HLL VM Implementation (Java)

CS 562: Virtual Machines

Kyle Hale

Reading: S&N Ch.5 & 6

Object Oriented Languages: Java

- Introduce the concept of "object," which captures both:
  - state
  - and methods (which manipulate the state)
- *class* defines object structure
- an *object* is an instance of a class
- fields can be shared among instances (`static` keyword)

OO languages allow **inheritance**

- enables **polymorphism:** binding of a function is based on object instance
  - we can have a make_sound method for objects that inherit from class Animal
  - make_sound for class HouseCat might print “meow”
  - make_sound for class Lion might print “roar”

Java allows us to define **interfaces**

- A list of methods that all classes that implement the interface must implement
- Does not have any state itself (interfaces cannot be instantiated)

Java Virtual Machine Overview

- Data types
- Internal data representation
- ISA (bytecode)
- Exceptions
- Class Representation

Primitive Data Types

- Primitive types defined based on *values* they can take on, **not** the bits
- E.g., an int can range between -2^31 and +2^31-1
- Types
  - int
  - char
  - byte
  - short
  - float
  - double
  - returnAddress

References

- Value that points to an object in memory (or null if the reference hasn’t been assigned)
- Internal representation depends on implementation!
  - (e.g., could be 32-bit pointer, 64-bit pointer, 256-bit pointer with a ton of metadata…)
- Note: programs cannot inspect (or use) the internal representation! (java does not have pointers/addresses!)

Objects

- constructed from primitive data types, and references which may refer to other objects
- Arrays are treated as special objects (the ISA has explicit support for them)

Internal JVM storage

- Global area: main memory, where globally declared variables live
- Local storage: local variables, this is attached to a method's stack frame
- Op storage: (operand stack)
- All storage areas store *cells*

Stack

- Each method gets its own stack frame
- Locals on the stack have a fixed size (this is known at compile time)
- Operand stack is used for arithmetic:
  - ONLY primitive types and references (objects and arrays cannot be put on the stack)

Global Memory

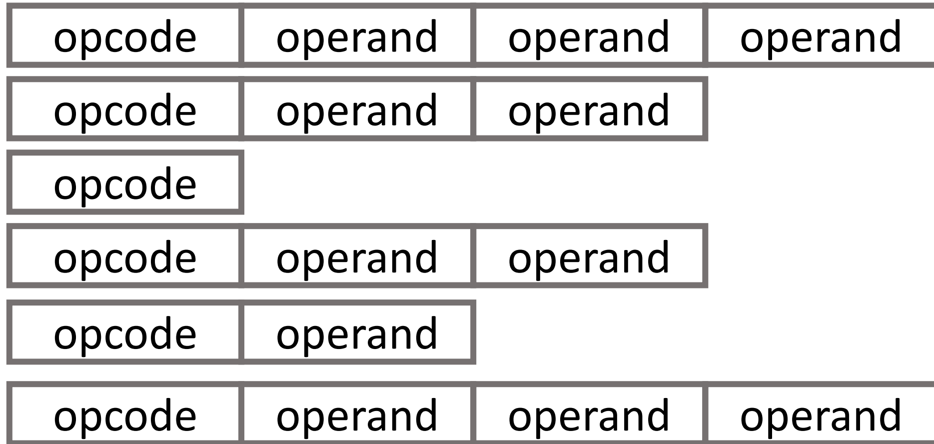
- Code (methods)
- Heap (holds objects and arrays)
- Size of global memory is unspecified (implementation dependent)
- When objects are created in heap, a reference is also created to point to it
- Objects can **only** be accessed via references, which much match type of object referred to

Constant Pool

- Constant values that are not encoded as instruction operands have to go somewhere!
- But, they can have a range of lengths (e.g., strings)
- These constants are placed in a ***constant pool*** (just a bag of bytes)
- Instructions that use them use ***indexes*** into the const. pool

Memory Layout

instructions (bytecode)

