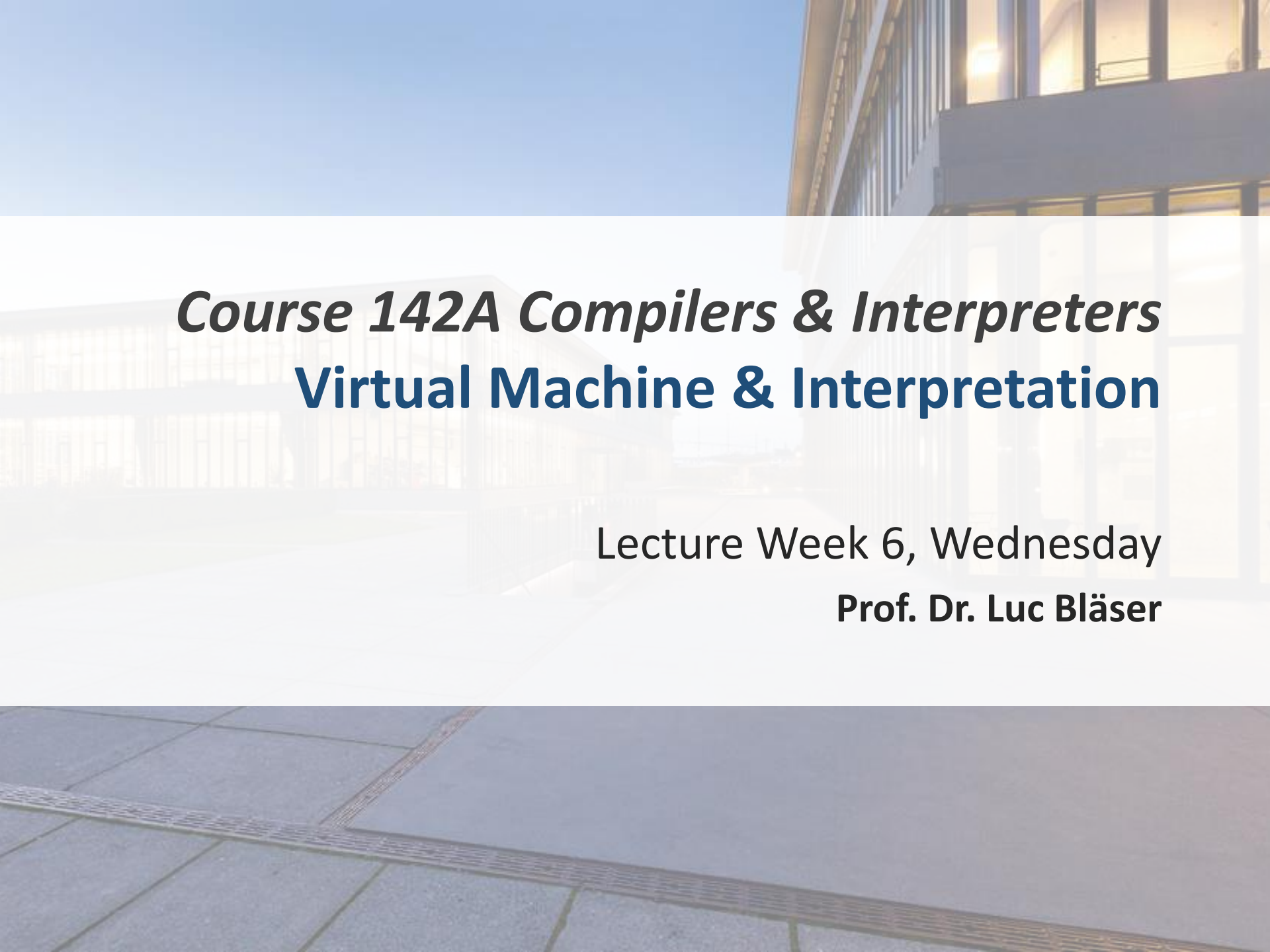


# *Course 142A Compilers & Interpreters*
# Virtual Machine & Interpretation

Lecture Week 6, Wednesday

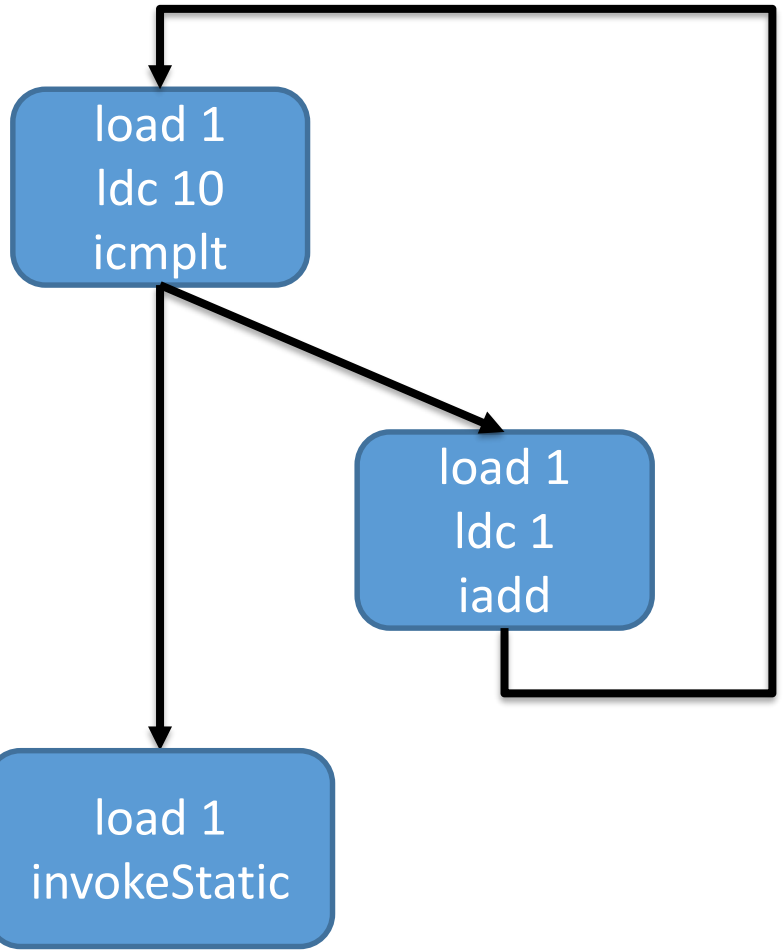
**Prof. Dr. Luc Bläser**

# Last Week - Quiz

```
label0: load 1
        ldc 10
        icmplt
        if_false label1
        load 1
        ldc 1
        iadd
        store 1
        goto label0
label1: load 1
        invokestatic writeInt
```

*How does the control flow graph look like?*

# Control Flow Graph




*What can we do with the control flow graph?*

# Dataflow Analysis

- Requires CFG (of bytecode or source code)
- Static verification
  - Type correctness
  - Bounded evaluation stack
  - …
- Static optimizations
  - Constant propagation
  - Dead code elimination
  - …

