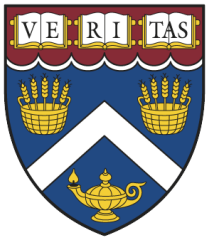
Virtual Machine Overview



Virtualization

- Indirection between application and platform
- We can virtualize at various levels
  - Single machine: virtual box, VMware Fusion
  - Infrastructure suite: AWS, vCloud
- We're interested in the application-level
  - The interface between the application and OS

Good old days

- Write code in C
- Run an optimizing compiler
  - May use cross-compilers to target other architectures
  - Complex make files and build infrastructure
- Link with static libraries
- Distribute the executable
- Link with runtime libraries

Why Virtualize?

- Separate the application from the platform
  - Target a single idealized architecture
  - Implement the architecture once per platform
- Bake in extra language features
  - Memory management
  - Type safety
  - Null and bounds checks
- Sandboxing
  - Don't give the application full access to the platform

Computer Architectures

- Complex Instruction Set Computer
  - Thousands of instructions
- Reduced Instruction Set Computer
  - Hundreds of instructions
- Idealized Bytecode Machine
  - Very small instruction sets
  - VM optimizes to make it run fast
- Direct source code
  - Complete compiler built into the VM

The Java Virtual Machine

- Bytecode-driven virtual machine
  - Based on a language designed for virtualization
- Extremely well-specified
  - Links to the various specs are on Canvas
- Current state of the art
  - Huge amounts of R&D effort over the years
  - Orders of magnitude improvement since early versions

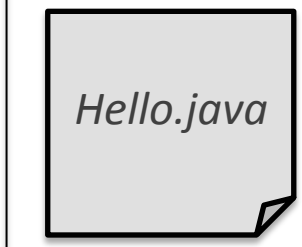
```java
package edu.harvard.cscie98.sample_code;

public class Hello {

  public static void main(final String[] args) {
    System.out.println("Hello World");
  }

}
```

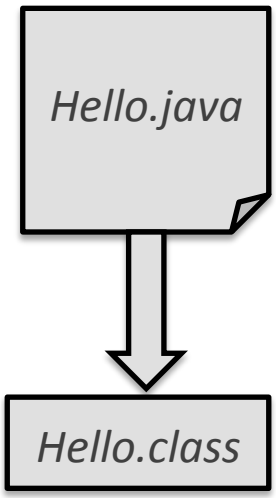
Bytecode Digression

- Compile the Hello.java class
- Inspect the bytecode
  - `javap -v Hello.class`
- See the terminal interaction PDF on Canvas
  - Published in the introduction module

Bytecode Digression

- Examine bytecode
  - Low complexity
    - Compare to i86 asssembly
  - Look at the contents of a single class
    - Constant pool
    - Fields
    - Methods
      - Max stack and locals
      - Bytecode
      - Line numbers
      - Variable table
      - Exception table


