



# Virtual Machines and Runtime

A focus into the Java and Javascript Runtime Environments

# Table of Contents

---

- Virtual Machines (VM)
- Java Virtual Machine (JVM)
- Runtime Environment (RTE)
- Java Runtime Environment (JRE)
- Interpreter vs Compiler
- Compiler
  - ◆ Steps
  - ◆ Strategies
  - ◆ Advantages/Disadvantages
- JIT Optimizations

- Benchmarks
- A look into some runtimes:
  - ◆ Hotspot
  - ◆ OpenJ9
  - ◆ GraalVM
  - ◆ Rhino
  - ◆ Nashorn
  - ◆ ChakraCore
  - ◆ V8

# What is a Virtual Machine?

---

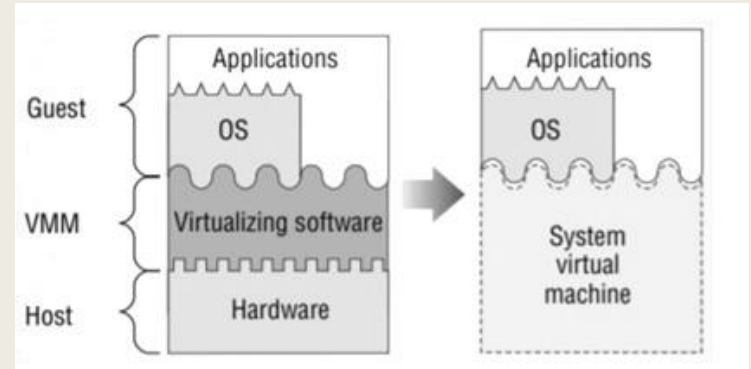
- A **virtual machine** is a **virtualized** environment, defined by software, with the objective of **simulating** a physical machine
- It is executed on a machine called "host" and is considered as a "guest" because it is **abstracted** from its execution environment

# Types of Virtual Machines

---

## SYSTEM VIRTUAL MACHINE

- Execute a whole operating system.
- Allows sharing of a host computer's physical resources between multiple VMs, running its own copy of the OS.
- Using an hypervisor (software that creates and runs VMs) that can run directly on hardware (VMware ESXI) or on top of an OS (VirtualBox).

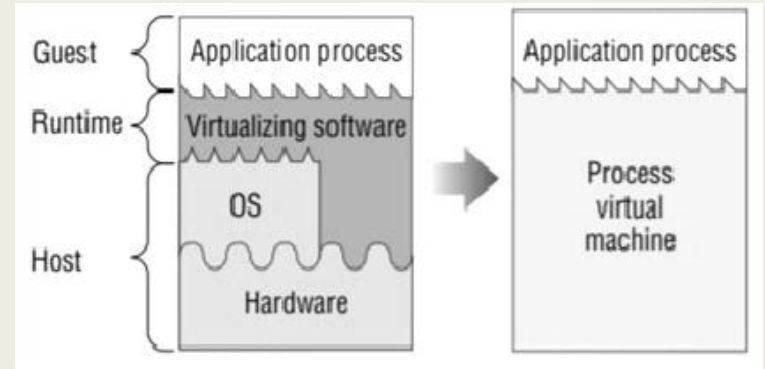


# Types of Virtual Machines

---

## PROCESS VIRTUAL MACHINE

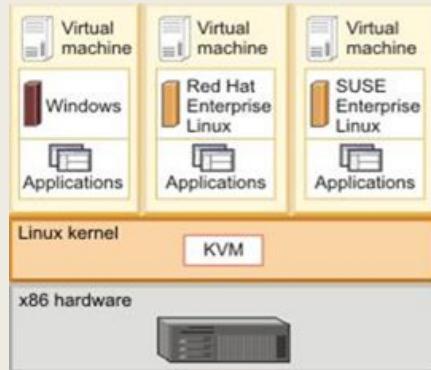
- Used for executing a single process.
- Masks the information of the underlying hardware/OS.
- Example with Java Virtual Machine (JVM) allows any system to run Java applications as if they were native to the system.