=> However, we can also check and optimize at runtime

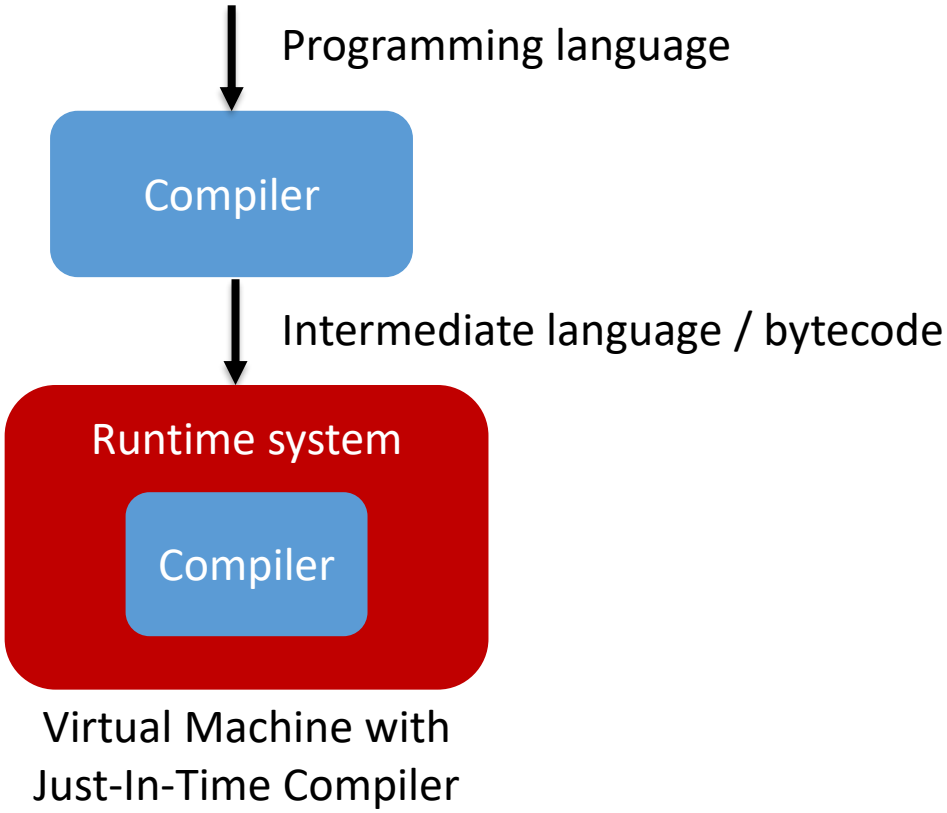
# Intermediate Code

```
label0: load 1
        ldc 10
        icmplt
        if_false label1
        load 1
        ldc 1
        iadd
        store 1
        goto label0
label1: load 1
        invokestatic writeInt
```

```
while (x < 10) {
    x = x + 1;
}
writeInt(x);
```

*How do we run our bytecode?*

# Big Picture


Programming language
Compiler
Intermediate language / bytecode
Runtime system
Compiler
Virtual Machine with
Just-In-Time Compiler

# Today's Topics

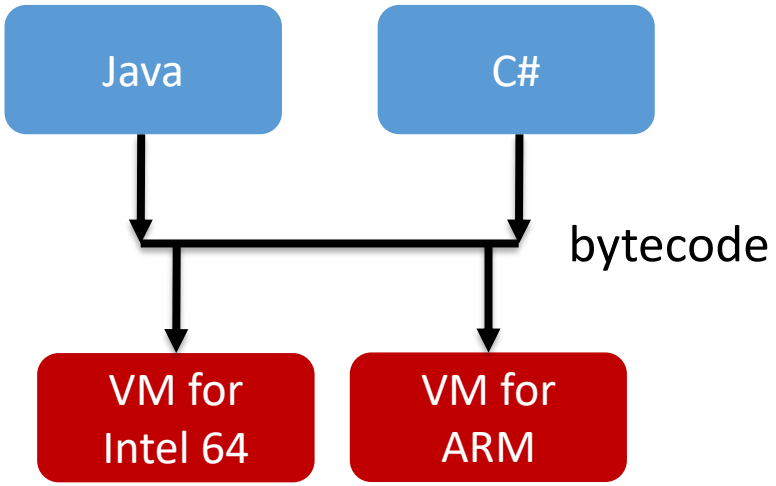
- Virtual Machine
- Loader
- Interpreter
- Call Stack

# Learning Goal

- Understand the architecture of a virtual machine
- Be able to implement an own interpreter
- Know the procedural runtime support

# Virtual Machine

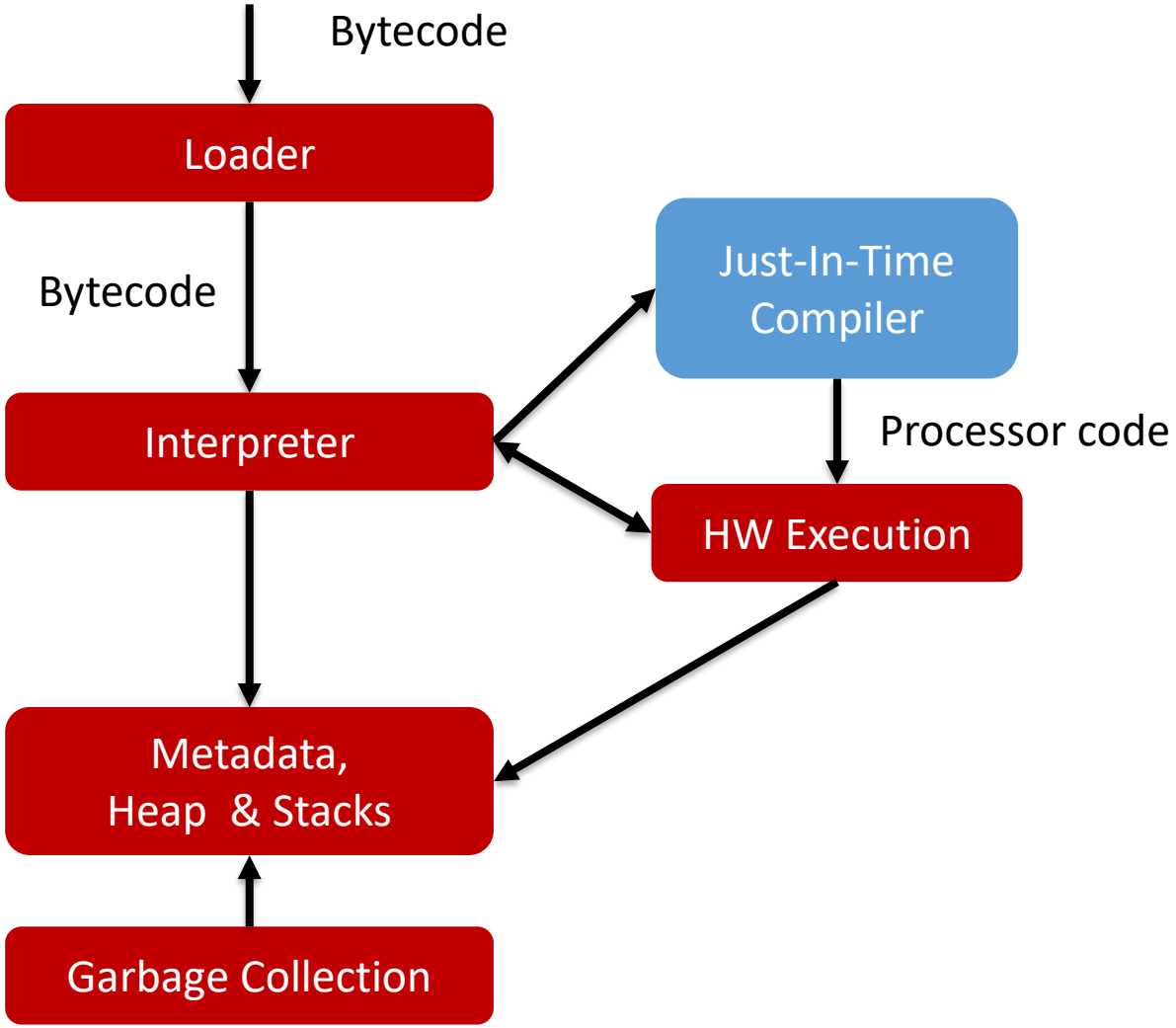
- Hypothetical machine with virtual processor
  - Custom instruction set (intermediate language/bytecode)
  - Wrapper around the real processor
- Advantages
  - Multi-platform
  - Multi-language
  - Security