Hello.java

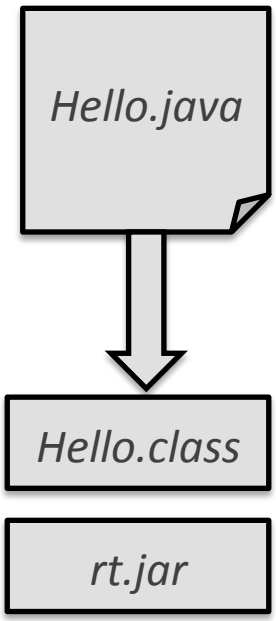

Hello.java
Hello.class

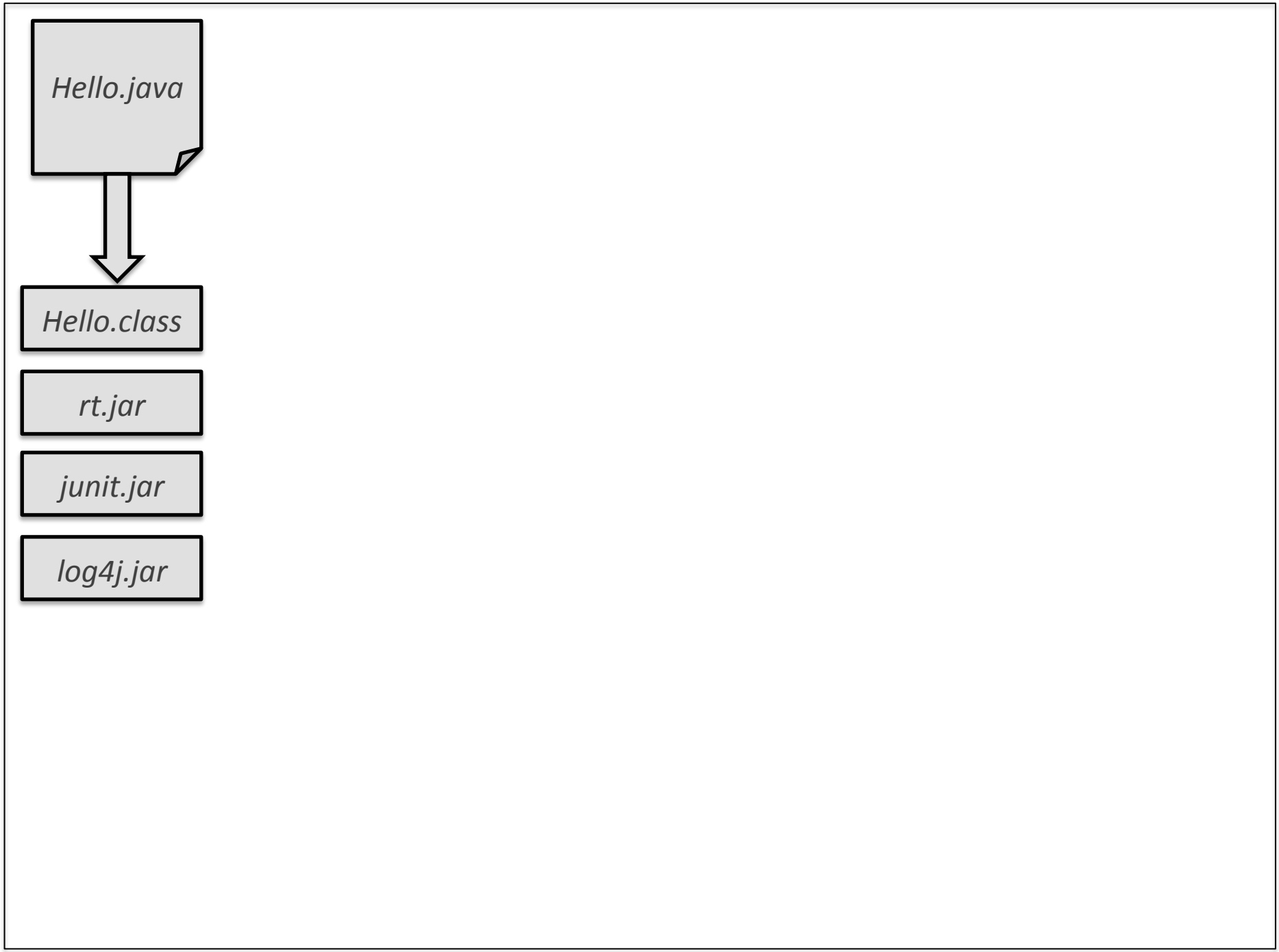
Class Libraries

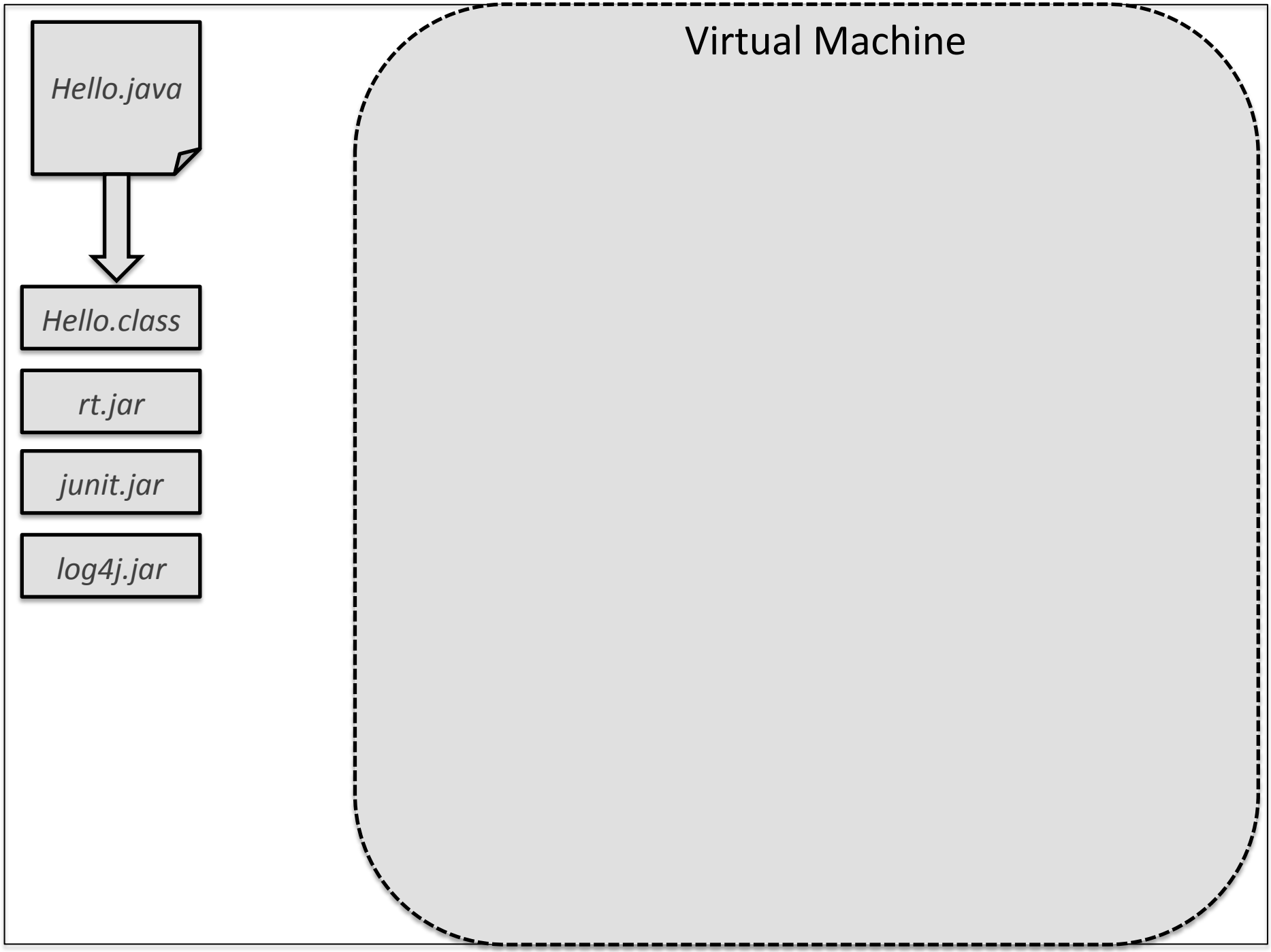
- The second half of any program
- Written in Java
- Trade-offs of language vs. native libraries

Class Libraries

- The VM is only a part of the runtime system
- Need class libraries to do anything useful
  - Even Hello World uses some complex libraries
    - java.lang.Object
    - java.lang.String
    - java.lang.System
    - java.io.PrintStream


