JAVA VIRTUAL MACHINE

MID DEFENSE

Lokesh Pandey **Mandip Thapa** **Manish Kunwar**

TABLE OF CONTENT

WHAT OUR PROJECT IS

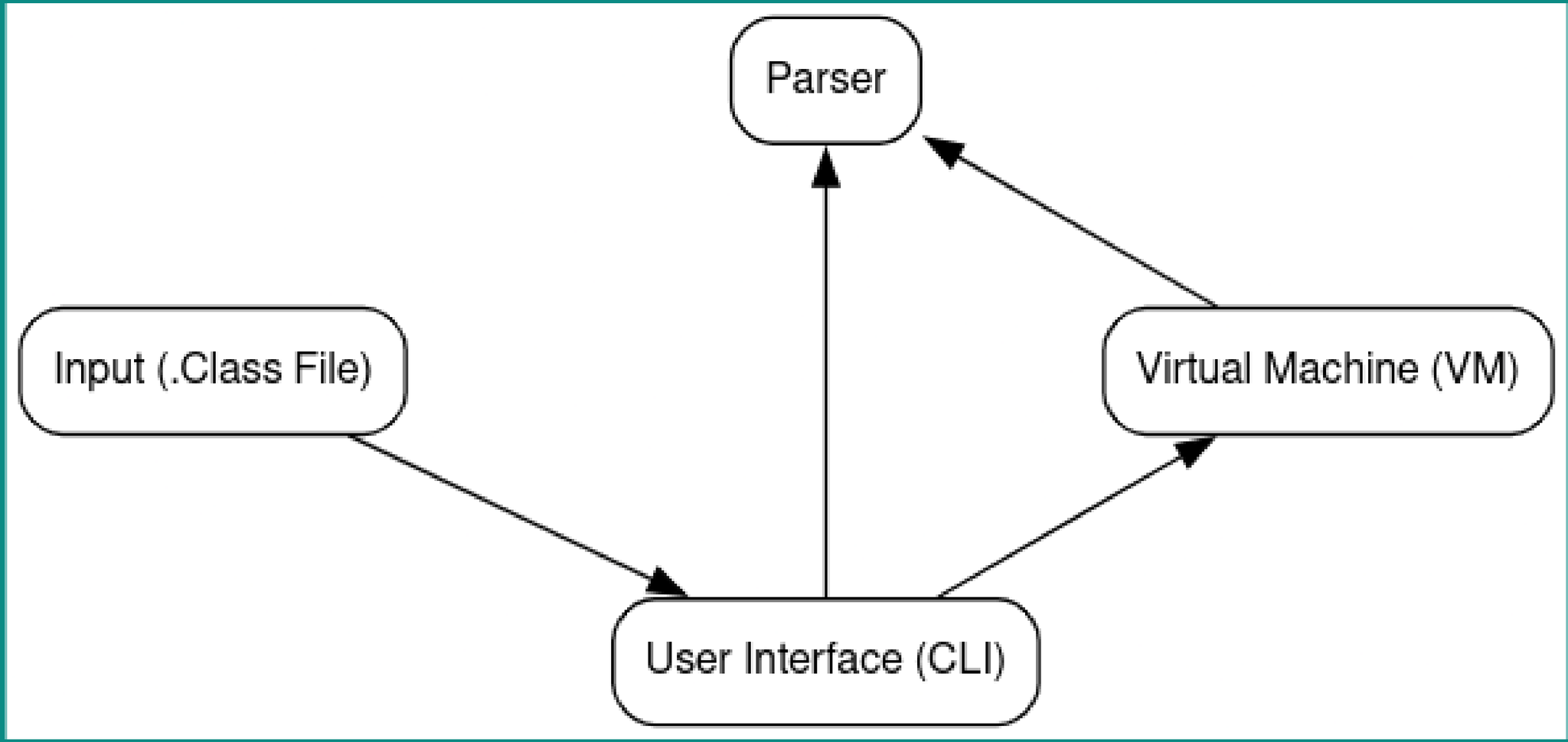
- A Java Virtual Machine (JVM) that executes Java bytecode.
- Parses and interprets .class files.
- Implements a basic runtime environment.
- Handles fundamental operations like stack management, method invocation, and basic memory management.