# Hypervisors

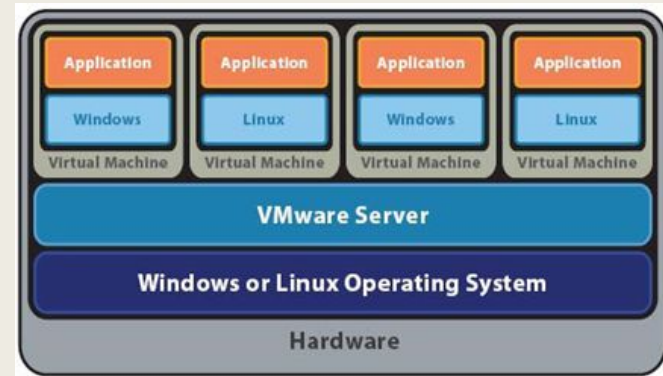
## TYPE I HYPERVISOR

- Runs directly on the hardware.  
Ex : KVM (Linux kernel)



## TYPE II HYPERVISOR

- Runs on top of an OS.  
Ex : VMware Workstation, Oracle VirtualBox



# Java Virtual Machine (JVM)

---

The Java Virtual Machine is an abstract computing machine that

- ◆ follows the JVM Specification
- ◆ contains an instruction set
- ◆ manipulates various memory areas at run time

