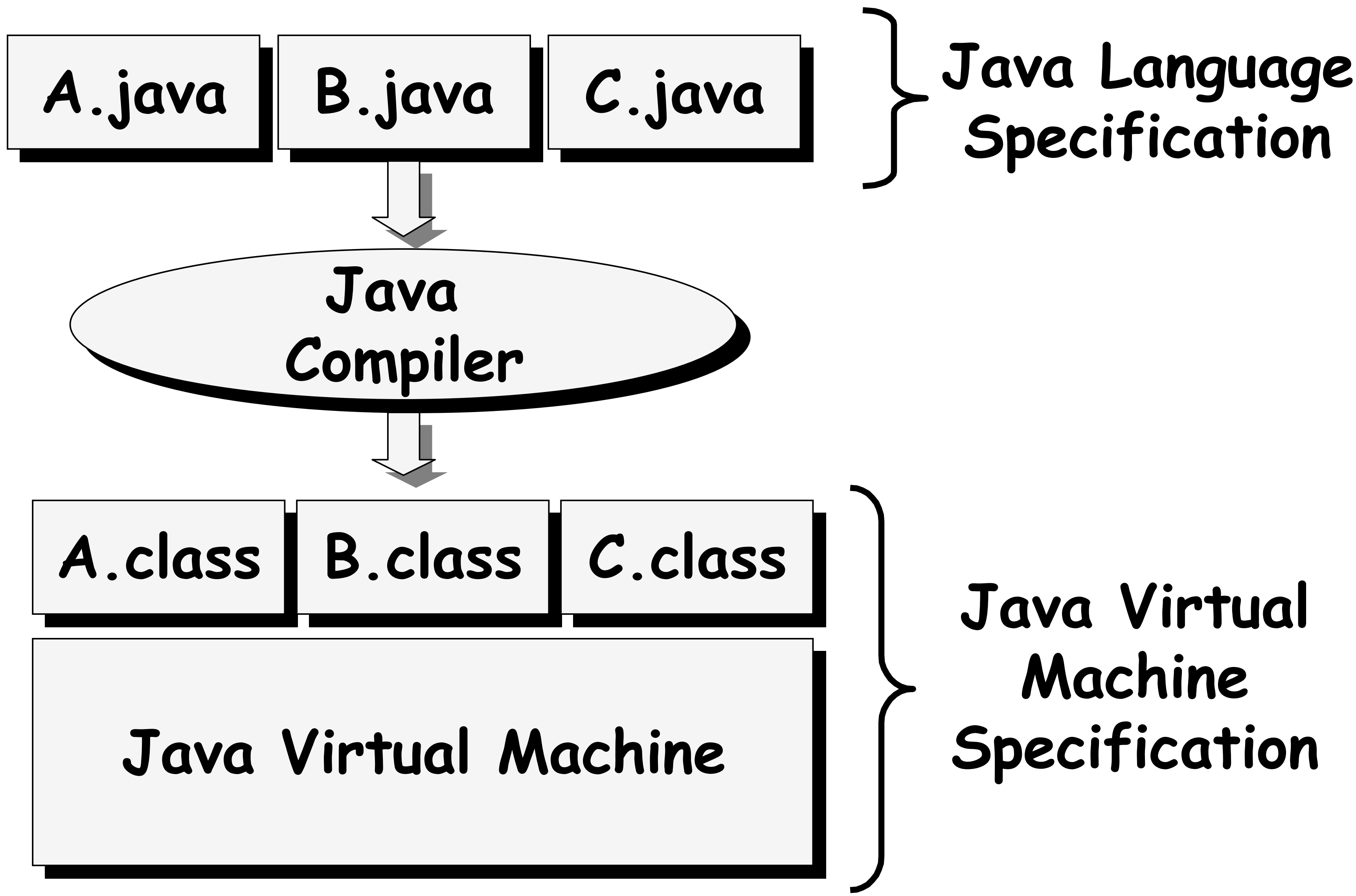
Introduction to Java Virtual Machine

Outline

- Java Language, Java Virtual Machine and Java Platform
- Organization of Java Virtual Machine
- Garbage Collection
- Interpreter and Just-In-Time Compiler