WHAT OUR PROJECT IS NOT

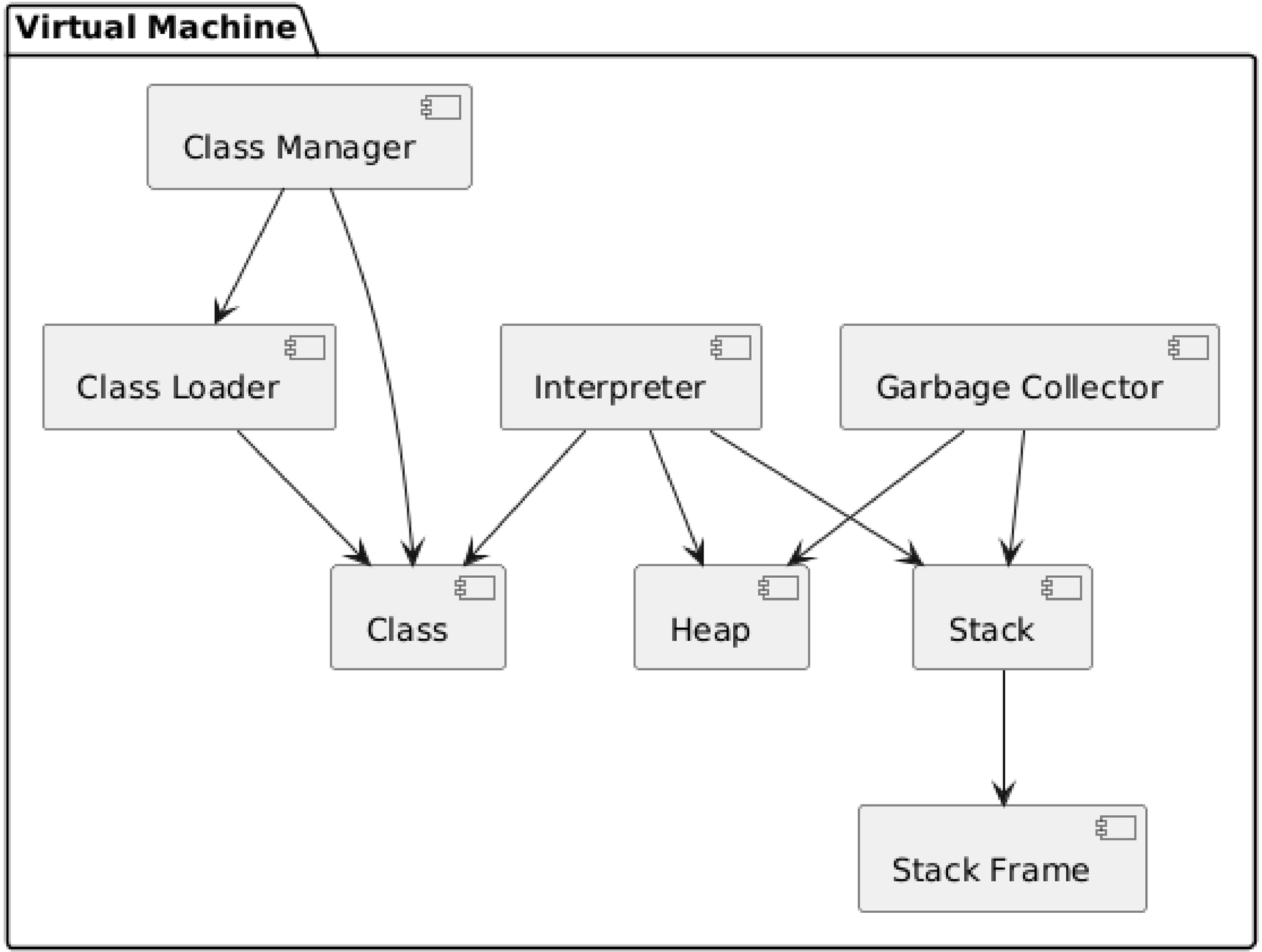
JDK (Java Development Kit)	JRE (Java Runtime Environment)	JVM (Java Virtual Machine)
Develop and run Java programs	Provides runtime for Java applications	Executes Java bytecode
Compiler (javac), JRE, JavaDocs, Debugger	JVM, core Java libraries, runtime environment	Class loader, execution engine, garbage collector
Developers (for coding, compiling, and debugging)	End-users running Java apps	Part of JRE, executes Java programs

SYSTEM DESIGN

System Overview



Virtual Machine



WORKING PRINCIPLE

CLASS FILE PARSER

Bytecode Parsing

Validates .class files and extracts metadata and bytecode.

Constant Pool Resolution

Resolves symbolic references into efficient data structures.

Intermediate Representation

Creates a structured representation of class file content.

Class File Structure

magic: U4 (32-bit)
version: U4 (32-bit)
constant_pool: Variable size
access_flags: Variable size
this_class: U2 (16-bit)
super_class: U2 (16-bit)
interfaces: Variable size
fields: Variable size
methods: Variable size
attributes: variable size

CLASS LOADER

Loading phase

Loads class files into memory using class loaders.

Linking Phase

Resolves references and allocates memory for class execution.

Initialization Phase

Executes static initializers to ensure thread-safe class setup.

Loaded Class

LoadedClass
Class Name: String
Super Class: Reference
Interfaces: References
Instance Fields: Feild Information
Instance Fields Indices: HashMap<String, index>
Static Fields: FieldInfo
Static Field Indices: HashMap<String, usize>
Static Values: Value[]
Methods: Methods Infos
Constant Pool: Reference
Access Flags: Class Access Flag
Code Cache: HashMap<NameDes, Code>>
Init State: Mutex<InitState>

INTERPRETER

The interpreter executes bytecode instructions using a fetch-decode-execute cycle, simulating a virtual stack machine.