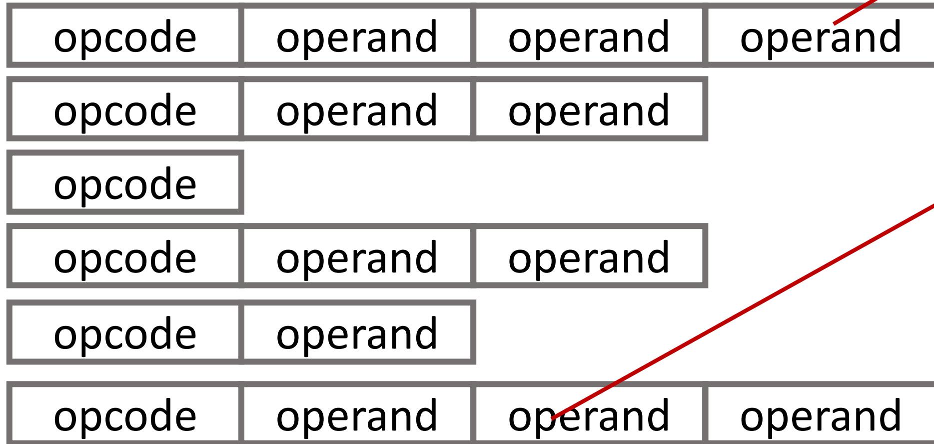


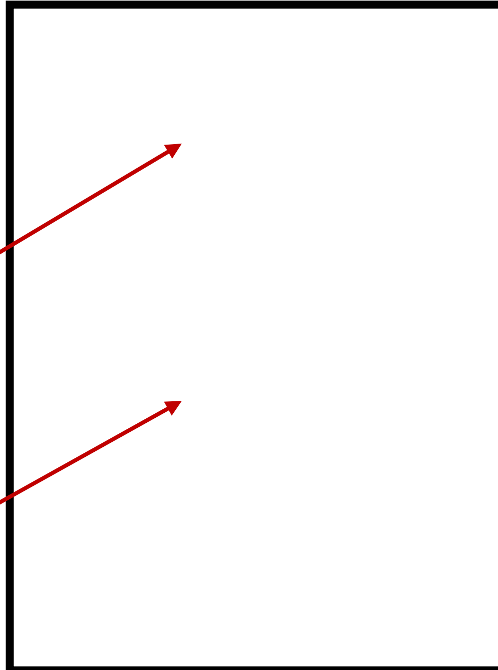
const pool

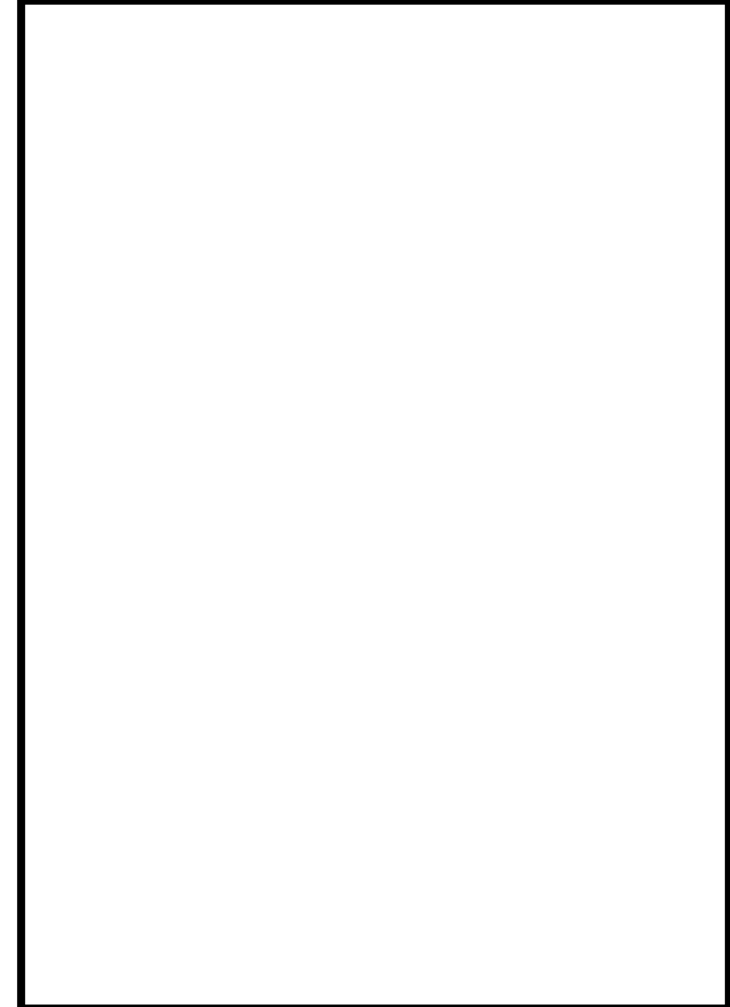
heap

stack frame

Operands can reference constants using indexes

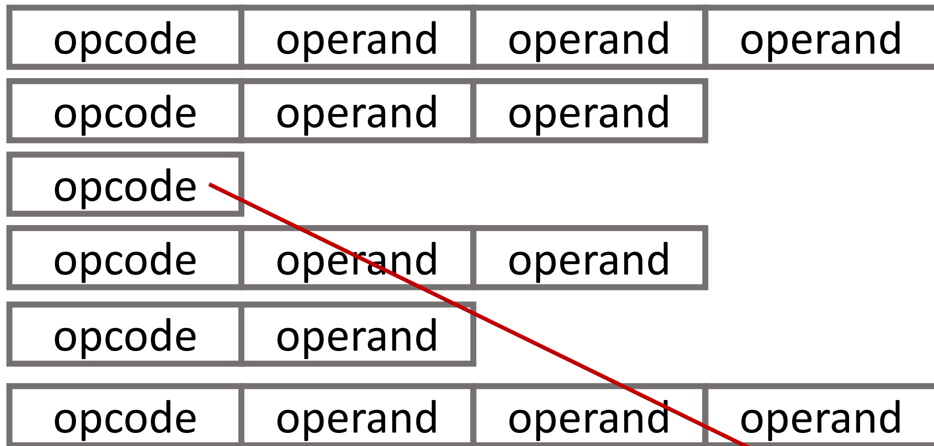






Some instructions have implied operands (e.g., **add**)

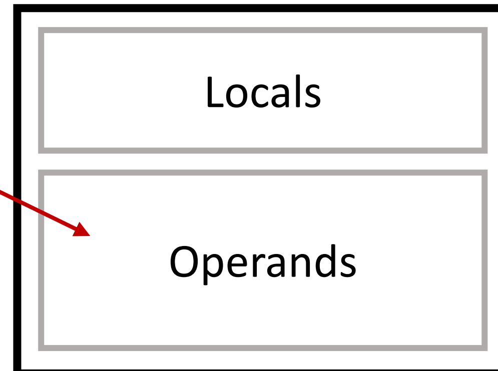
instructions (bytecode)



const pool

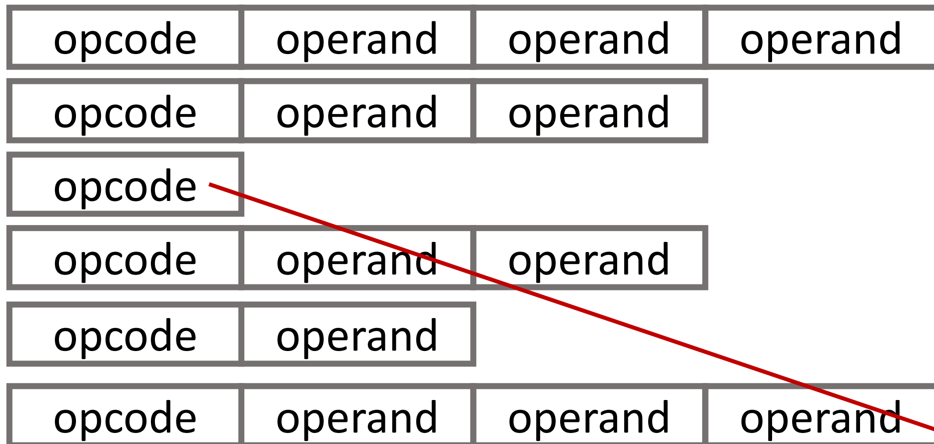
heap

stack frame



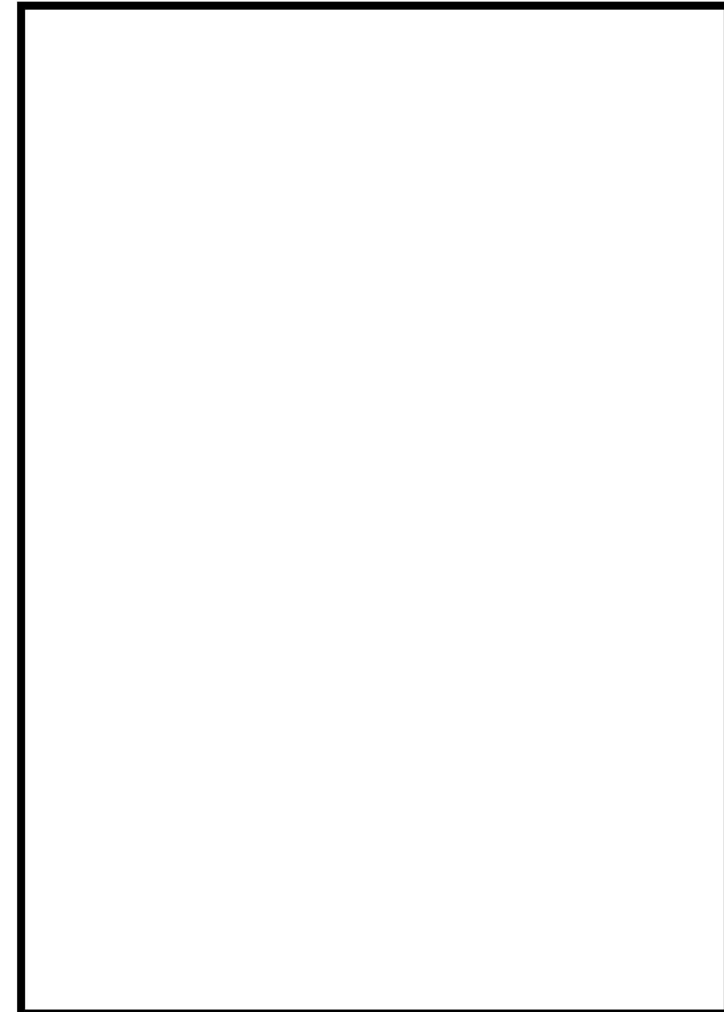
Some instructions have implied operands (or `iload`)

instructions (bytecode)

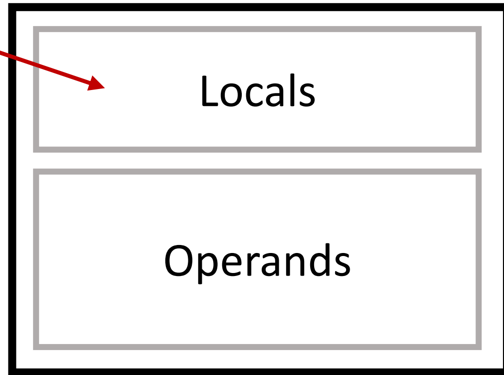


const pool

heap

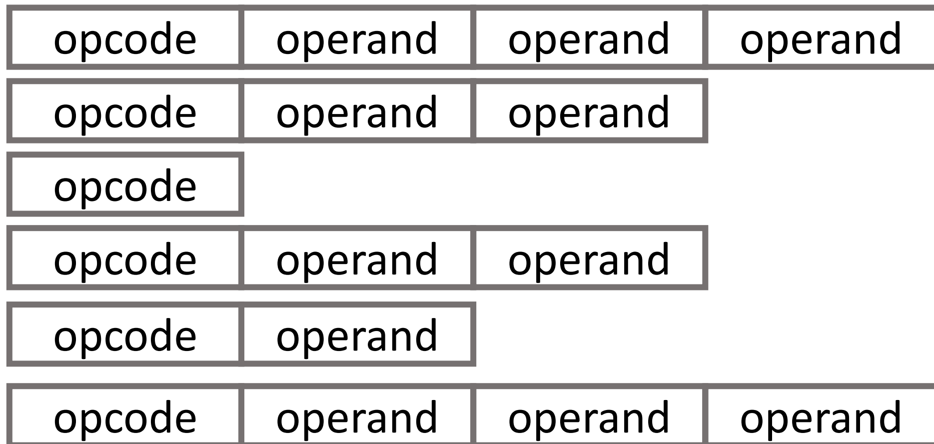


stack frame



Constants and operands point to the heap with references

instructions (bytecode)



const pool

heap

obj a

array

obj b

obj c

stack frame

Locals

Operands

Bytecode Instruction formats

opcode			no operands, or implied operands
opcode	index		one operand, either index into const pool or local var array
opcode	index1	index1	two operands, both indexes into const pool or local vars
opcode	data		one immediate operand
opcode	data1	data2	two immediate operands

Bytecode Instruction formats

opcode			no operands, or implied operands
opcode	index		one operand, either index into const pool or local var array
opcode	index1	index1	two operands, both indexes into const pool or local vars
opcode	data		one immediate operand
opcode	data1	data2	two immediate operands

could be an immediate offset (e.g., for a PC-relative branch), or just an immediate operand (e.g., a constant)

Bytecode instructions are typed

- e.g., iadd or dadd
- iload, etc.