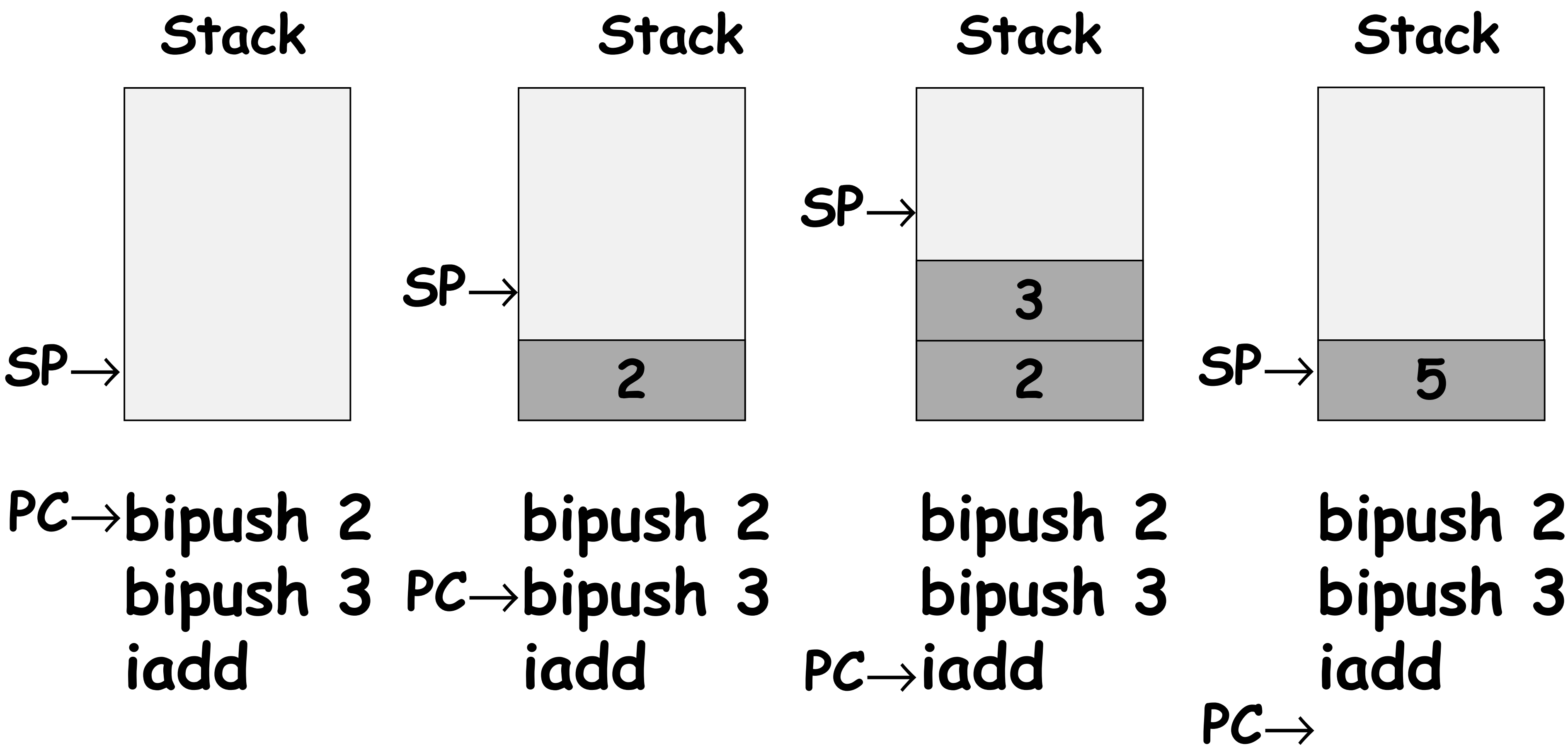
The Big Picture



What Is in the JVM Spec?

- Bytecodes – the instruction set for Java Virtual Machine
- Class File Format – The platform independent representation of Java binary code
- Verification Rules – the algorithm for identifying programs that cannot compromise the integrity of the JVM

Bytecode example



Class File Example (1)

HelloWorld.java

```
public  class HelloWorld extends Object {
    private String s;
    public HelloWorld() {
        s = ‘Hello World!’;
    }
    public void sayHello() {
        System.out.println(s);
    }
    public static void main(String[] args) {
        HelloWorld hello = new HelloWorld();
        hello.sayHello( );
    }
}
```

Class File Example (2)

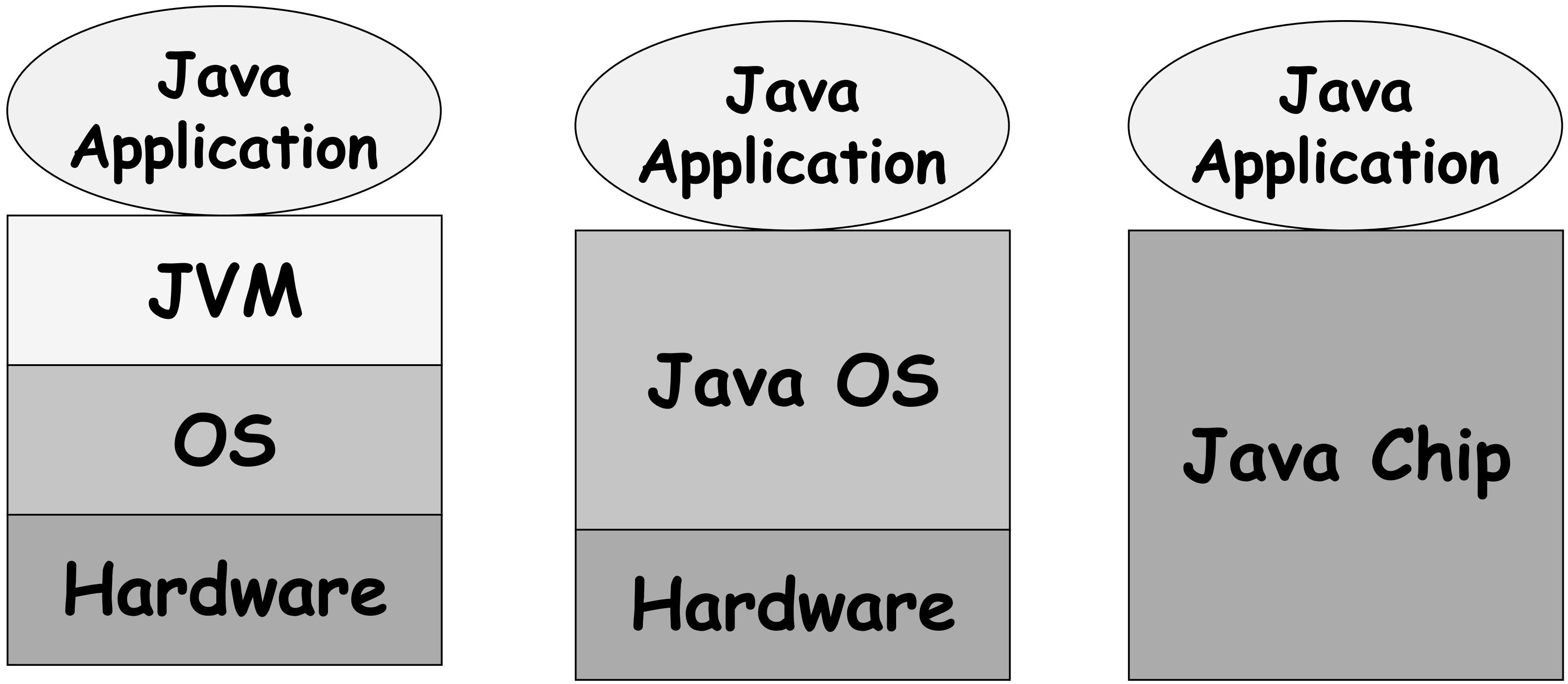
HelloWorld.class

```
class HelloWorld
Superclass java/lang/Object
Constant Pool
  #0: 'Hello World'
Fields
  s descriptor : Ljava/lang/String;
    modifiers  : private
Methods
  <init>     descriptor : ()V
             modifiers  : public
  sayHello   descriptor : ()V
             modifiers  : public
  main       descriptor : (Ljava/lan/String[;)V
              modifiers : public, static
Bytecodes
  Bytecodes for <init> (the constrtuctor)
  Bytecodes for sayHello
  Bytecodes for main
```