# Java Virtual Machine (JVM)

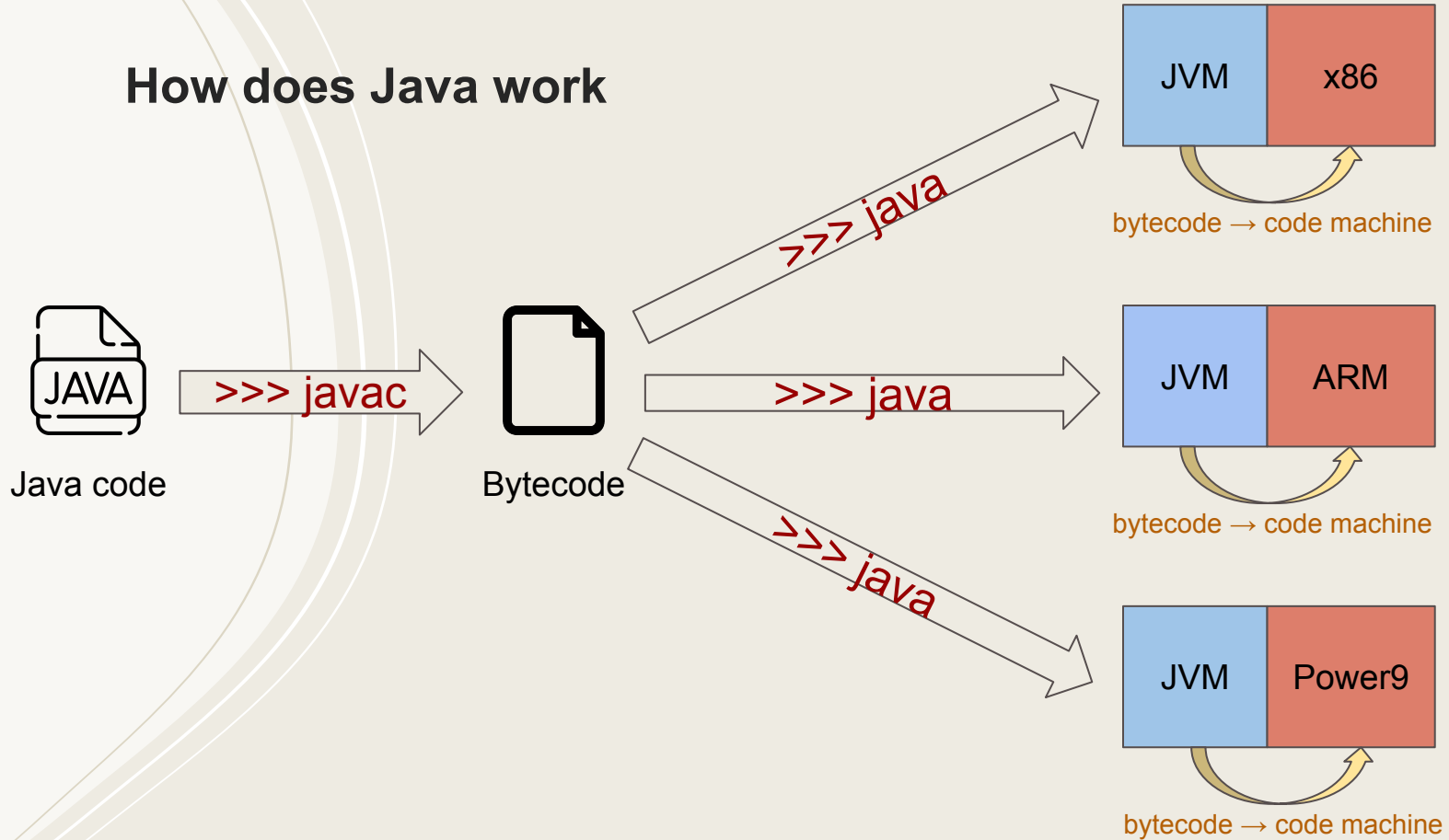
---

The JVM:

- ◆ offers a garbage collector for removing unreferenced objects.
- ◆ has an instruction set called java bytecode and is described in the *JVM Specification*
- ◆ runs binary files that contains JVM instructions
- ◆ is machine dependant, but is used to run programmes in a machine-independent environment
- ◆ can run any language that can be compile into JVM instructions



# How does Java work



# Runtime Environment (RTE)

---

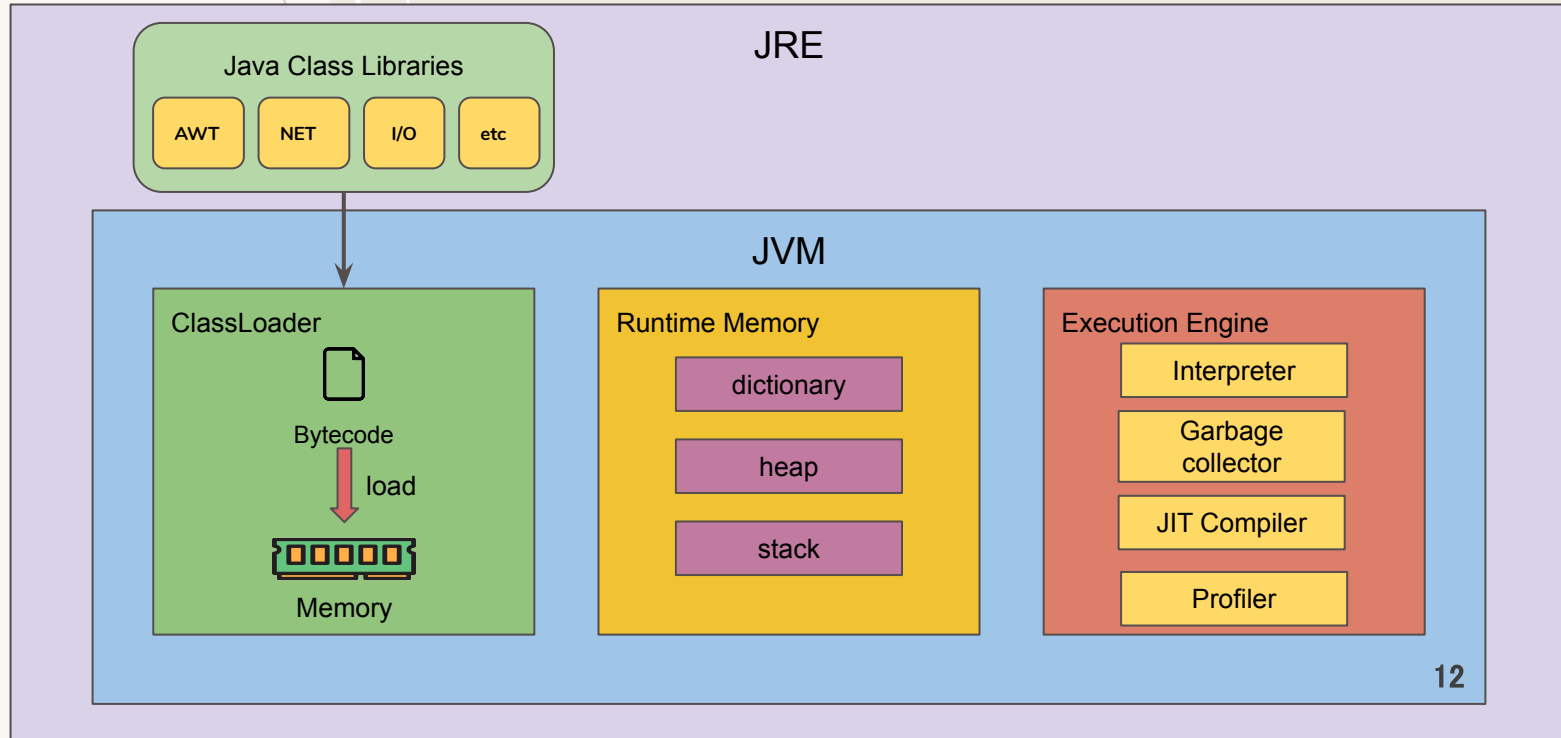
- The runtime of a program is the execution and running phase.
- A Runtime Environment provides all the functionalities and dependencies necessary to run a program independently of the underlying operating system, which allows the program to have the same user interfaces regardless of the OS.
- Provides basic functions for memory, network and hardware.