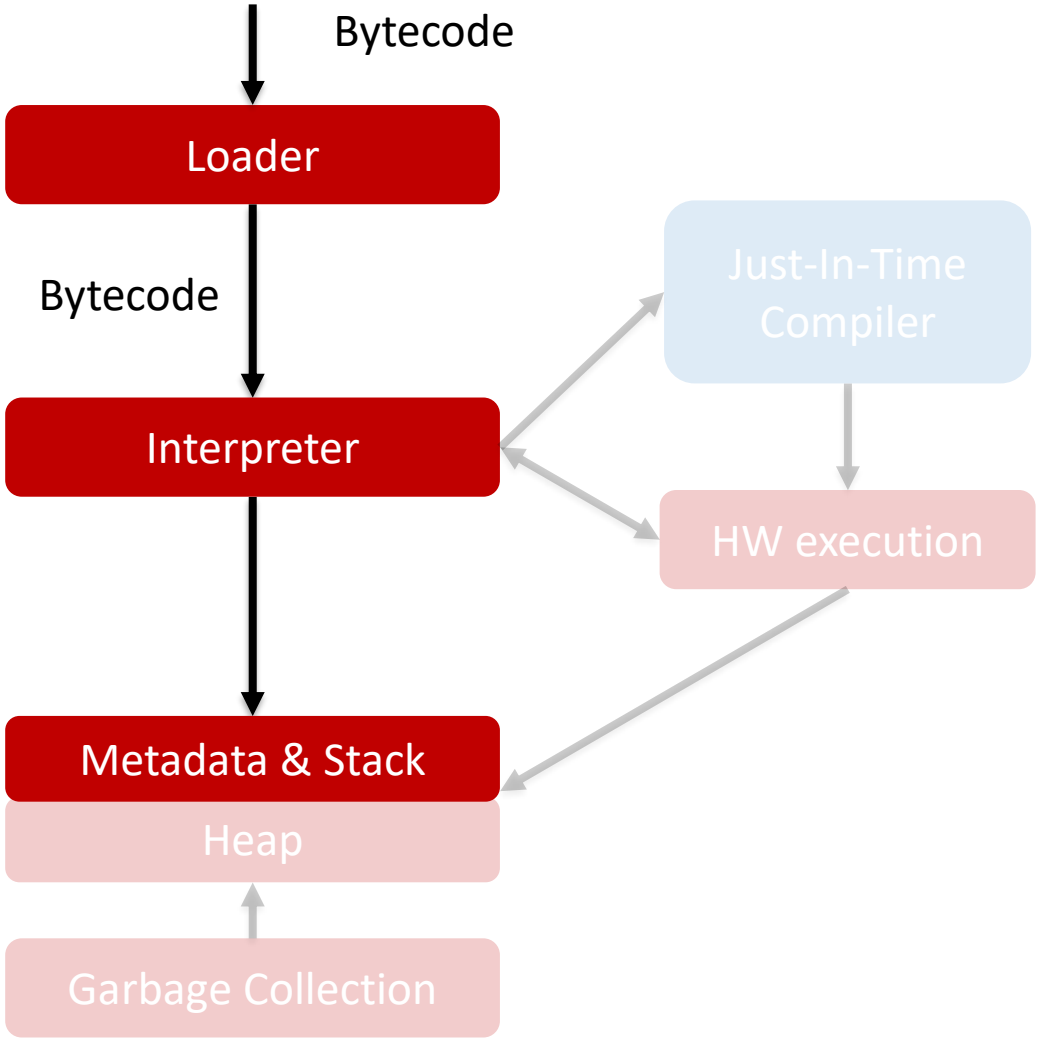
# History

- First intermediate language in 1966 (O-Code)
- 1975 Pascal P-Code
  - Stack machine (postfix notation)
  - One frontend, multiple backends
  - UCSD Apple II implementation
- 1980 Modula M-Code
  - High code density
  - VM in firmware
- 1995 JVM Java Virtual Machine
- 2000 Microsoft .NET Framework