Data movement instructions

- all loads and stores from global or local memory must be to the op stack
- all functional instructions operate on operands on the op stack
- some instructions have constants hard coded
  - e.g., **`iconst1`**: pushes int constant 1 onto op stack
  - **`bipush`** for two small constants
  - **`ldc`** for arbitrary constant
- **`pop`** discards top of op stack
- **`swap`** swaps two top elements of op stack

Data movement instructions

- Some data movement instructions move between local storage and stack
  - **`iload_1`**: take int from local storage location 1 and push on op stack
  - **`iload idx`**, **`idx`** refers to const pool entry
  - **`istore_1`**/**`istore idx`**
- Others involve the heap
  - **`new idx1 idx2`**: two bytes form idx into const pool, whose entry specifies object. New object instance is allocated on heap, and reference pushed on op stack
  - **`getfield`** and **`putfield`** access object fields (described by a const pool entry)

Runtime type conversion (casting)

- supported by explicit instructions, e.g., **`i2f`**
  - this pops an int from stack, converts it to float and pushes the result

Control flow instructions

- **`ifeq data1 data2`**: compare to zero (PC-rel offset is 2)
- **`if_icmpeq data1 data2`**: compare int equality (PC-rel offset is 2)
- **`ifnull data1 data2`**: pop obj. reference, branch if null
- methods are invoked with invoke family of instructions
  - **`invokevirtual idx1 idx2`**: typical method invocation (virtual functions!)
  - **`invokeinterace`**: invoke interface methods
  - **`invokespecial`**: invoke instance init. methods, private methods, superclass methods
  - **`invokestatic`**: static methods

Exceptions

- some are defined as part of the ISA
- others defined by programmers
- all exceptions have to be handled (cannot be turned off!)
- hot potato model: current method tries to handle
  - if it can’t, pop a stack frame and allow calling method to handle
  - …and so on up the stack
  - eventually the JVM will have to handle fallthrough exceptions

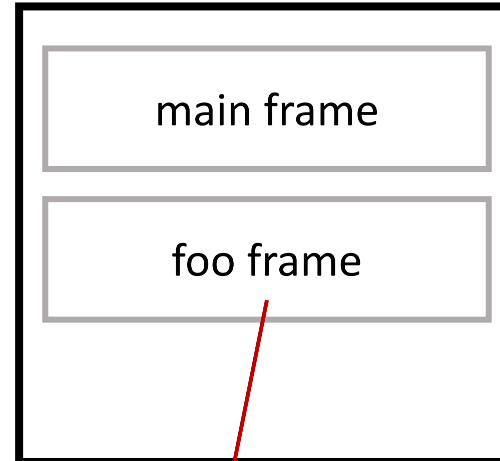
Exception Examples

- NullPointerException
- ArrayIndexOutOfBoundsException
- These imply runtime checking of invariants by the VM!

How does it work?

- Each method has a table of exception handlers
- Each table entry contains a type, a scope, and a reference to a handler
- When an exception is thrown, op stack is flushed, table looked up

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

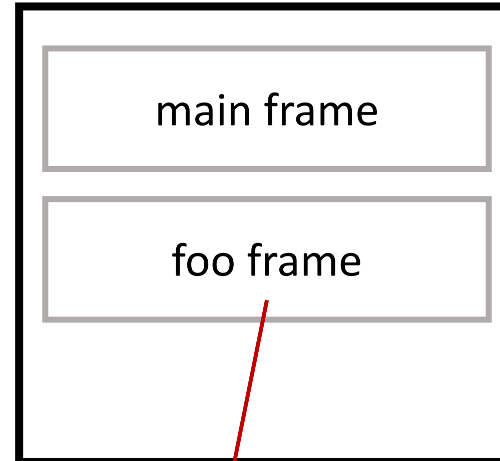


foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

stack

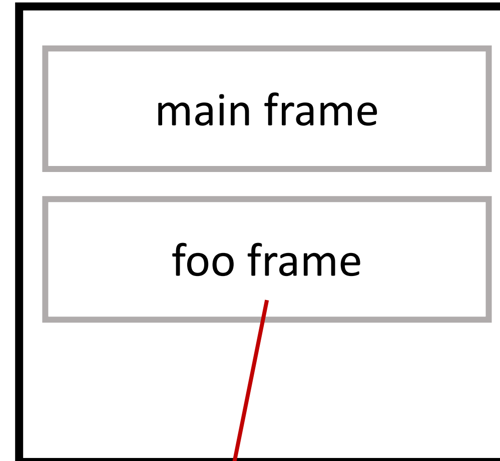


foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

stack

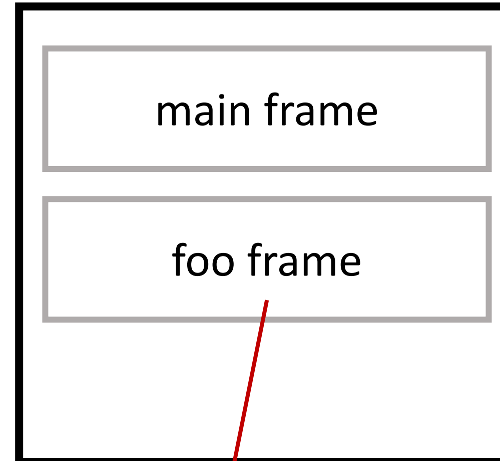


foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

stack



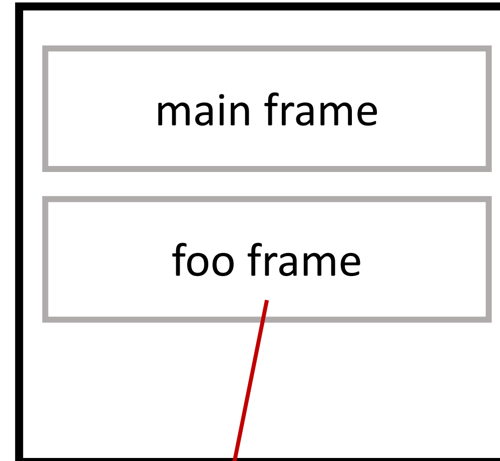
foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