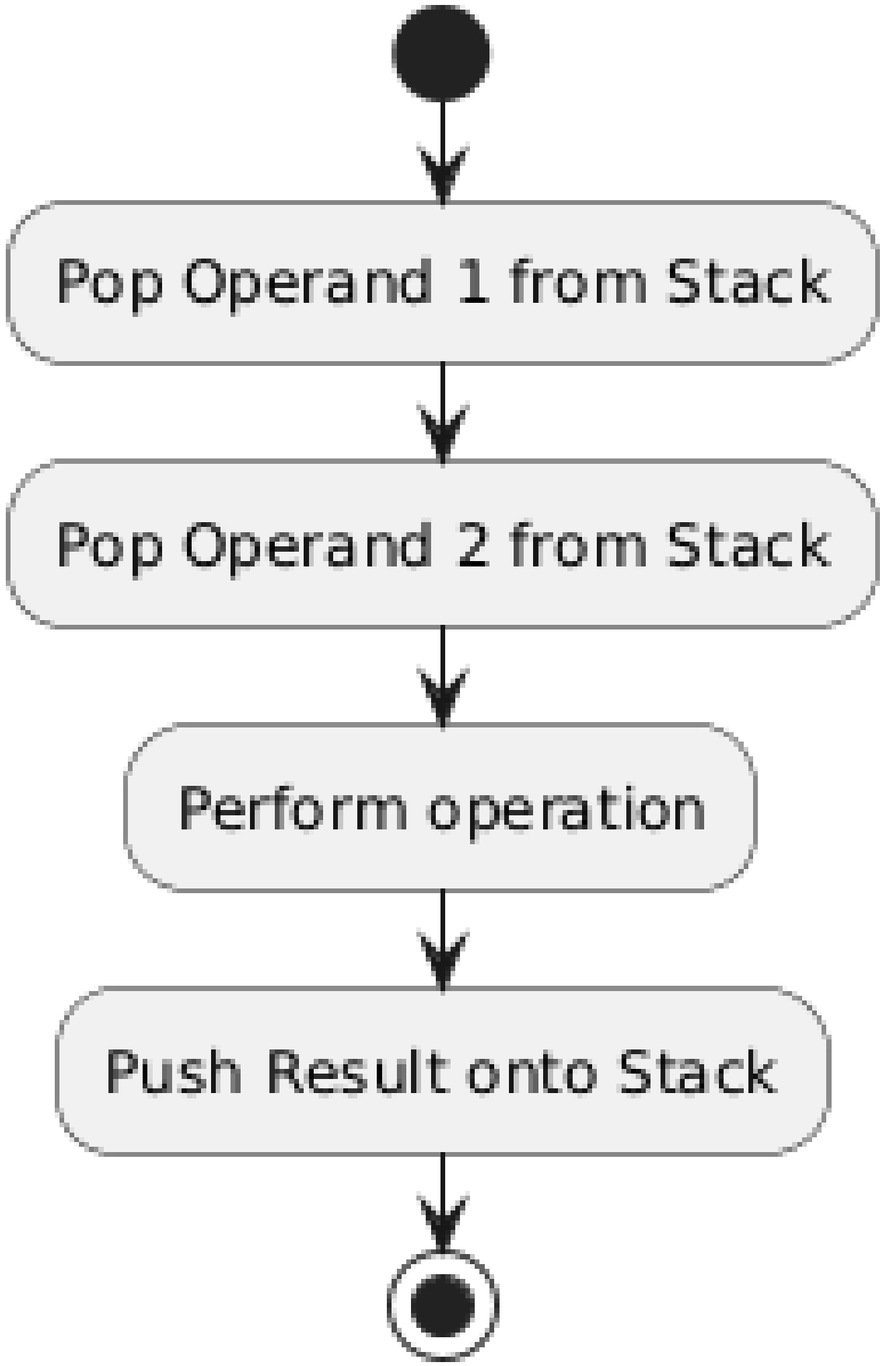
Runtime Data Area:

Each method call creates a new stack frame, which includes local variables, operand stack, constant pool reference, and a program counter.

Bytecode Execution:

Executes bytecode instructions for arithmetic, control flow, object manipulation, and exception handling in a stack-based environment.