# Java Runtime Environment (JRE)

---

- Combines the Java code created by the JDK with the dependencies required (libraries) and creates an instance of the JVM to run the resulting program.
- Enables a Java program to run in any OS without modification.
- Automatic memory management via Garbage Collection

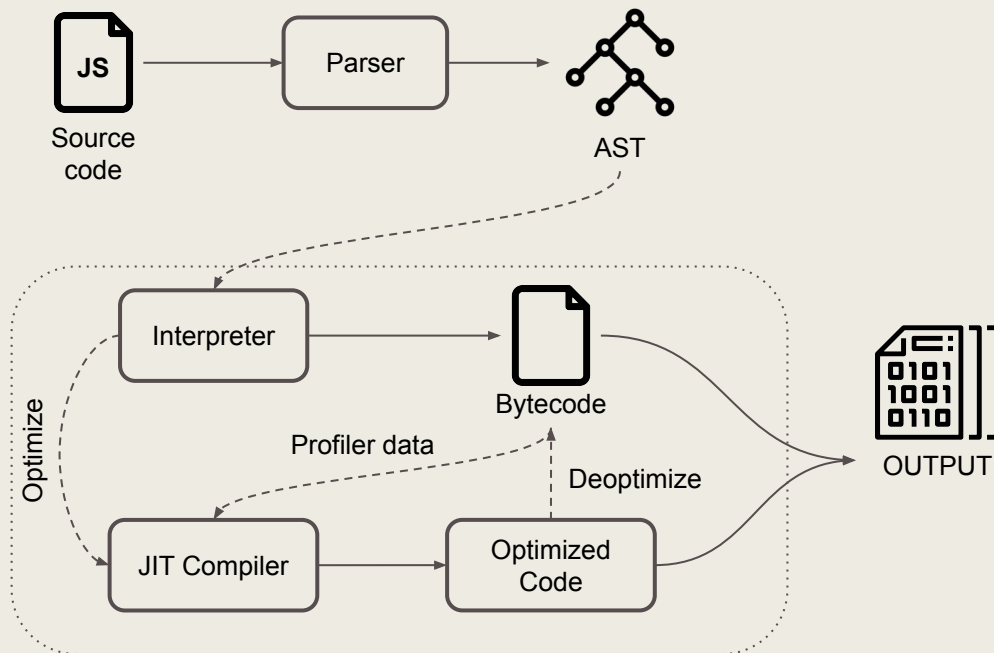
# The Java Runtime Environment (JRE)