# Virtual Machine Structure



# Our Focus Today



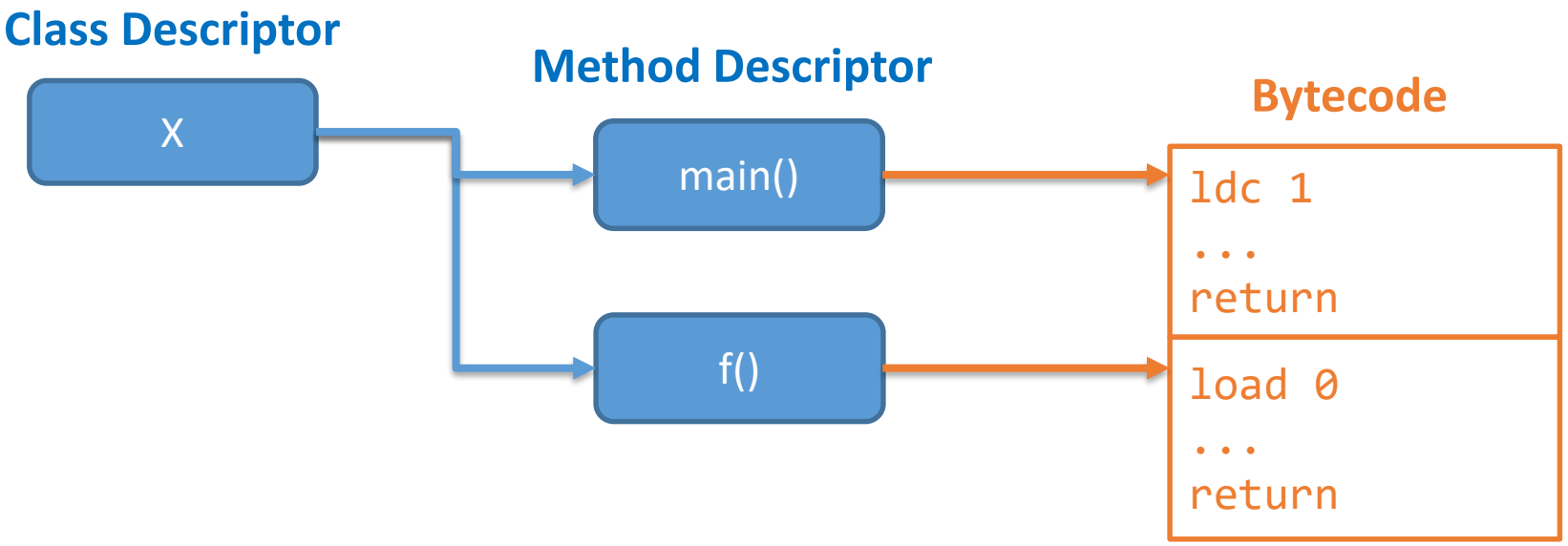
# Loader

- Loads bytecode code (file) in memory
- Allocates memory
  - Metadata for classes, methods, variables, code
- Defines layouts
  - Memory regions for fields/variables/parameters
- Address relocation
  - Resolves references to methods, types, other assemblies
- Initiates program execution
  - interpretation or compilation (JIT)
- Optional: Verifier

# Verifier

- Detect and prevent invalid bytecode
  - Static analysis at loading time
  - Compiler errors, malicious manipulation etc.
- Possible errors
  - Type errors
  - Evaluation stack overflow/underflow
  - Undefined variables/methods/classes
  - Illegal branches
  - …
- Alternative: Runtime checks
  - Our approach

# Metadata

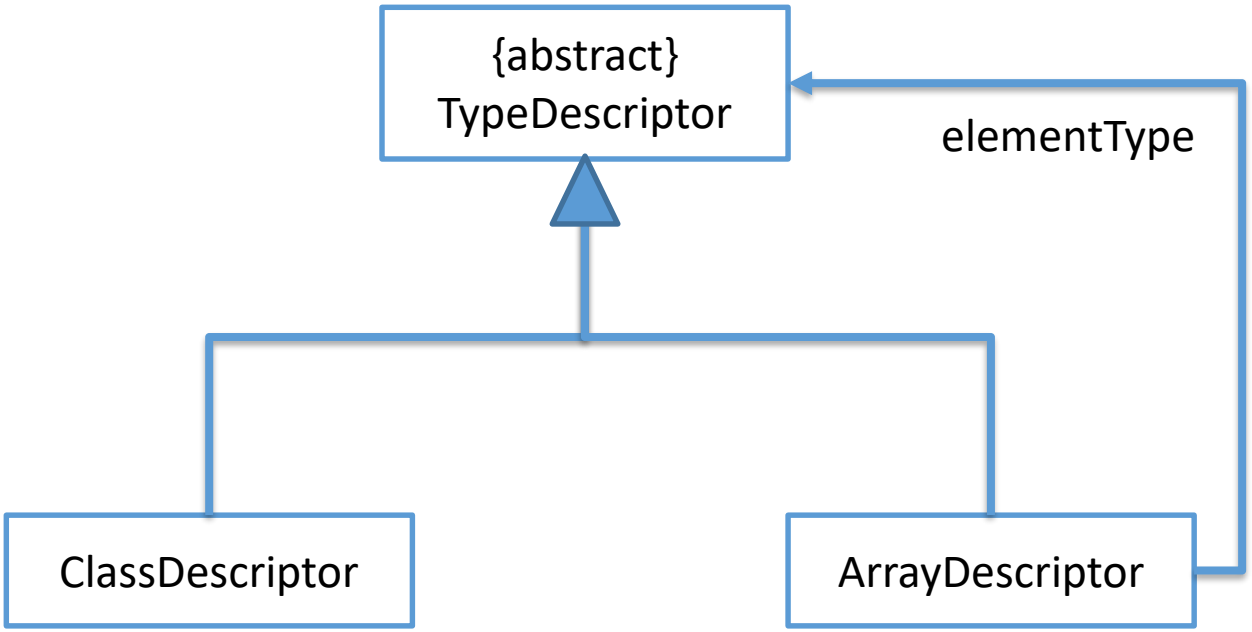

Class Descriptor
Method Descriptor
Bytecode
X
main()
f()
ldc 1
...
return
load 0
...
return

# Descriptors

- Runtime information for types & methods
  - Types: Classes, arrays, or base types
  - Classes: Field types
  - Methods: Types of parameters & locals, return type, bytecode