Binary Arithmetic Opertaion

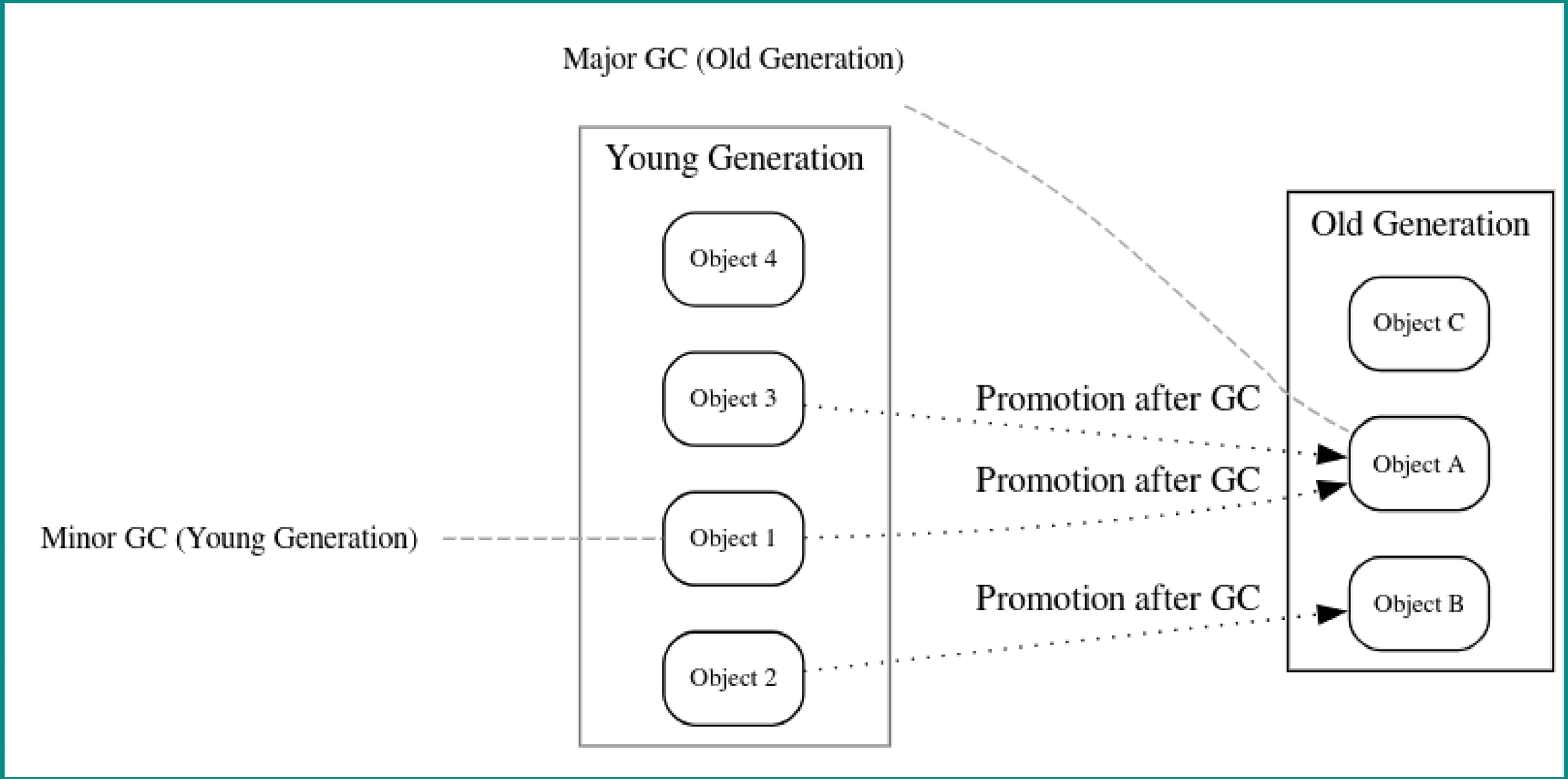


Frame Structure

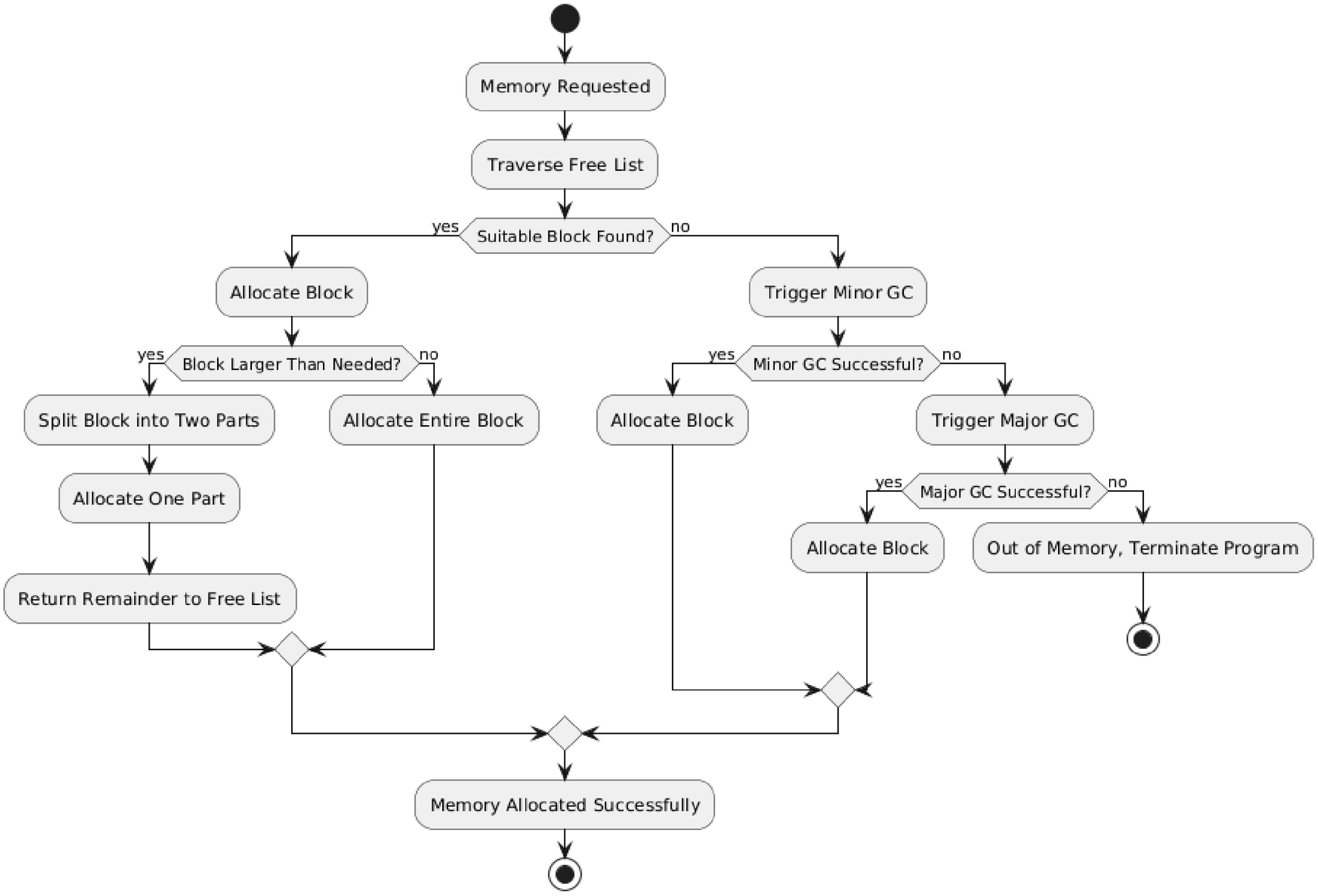
Frame
Constant Pool: Reference
Method Name Descriptor: NameDes
Code: Reference Code block
PC: 32-bit
Locals: Value[]
Operands: Stack of values