Verification

- Is it a structurally valid class file?
- Are all constant references correct?
- Will each instruction always find a correct formed stack and local variable array?
- Check out external references
- Other safety requirements

Implementations of JVM

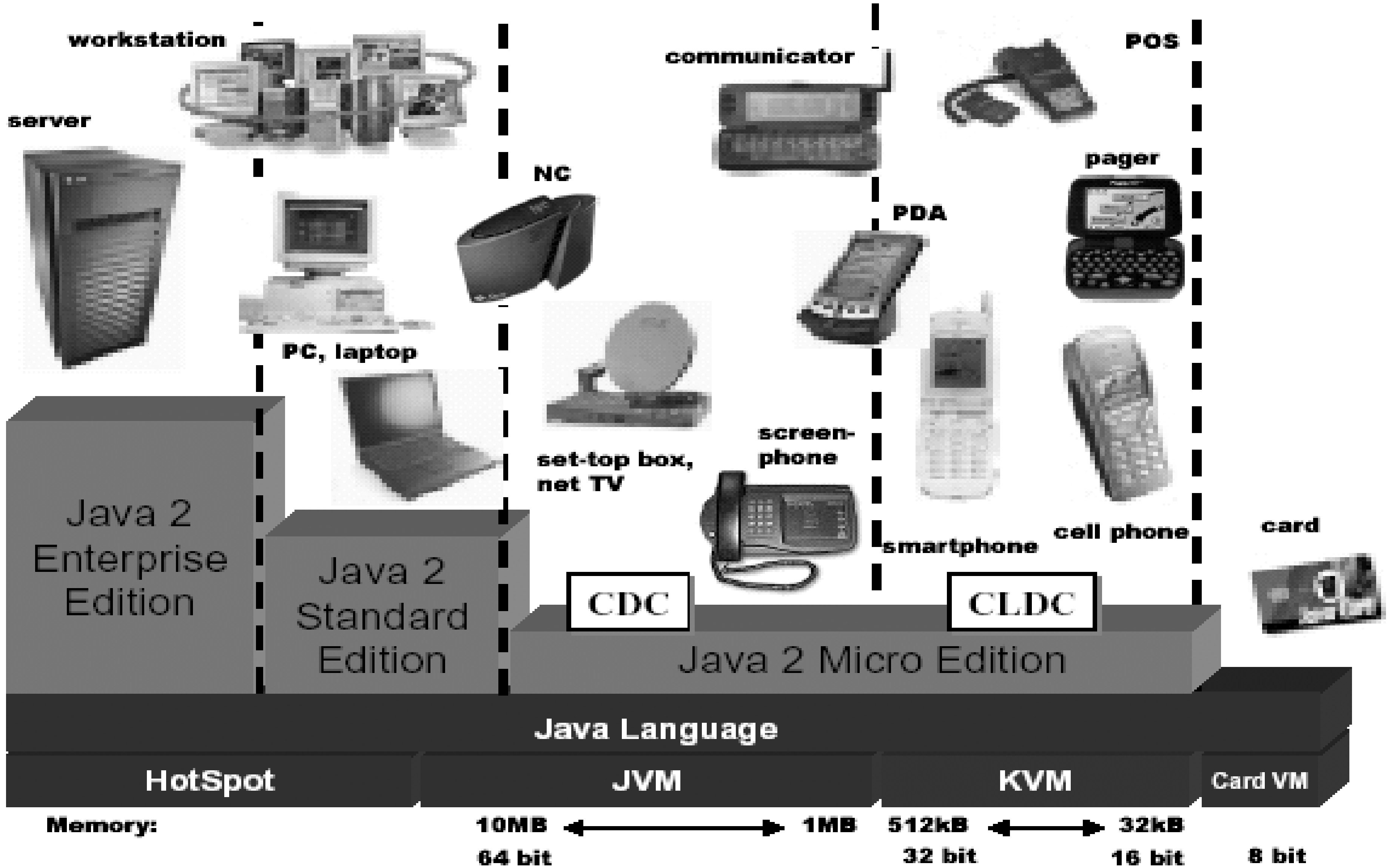


JVM Specification does not specify how a JVM is implemented

Java Platforms (1)