*What else could be recorded in class descriptors?*

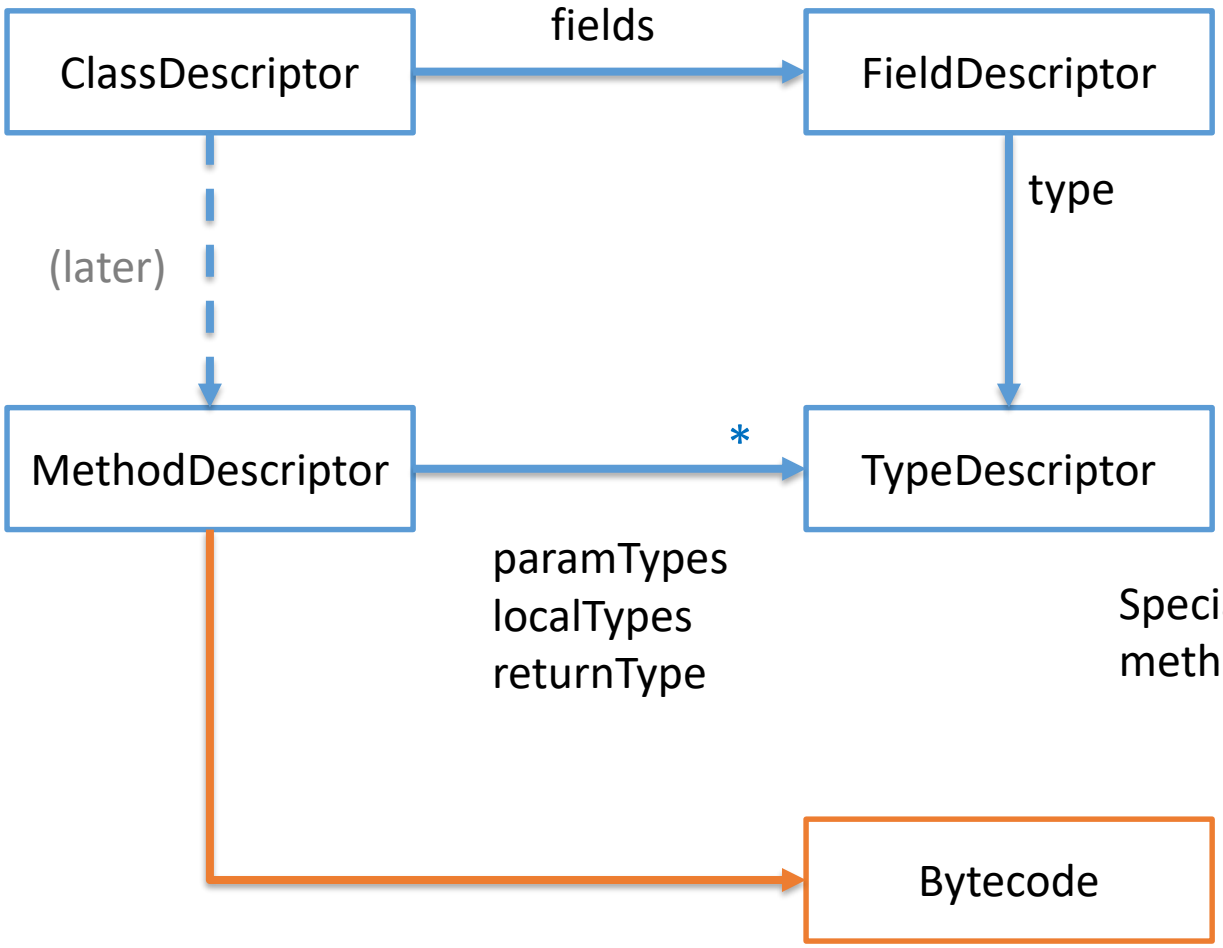
# Type Descriptors




Special inbuilt types:

- boolean
- int
- string

# Class & Method Descriptors

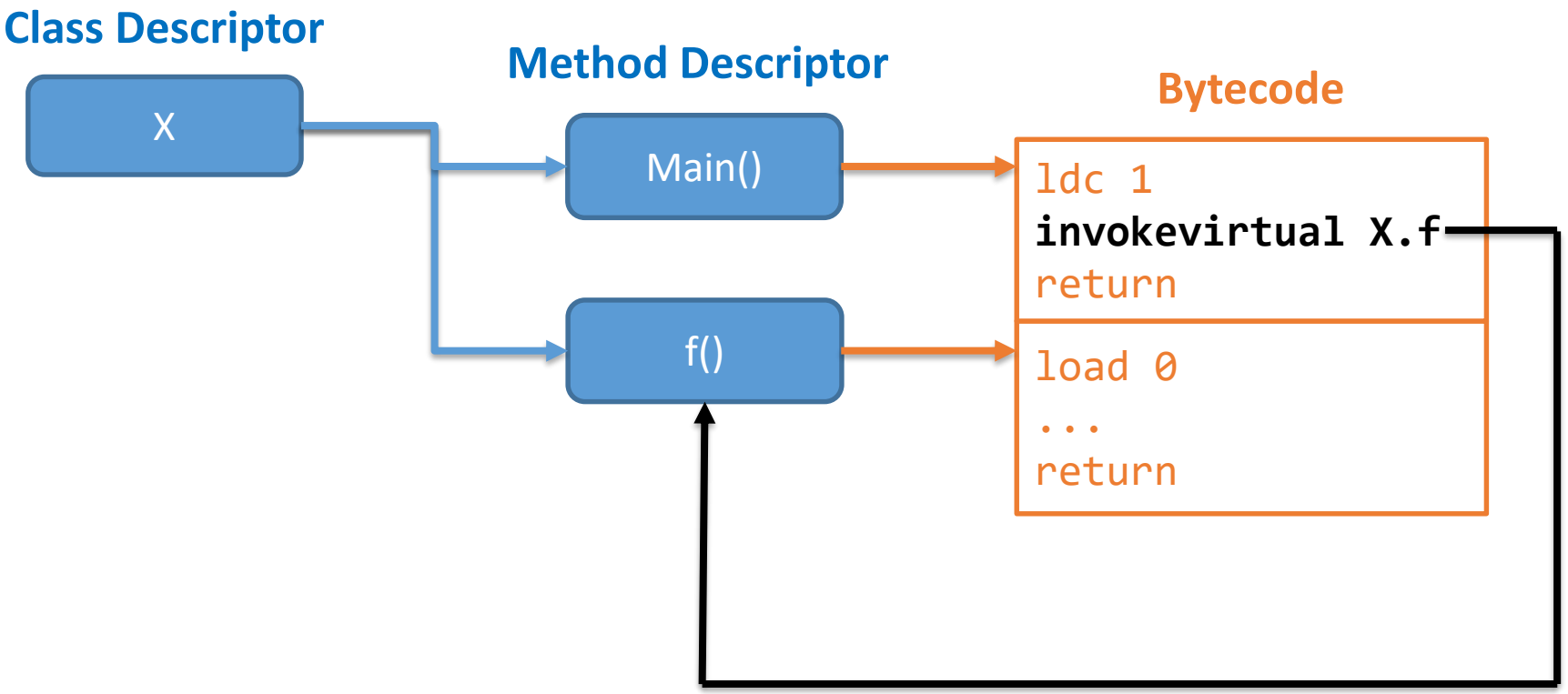


# Bytecode

- Loaded from file directly in memory
- Patching: Adjust instruction arguments
- References to corresponding descriptors

| Original | Patched |
|---|---|
| `invokevirtual MyMethod` | `invokevirtual <method_desc>` |
| `new MyClass` | `new <class_desc>` |
| `newarr MyType` | `newarr <type_desc>` |
| `getfield MyField` | `getfield <field_desc>` |
| `putfield MyField` | `putfield <field_desc>` |
| … | |

# Patching

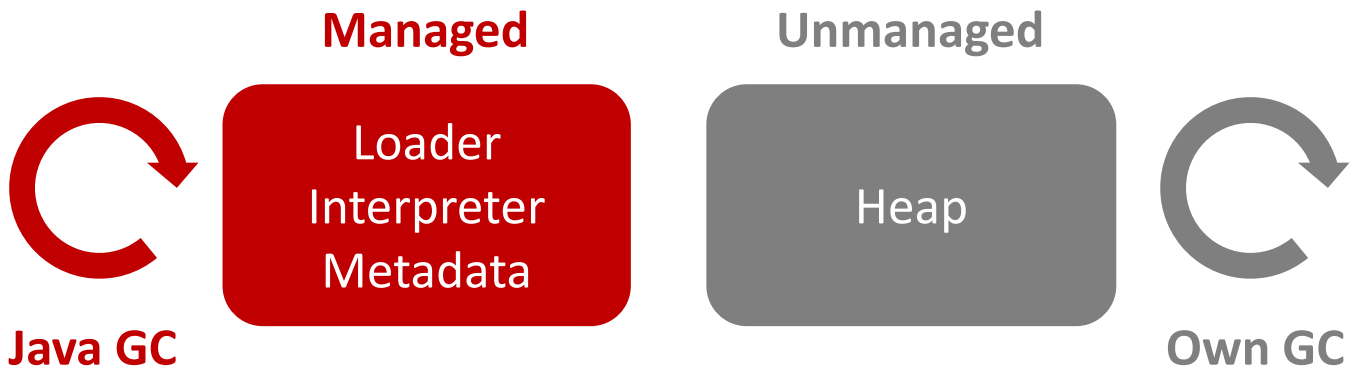

Class Descriptor
Method Descriptor
Bytecode
X
Main()
f()
ldc 1
invokevirtual X.f
return
load 0
...
return

