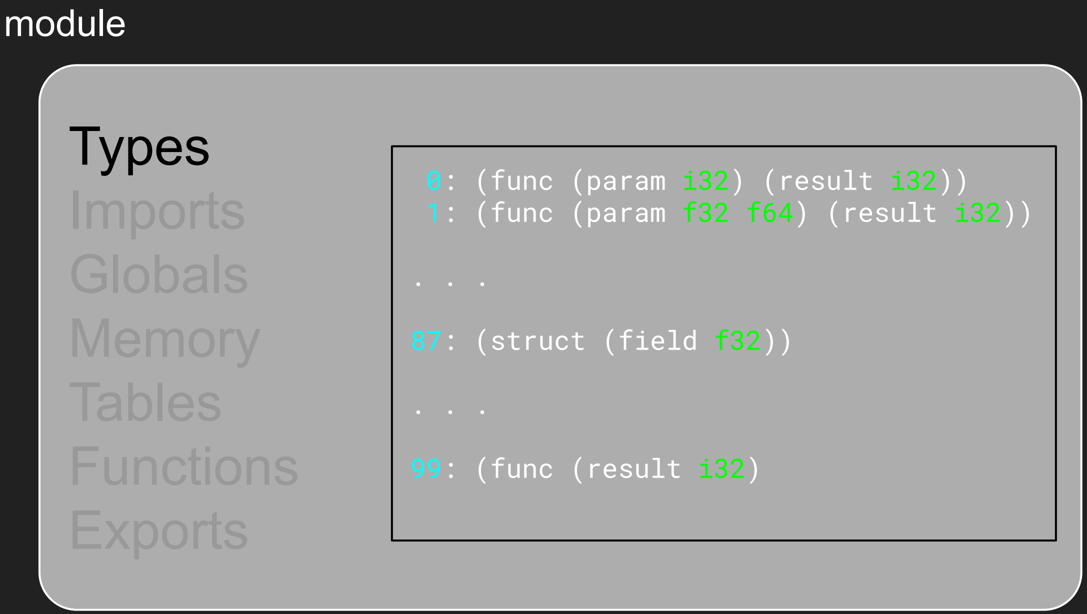
Wasm Example

module

Types
Imports
Globals
Memory
Tables
Functions
Exports

- Unit of code is a *module*
- Like an “executable file” and not a program
- *Instantiate* to get a instance (program)
- Must be *validated* (typechecked)

Wasm Example

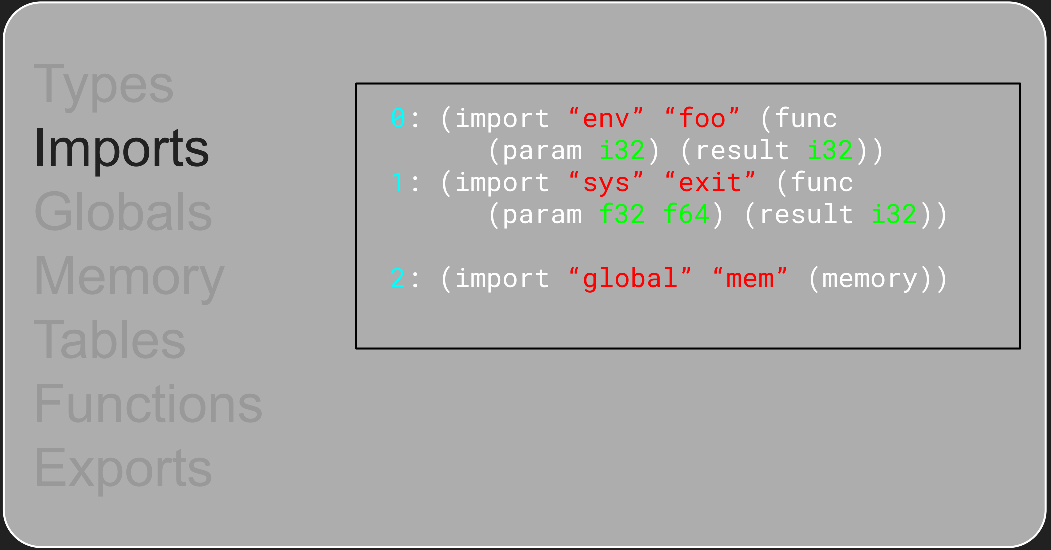


```
 0: (func (param i32) (result i32))
 1: (func (param f32 f64) (result i32))

. . .

87: (struct (field f32))

. . .

99: (func (result i32)
```

- Types section defines
  - Function signatures
  - structs (extension)
  - arrays (extension)
  - exceptions (extension)

Wasm Example

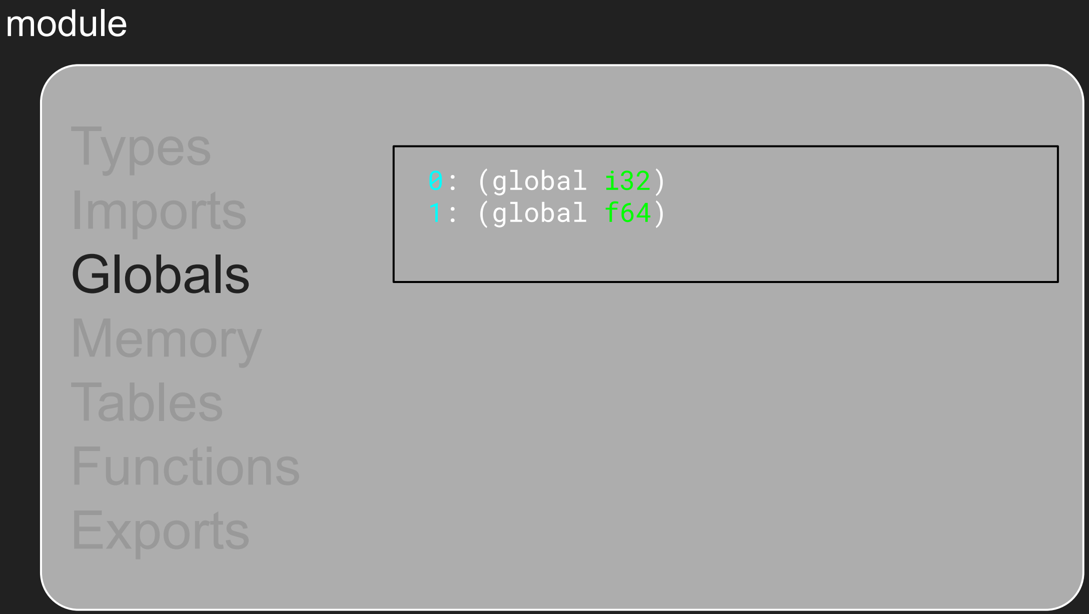
module



```
0: (import “env” “foo” (func
         (param i32) (result i32))
1: (import “sys” “exit” (func
         (param f32 f64) (result i32))

2: (import “global” “mem” (memory))
```

- Imports section denotes
  - Functions
  - Globals
  - Memory
  - Tables
- Imports are named and constrained by an expected type

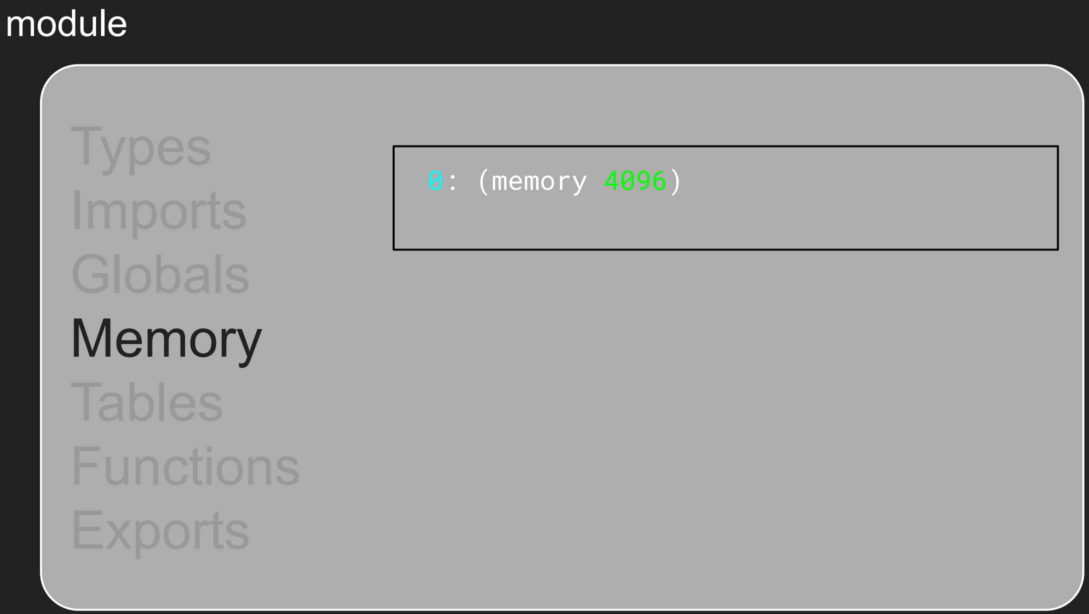
Wasm Example



```
0: (global i32)
1: (global f64)
```

- Global section declares statically-typed, possibly immutable global variables

Wasm Example



```
0: (memory 4096)
```

- Memory section declares (possibly more than one) memory with size in pages

Wasm Example

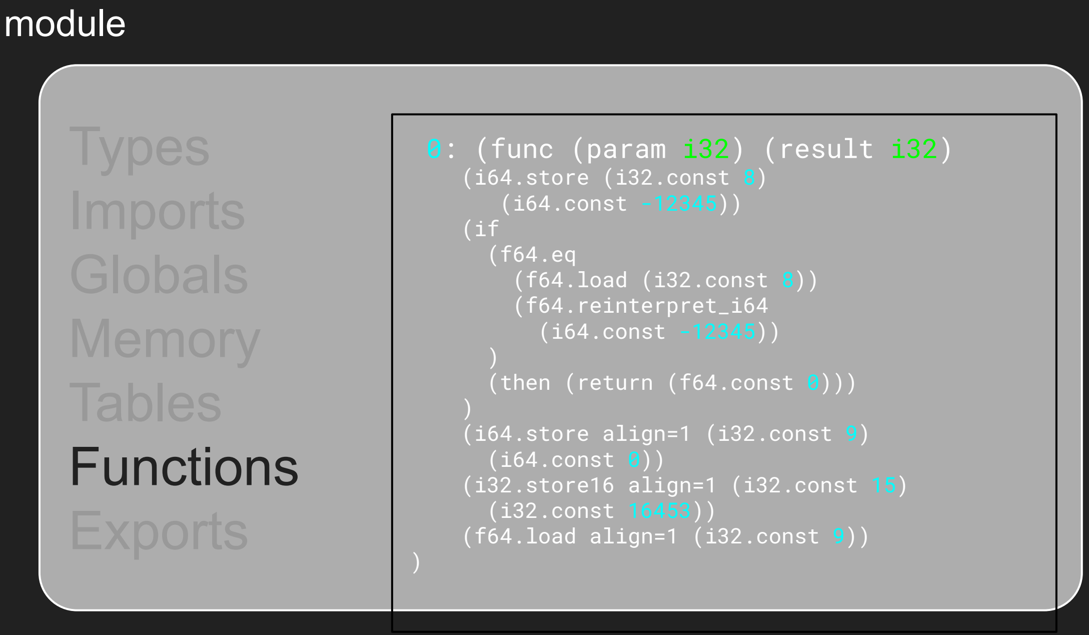
module

Types
Imports
Globals
Memory
Tables
Functions
Exports

```
0: (table func 0 89)
```

- Table section declares statically typed tables that can store (e.g. function references)