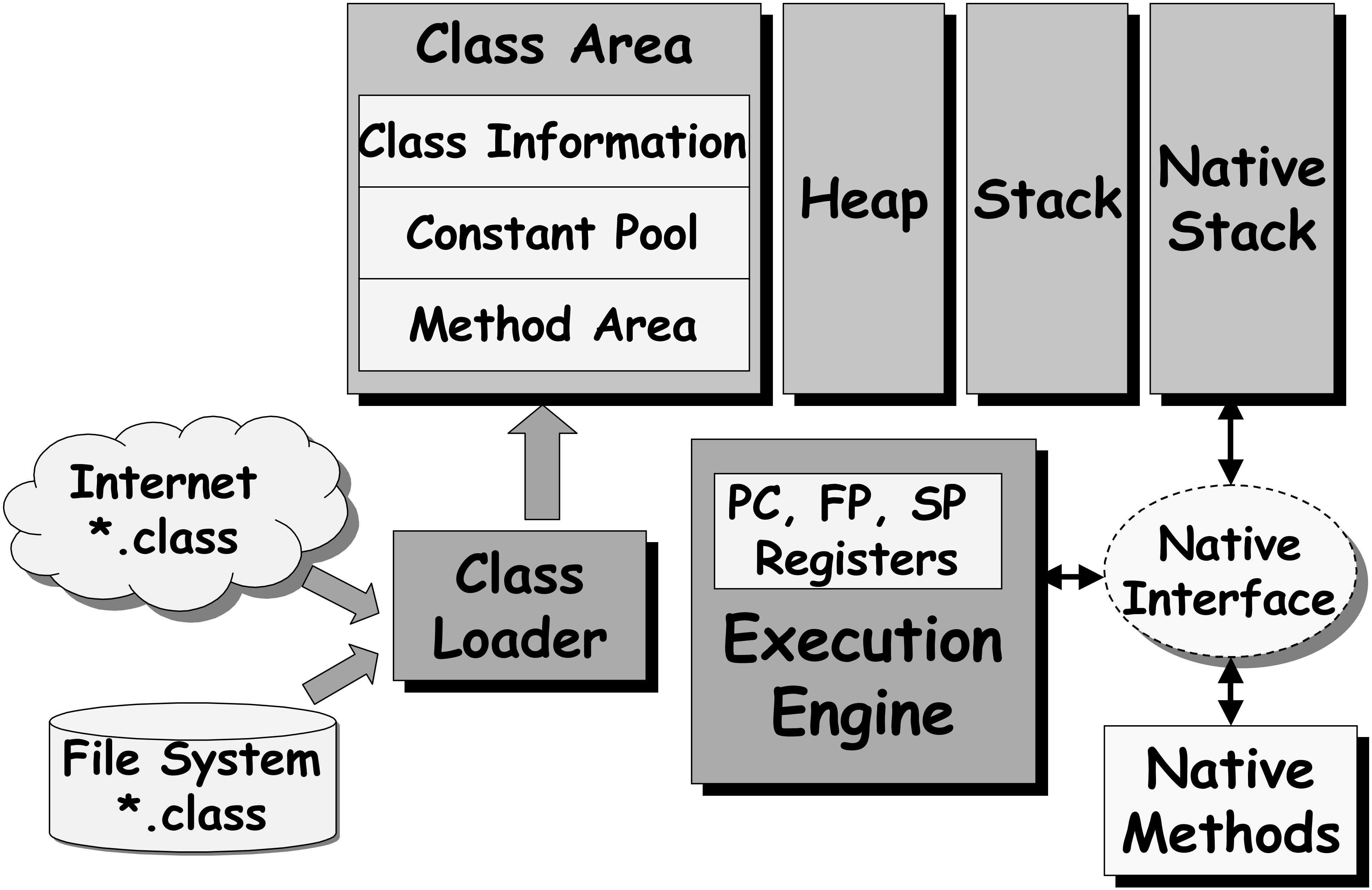
- A Java Platform consists of Java Virtual Machine and a set of standard classes
- JVM in all platforms must satisfy JVM Specification
- Standard classes can be tailored according the the resource constraints
- Three levels of Java platforms: J2EE, J2SE and J2ME

Java Platforms (2)



From KVM White Paper (Sun Microsystem)

Organization of JVM



Execution Engine

- Executes Java bytecodes either using interpreter or Just-In-Time compiler
- Registers:
  - PC: Program Counter
  - FP: Frame Pointer
  - SP: Operand Stack Top Pointer

Class Loader

1. Loading: finding and importing the binary data for a class
2. Linking:
   - Verification: ensuring the correctness of the imported type
   - Preparation: allocating memory for class variables and initializing the memory to default values
   - Resolution: transforming symbolic references from the type into direct references.
3. Initialization: invoking Java code that initializes class variables to their proper starting values

Class Area

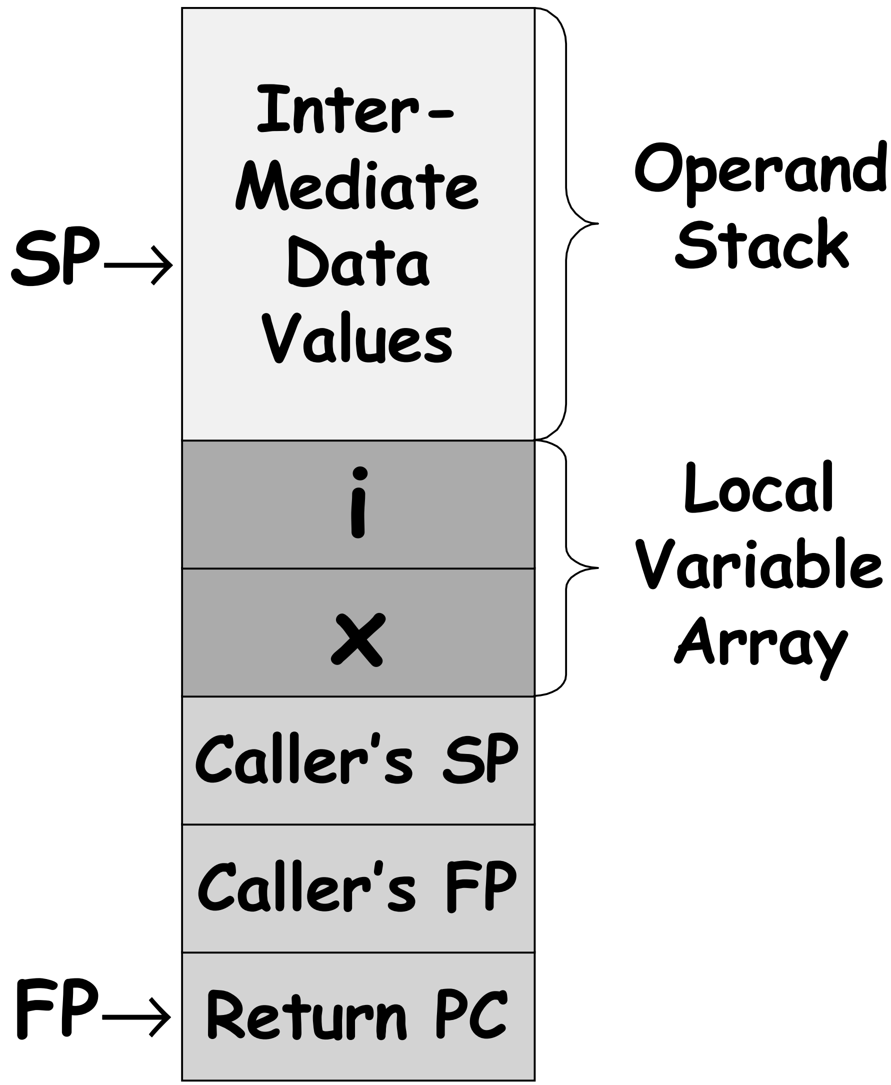
- Class Information
  - Internal representation of Java classes
  - Information about the superclass and implemented interfaces
  - Information about the fields and methods
- Constant Pool
- Method Area
  - Contains the bytecodes of the methods of the loaded classes