MEMORY MANAGEMENT

- The heap is divided into generations (Young and Old) to optimize garbage collection efficiency.
- Uses a double-linked list to track unallocated memory blocks, facilitating efficient allocation and deallocation.
- Allocates memory by searching the free list for suitable blocks, which are then marked as allocated or split if necessary.



Memory Overview



Memory Allocation Process

GARBAGE COLLECTION

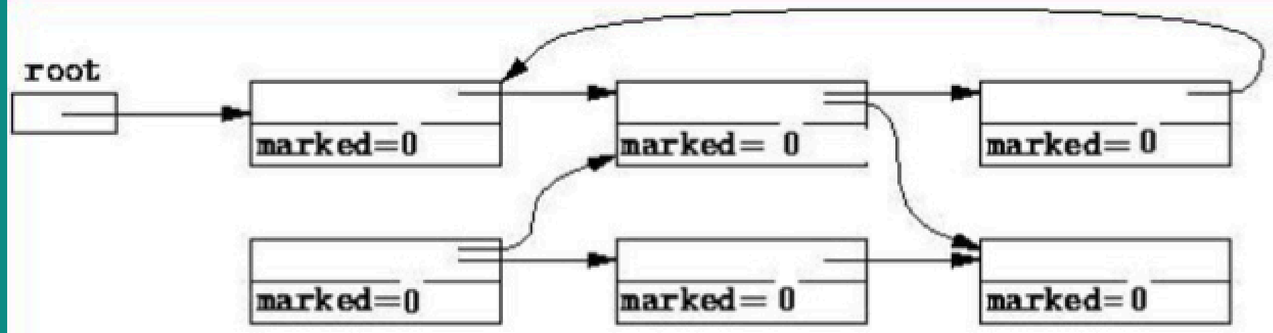
Mark Phase

Identifies reachable objects starting from root references and marks them as live to prevent accidental freeing.

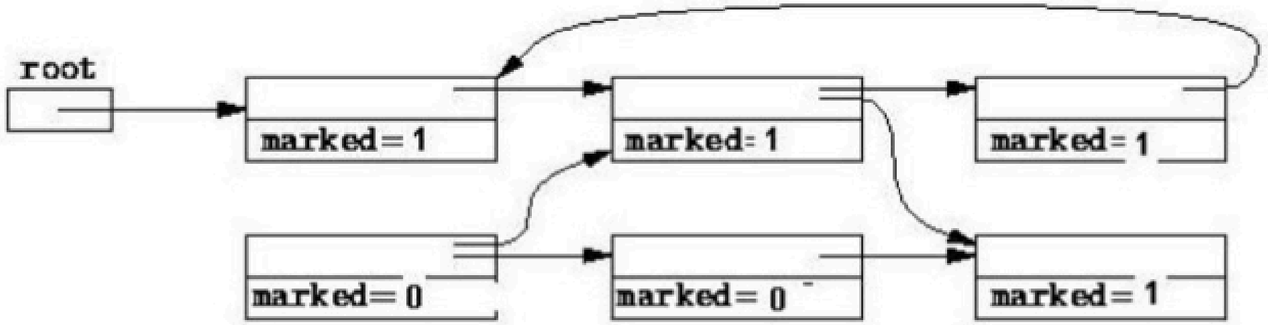
Sweep Phase

Reclaims memory from unreachable objects not marked in the mark phase, returning it to the free list.