Wasm Example



```
0: (func (param i32) (result i32)
     (i64.store (i32.const 8)
        (i64.const -12345))
     (if
       (f64.eq
         (f64.load (i32.const 8))
         (f64.reinterpret_i64
           (i64.const -12345))
       )
       (then (return (f64.const 0)))
     )
     (i64.store align=1 (i32.const 9)
       (i64.const 0))
     (i32.store16 align=1 (i32.const 15)
       (i32.const 16453))
     (f64.load align=1 (i32.const 9))
)
```

- Functions are statically typed, fixed arguments, filled with stack-machine bytecode
- Instructions for arithmetic with i32, i64, f32, f64
- Load and store from memory, tables
- Direct and indirect calls

Wasm Example

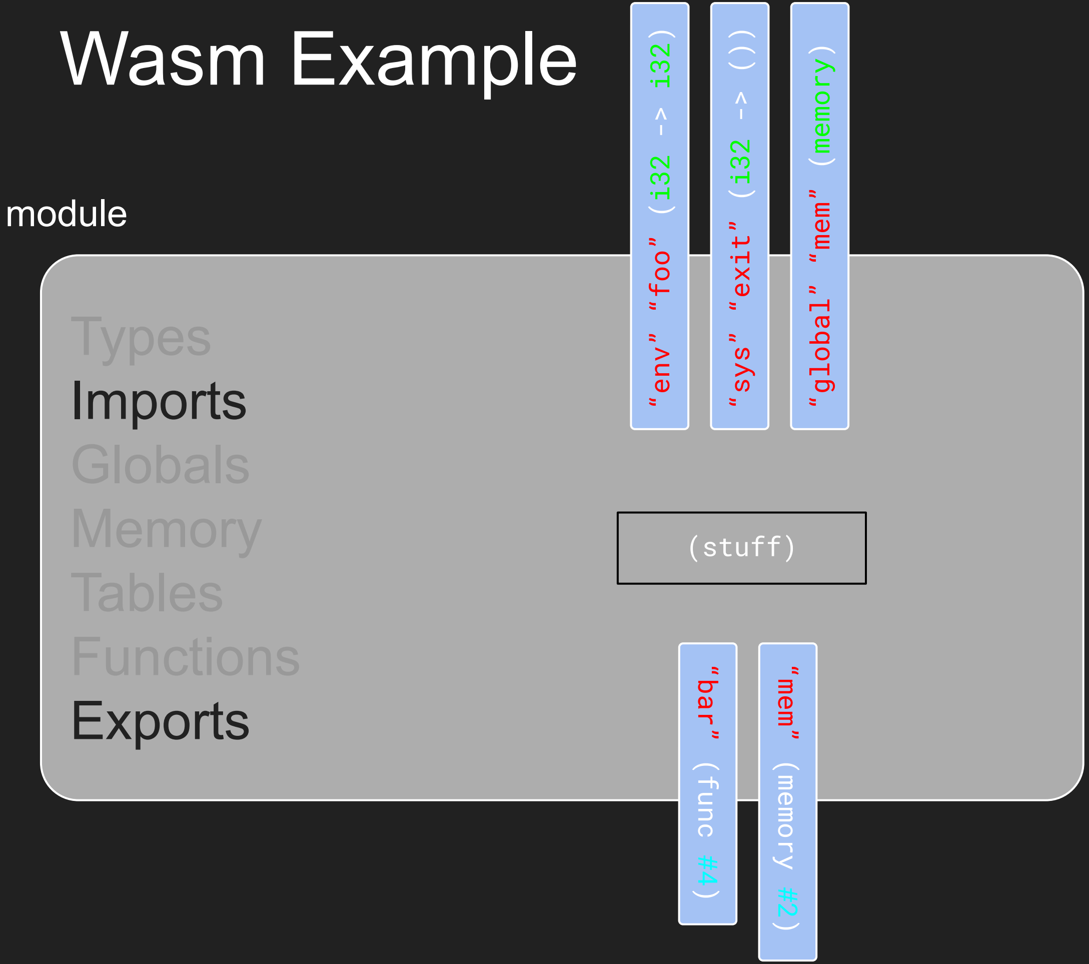
module

Types
Imports
Globals
Memory
Tables
Functions
Exports

```
0: (export “bar” (func #4))
1: (export “mem” (memory #2))
```

- Export section declares any internal or imported definition to be exported from the module, with a name

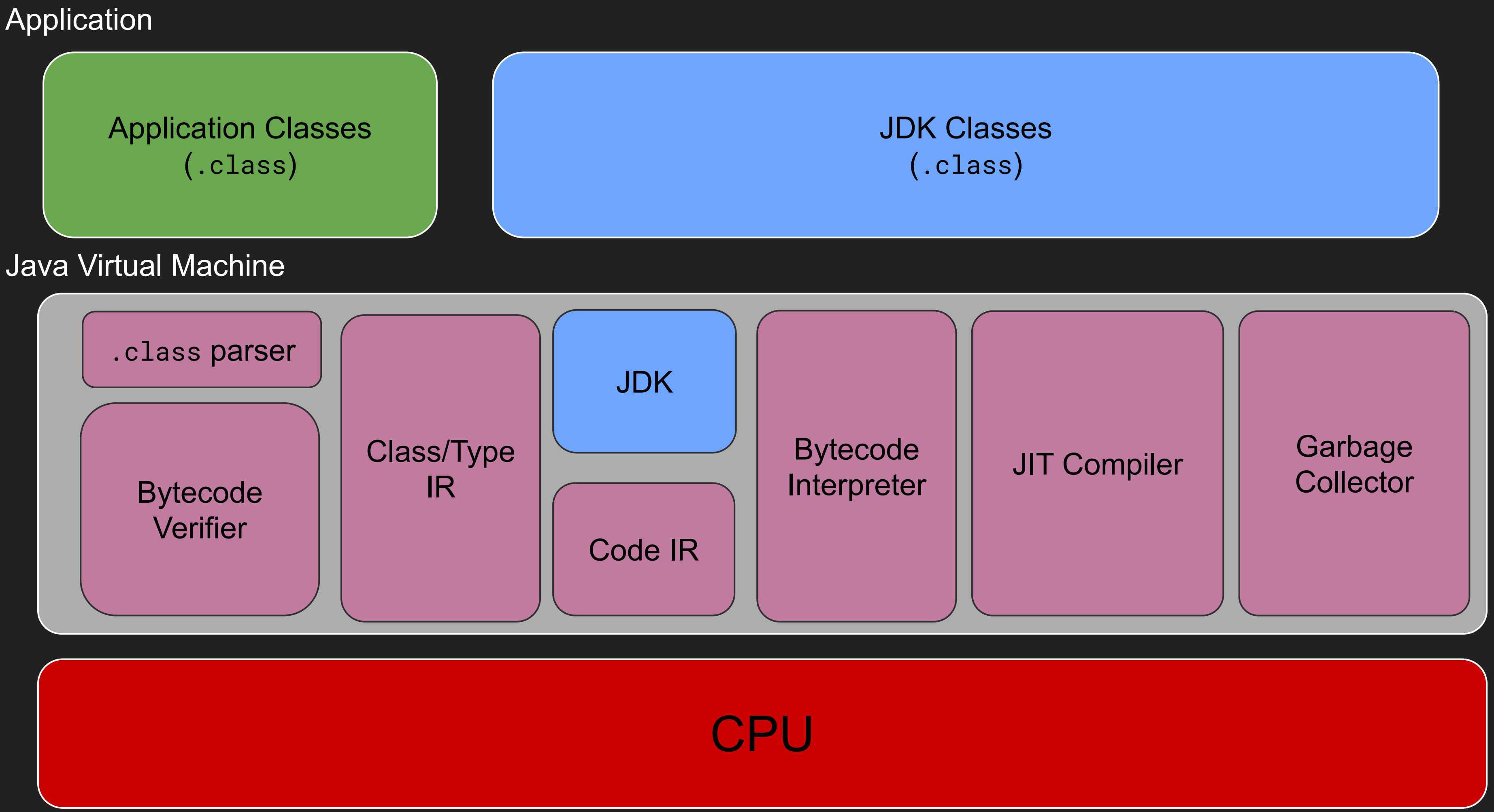
Wasm Example

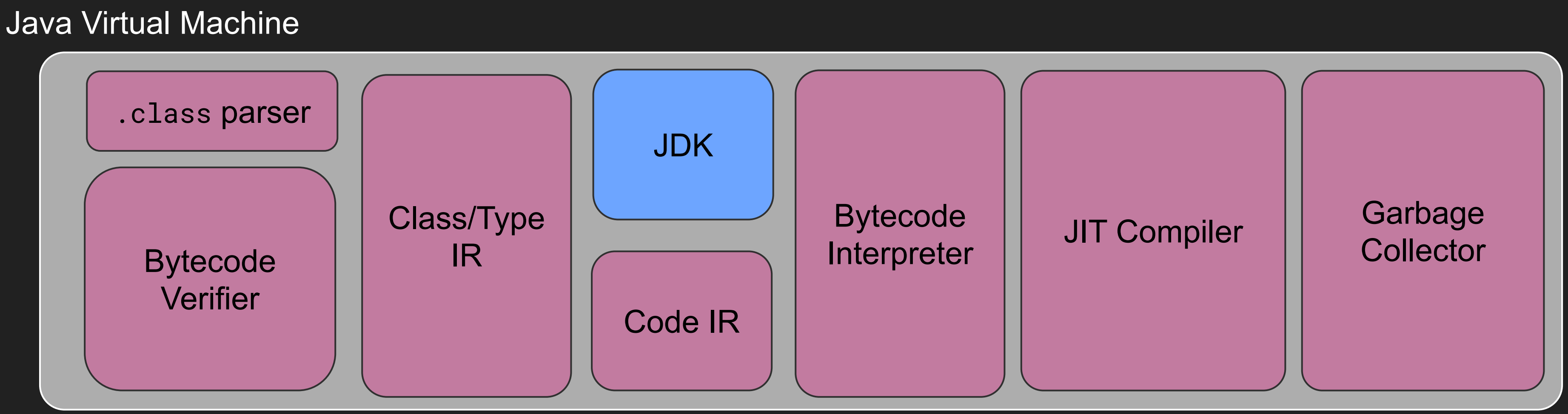


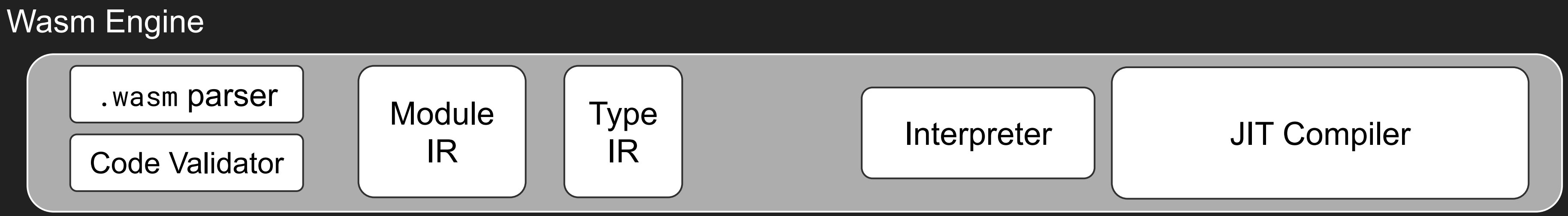
- Modules from the outside pretty much look like a set of imports and exports, with stuff in-between
- "Stuff" only depends on imports and core wasm
- Engine understands "stuff" but not externals