# Interpreter vs Compiler

INTERPRETER	COMPILER
Translates one line at a time into intermediary code at runtime.	Scans the entire program and translates it into machine code at once
Takes less time to analyse the source code, but the process execution time is much slower	Takes a lot of time to analyse the source code, but the process execution time is much faster
Does not store intermediary code on the disk.	Always generates intermediary object code. Needs further linking, so more memory is needed
Keeps translating the program until an error is found. Execution is stopped if an error is spotted	Generates error messages when a syntax error is found.
Languages using interpreters : Ruby or Python, JavaScript	Languages using compilers : C, C++, Java

# Javascript Interpreter: V8 as an example



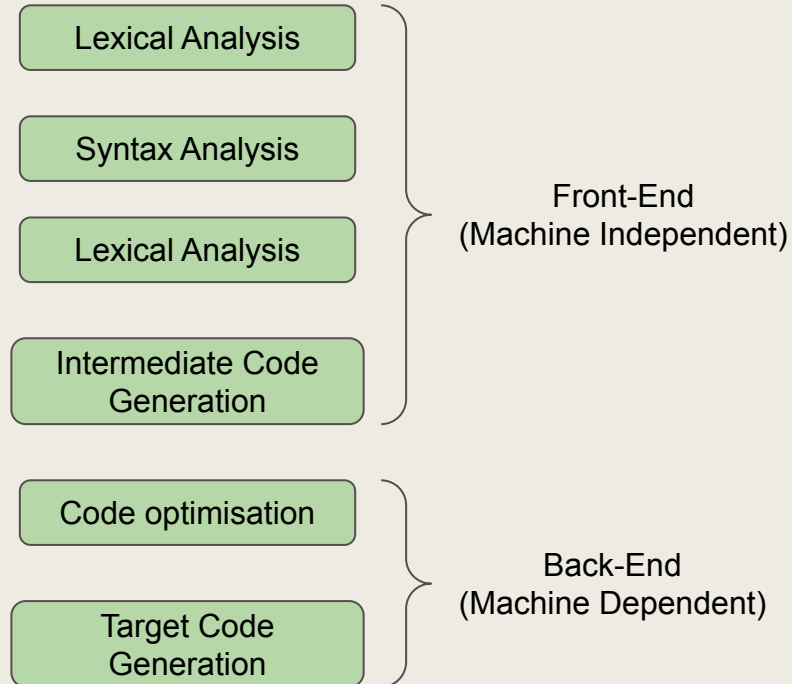
# Compiler

---

- A compiler is a program that translates source code into another lower-level language.
- Its role is to check for all possible errors in a source program, such as spelling errors, variables, types.
- The executable result is some form of machine-specific binary code if no errors are found.
- There are different types of compilers : we are going to talk about the Java JIT compiler.