scenario 1

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

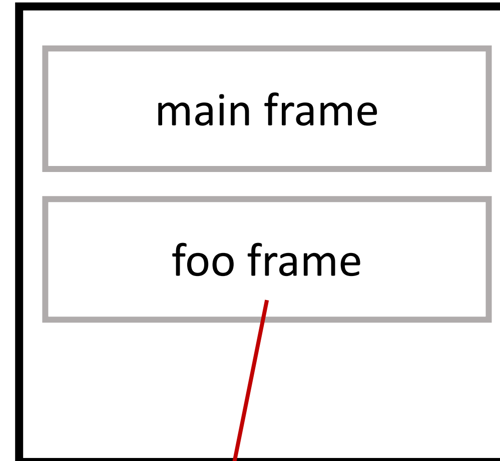
stack



foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 
0");
        }

    }
}
```

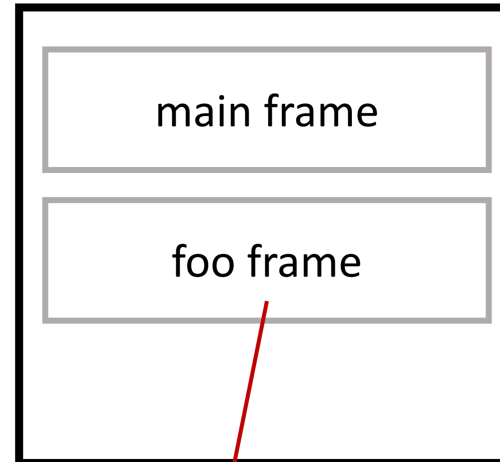


foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

EXCEPTION!

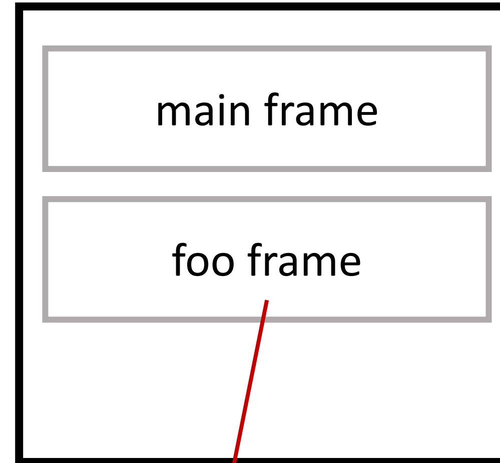


foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```

EXCEPTION!

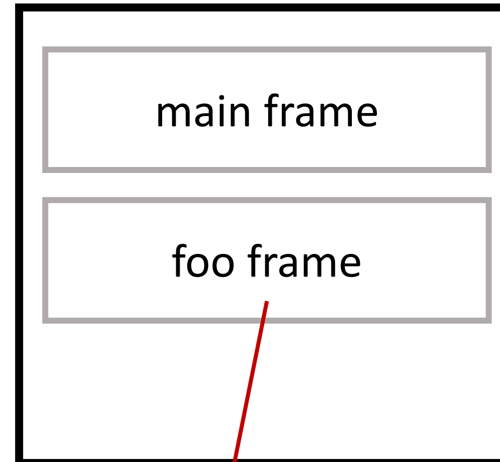


foo method info

From	To	Target	Type
40	45	49	ArithmeticException
...			

table lookup (match!)

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        try {
            int a = 30, b = 0;
            int c = a/b; // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by
0");
        }

    }
}
```



EXCEPTION!

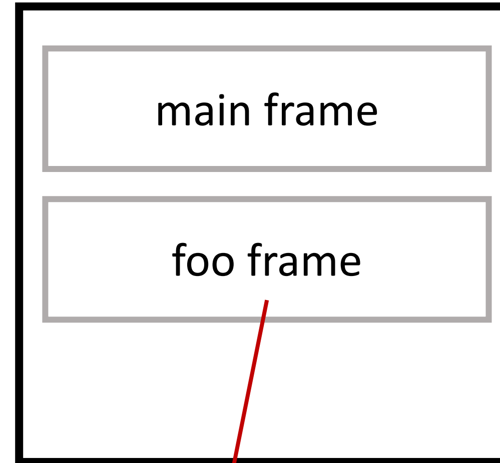
foo method info

From	To	Target	Type
40	45	49	ArithmeticException
…			

jump to matched excp. handler

scenario 2

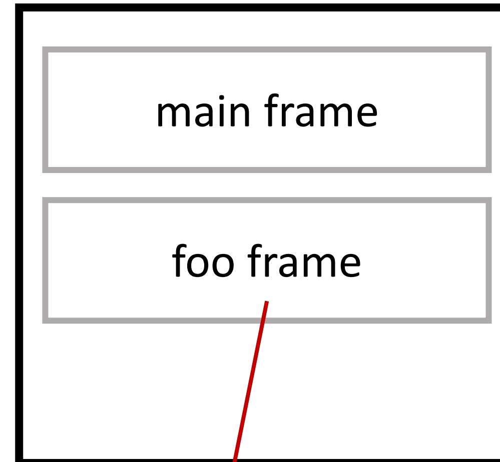
```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```



foo method info

From	To	Target	Type
...			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```



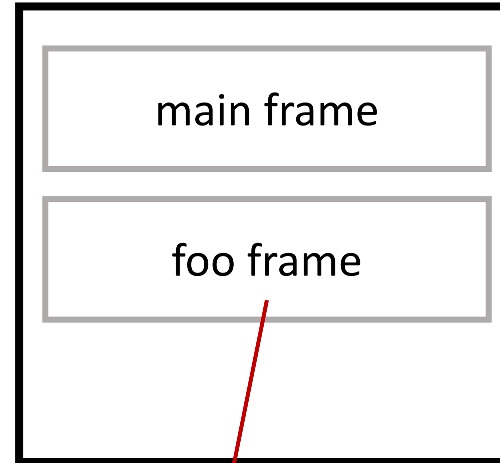
EXCEPTION!

foo method info

From	To	Target	Type
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```

EXCEPTION!

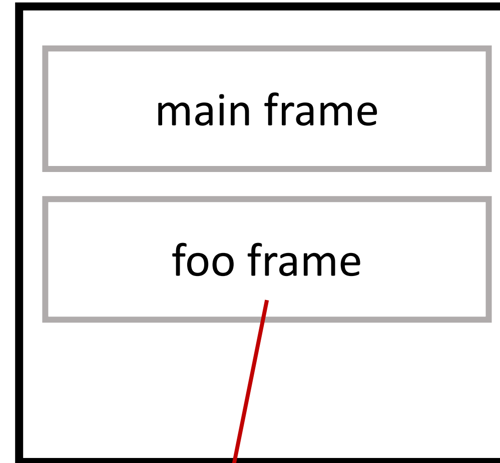


foo method info

From	To	Target	Type
…			

lookup FAILS

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```



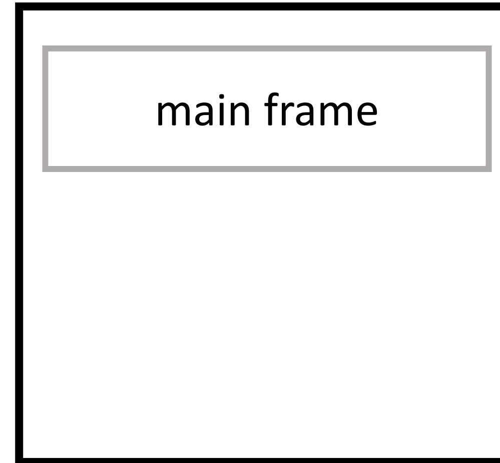
foo method info

From	To	Target	Type
...			

lookup FAILS

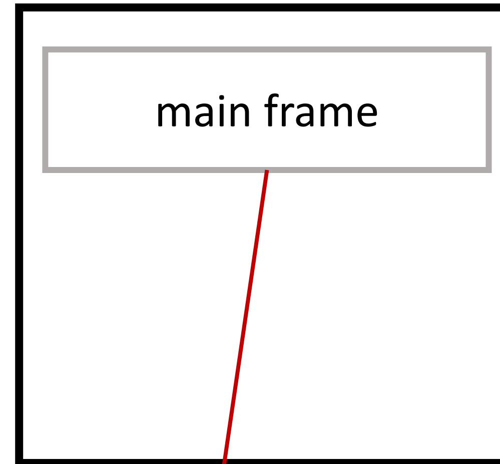
What to do?

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```



pop frame!

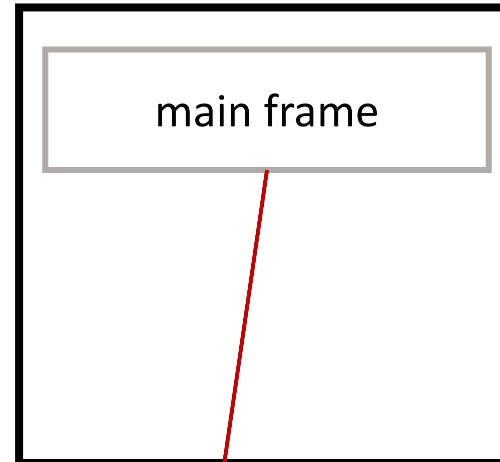
```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```



main method info

From	To	Target	Type
…			

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```

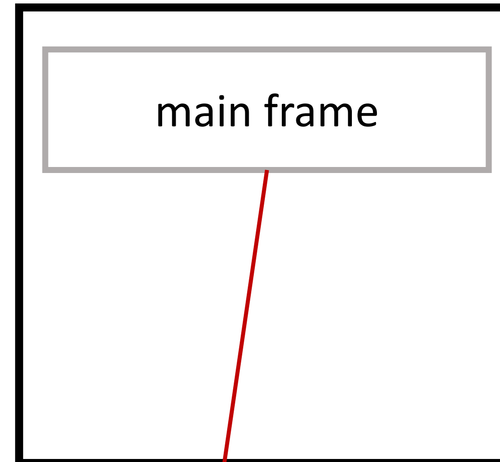


main method info

From	To	Target	Type
…			

still no handler!

```java
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        foo()
    }

    public void foo()
    {
        int a = 30, b = 0;
        int c = a/b; // cannot divide by zero
        System.out.println ("Result = " + c);
    }
}
```



main method info

From	To	Target	Type
…			

JVM has to handle…

Errors and Exceptions *are not the same*

- Error: something is wrong internally (e.g. with the host or the VM)
  - example: `StackOverflowError`
  - example: `InternalError`