Building a High-performance Runtime

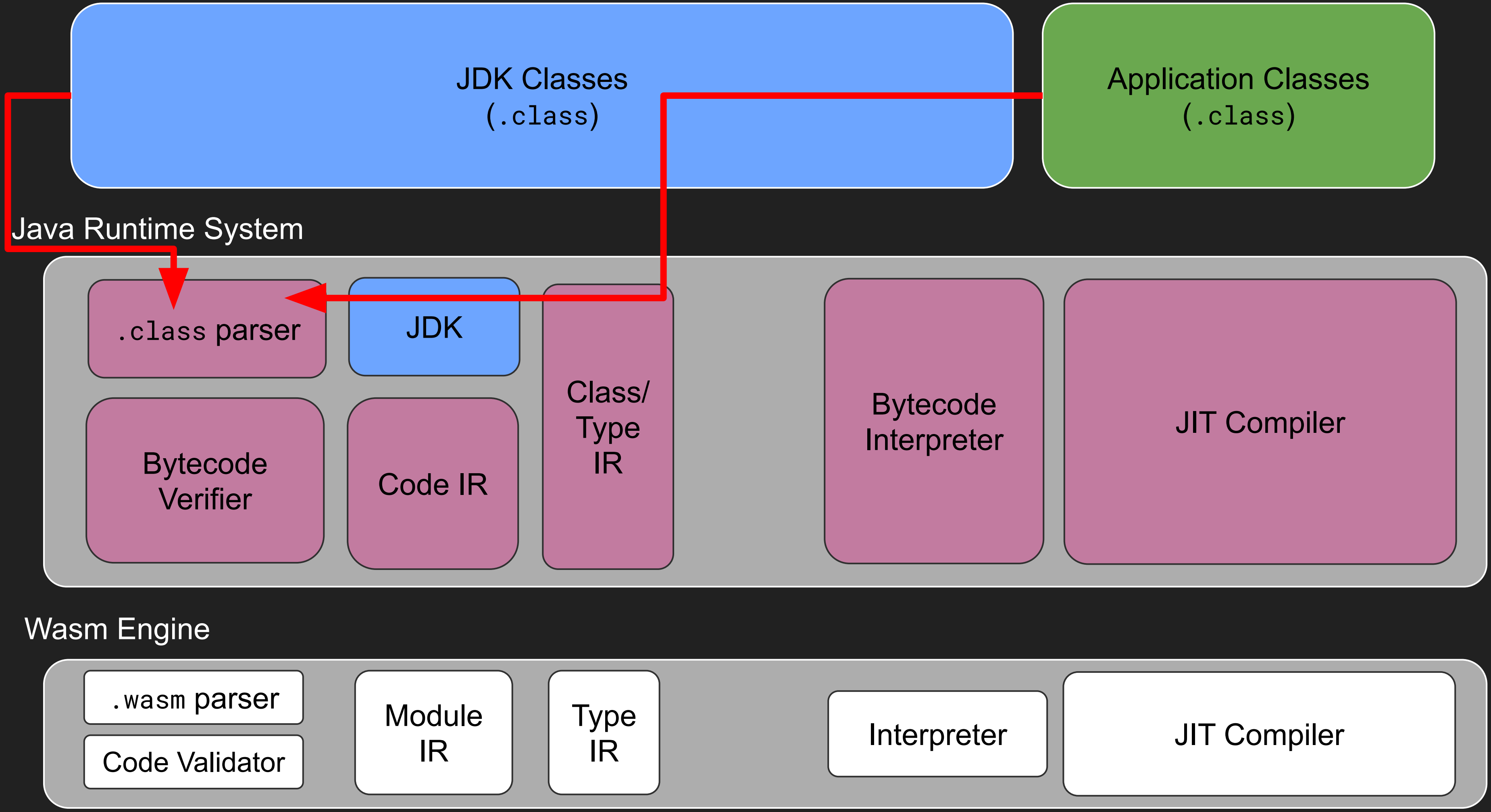
- Fast parsing/loading of code
- Efficient data structures in the middle
- Appropriate execution tiers (interpreter, JITs)
- High-performance garbage collector(s)

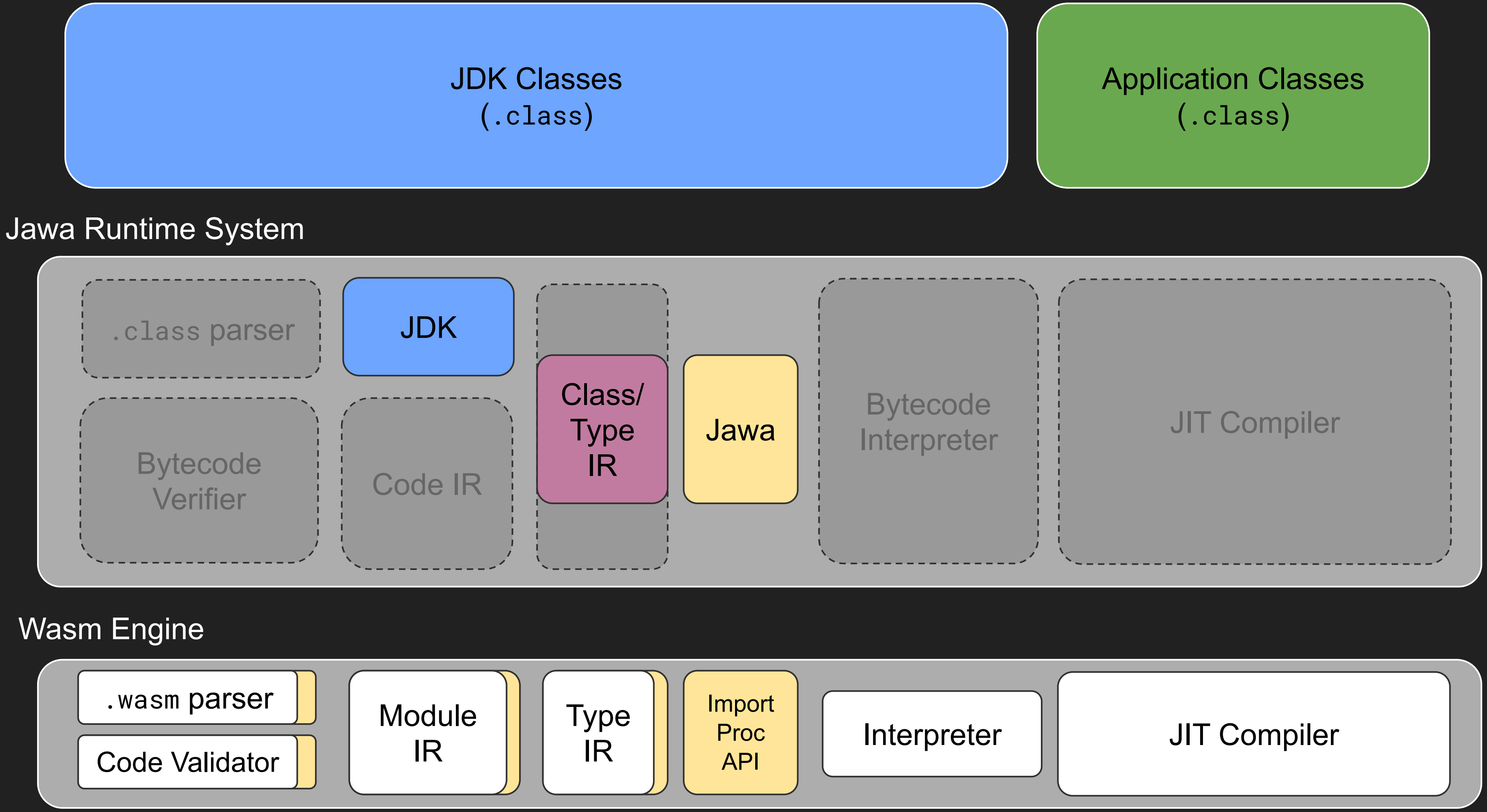

Application
Application Classes
(.class)
JDK Classes
(.class)
Java Virtual Machine
.class parser
Bytecode Verifier
Class/Type IR
JDK
Code IR
Bytecode Interpreter
JIT Compiler
Garbage Collector
CPU


Java Virtual Machine
.class parser
Bytecode Verifier
Class/Type IR
JDK
Code IR
Bytecode Interpreter
JIT Compiler
Garbage Collector


Wasm Engine
.wasm parser
Code Validator
Module IR
Type IR
Interpreter
JIT Compiler

Layered Runtime Systems


JDK Classes
(.class)
Application Classes
(.class)
Java Runtime System
.class parser
JDK
Class/ Type IR
Bytecode Interpreter
JIT Compiler
Bytecode Verifier
Code IR
Wasm Engine
.wasm parser
Code Validator
Module IR
Type IR
Interpreter
JIT Compiler


JDK Classes
(.class)
Application Classes
(.class)
Jawa Runtime System
.class parser
JDK
Class/
Type
IR
Jawa
Bytecode
Interpreter
JIT Compiler
Bytecode
Verifier
Code IR
Wasm Engine
.wasm parser
Code Validator
Module
IR
Type
IR
Import
Proc
API
Interpreter
JIT Compiler

Java to Wasm (GC): the obvious

- Essentially all primitive arithmetic maps 1-to-1 to Wasm
- Objects represented as `struct` instances
- Instance fields become `struct` fields, i.e. single indirection load/store
- Objects have a “user-level header” pointing to meta-object with vtable
- Virtual dispatch sequence consist of load of header, load of method, then `call.ref`
- Static fields/methods can become `global` variables

Java to Wasm (GC): the less obvious

- Interface dispatch requires linear search
  - I-tables duplicated per implemented interface (JVM shares them)
  - Can be faster if interface references represented as tuple (object, i-table)
- Wasm `struct` cast should imply Java-level cast
  - RTTs need to be generative to encode nominal hierarchy
- No variance for arrays =>
  - `array` of java/lang/Object and cast on get
- Annotated `ref.null` instruction =>
  - No bottom (`null`) type for Jawa, need to import one
- More space overhead in object model than anticipated

Java to Wasm: the totally surprising

- Late binding ruins everything
  - Field offset resolution
  - Field name resolution
  - Method overloading/overriding
  - Depth in class hierarchy
  - Implemented interfaces
- Source-level types and names required, not just for reflection

Late-binding problems in Java

```java
// file: A.java
public class A {
    public int x;
    public short y;
    public void m(A a) { ... }
}
```

```java
// file: B.java
public class B extends A {
    public void m(A a) {
        this.x += a.x;
        this.y = a.y;
    }
}
```