# Phases of a compiler

---

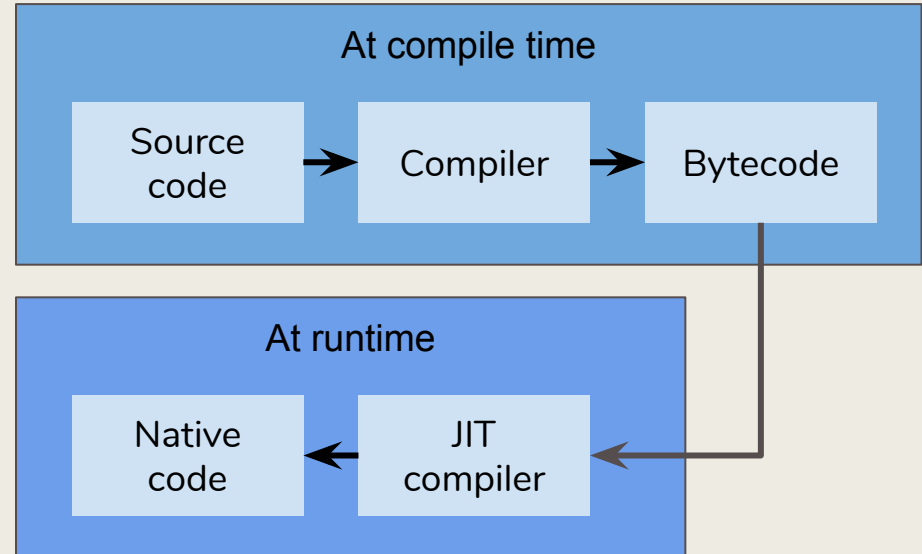




# Just-In-Time (JIT) Compiler

---

- Used to improve the performance of applications at run time.
- Compiles « just in time » bytecodes into native machine code at runtime.
- The JVM calls the compiled code instead of interpreting it.
- JIT compilers combine interpreter and compiler principals to get the best of both worlds.



# Static Single Assignment (SSA)

---

- Property of an intermediate representation (IR), which requires that each variable is assigned exactly once.
- Every variable must be defined before it is used.
- Existing variables in the original IR are split into versions:
  - ◆ Every definition gets its own version.

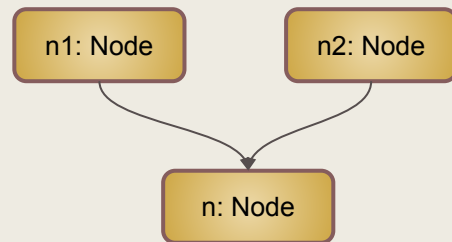
# Sea of Nodes

---

The sea of nodes is a data structure for representing a program in SSA form.

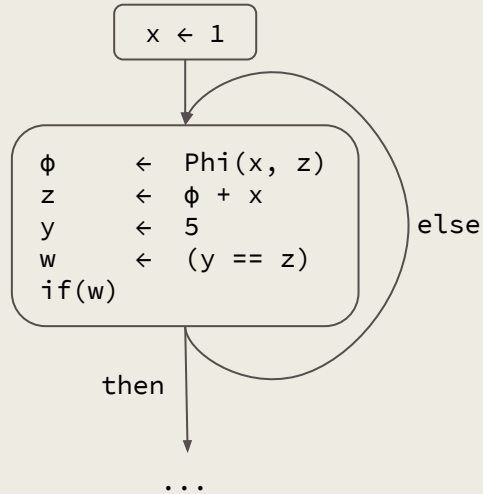
- Instructions and control flow auxiliary helpers are put in “nodes”, represented in an oriented graph.
- Programs become more flexible to optimize.

In this example, n1 and n2 nodes can be seen as inputs or predecessors of the node n.