Garbage Collection

- JVM can **automatically** free unused objects
  - Detect when last reference to object is destroyed
- JVM spec does not *require* GC, but most use it
- It uses an algorithm to do this (which one? more later…)

Emulation Engine

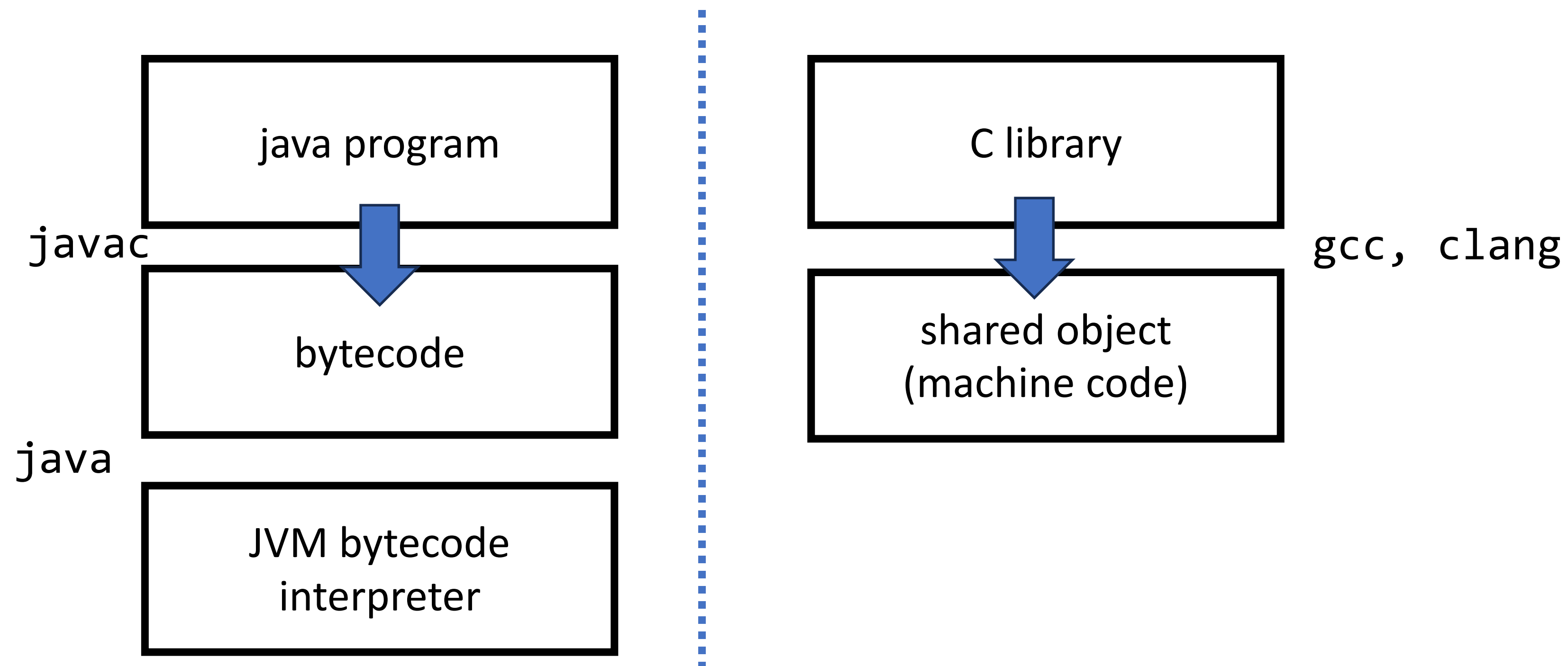
- Can be a simple instruction interpreter (like the 6502 emulator)
- Or something more advanced (binary translation)
- Industrial JVMs use just-in-time compilation (dynamic byn. tran.) with profiling
  - Profiler translates **hot** functions to native code, others are emulated
- Constant pool and string lookups are expensive, so most of this indirection is removed at runtime (more later)

Java Native Interface (JNI)

- Gives us interoperability between languages
  - E.g., call C code from Java
  - and other way around

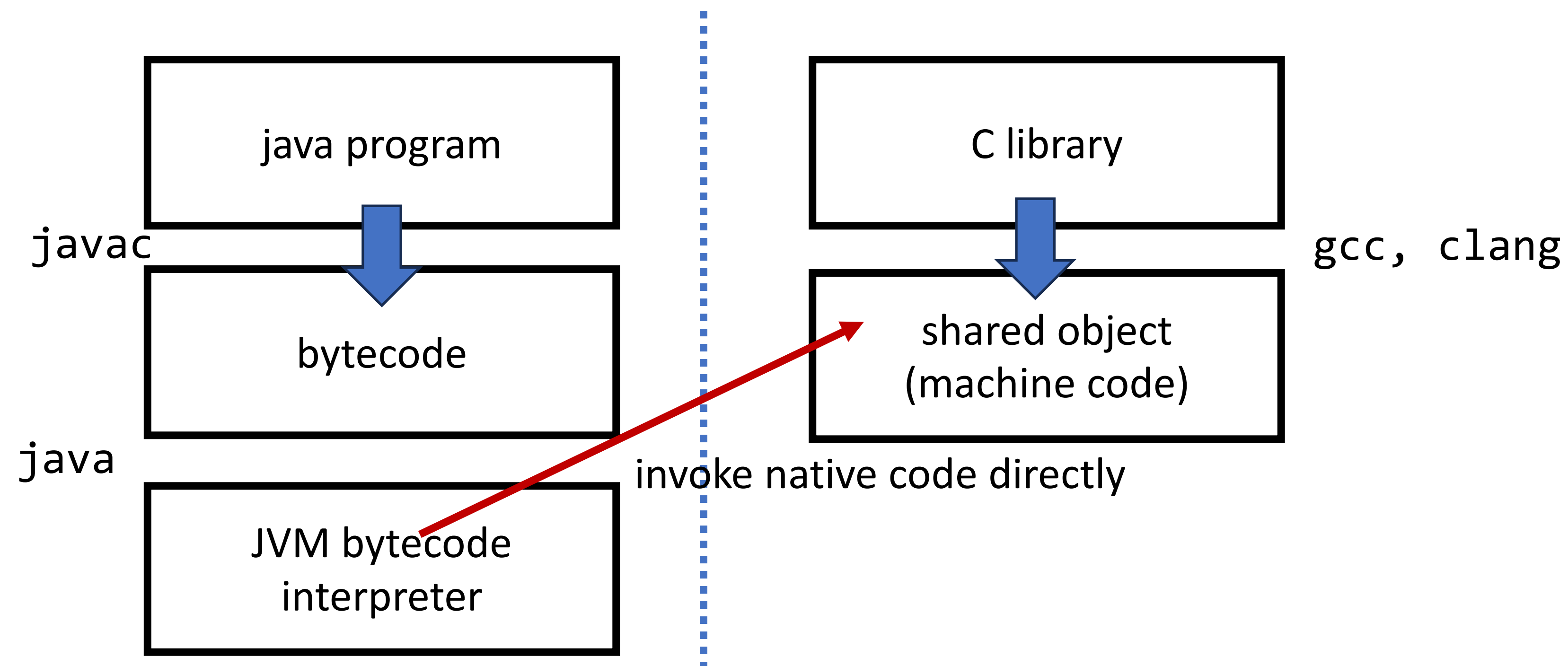
Java Native Interface (JNI)