Stack

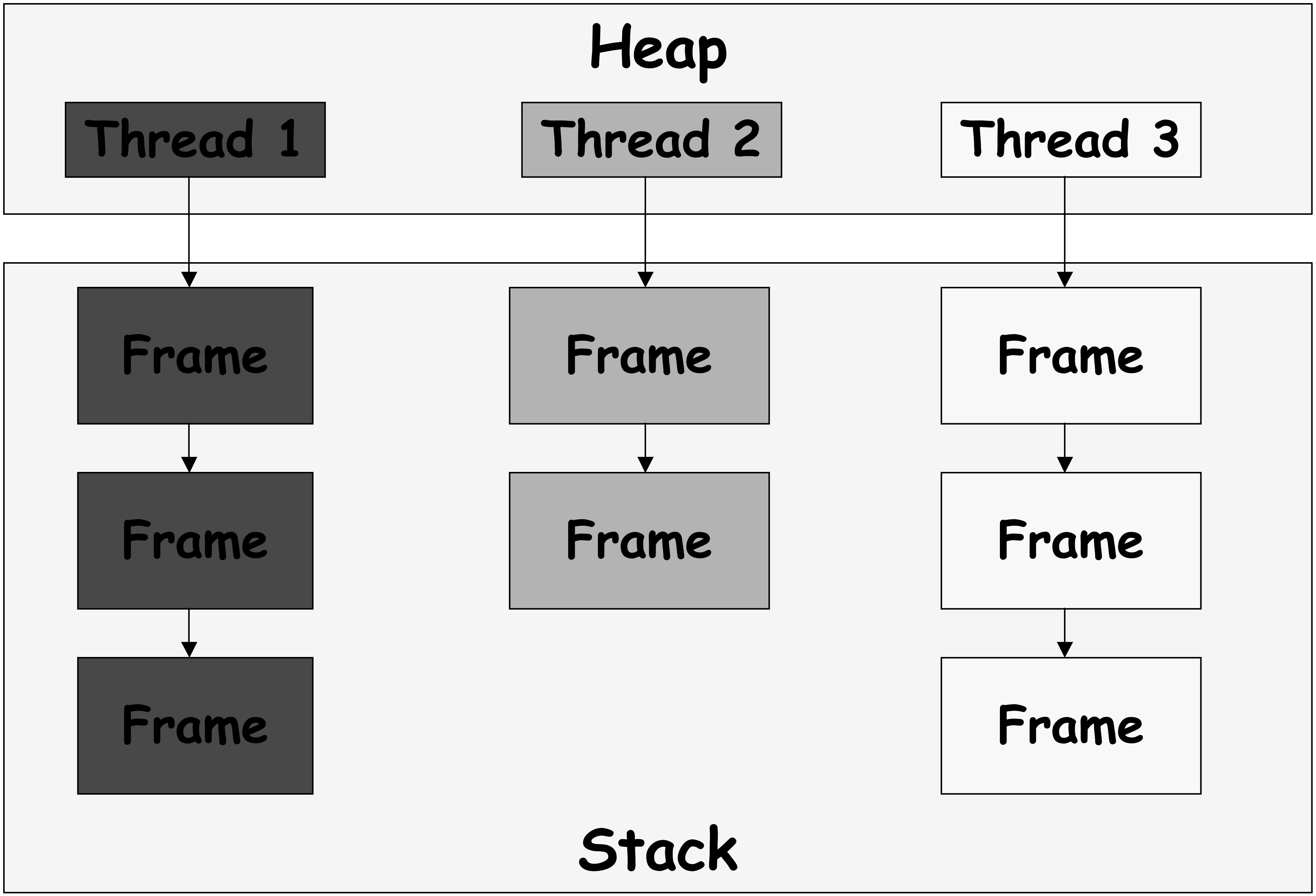
- The Java stack is composed of frames
  - A frame contains the state of one Java method invocation
  - Logically, a frame has two parts: local variable array and operand stack
- JVM has no registers; it uses the operand stack for storage of intermediate data values
  - to keep the JVM's instruction set compact
  - to facilitate implementation on architectures with limited number of general purpose registers
- Each Java thread has its own stack and cannot access the stack of other threads

Stack Frame

```
public class A
{
   ... ...
   void f(int x)
   {
     int i;
     for(i=0; i<x; i++)
     {
       ... ...
     }
   ... ...
}
```