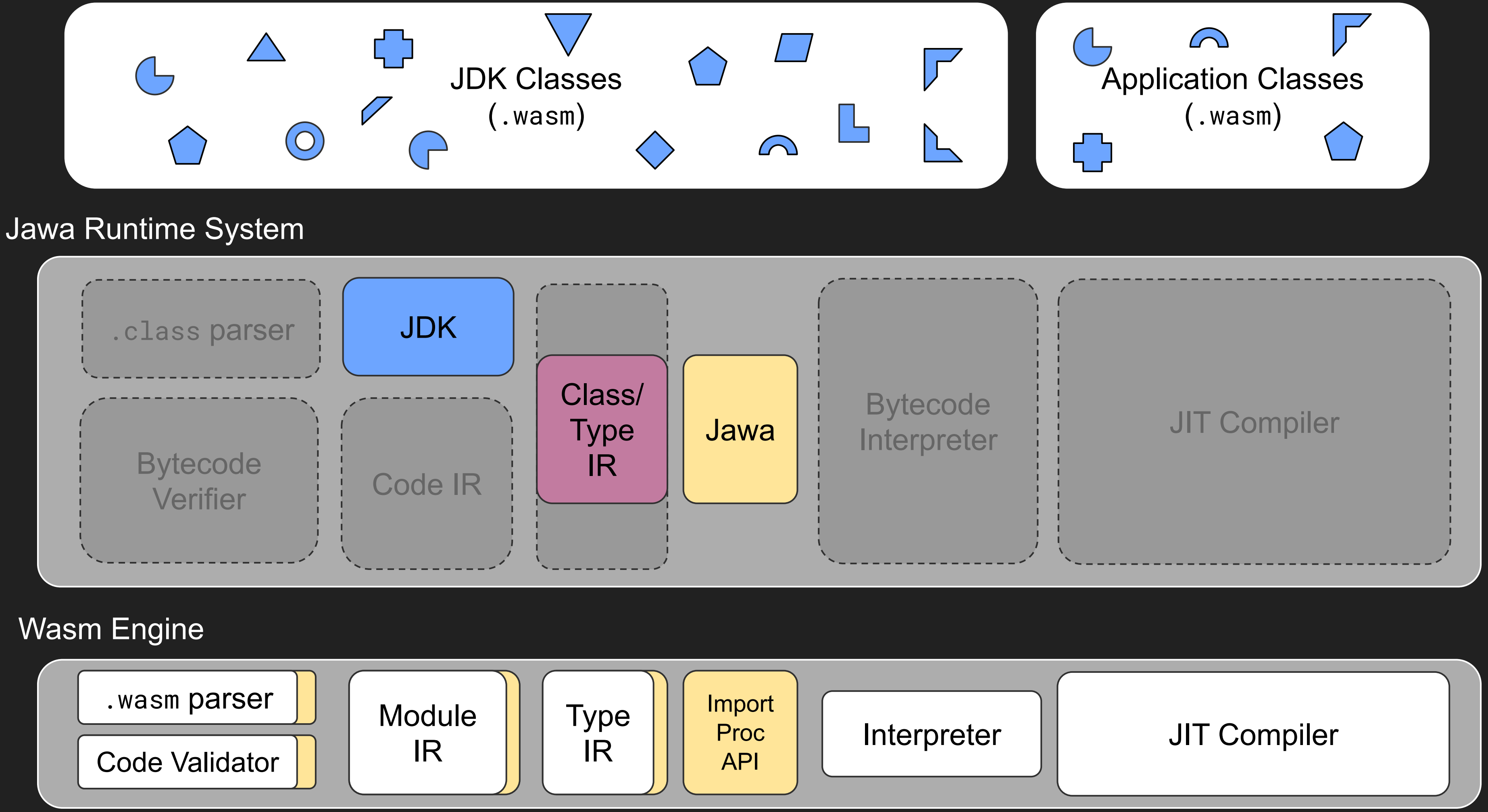
Change to A.class	**Outcome**
Add field	✅ No error
Change declaration order	✅ No error
`short y` ➜ `char y`	`NoSuchFieldError` upon access
`short y` ➜ `short z`	`NoSuchFieldError` upon access
`short y` ➜ `static short y`	`IncompatibleClassChangeError` upon access
`void m(A a)` ➜ `int m(A a)`	`B.m` silently no longer overrides

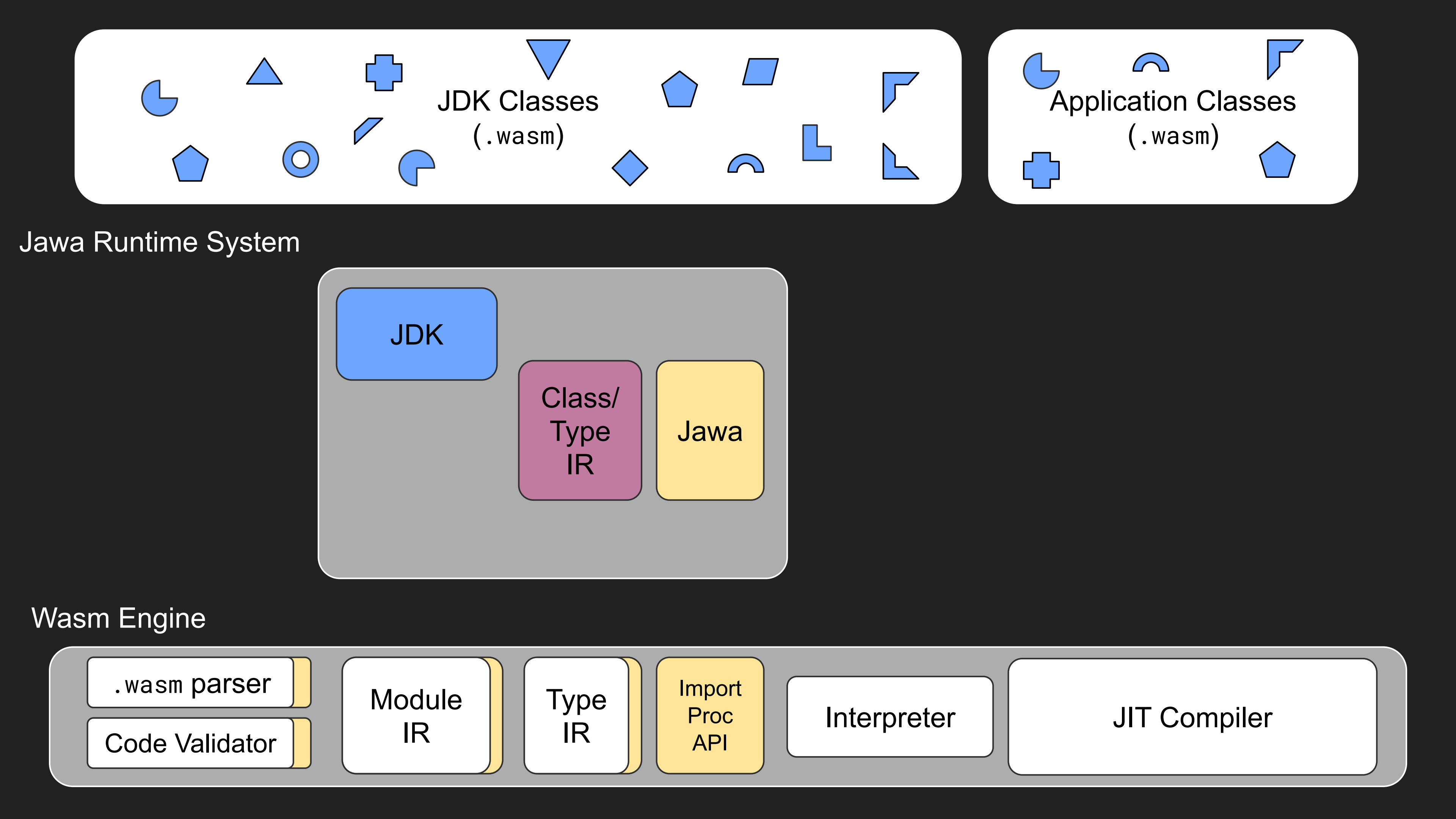
Solving the Late-Binding Problem

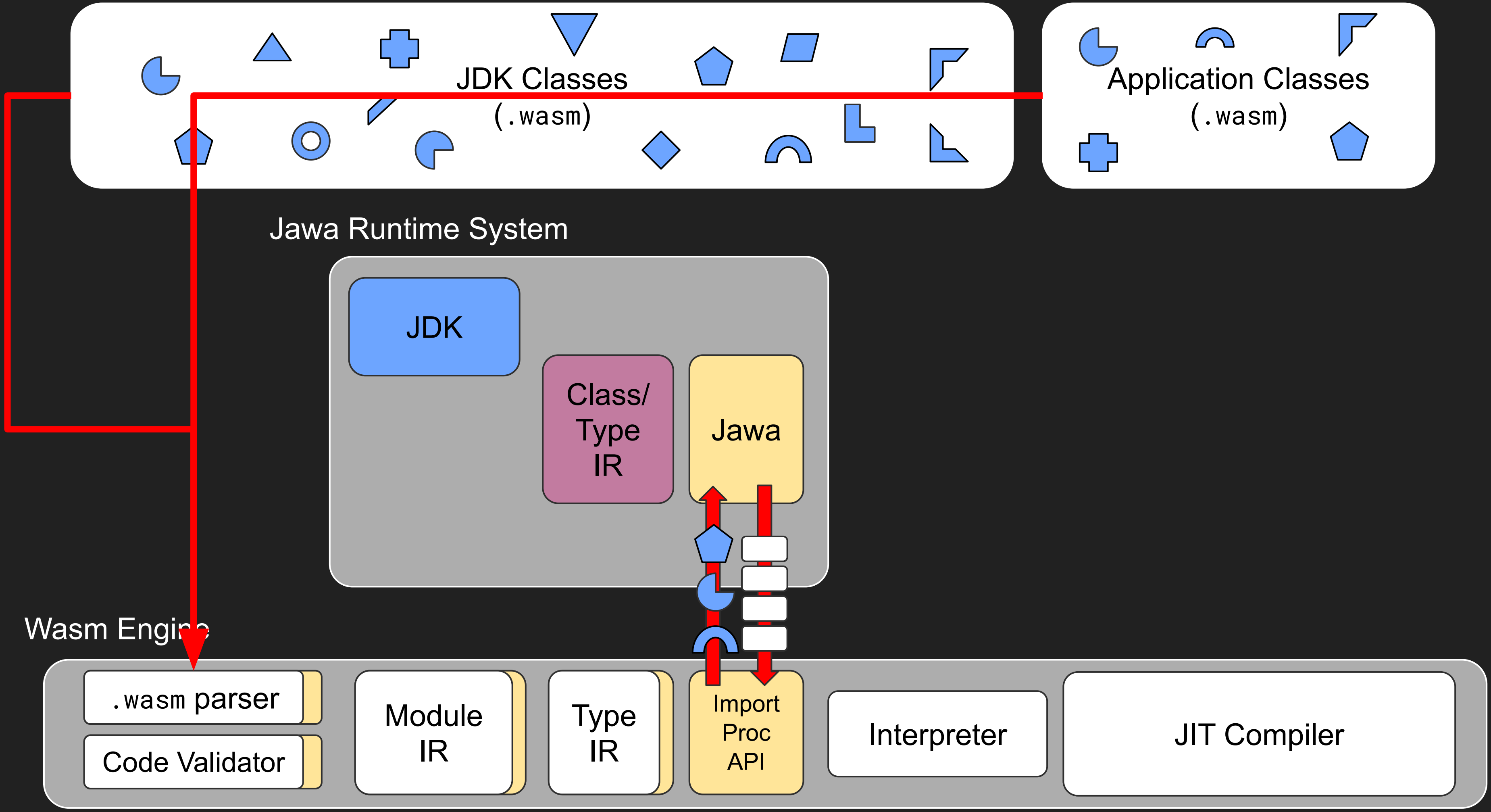
- Java late binding (i.e. across modules) require full source-level names and types
- A runtime system with centralized information about (incrementally) loaded modules is required to process source names
  - I.e. no “name mangling” and “name matching” system fits all
- Lowering early (i.e. statically) cannot in general solve this problem, no matter how complicated (and inefficient) an object model it uses (e.g. arrays of arrays)

The Jawa Approach

- Embrace the source-level information!
- Provide a way to encode *all* relevant source-level information in Wasm binaries
- A Runtime system performs late binding
- Runtime system reuses as much as possible from the Wasm engine:
  - Code format
  - Code loading
  - Code verification
  - Execution tiers
- No more tiers! Don’t reintroduce a new bytecode “one level up” and compile an interpreter and JIT--too complex overall


JDK Classes
(.wasm)
Application Classes
(.wasm)
Jawa Runtime System
.class parser
JDK
Class/ Type IR
Jawa
Bytecode Interpreter
JIT Compiler
Bytecode Verifier
Code IR
Wasm Engine
.wasm parser
Code Validator
Module IR
Type IR
Import Proc API
Interpreter
JIT Compiler