Hello.java
Hello.class
rt.jar

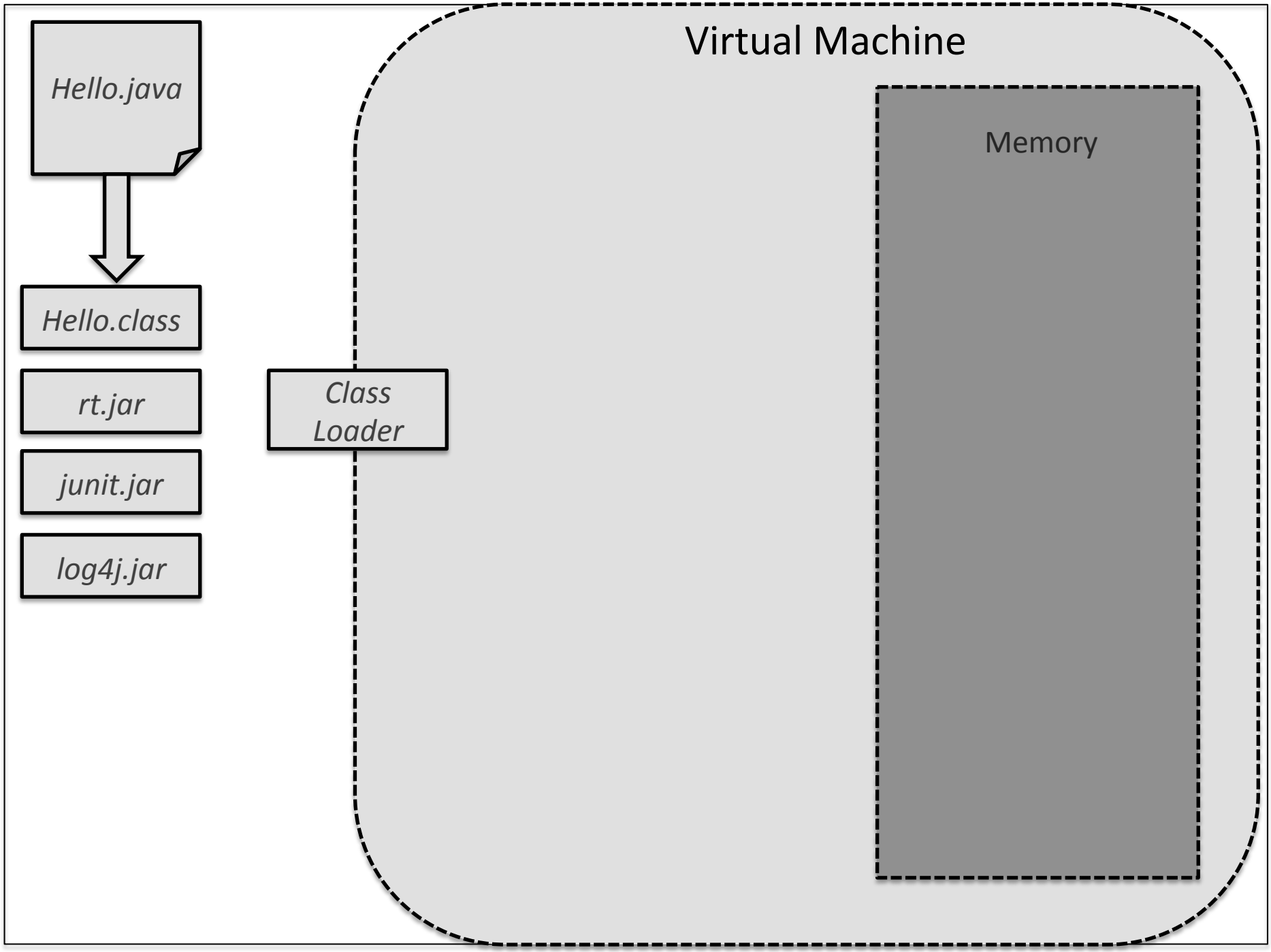

Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine

Launching the VM

- There's a lot of work to do before `main`
  - Map the heap and other memory
  - Load primordial classes
  - Parse command line arguments
  - Initialize VM components
  - Set up threads and internal structures

- VMs have a longer startup time than native code
  - We'll see more examples of this later


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Class Loader
Virtual Machine
Memory

Class Loading

- The process by which code enters the VM
  - Code can come from many sources
    - Read from class files
    - Loaded from JAR files
    - Downloaded from the network
    - Created on the fly
- Classes are loaded lazily
  - Triggered by the first use of a class