# Programming Language

- For convenience, we implement the VM in Java
  - No C or C++
- Disadvantage: We get managed runtime support
  - But we want to build our own GC later
  - “Managed in managed” does not make any sense
- Solution: Managed Java + Unmanaged Native Access

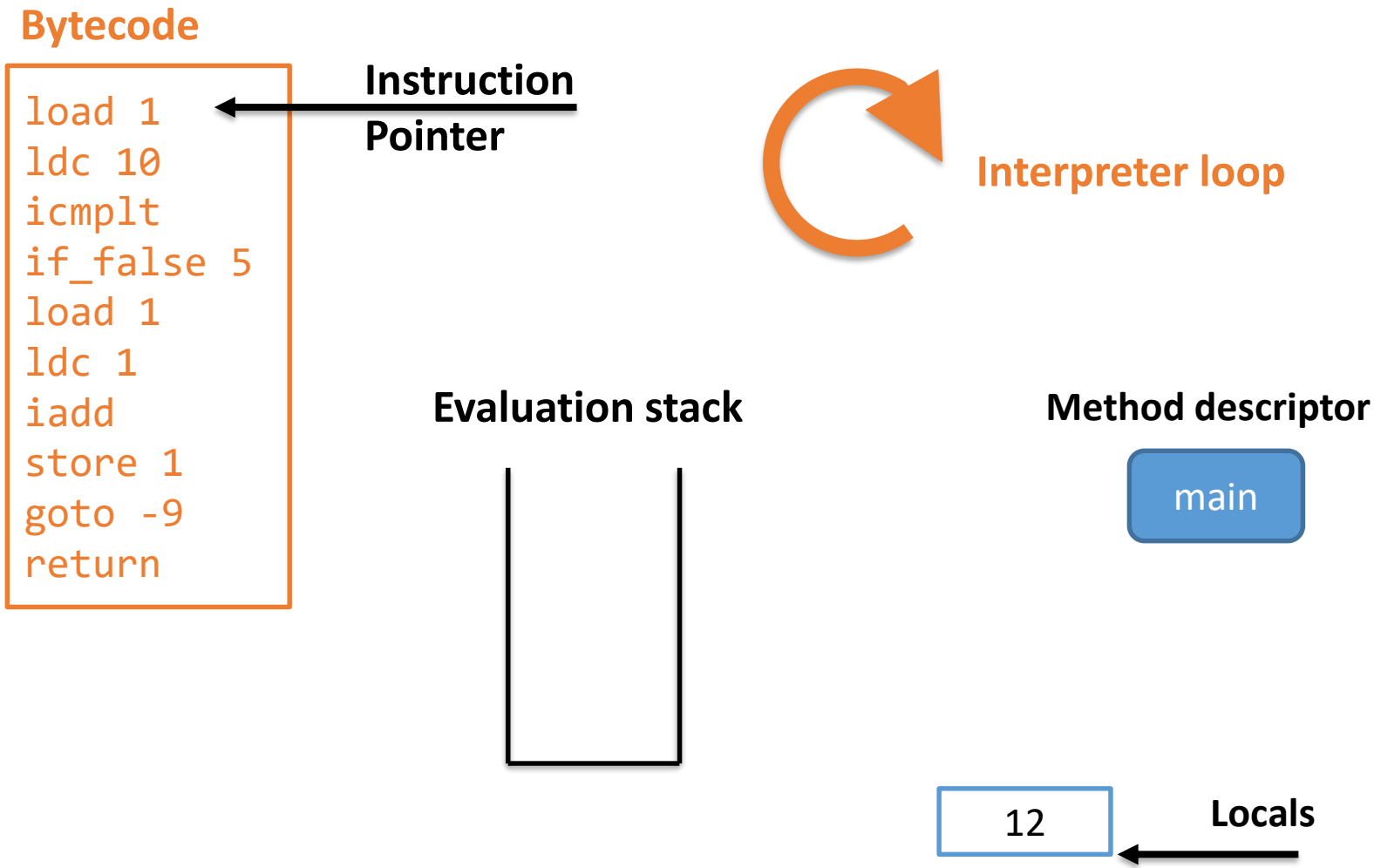
# VM: Managed & Unmanaged

- Unmanaged pieces besides Java VM
  - Heap and HW execution (JIT)




We skip unmanaged for the moment: Covered later

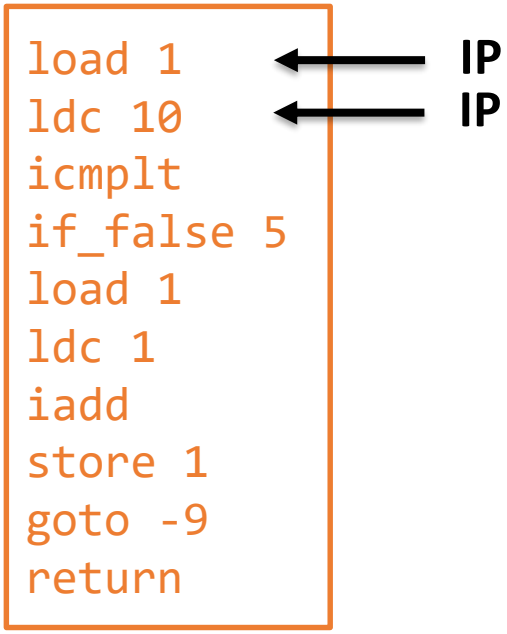
# Interpreter



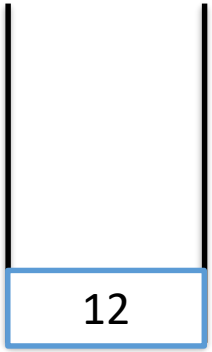
# Interpreter Components

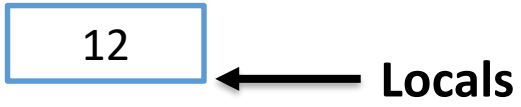
- Interpreter loop
  - Emulates instructions one after the other
- Instruction Pointer (IP)
  - Address of the next instruction
- Evaluation stack
  - For virtual stack processor
- Locals & parameters
  - For active method
- Method descriptor
  - For active method