JDK Classes
(.wasm)
Application Classes
(.wasm)
Jawa Runtime System
JDK
Class/ Type IR
Jawa
Wasm Engine
.wasm parser
Code Validator
Module IR
Type IR
Import Proc API
Interpreter
JIT Compiler


JDK Classes
(.wasm)
Application Classes
(.wasm)
Jawa Runtime System
JDK
Class/
Type
IR
Jawa
Wasm Engine
.wasm parser
Code Validator
Module
IR
Type
IR
Import
Proc
API
Interpreter
JIT Compiler

Encoding Java (.class files) into Wasm Imports

- Type imports
  - External classes (by name)
  - Primitive arrays
  - Reference arrays (apply type constructor, by imported type index)
  - Forward-declare classes
- Function imports
  - Missing floating point operators
  - All “complex” bytecodes dealing with object model
- Global imports
  - String constants, class object constants
- Command imports
  - Finish class/interface definitions

Jawa Type Imports (1)

- Import *abstract* types from a special “jawa” module using our enhanced type import mechanism.
    - Encode source-level names as part of the import (field) name.

```
(import "jawa" "EXT_CLASS[jawa/lang/Object]" (type $jlo))
```

Jawa Type Imports (2)

- Use multiple subtype constraints to express this module’s view (or requirements) of the imported Java type.

```
(import "jawa" "EXT_INTERFACE[jawa/lang/Cloneable]" (type $jlcl (sub $jlo)))

(import "jawa" "EXT_CLASS[jawa/lang/String]" (type $jls (sub $jlo $jlcl)))
```

Jawa Type Imports (3)

- Import primitive arrays individually (finite set)

```
(import "jawa" "BYTE_ARRAY" (type $ab) (sub $jlo)))
```

- Use type *arguments* to apply type constructors, e.g. arrays

```
(import "jawa" "REF_ARRAY" <$jls> (type $at (sub $jlo))
```

Jawa Type Imports (4)

- Forward-declare classes/interfaces that we will *define* later
  - Provide superclass and implemented interfaces as type arguments

```
(import "jawa" "DECL_CLASS[my/Name]" <$sc,$its...>

      (type $ct (sub $sc $its...))
```

Jawa Type Import Summary

- All source information to describe the Java type is encoded into an import
- Different encodings for different kinds of Java types
- Multiple type constraints for expressing Java's multiple-inheritance of interfaces
- In general, abstract type imports have *unspecified* representation
    - Complete implementation freedom, e.g. to use pair for interface references
    - In practice, imported Jawa types are reference types

Jawa Global Imports (1)

- Import java/lang/String constants as immutable globals
  - Encode the *value* of the string into the import name

```
(import "jawa" "STRING_CONST[value]" (global $g ($jls))
```

- Import java/lang/Class constants as immutable globals
  - Encode the *name* of the class into the import name

```
(import "jawa" "CLASS_CONST[jawa/lang/Object]" (global $g ($jlc))
```

Jawa Function Imports (1)

- Import the missing floating point operations as functions

```
(import "jawa" "DCMPG" (func (param f64 f64) (result i32))

(import "jawa" "DCMPL" (func (param f64 f64) (result i32))

(import "jawa" "DREM" (func (param f64 f64) (result f64))

(import "jawa" "FCMPG" (func (param f32 f32) (result i32))

(import "jawa" "FCMPL" (func (param f32 f32) (result i32))

(import "jawa" "FREM" (func (param f32 f32) (result f32))
```

Jawa Function Imports (2)

- Import array allocation bytecodes as functions

```
(import "jawa" "NEWARRAY" <$at> (func (param i32) (result $at))

(import "jawa" "ANEWARRAY" <$at> (func (param i32) (result $at))

(import "jawa" "MULTIANEWARRAY[dims]" <$at>

     (func (param i32 x dims) (result $at))
```

Jawa Function Imports (3)

- Import array access bytecodes as functions

```
(import "jawa" "primALOAD" (func (param ($aprim i32) (result prim))

(import "jawa" "primASTORE" (func (param $aprim i32 prim))

(import "jawa" "AALOAD" <$at> (func (param $at i32) (result $at.elem))

(import "jawa" "AASTORE" <$at> (func (param $at i32 $at.elem))
```

Jawa Function Imports (4)

- Import cast/instanceof bytecodes as functions

```
(import "jawa" "CHECKCAST" <$t> (func (param $jlo) (result $t))

(import "jawa" "INSTANCEOF" <$t> (func (param $jlo) (result i32))
```

Jawa Function Imports (5)

- Import general reference manipulations as functions

```
(import "jawa" "MONITORENTER" (func (param $jlo))
(import "jawa" "MONITOREXIT" (func (param $jlo))
(import "jawa" "ISNULL" (func (param $jlo) (result i32))
(import "jawa" "ACMPEQ" (func (param $jlo $jlo) (result i32))
```

- Import the throw operation (takes java/lang/Throwable)

```
(import "jawa" "ATHROW" (func (param $jlt))
```

Jawa Function Imports (6)

- Import object allocation/field access as functions

```
(import "jawa" "NEW" <$ct> (func (result $ct))

(import "jawa" "GETFIELD[field.name]" <$ct>

     (func (param $ct) (result field.type))

(import "jawa" "PUTFIELD[field.name]" <$ct> (func (param $ct field.type))

(import "jawa" "GETSTATIC[field.name]" <$ct> (func (result field.type))

(import "jawa" "PUTSTATIC[field.name]" <$ct> (func (param field.type))
```

Jawa Function Imports (7)

- Import method call bytecodes as functions

```
(import "jawa" "INVOKEINTERFACE[name,sig]" <$it>

    (func (param $it sig.params) (result sig.result))

(import "jawa" "INVOKESPECIAL[name,sig]" <$ct>

    (func (param $ct sig.params) (result sig.result))

(import "jawa" "INVOKESTATIC[name,sig]" <$ct>

    (func (param sig.params) (result sig.result))

(import "jawa" "INVOKEVIRTUAL[name,sig]" <$ct>

    (func (param $ct sig.params) (result sig.result))
```

Jawa Command Imports (1)

- Still not done! Need to finish class definitions

```
        (import "jawa" "DEFINE_CLASS[attributes,fields,methods]"
            <$ct,$types,$funcs> (command))
```

- A *command import* is a side-effecting operation (no return value)
- The entire class body is encoded entirely into the name
- External types are type arguments
- Wasm functions define the classes' method bodies are import arguments

Jawa Import Summary

- All imports can be topologically sorted for single-pass verification
- Requires relaxed section order because:
    - Imports of Jawa types must occur before Wasm types section
    - Must define Wasm signatures/functions to define classes
    - Classes must be defined before bytecodes using their fields/methods
- Requires generalized import args because:
    - Java reference arrays are *compound*
    - Java bytecodes are parameterized by types
    - We use Wasm functions to define the bodies of Java classes

$$\begin{tikzcd}
\text{Jawa Types} \arrow[d] \\
\text{Wasm Signatures} \arrow[d] \\
\text{Wasm Functions} \arrow[d] \\
\text{Jawa Classes} \arrow[d] \\
\text{Jawa Bytecodes} \arrow[d] \\
\text{Function Bodies}
\end{tikzcd}$$

Wutudeee!

What does Jawa accomplish?

- All Java source-level constructs present in class files encoded into Wasm
    - Java types exposed as a checkable nominal subtyping lattice
    - Java bytecodes expressed as function imports: effectively a Wasm bytecode extension
    - Source-level signatures and names encoded without mangling for binding, reflection
    - Effectively a replacement for class files that is understood by a Wasm engine
- Defer the lowering of types and operations until later (e.g. runtime)
- Separate producer from implementation details
- Create an opportunity for a runtime system to be simpler than it otherwise would be
    - No code loader, no code verifier, no execution tiers

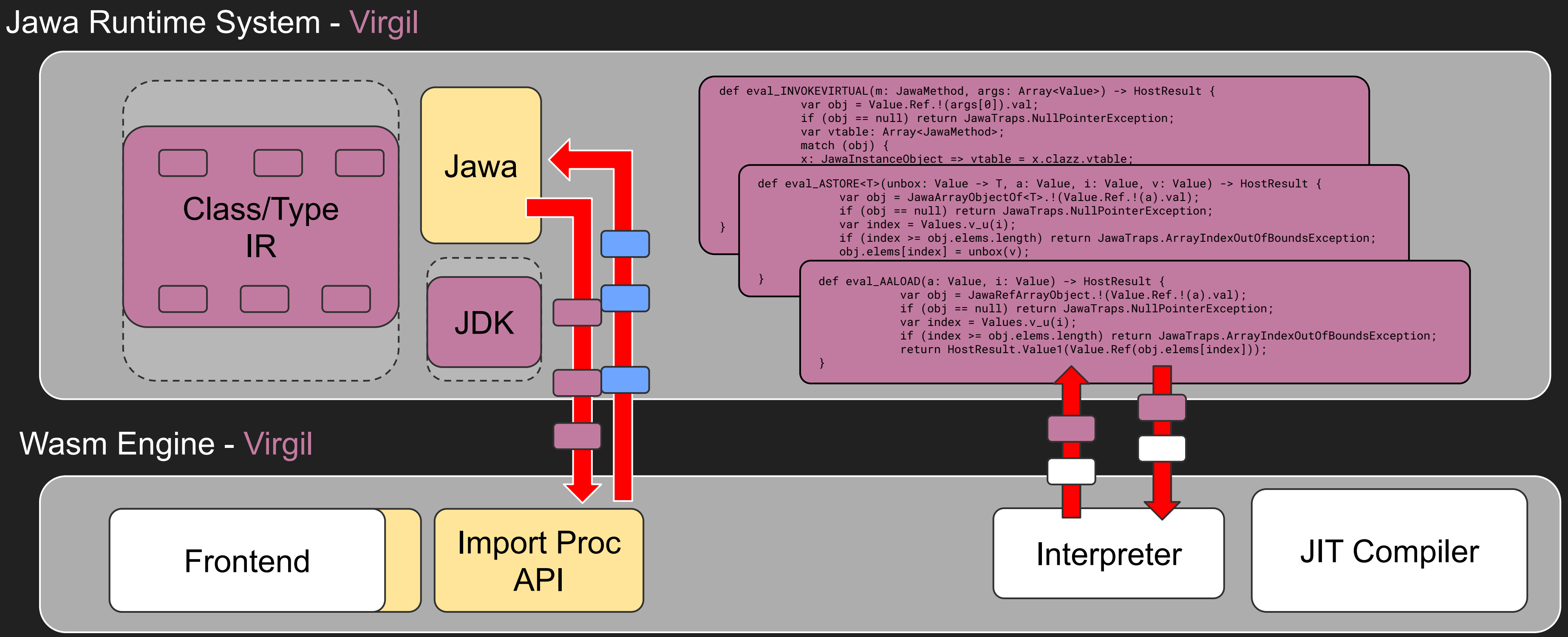
Building the Jawa Runtime System

- Step 0: We built a new Wasm Research VM, the Wizard Engine
  - https://github.com/titzer/wizard-engine/
- While not specifically tied to Jawa, Wizard facilitates research by being simple and easy to extend
  - Written in Virgil, about 7000 lines of code
  - No frills, simple *low-overhead* interpreter
  - Internal data structures designed to eventually support *union* of all proposals
- Wizard Supports
  - ✅ MVP, sign-extension, non-trapping conversions
  - ✅ multi-value, reference-types, bulk-memory
  - ✅ GC, function-references
  - 🔍 Exception-handling (wip)