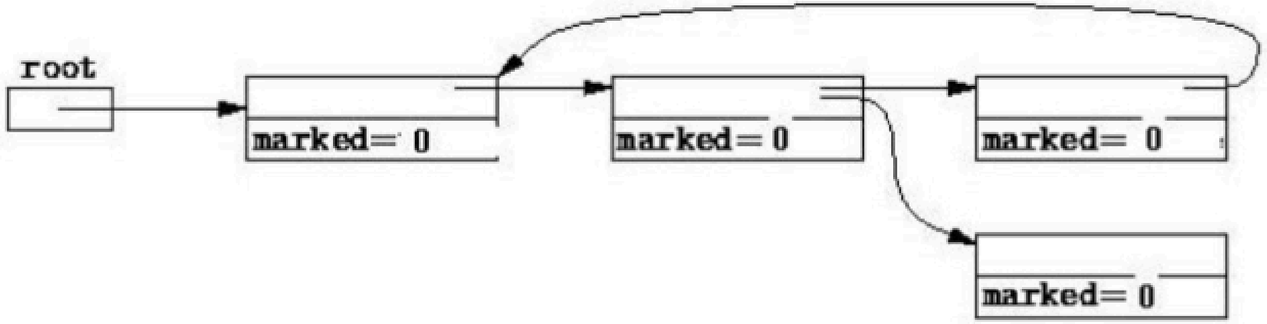
Garbage collection Mechanism



Stage :- Before Garbage Collection

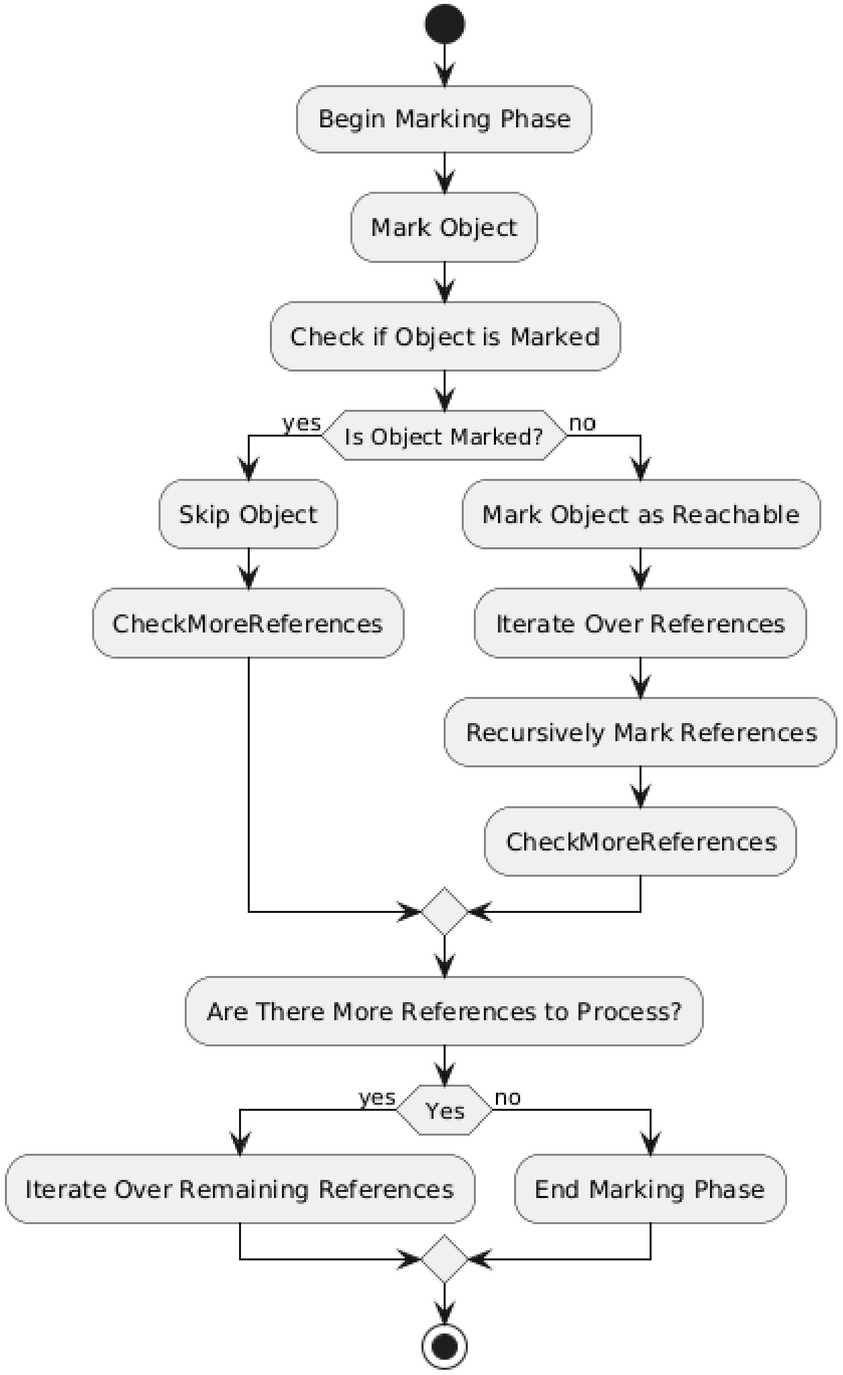


Stage: After Mark Phase

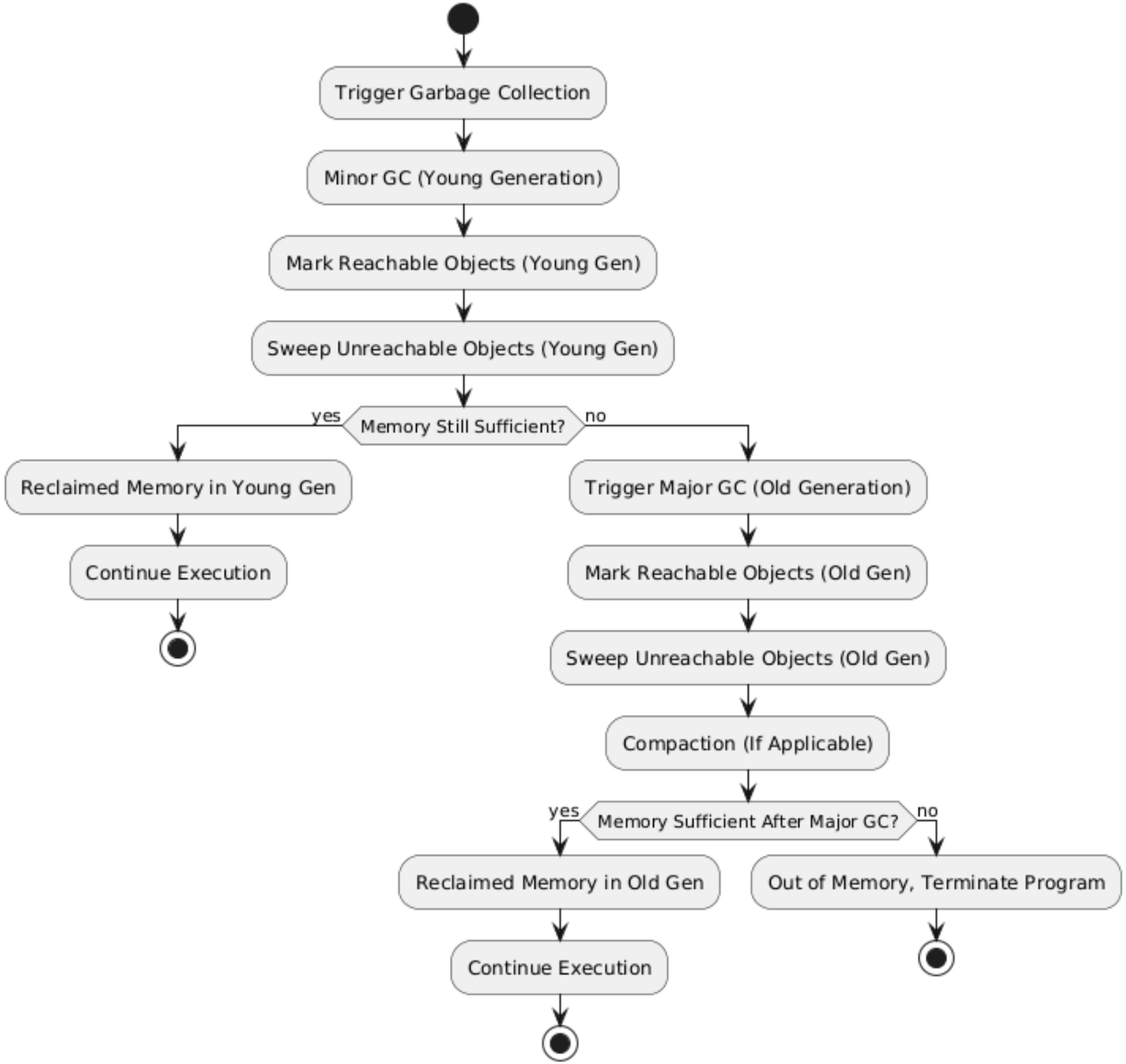


Stage: After Sweep Phase

Marking


Begin Marking Phase
Mark Object
Check if Object is Marked
Is Object Marked?
yes
no
Skip Object
Mark Object as Reachable
CheckMoreReferences
Iterate Over References
Recursively Mark References
CheckMoreReferences
Are There More References to Process?
Yes
yes
no
Iterate Over Remaining References
End Marking Phase