Verification

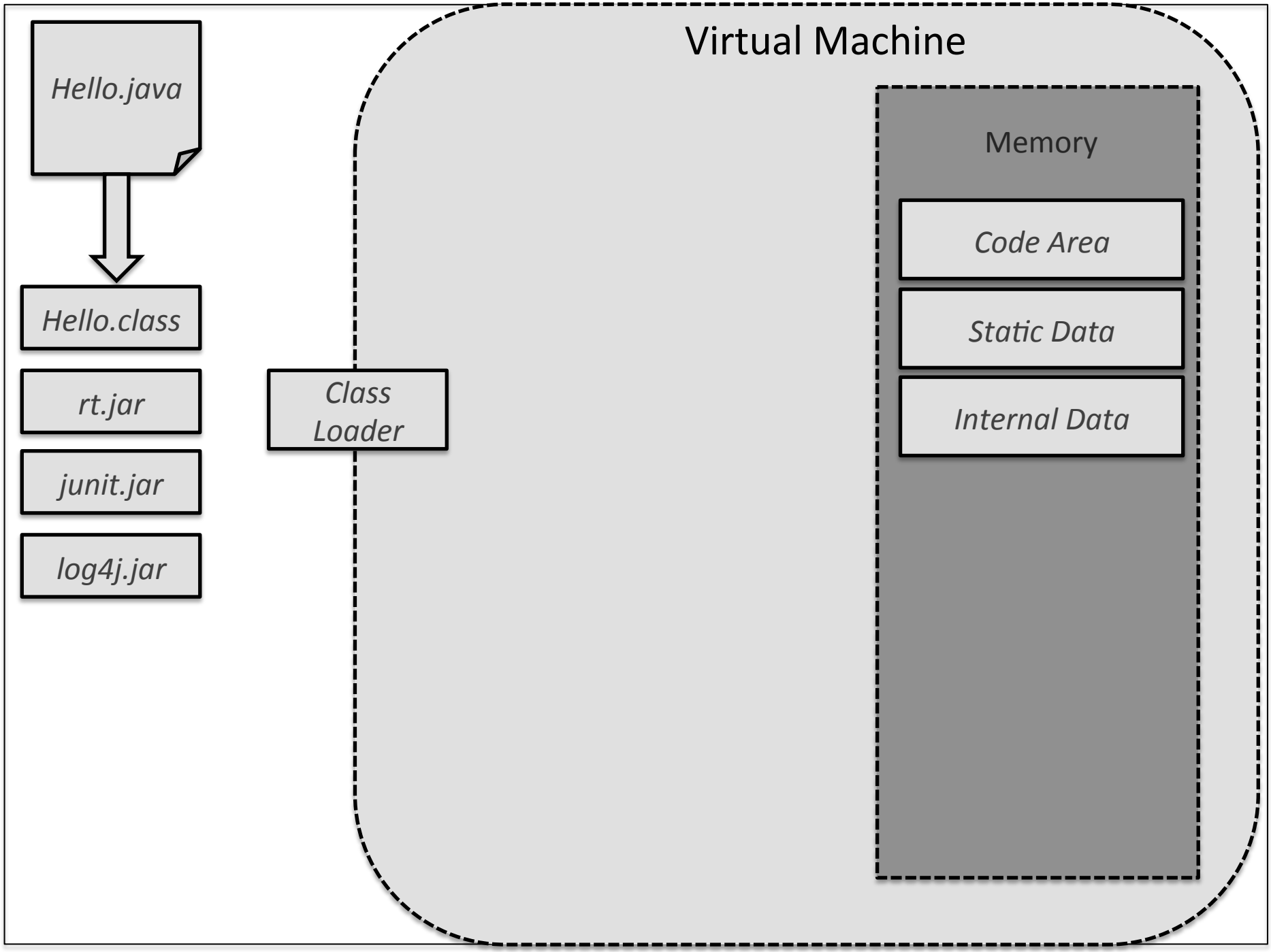
- Bytecode is verified before loading
  - Opcodes and operands must be valid
- Recall the data stored in the bytecode
  - Maximum locals and stack size
  - Exception handler blocks
  - Constant pool types
  - Jump targets