# Interpretation

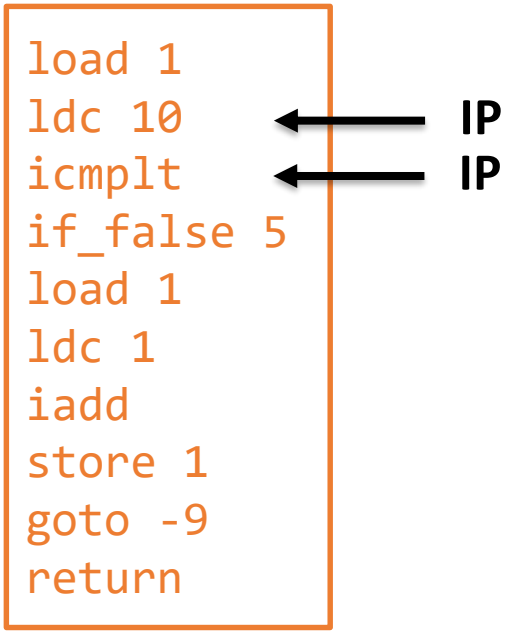



```
value = locals[1]; // 12
push(value);
```

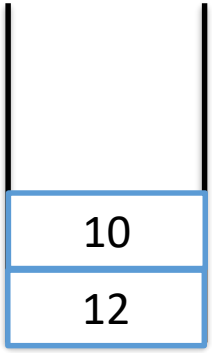


# Interpretation




push(10)



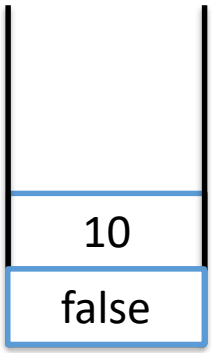
# Interpretation

```
load 1
ldc 10
icmplt
if_false 5
load 1
ldc 1
iadd
store 1
goto -9
return
```

IP

IP

```
right = pop(); // 10
left = pop(); // 12
result = left < right; // false
push(result);
```



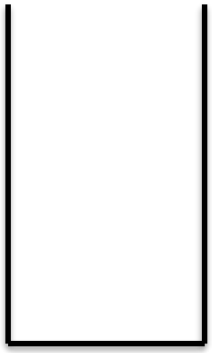
# Interpretation

```
load 1
ldc 10
icmplt
if_false 5
load 1
ldc 1
iadd
store 1
goto -9
return
```

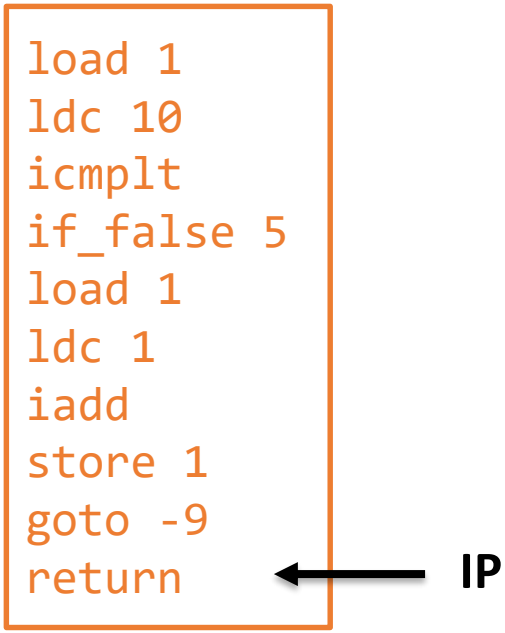
IP

IP

```
condition = pop(); // false
if (!condition) {
   IP += 5;
}
```



# Interpretation



# Interpreter Loop

```
while (true) {
  var instruction = code[instructionPointer];
  instructionPointer++;
  execute(instruction);
}
```

# Execution

- execute() emulates instruction depending on opcode

```
switch(instruction.getOpCode()) {
  case LDC:
    push(instruction.getOperand());
    break;
  case IADD:
    var right = pop();
    var left = pop();
    var result = left + right;
    push(result);
    break;
  …
}
```

# Interpretation Patterns

ldc

```
push(instruction.getOperand())
```

load

```
var index = instruction.getOperand();
push(getParamOrLocal(index));
```

store

```
var index = instruction.getOperand();
setParamOrLocal(index, pop());
```

get/setParamOrLocal

| | |
|---|---|
| 0 | “this” (read-only) |
| 1..N | N parameters |
| N+1..M | M locals |

# Interpretation Patterns

iadd

```
var right = pop(), left = pop();
push(left + right);
```

Analogous for isub, imul, idiv, irem etc.

goto

```
instructionPointer += (int)instruction.getOperand();
```

if_true

```
if (pop()) {
  instructionPointer += (int)instruction.getOperand();
}
```

Analogous for if_false

# Procedural Support

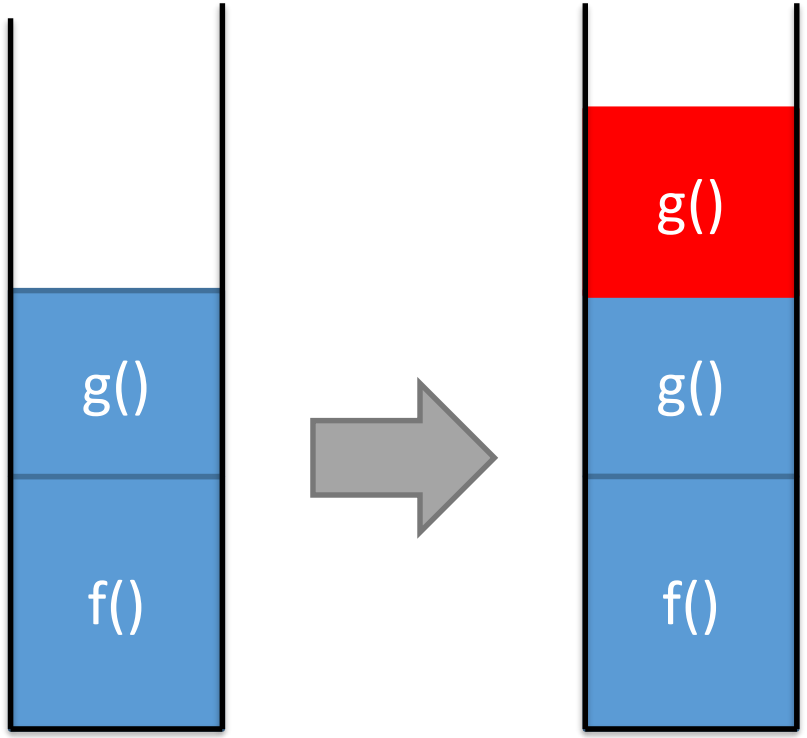
- Method calls
  - invokevirtual = Call new method
  - return = Exit from method
- Activation frame
  - Memory space for method
  - Parameter, local variables, temporary evaluation
- Call stack
  - Stack of activation frames according to call order

# Call Stack

Call f()

Call g()

Call g()