Sweeping


Trigger Garbage Collection
Minor GC (Young Generation)
Mark Reachable Objects (Young Gen)
Sweep Unreachable Objects (Young Gen)
Memory Still Sufficient?
yes
no
Reclaimed Memory in Young Gen
Continue Execution
Trigger Major GC (Old Generation)
Mark Reachable Objects (Old Gen)
Sweep Unreachable Objects (Old Gen)
Compaction (If Applicable)
Memory Sufficient After Major GC?
yes
no
Reclaimed Memory in Old Gen
Continue Execution
Out of Memory, Terminate Program

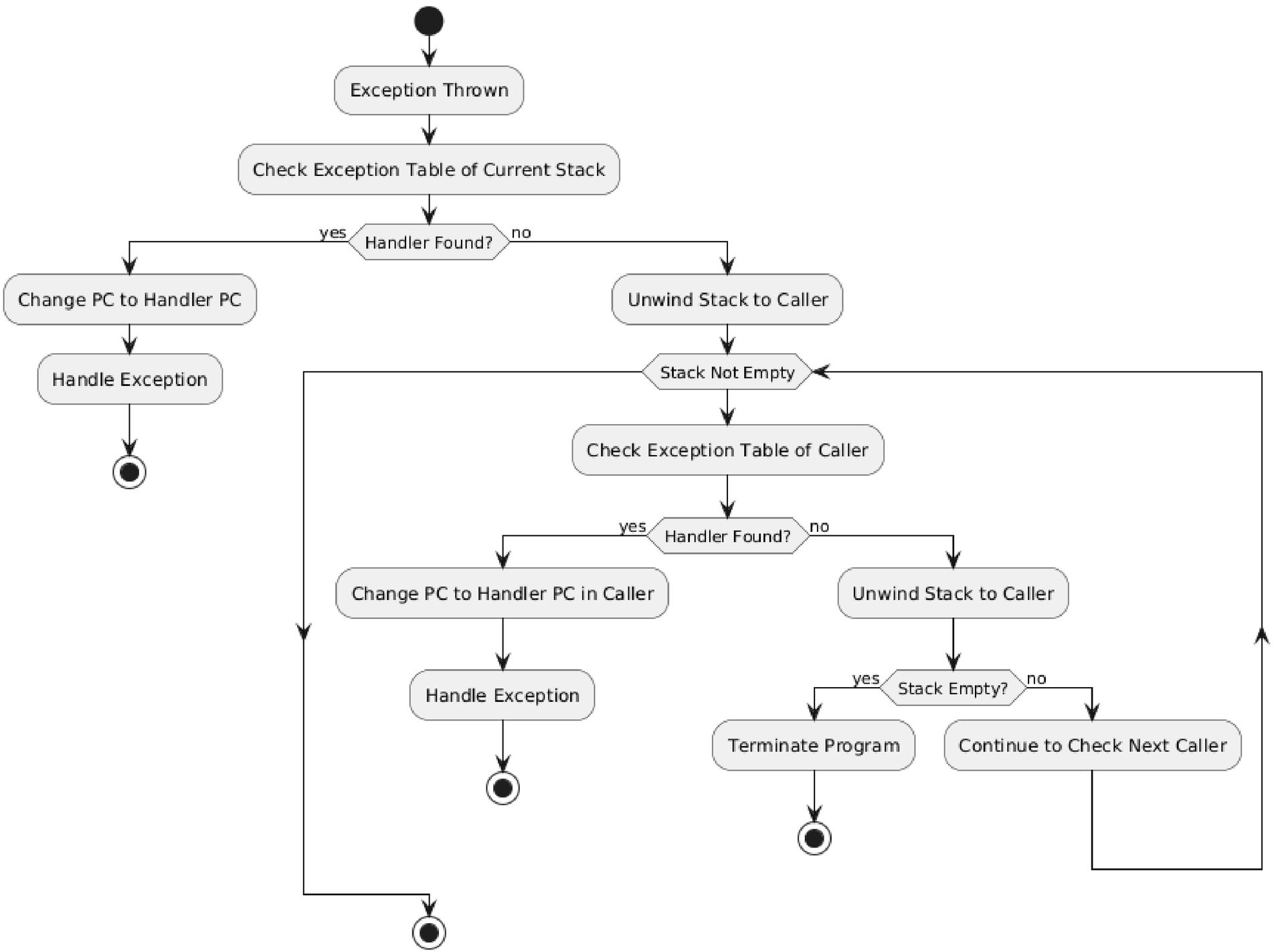
EXCEPTION HANDLING

- The JVM uses an exception table to map exceptions to catch blocks, directing the flow to the appropriate handler when an exception occurs.

- When an exception is thrown, the JVM unwinds the stack, popping method frames until it finds a matching catch block or the program terminates.

- During stack unwinding, the JVM cleans up method frames, destroying local variables and references to prevent memory leaks.

Exception Handling



PROJECT DEMONSTRATION

LIMITATIONS

- No JIT compilation, leading to slower execution.
- No multithreading support.
- Limited I/O handling (only basic terminal output).
- Invokedynamic instruction not implemented.
- No full compatibility with Java standard libraries.

FUTURE ENHANCEMENTS

- Just-In-Time Compilation
- Command Line Options for Runtime Configuration
- Bytecode Verification
- GUI for Monitoring

THANK YOU!