Five Options for a X-wa Runtime System

1. Host language emulation
    a. Jawa implemented *in* Wizard with Virgil
    b. Pywa implemented *on* a Web Engine using JavaScript
2. Translation to linear memory
3. Translation to Wasm GC
    a. Jawa implemented in Wizard
4. Pass-through emulation
5. Native implementation integrated into an engine
    a. TBD: performance experiment in V8

Host language emulation

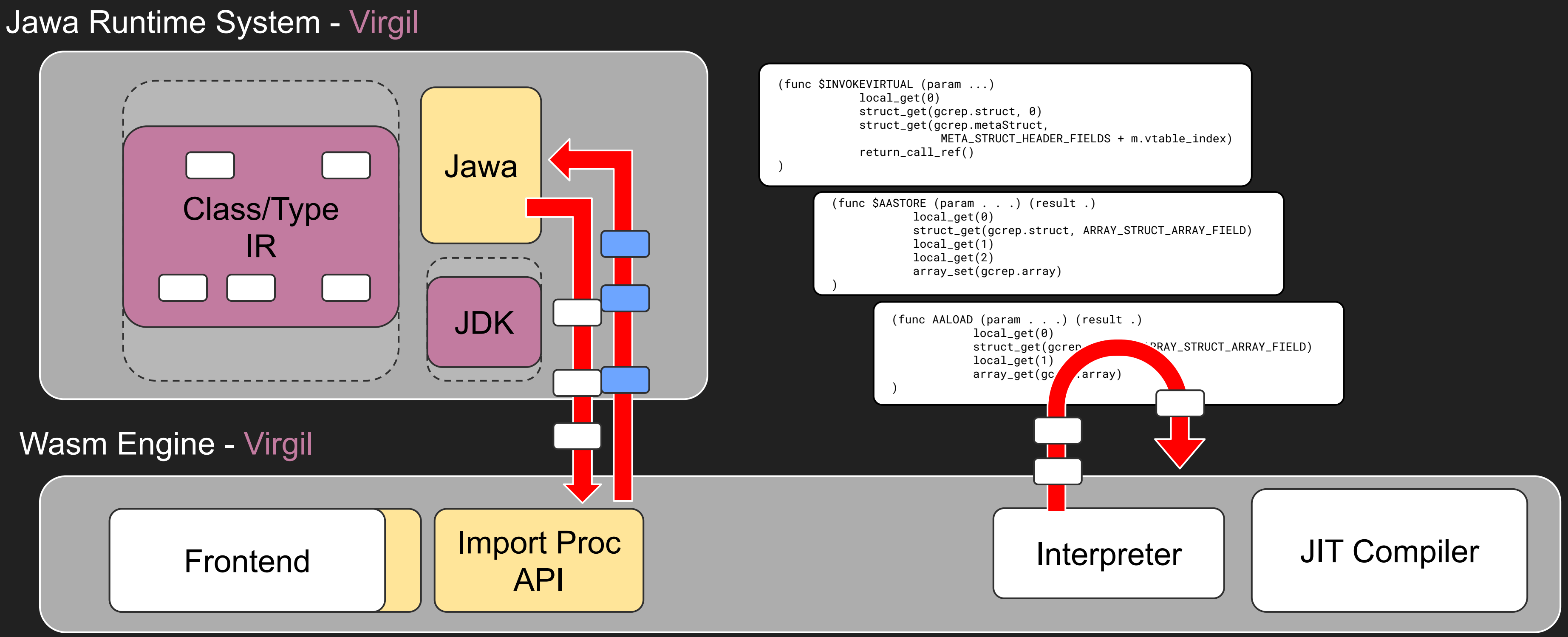


```
def eval_INVOKEVIRTUAL(m: JawaMethod, args: Array<Value>) -> HostResult {
            var obj = Value.Ref.!(args[0]).val;
            if (obj == null) return JawaTraps.NullPointerException;
            var vtable: Array<JawaMethod>;
            match (obj) {
            x: JawaInstanceObject => vtable = x.clazz.vtable;
}
```

```
def eval_ASTORE<T>(unbox: Value -> T, a: Value, i: Value, v: Value) -> HostResult {
            var obj = JawaArrayObjectOf<T>.!(Value.Ref.!(a).val);
            if (obj == null) return JawaTraps.NullPointerException;
            var index = Values.v_u(i);
            if (index >= obj.elems.length) return JawaTraps.ArrayIndexOutOfBoundsException;
            obj.elems[index] = unbox(v);

}
```

```
def eval_AALOAD(a: Value, i: Value) -> HostResult {
            var obj = JawaRefArrayObject.!(Value.Ref.!(a).val);
            if (obj == null) return JawaTraps.NullPointerException;
            var index = Values.v_u(i);
            if (index >= obj.elems.length) return JawaTraps.ArrayIndexOutOfBoundsException;
            return HostResult.Value1(Value.Ref(obj.elems[index]));
}
```

Translation to Wasm [GC]

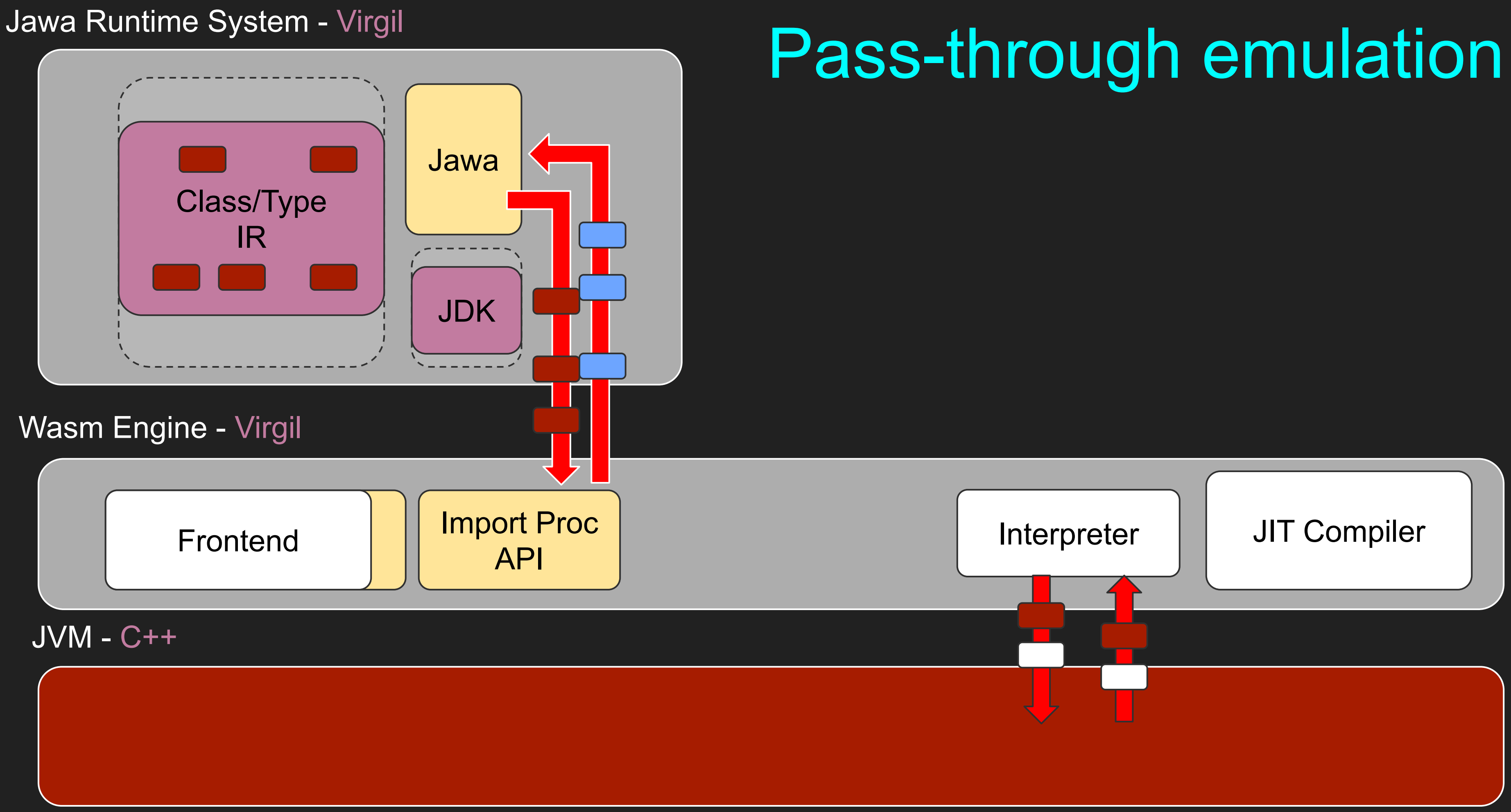


```
(func $INVOKEVIRTUAL (param ...)
          local_get(0)
          struct_get(gcrep.struct, 0)
          struct_get(gcrep.metaStruct,
                    META_STRUCT_HEADER_FIELDS + m.vtable_index)
          return_call_ref()
)
```

```
(func $AASTORE (param . . .) (result .)
          local_get(0)
          struct_get(gcrep.struct, ARRAY_STRUCT_ARRAY_FIELD)
          local_get(1)
          local_get(2)
          array_set(gcrep.array)
)
```

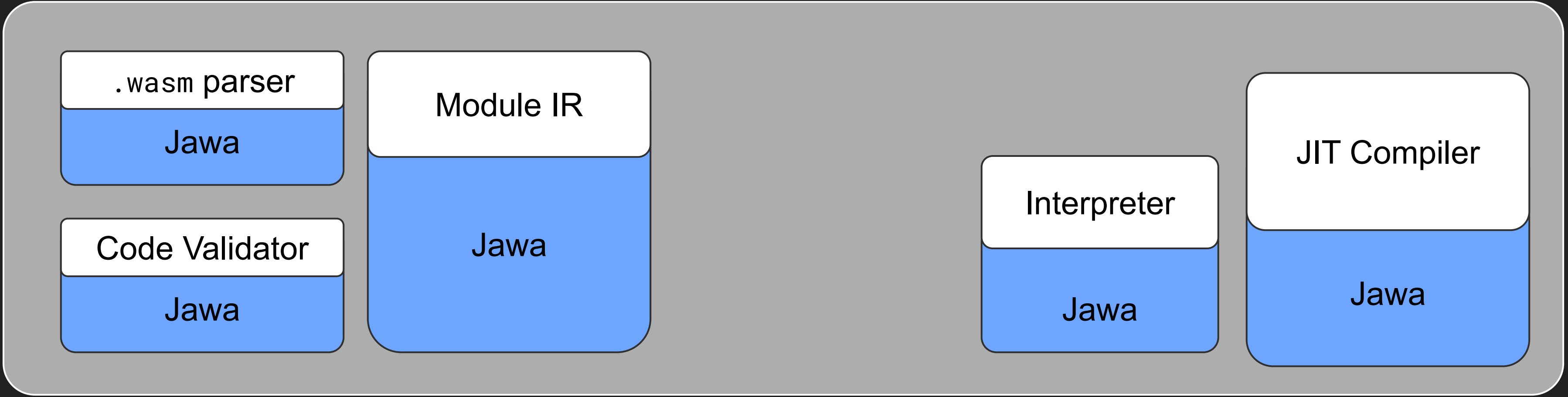
```
(func AALOAD (param . . .) (result .)
          local_get(0)
          struct_get(gcrep        RRAY_STRUCT_ARRAY_FIELD)
          local_get(1)
          array_get(gc    .array)
)
```

Pass-through emulation



Native Implementation

Wasm Engine (Wizard/Virgil, V8/C++)



Pros and Cons

1. **Host language emulation**
2. Translation to linear memory
3. Translation to Wasm GC
4. Pass-through emulation
5. Native implementation