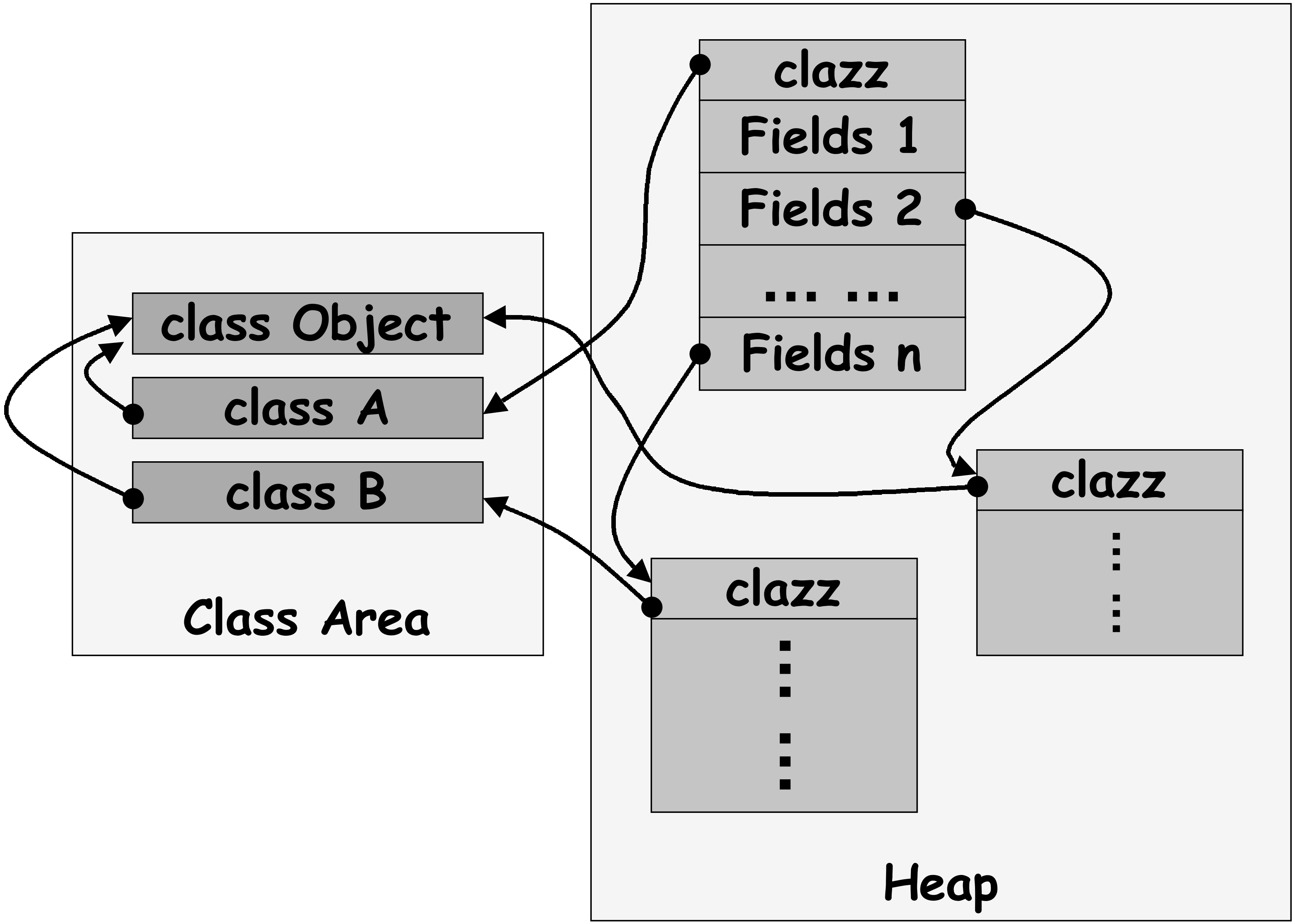
Stack - Each Thread Has its own Stack



Heap

- All Java objects are allocated in the heap
- Java applications cannot explicitly free an object
- The Garbage Collector is invoked from time to time automatically to reclaim the objects that are no longer needed by the application
- The heap is shared by all Java threads