- Gives us interoperability between languages
  - E.g., call C code from Java
  - and other way around



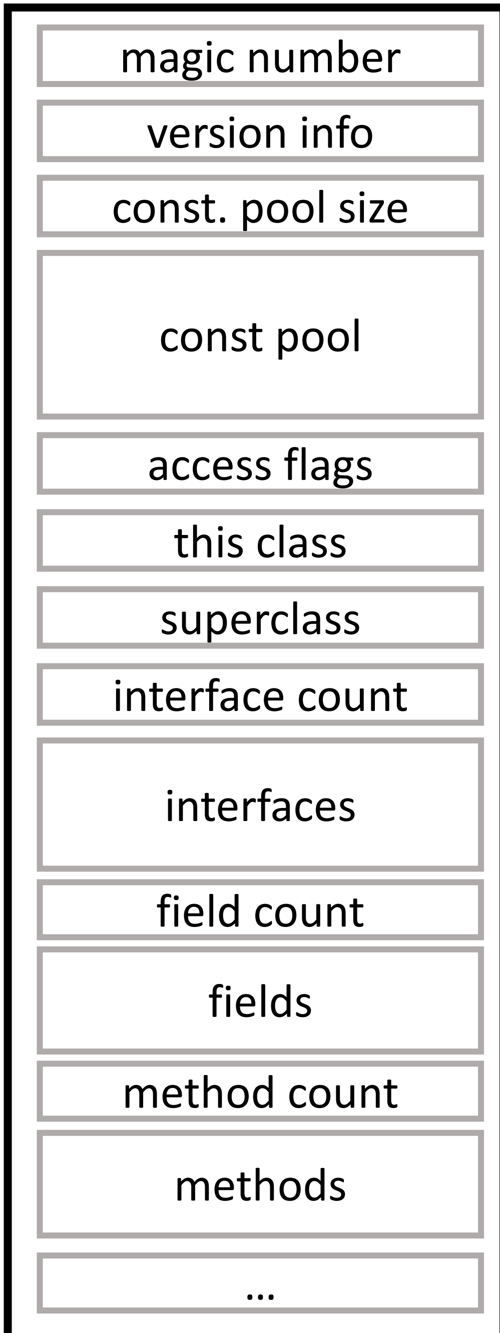
Java Native Interface (JNI)

- Gives us interoperability between languages
  - E.g., call C code from Java
  - and other way around

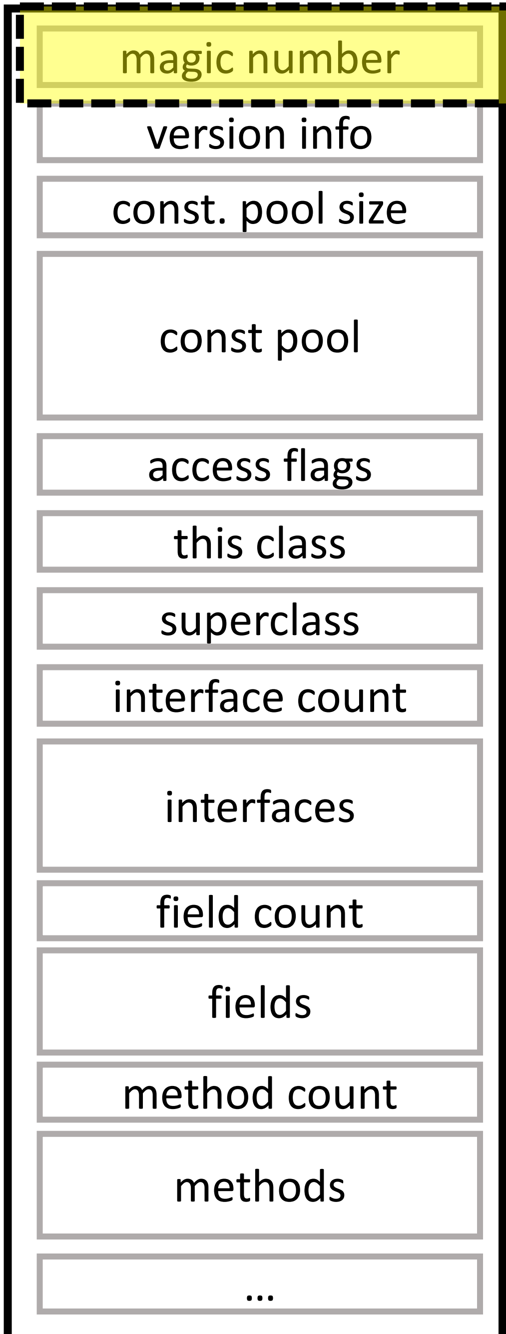


Java Binary Classes

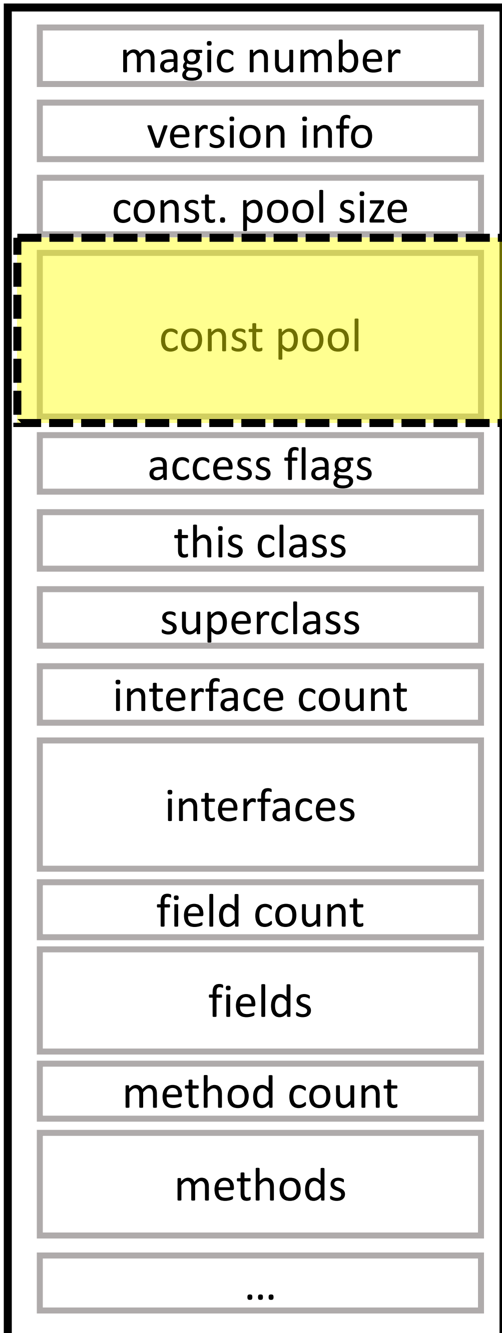
- i.e., .class files
- These define both the code for a Java program, but also the ***metadata***
- not necessarily loaded at program startup
    - classes can be loaded *lazily*



.class file format
magic number
version info
const. pool size
const pool
access flags
this class
superclass
interface count
interfaces
field count
fields
method count
methods
…



“cafebabe” (hex)



constants live here:

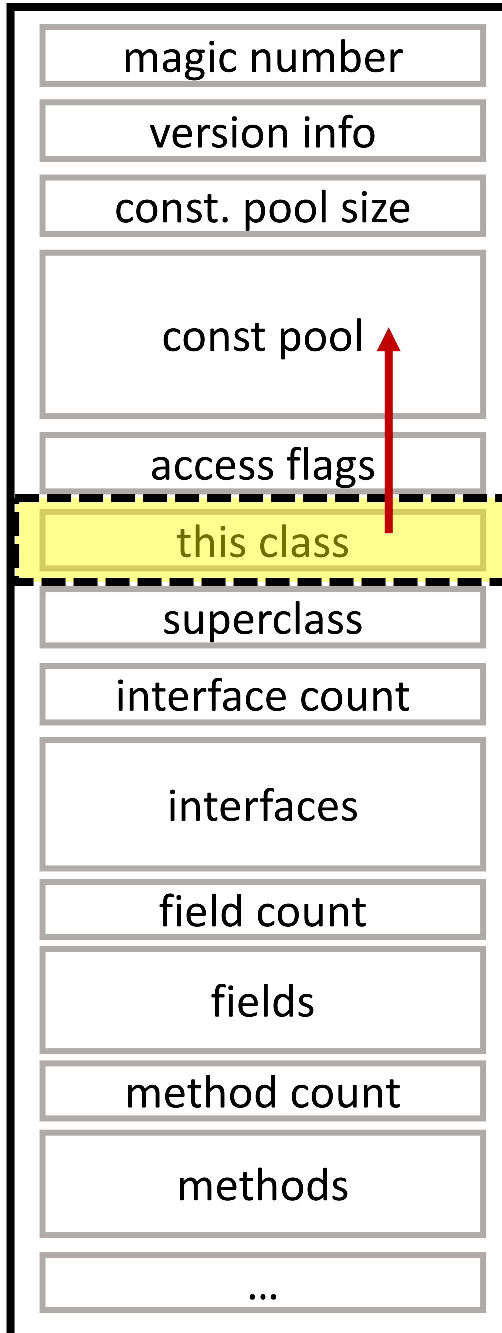
- strings

- class names

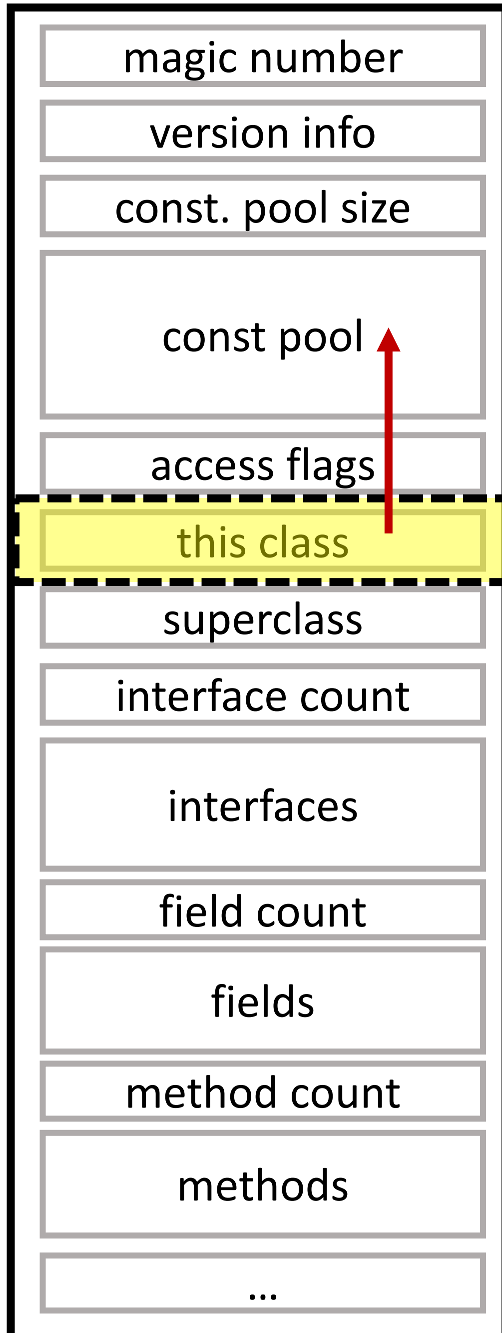
- method names

- references (including to other obj.)

- etc.

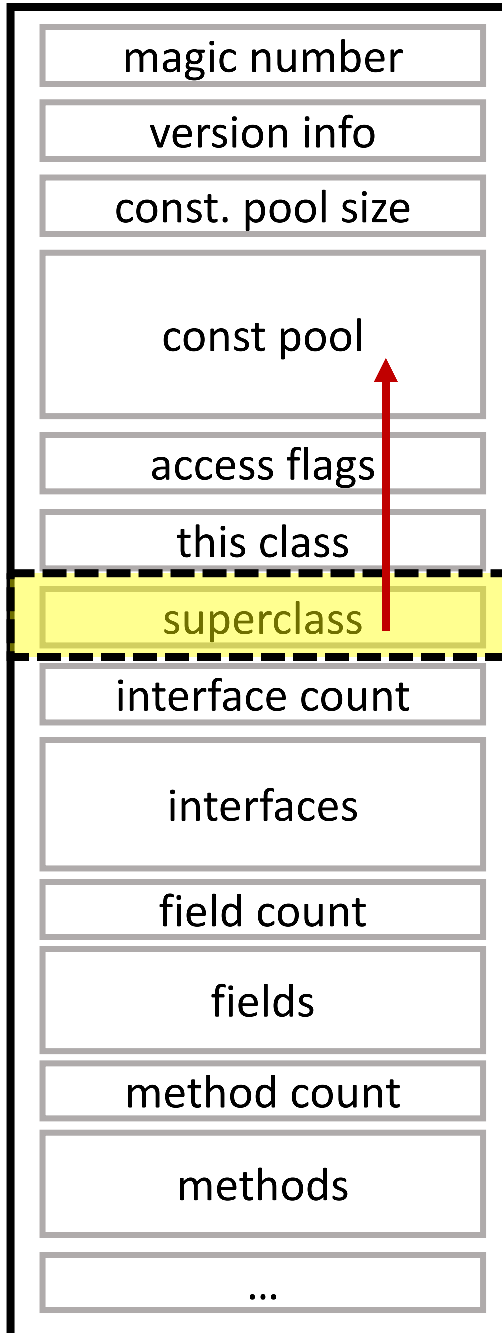


reference to const pool entry to this class (index)

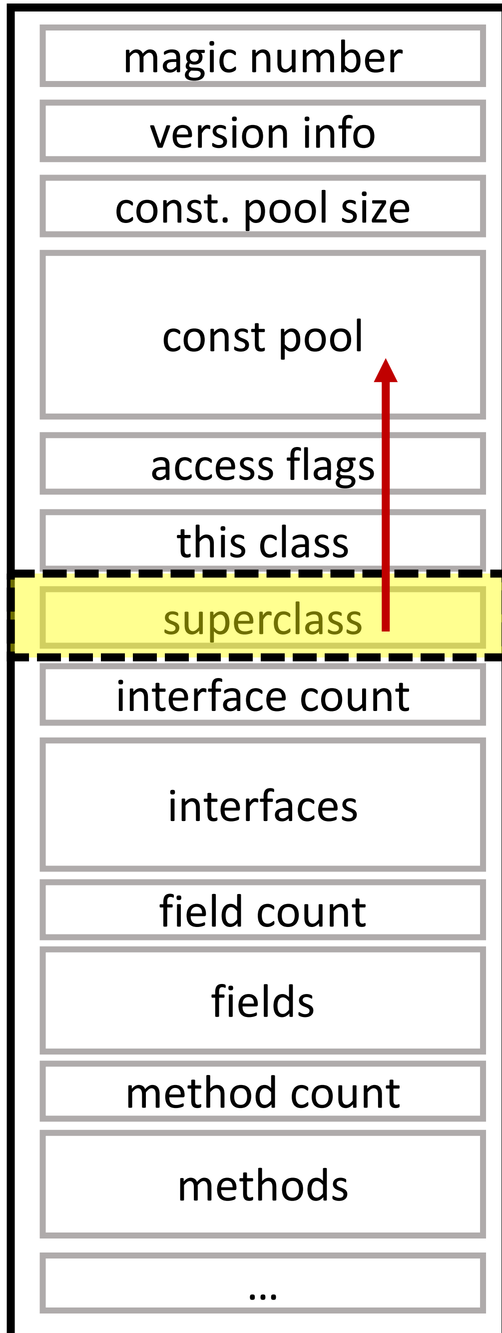


reference to const pool entry to this class (index)

const pool entry can look like "MyClass.Foo" (symbolic)