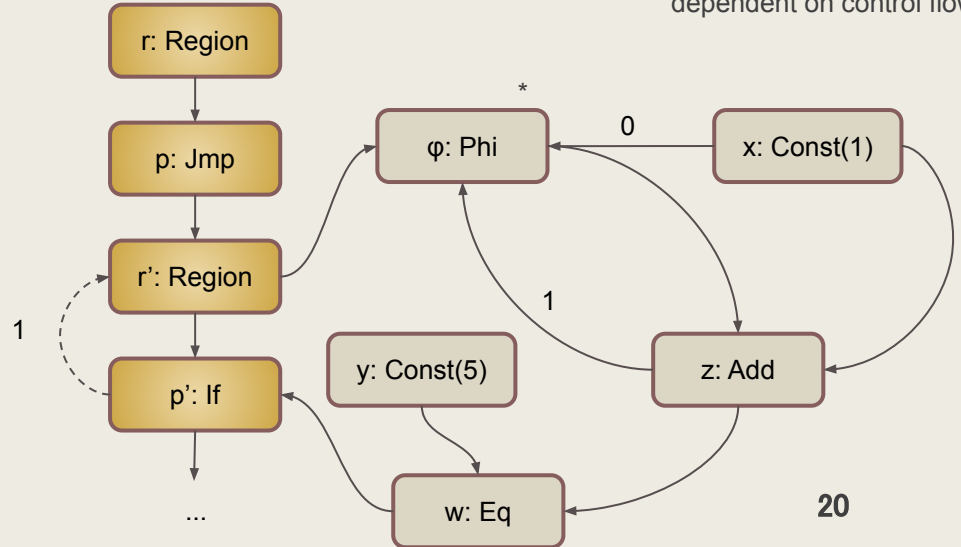


# Sea of Nodes

## PROGRAM WITH FLOW-CONTROL



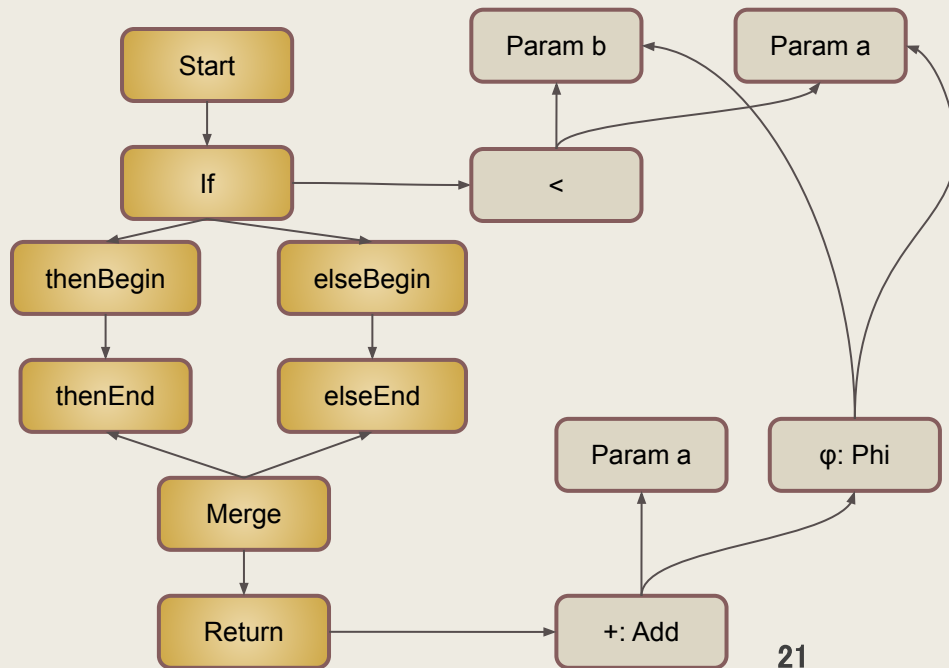
## SEA OF NODES GRAPH



\* Phi nodes merge data flow and other side effects (memory, I/O) dependent on control flow.

# Sea of Nodes

```
int addMin(int a, int b, int sum) {  
    int min;  
    if (a > b) {  
        min = b;  
    } else {  
        min = a;  
    }  
    return min + sum;  
}
```



## JIT : Multiple strategies

---

- Just in time : transform into assembly at the first call
- Tiered - 2 JITs : 1 fast and simple and 1 optimized (slower)
- Mixed-mode : Interpreter + 1 JIT or more
- Ahead of time : code is optimized at the installation

## Advantages / Disadvantages of a JIT compiler

---

ADVANTAGES	DISADVANTAGES
Run after a program starts	Startup time can be slow
Code optimization is done while the code is running	Heavy usage of cache memory
Can utilize different levels of optimization	Can increase the level of complexity of a program
Less memory usage : only methods required at run-time are compiled into machine code	