Java Objects in the Heap

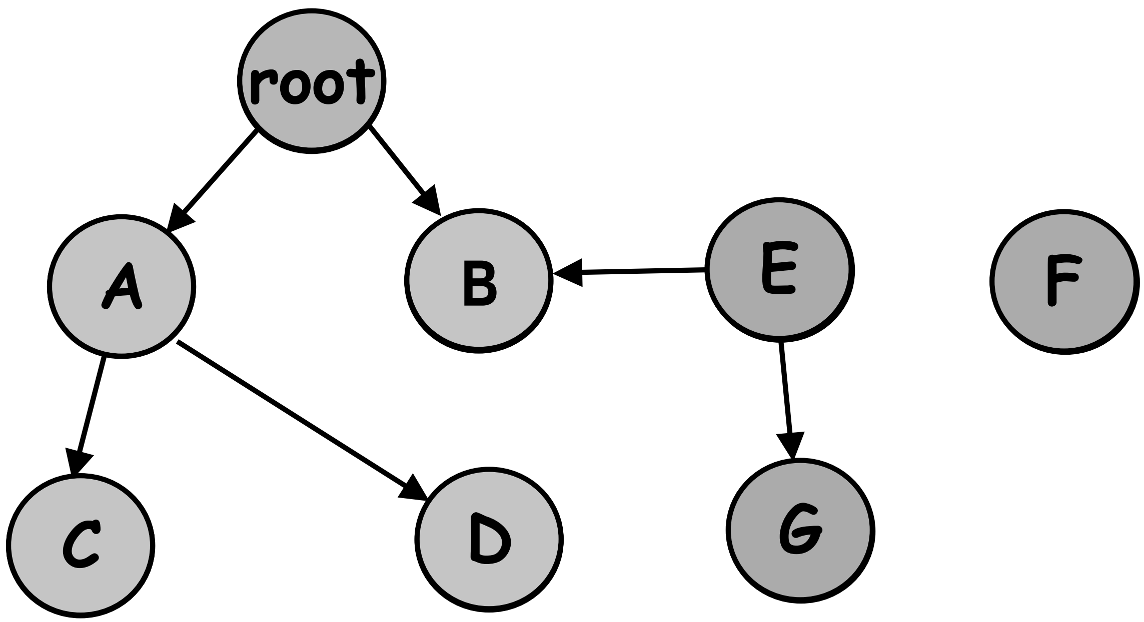


Garbage Collector

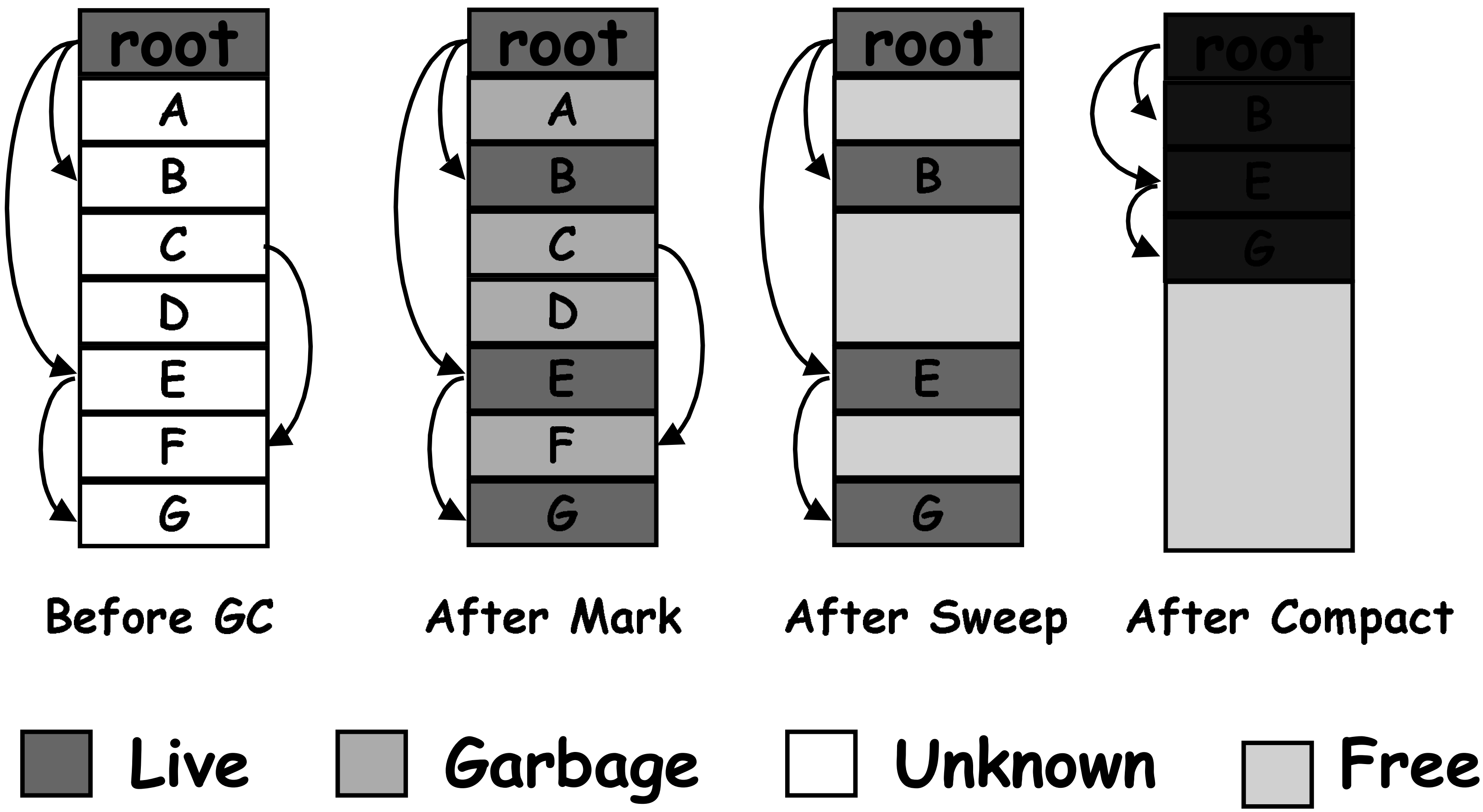
Roots: internally defined by the JVM implementation

Live Objects: reachable from the roots

Garbage (Dead Objects): not reachable from the roots, not accessible to the application