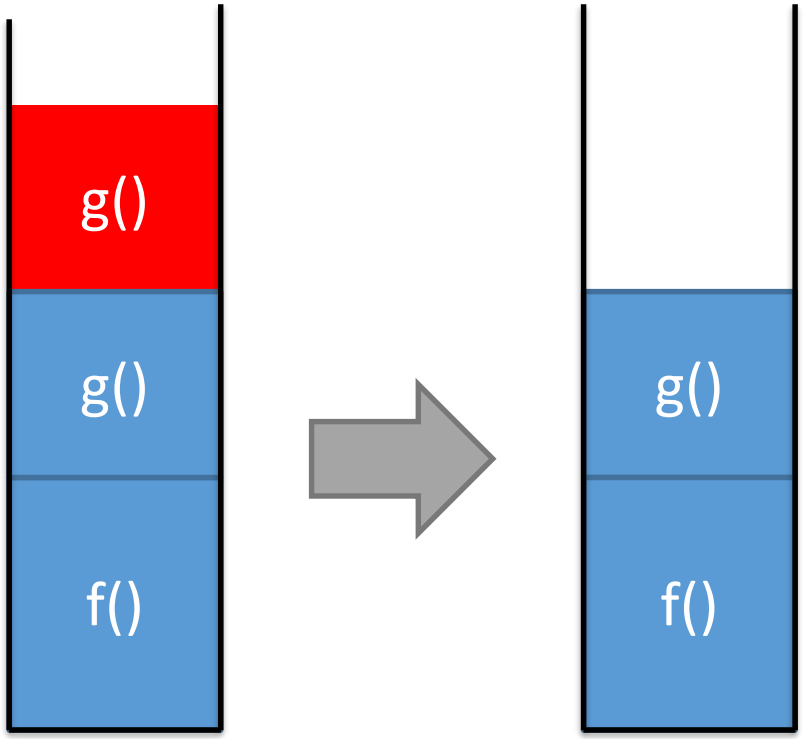



New activation frame

For each method call, push new frame on call stack

# Method Return

return




Pop frame from call stack

# Call Stack Design

- Managed call stack in interpreter
  - Object-oriented representation => convenience
- Unmanaged call stack in HW execution
  - Contiguous memory block => efficiency

# Managed Call Stack

```
class ActivationFrame {
  private MethodDescriptor method;
  private Pointer thisReference;
  private Object[] arguments;
  private Object[] locals;

  private EvaluationStack evaluationStack;
  private int instructionPointer;
  …
}

class CallStack {
  private Deque<ActivationFrame> stack;
}
```

*Why does every frame have its own evaluation stack and instruction pointer?*

# Method Call

```
var method = (MethodDescriptor)instruction.getOperand();

var nofParams = method.getParameterTypes().length;
var arguments = new Object[nofParams];
for (int i = arguments.length – 1; i >= 0; i--) {
  arguments[i] = pop();
}
var target = pop();

var frame = new ActivationFrame(method, target, arguments);
callStack.push(frame);
```

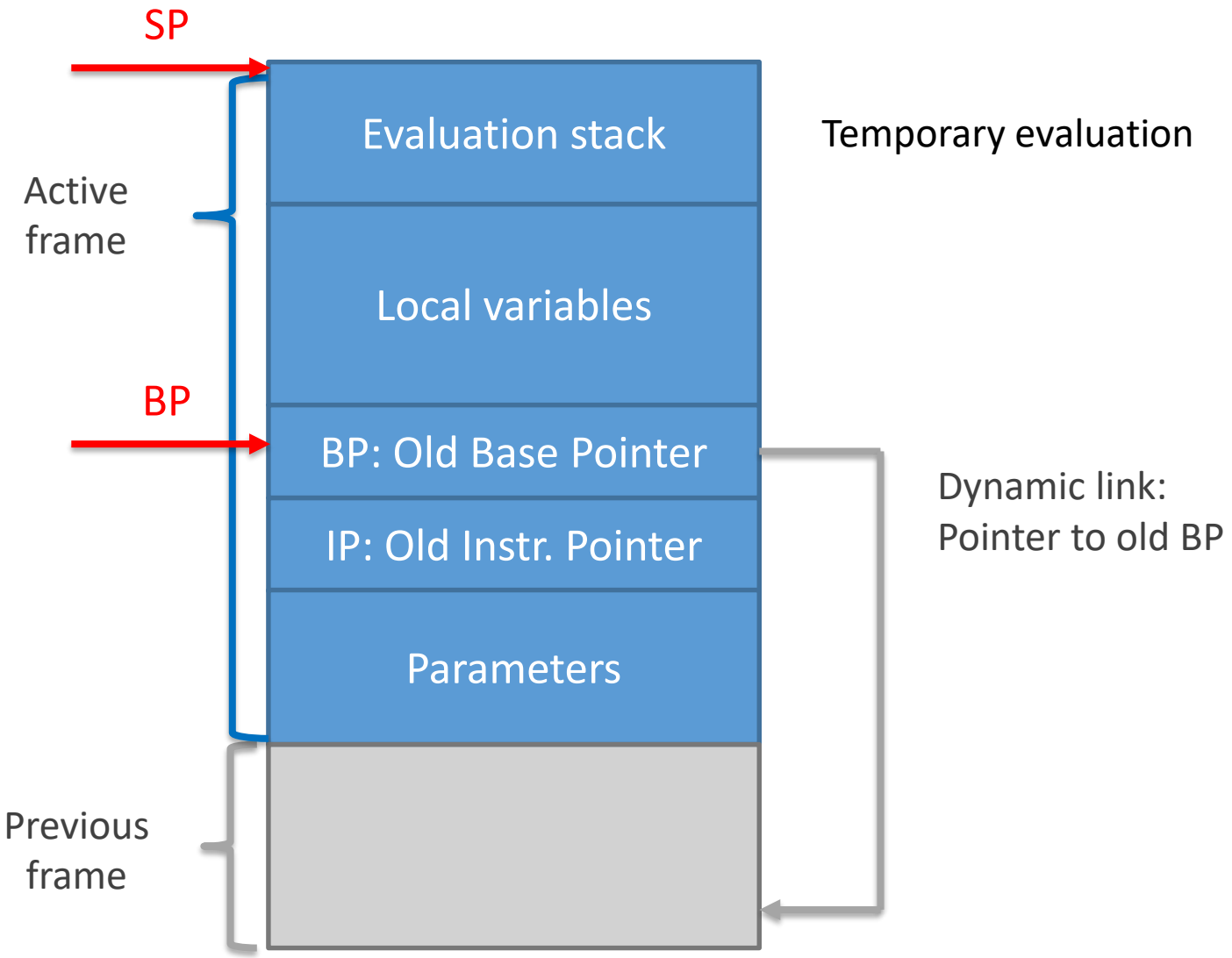
*Do we need any additional logic?*

# Method Return

```
var method = activeFrame().getMethod();
var hasReturn = method.getReturnType() != null;
Object result = null;
if (hasReturn) {
  result = pop();
}
callStack.pop();
if (hasReturn) {
  push(result);
}
```

*What should we add?*

# Unmanaged Call Stack (Alternative)



# Review: Learning Goal

- ✓ Understand the architecture of a virtual machine
- ✓ Be able to implement an own interpreter
- ✓ Know the procedural runtime support

# Further Reading

- Dragon Book, Runtime Environment
  - Section 7.1-7.2: (Unmanaged) call stacks
- Optional, if interested
  - B. Venners. Inside the Java Virtual Machine.
    - https://www.artima.com/insidejvm/ed2
    - Chapter 5 (JVM)