## JIT optimizations on loops

---

- Loop Reduction
- Loop Unrolling
- Loop Peeling
- Loop Striding
- Loop-Invariant Code Motion
- Loop Versioning & Specialization



## Optimization: Loop unrolling

---

- The JIT compiler opens up the loop and repeats the corresponding Assembly instructions one after another.

```
public void foo (int [] arr, int a) {  
    for (int i= 0; i < arr.length; i++) {  
        arr [i] += a;  
    }  
}
```



```
public void foo (int [] arr, int a) {  
    int i = 0;  
    for (; i < (arr.length - 4); i += 4) {  
        arr [i] += a;  
        arr [i+1] += a;  
        arr [i+2] += a;  
        arr [i+3] += a;  
    }  
    for (; i < arr.length; i++) {  
        arr [i] += a;  
    }  
}
```

## JIT optimizations on Method Calls

- When possible, the JIT compiler would try to inline method calls and eliminate the jumps of going there and back, the need to send arguments, and returning a value and transferring its whole content to the calling method.

```
private static void calcLine(int a, int b, int from, int to) {  
    Line l = new Line(a, b);  
    for (int x = from; x <= to; x++) {  
        int y = l.getY(x);  
        System.err.println("(" + x + ", " + y + ")");  
    }  
}  
  
static class Line {  
    public final int a;  
    public final int b;  
    public Line(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
    // Inlining  
    public int getY(int x) {  
        return (a * x + b);  
    }  
}
```

```
private static void calcLine(int a, int b, int from,  
int to) {  
    Line l = new Line(a, b);  
    for (int x = from; x <= to; x++) {  
        int y = (l.a * x + l.b);  
        System.err.println("(" + x + ", " + y + ")");  
    }  
}
```

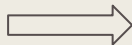
- Eliminate the jump, the send of the arguments l and x, and the returning of y

# JIT Optimizations - Null Check Elimination

---

- Checking for null is common in Source Code
- But these checks can be eliminated from optimized Assembly Code
  - ◆ To avoid unnecessary jumps that slow down execution
- At runtime, the JIT knows if an object is null or not
- So it can perform this kind of transformation :

```
private static void runSomeAlgorithm(Graph graph) {  
    if (graph == null) {  
        return;  
    }  
    // do something with the graph  
}
```



```
private static void runSomeAlgorithm(Graph graph) {  
    // do something with the graph  
}
```

- “Uncommon trap” mechanism if the condition eventually happens:

- ◆ JIT deoptimize code

# JIT Optimization - Branch Prediction

→ Try to decide which lines of code are “hotter”  
(*ie. happen more often*) during runtime and fetch them in advance

**Without branch prediction**

Time	1	2	3	4	5	6	7	8	9	10
a += 2	F	D	E	S						
if (a > 10)		F	D	E	S					
a *= 3						F	D	E	S	
a -= 4										
a -= 1							F	D	E	S

F : Fetch  
D : Decode  
E : Execute  
S : Store

**With right branch prediction**

Time	1	2	3	4	5	6	7	8
a += 2	F	D	E	S				
if (a > 10)		F	D	E	S			
a *= 3			F	D		E	S	
a -= 4								
a -= 1				F		D	E	S


**With wrong branch prediction**

Time	1	2	3	4	5	6	7	8	9	10
a += 2	F	D	E	S						
if (a > 10)		F	D	E	S					
a *= 3			F	D						
a -= 4					F	D	E	S		
a -= 1				F		F	D	E	S	

## Benchmarks performed

---

- Benchmark performed on two big arrays:
  - ◆ Array of native type (int)
  - ◆ Array of two different objects (boxed ints)
- Focus on the loop iteration performance by going through all elements of the arrays.
- Focus on the access to a variable boxed/non boxed by a class.



Let's talk about some specific runtimes.