✅ Easy to experiment with

✅ Limited only by host language functionality, not wasm or target machine

✅ Host interoperability

❌ Host call overhead for all bytecode extensions

❌ All added values must be host references

Pros and Cons

1. Host language emulation
2. **Translation to linear memory**
3. Translation to Wasm GC
4. Pass-through emulation
5. Native implementation

✅ Least Wasm engine support

✅ Least controversial

❌ GC root finding still a problem

❌ No host interoperability

❌ Maximum implementation effort

Pros and Cons

1. Host language emulation
2. Translation to linear memory
3. **Translation to Wasm GC**
4. Pass-through emulation
5. Native implementation

✅ Best engine / runtime modularity

✅ Host interoperability possible

✅ Possibility for completely untrusted runtimes through an API

❌ Limited by Wasm GC capabilities

Pros and Cons

1. Host language emulation
2. Translation to linear memory
3. Translation to Wasm GC
4. **Pass-through emulation**
5. Native implementation

✅ Not limited by Wasm or host language capabilities

✅ Host interoperability possible

❌ Must be trusted engine component

❌ Only works in narrow circumstances

❌ May still have call overhead

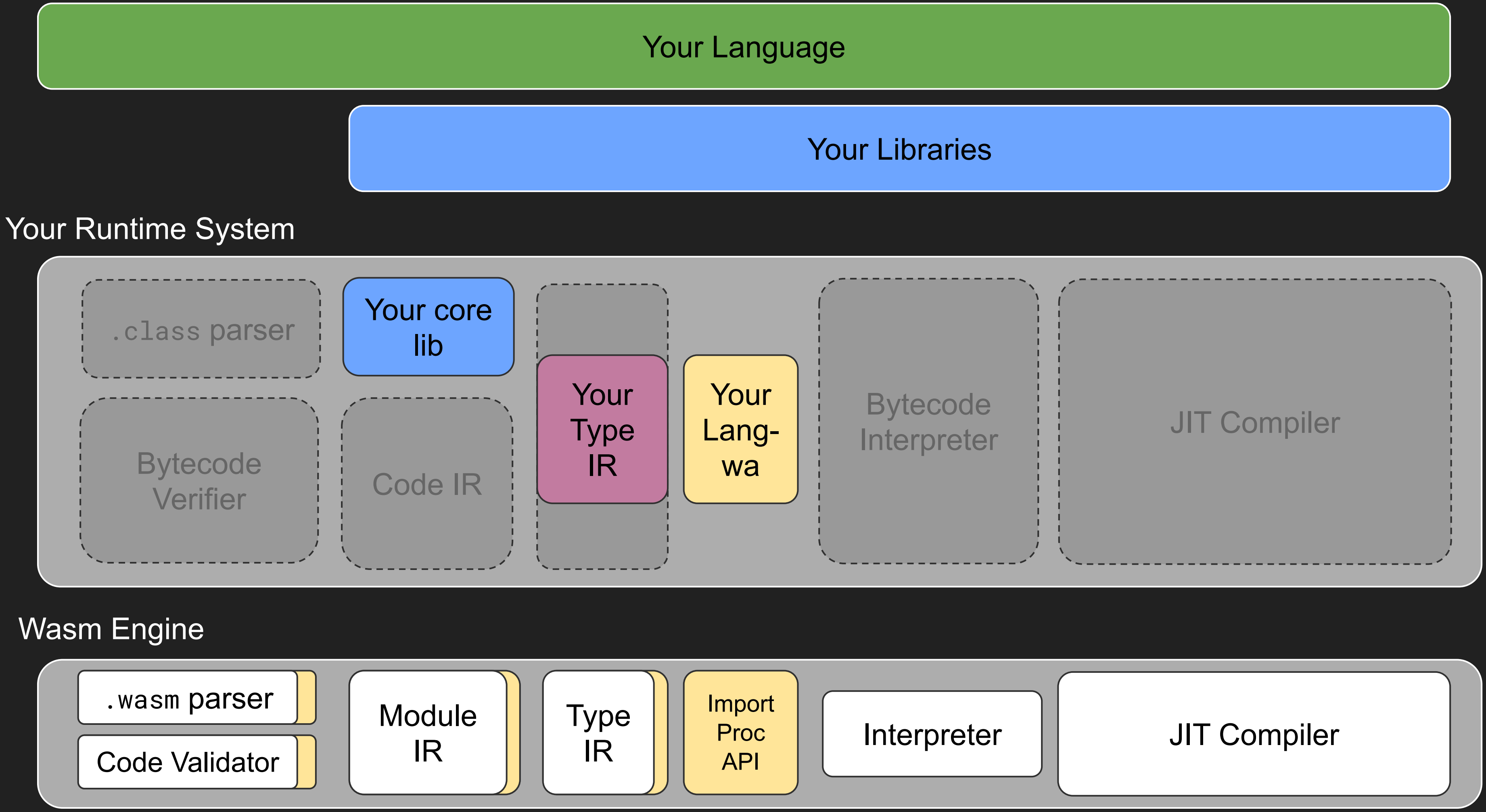
Pros and Cons

1. Host language emulation
2. Translation to linear memory
3. Translation to Wasm GC
4. Pass-through emulation
5. **Native implementation**

✅ Best possible performance

❌ Intrusive modifications to engine

❌ Doesn’t scale to many languages


Your Language
Your Libraries
Your Runtime System
.class parser
Your core lib
Bytecode Verifier
Code IR
Your Type IR
Your Lang-wa
Bytecode Interpreter
JIT Compiler
Wasm Engine
.wasm parser
Code Validator
Module IR
Type IR
Import Proc API
Interpreter
JIT Compiler

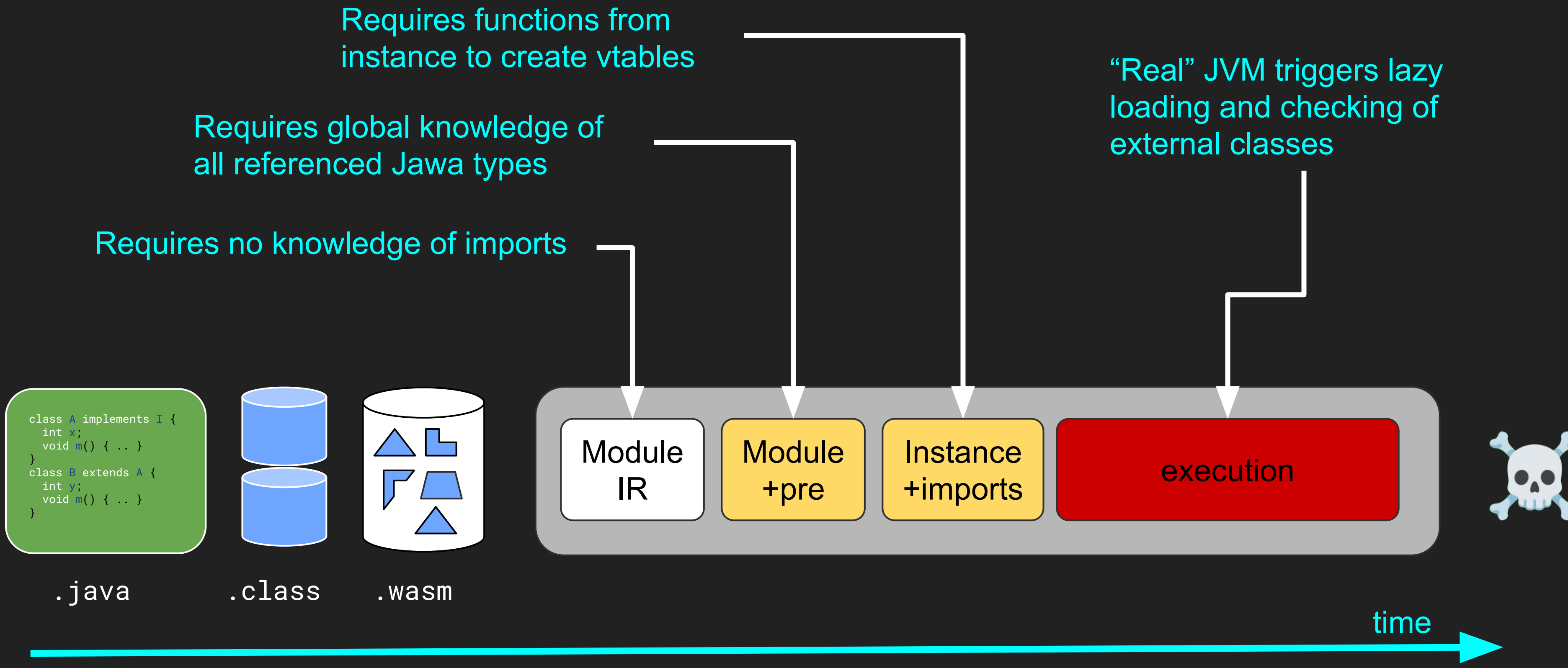
Questions?

http://github.com/titzer/virgil
http://github.com/titzer/wizard-engine

FAQ

- Why not just use custom sections?
  - Jawa concepts are intertwined with Wasm concepts, such as functions, so we want to be robust to module transforms and not encode member indices into custom section
- Can the X proposal express this?
  - type-imports: need type args, function import args, multiple subtype constraints
  - module-linking: not really, unless in a very clunky and inefficient way
  - interface-types: lacks parameterization mechanism for types, functions
- Can this proposal express X?
  - ✅ Python: (parallel Pywa project)
  - ✅ GC: (if equivalent intra-module types canonicalized)
  - ✅ function-references: (if we also allow signatures to be exported)
  - ✅ module-linking: (at least, I think so)
  - 🔍 interface-types: (maybe…)
  - ✅ SIMD: yes, but not yet implemented

Staging and Linking Jawa (Wasm GC)



```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```

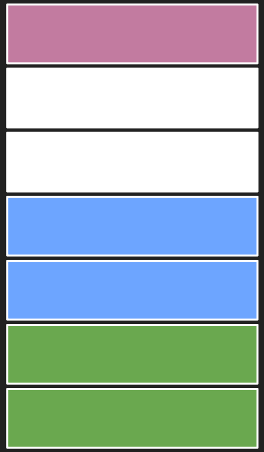
Requires functions from instance to create vtables
Requires global knowledge of all referenced Jawa types
Requires no knowledge of imports
"Real" JVM triggers lazy loading and checking of external classes
Module IR
Module +pre
Instance +imports
execution
.java
.class
.wasm
time

The case for Jawa

- Jawa encoding is ***fully independent*** of implementation strategy
- ***No changes*** to application binaries to run on a different kind of VM
- Once Jawa binary format is stable, it is ***forward-compatible*** with
  - new optimizations
  - new Wasm features
  - new Engine / runtime system designs
- Jawa could bring the JVM ecosystem to Wasm much more easily
  - Simple, straightforward translation from `.class` files
  - Hundreds of source compilers need no modification

A little deeper on the Jawa GC Object Model

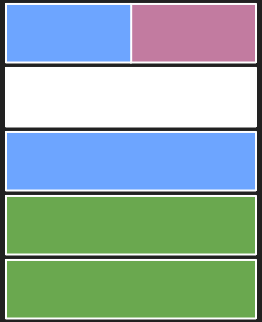
```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```



B object
in memory

Jawa GC Object Model (improved)