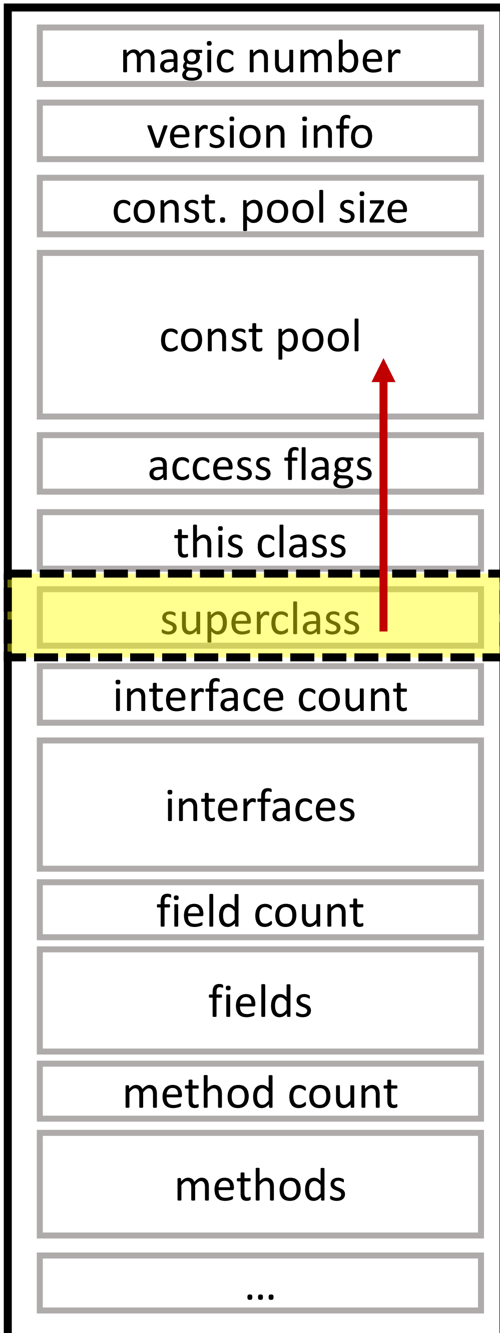


reference to const pool entry to my superclass (who I inherit from)



reference to const pool entry to my superclass (who I inherit from)

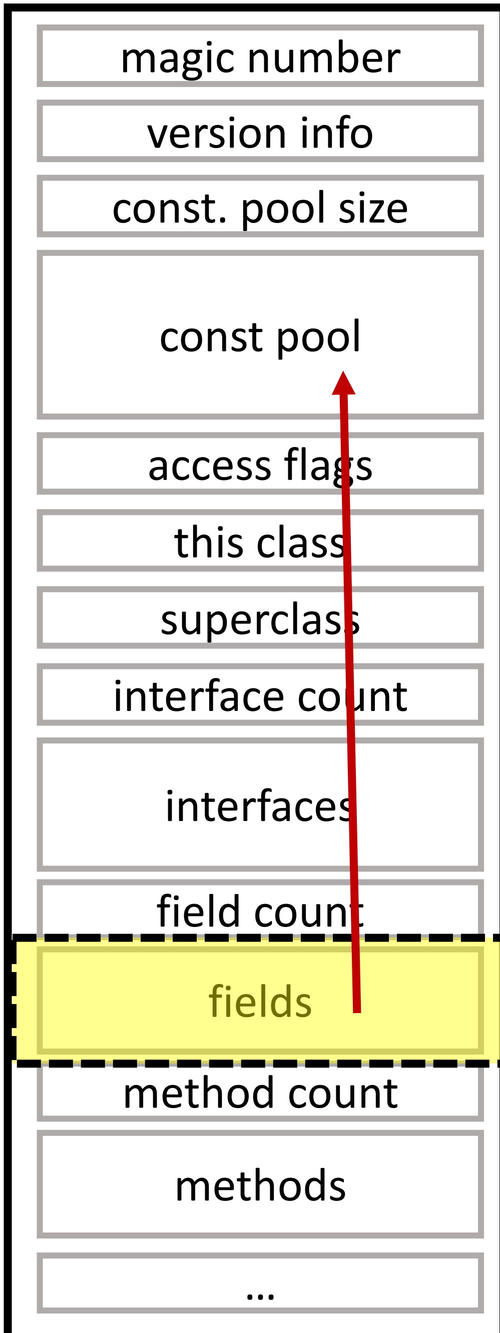
every class must have a superclass



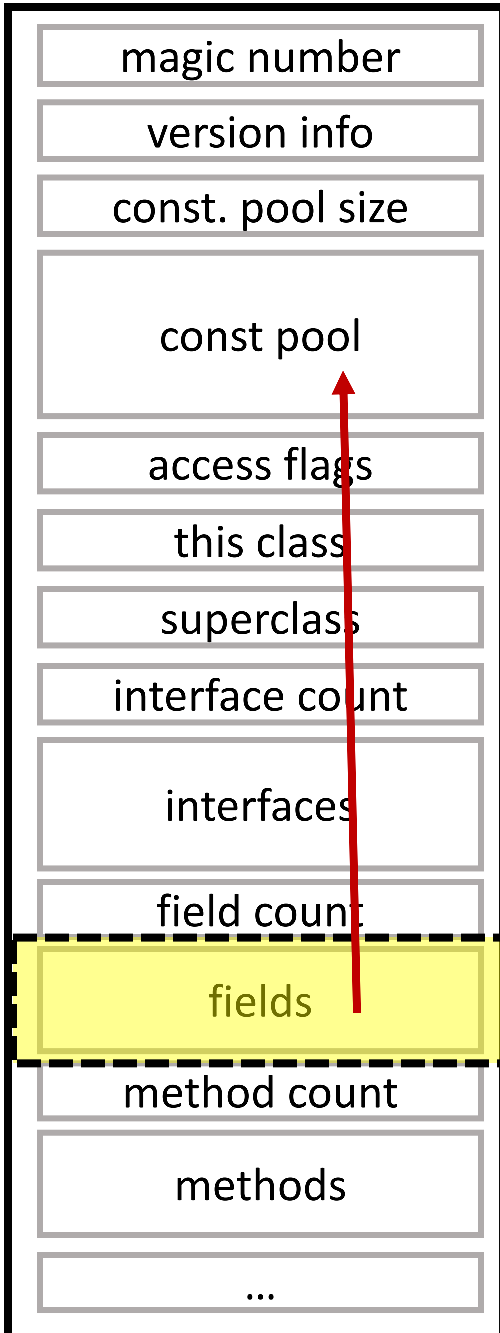
reference to const pool entry to my superclass (who I inherit from)

every class must have a superclass

***except `Object`, which is the root class (this val will be 0)**

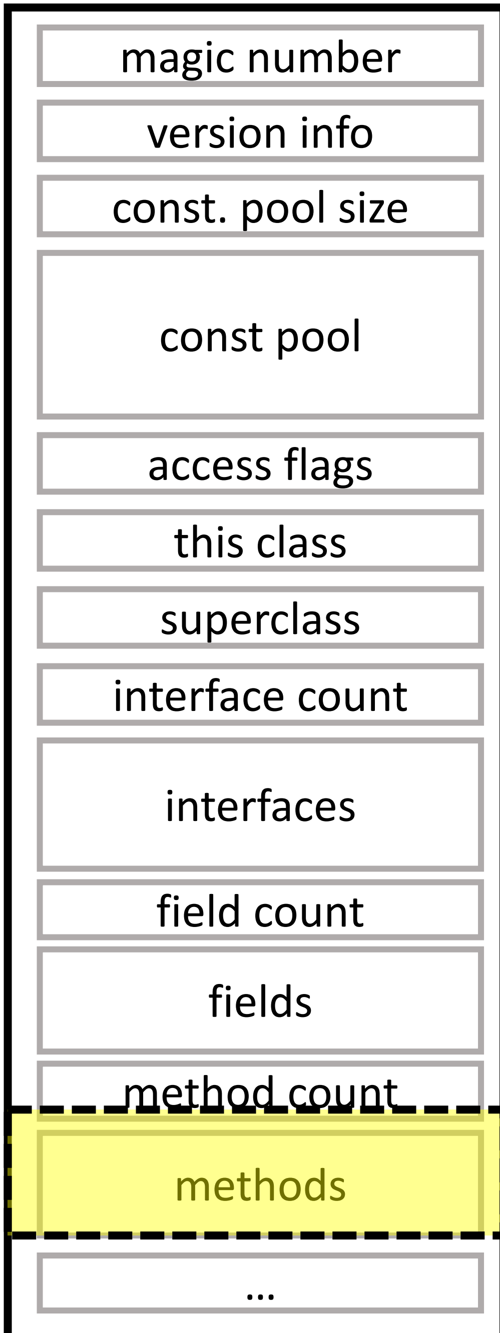


fields are also symbolic references in const pool (field names and types)



fields are also symbolic references in const pool (field names and types)

can refer to other objects, so access to them can trigger class loading



this is where the bytecode for each class method lives!