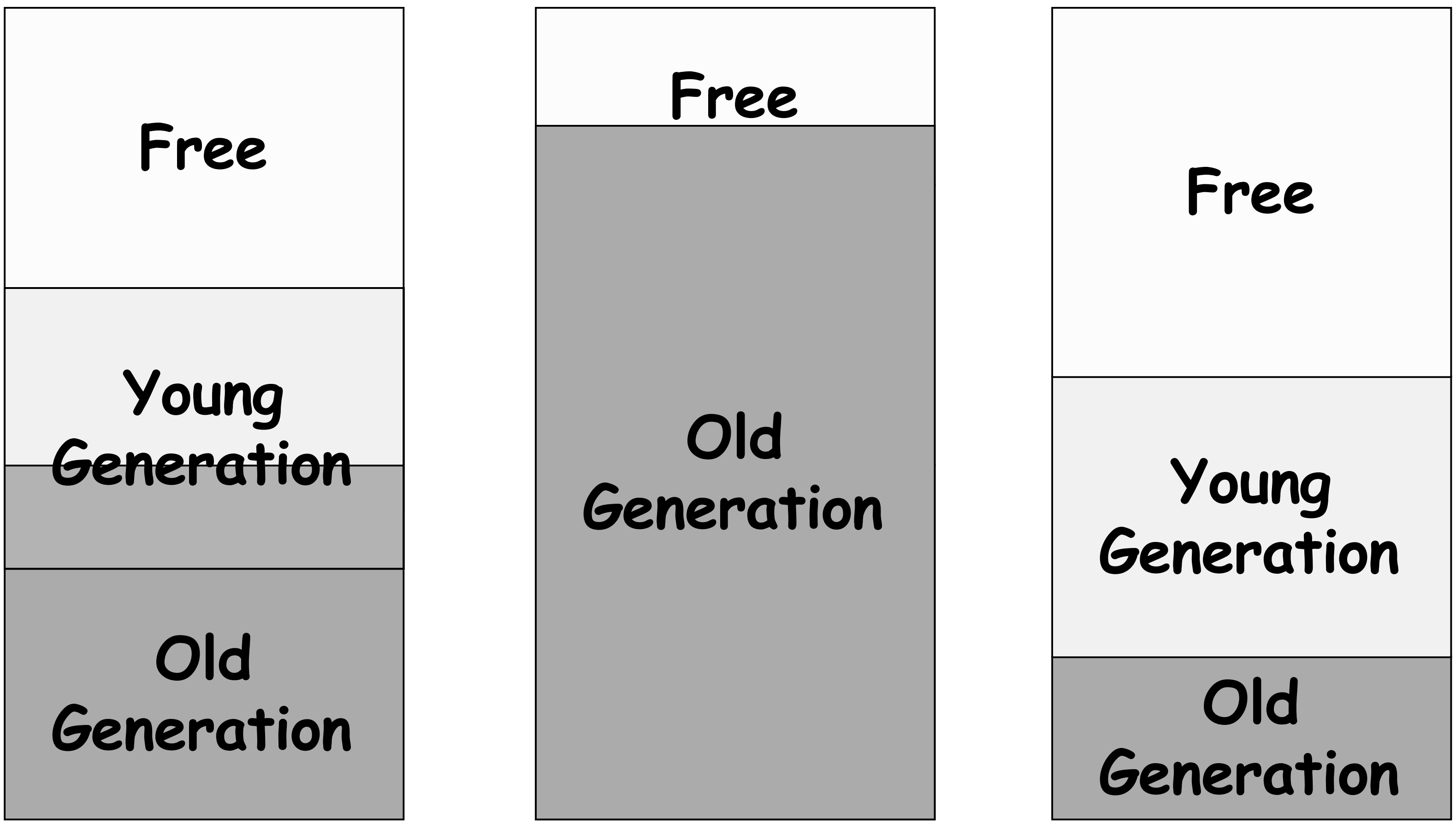


Mark / Sweep / Compaction

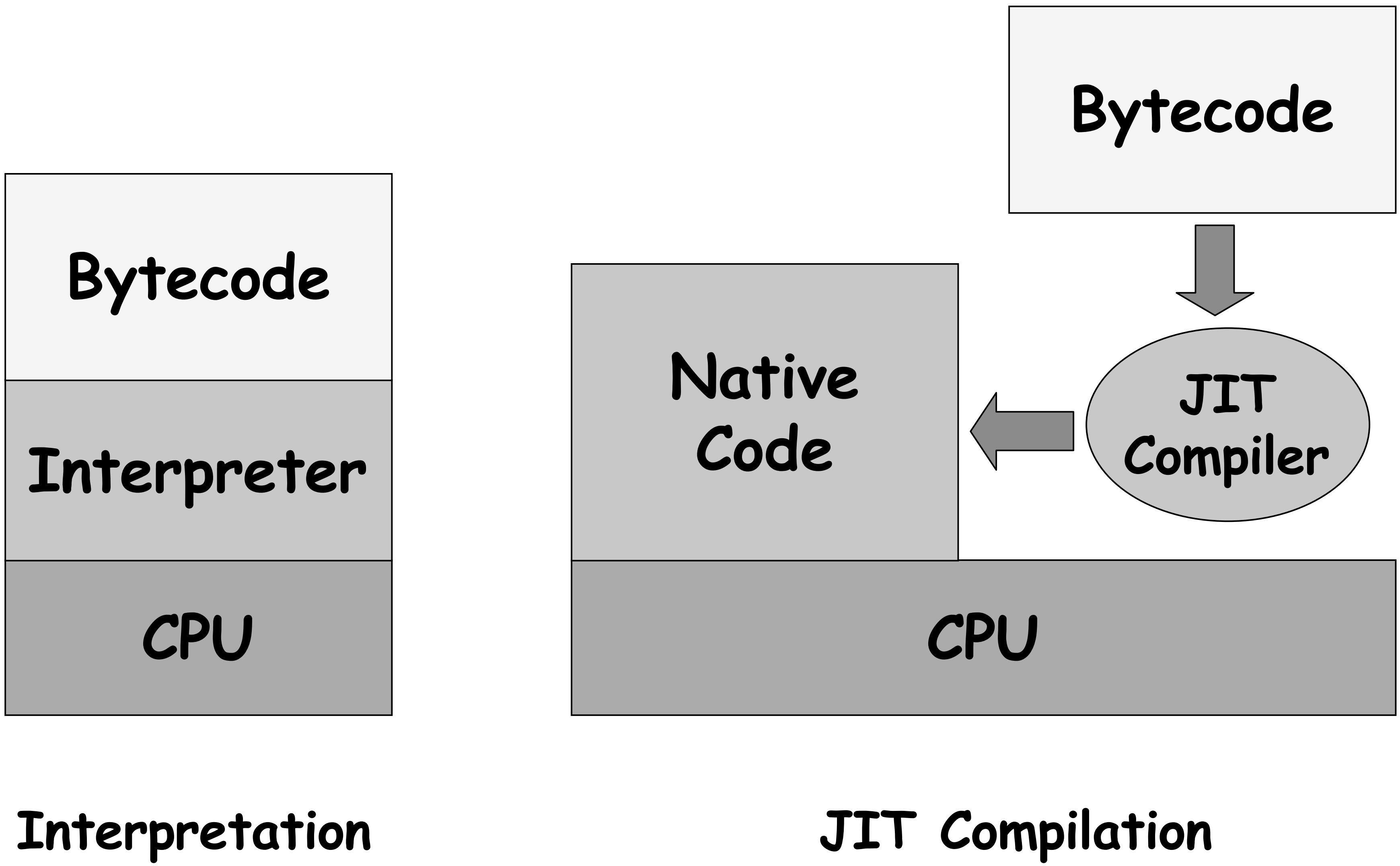


Generational Garbage Collection

- Most objects live for very shot time
- A small percentage of them live much longer



Interpreter vs Just-In-Time Compiler



Bytecode Interpreter (1)

```
while(program not end ) {
  fetch next bytecode => b
  switch(b) {
    case ILOAD:
      load an integer from the local
      variable array and push on top
      of current operand stack;
    case ISTORE:
      pop an integer from the top of
      current operand stack and store
      it into the local variable array;
    case ALOAD:
      ….. ...
  } // end of switch
} // end of while
```

Bytecode interpreter (2)

- Advantage
  - Ease to implement
  - Does not need extra memory to store compiled code
- Disadvantage
  - Very Slow: 10 ~ 100 times slower than execution of native code

Just-In-Time Compiler

- Dynamically compiles bytecode into native code at runtime, usually in method granularity
- Execution of native code is much faster than interpretation of bytecode
- Compilation is time consuming and may slow down the application
- Tradeoffs between execution time and compilation time

Adaptive Compiler

- Observation: less than 20% of the methods account for more than 80% of execution time
  - Methods contains loop with large number of iteration;
  - Methods that are frequently invoked
- Idea 1: only compile the methods where the application spends a lot of time
- Idea 2: perform advanced compiler optimization for the hottest methods, simple or no compiler optimization for less hot methods

How Adaptive Compiler Works

- Set three thresholds T1, T2 (T1<T2)
- Each method has a counter that is initialized to 0. Whenever the method is invoked, increase its counter by 1
- The methods with counter lower than T1 are executed using interpreter
- When a method's counter reaches T1, compile this method with simple optimizations
- When a method's counter reaches T2, recompile this method with advanced optimizations