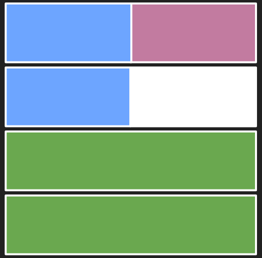
```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```



B object
in memory

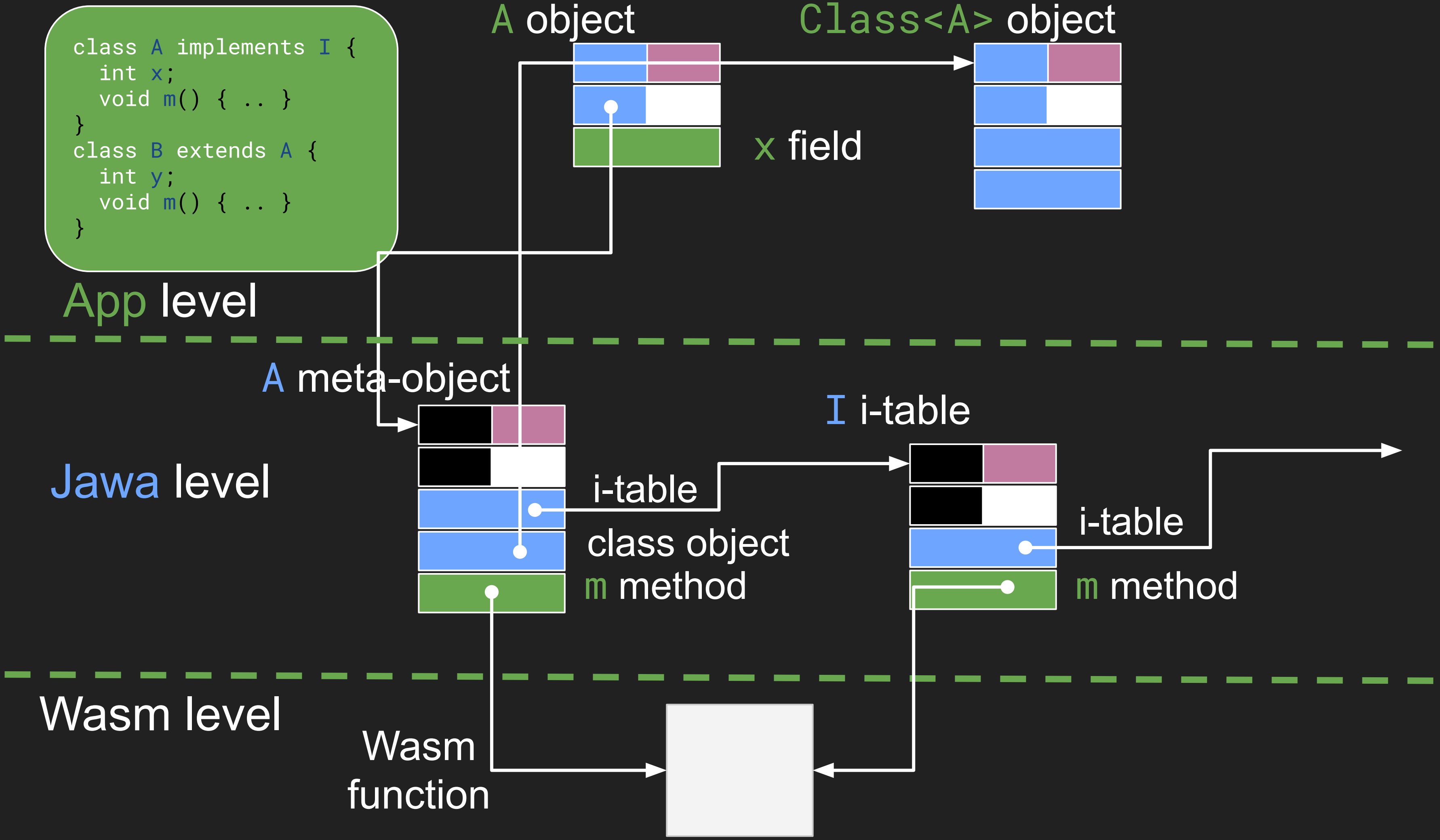
Jawa GC Object Model (improved++)

```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```

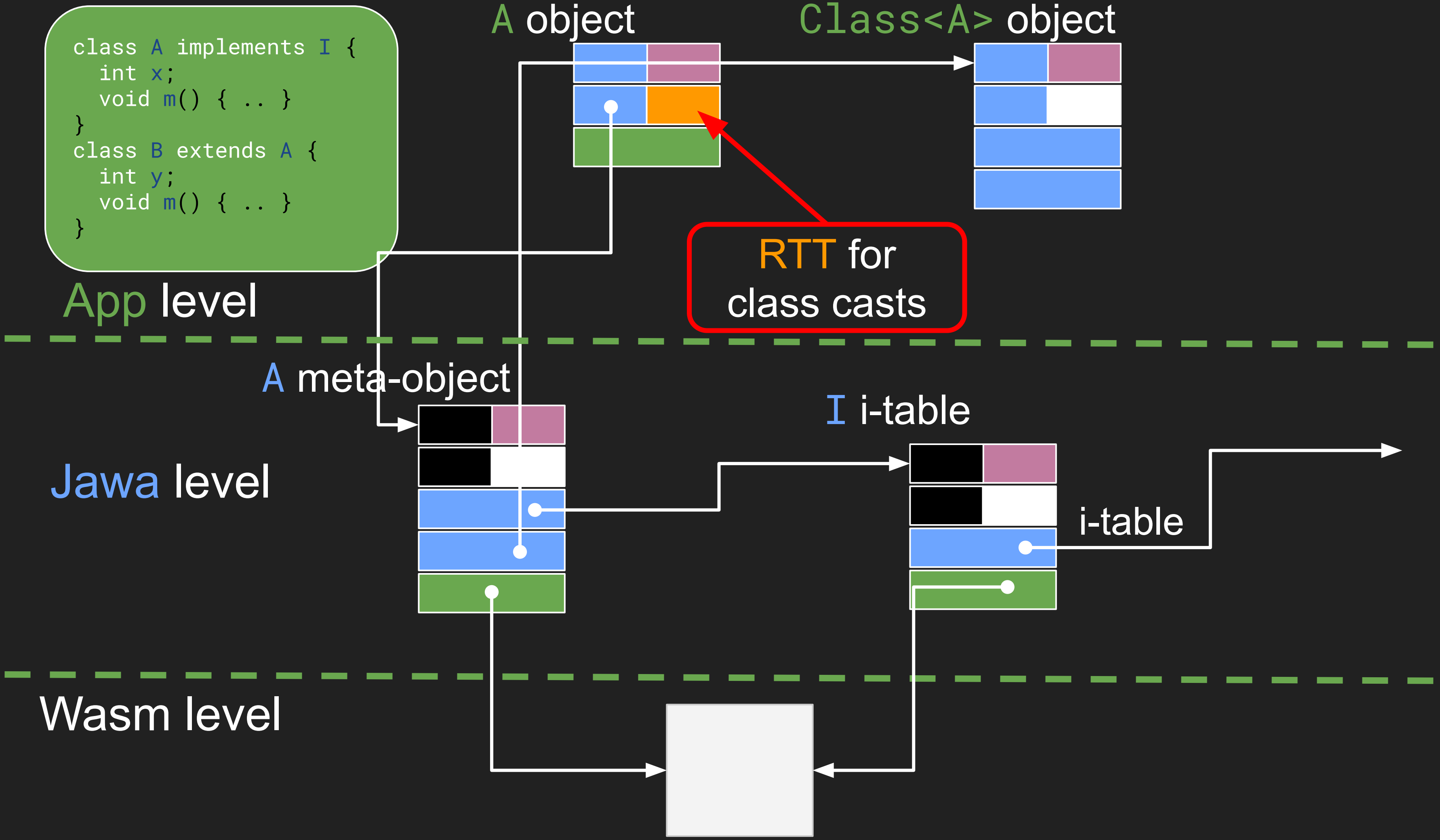


Jawa hashcode / Virgil header
Jawa metaobject
reference/Wasm RTT/struct
x field
y field

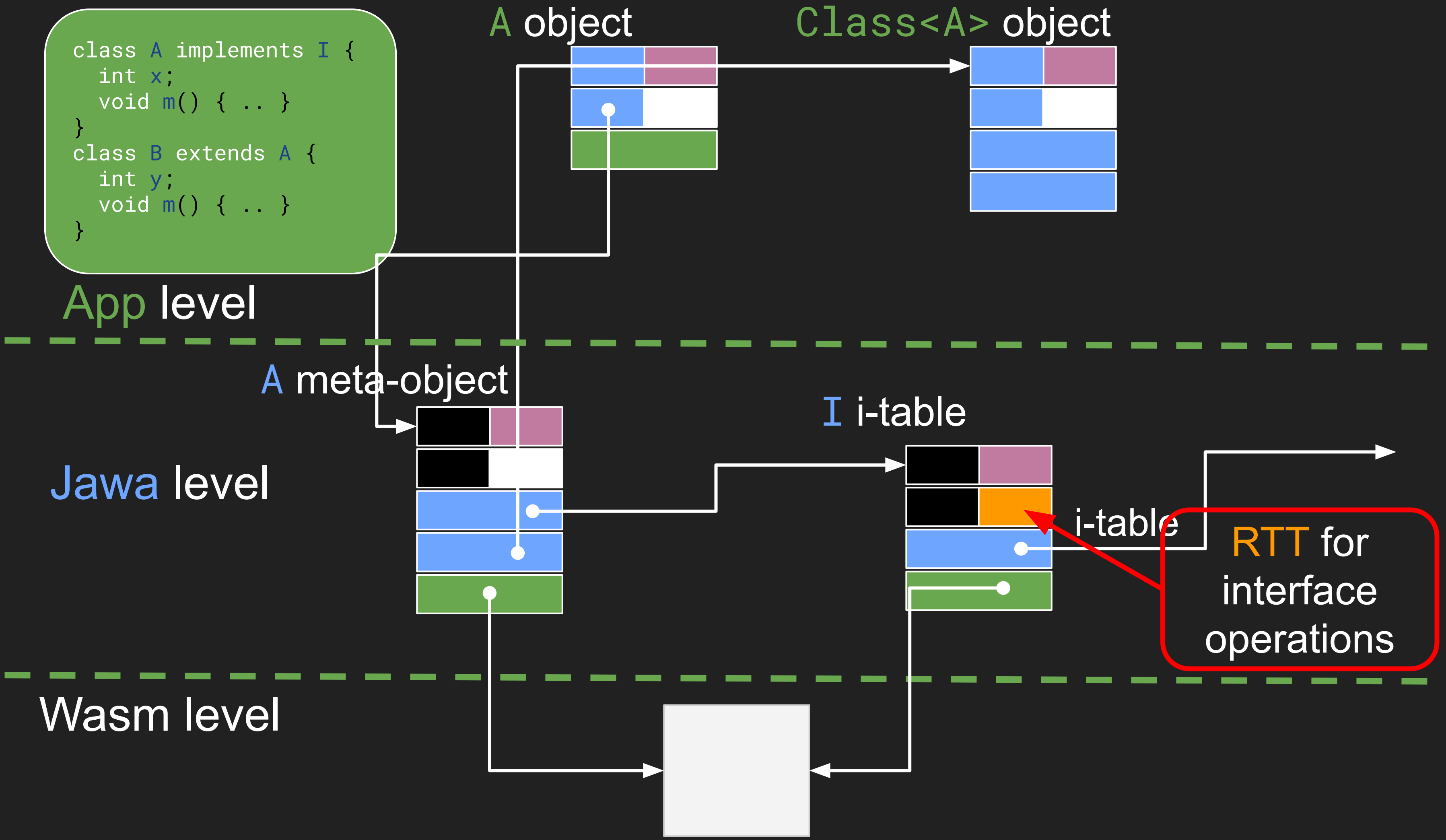
B object
in memory



```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```



```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```



```
class A implements I {
  int x;
  void m() { .. }
}
class B extends A {
  int y;
  void m() { .. }
}
```