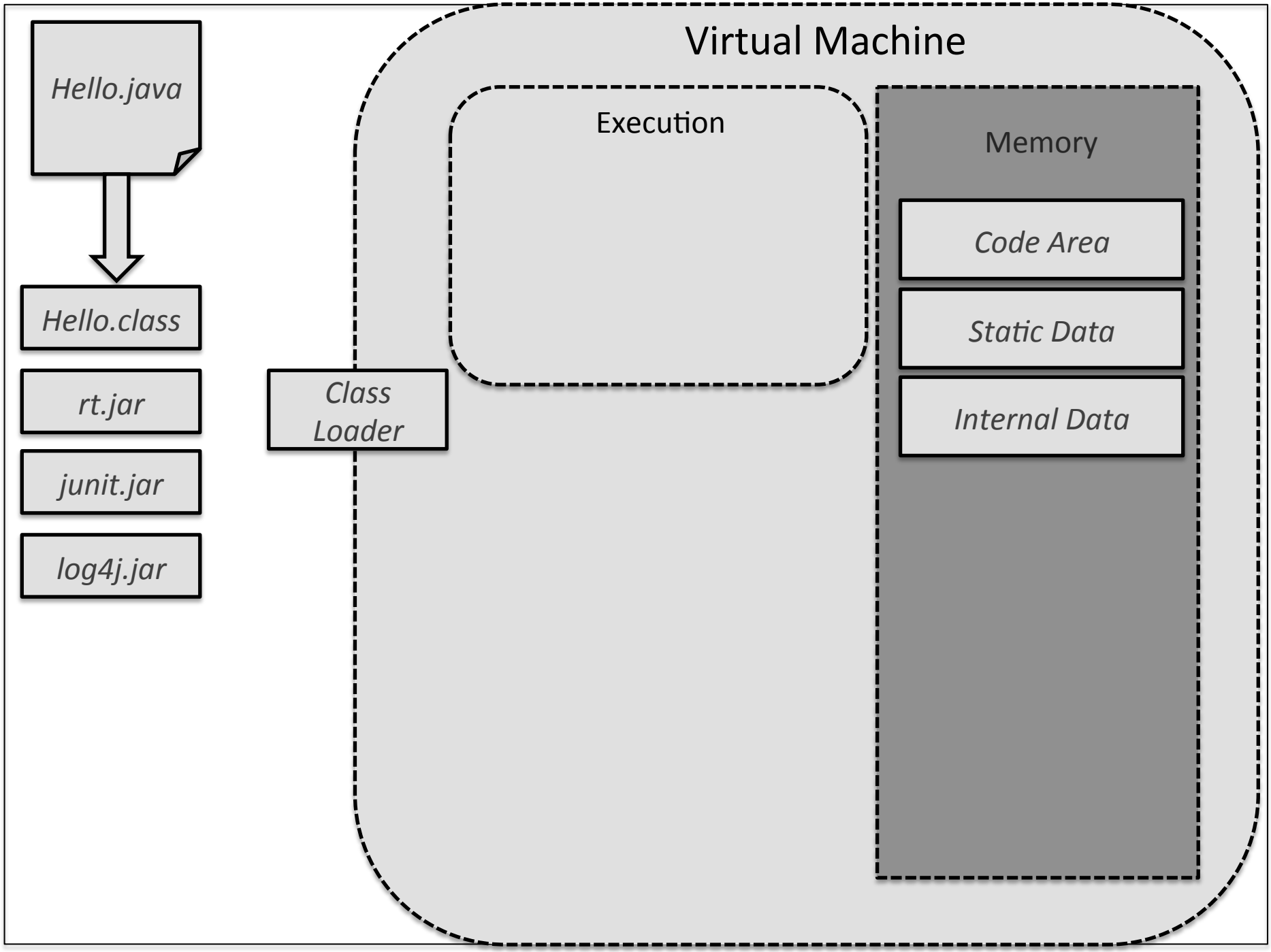
Class Data

- The class file has some internal data structures
  - Constant pool
  - Method bytecodes
  - Mapping tables
- It may also have static initializers
  - `static {}`
  - Static fields

Running the program

- Start with the main class
  - Recall how we identify methods
    - Class: `Hello`
    - Name: `main`
    - Signature: `([Ljava/lang/String;)V`
    - Classloader: `root class loader`
  - Method must also be `static` and `public`
- Program terminates when main exits


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Class Loader
Virtual Machine
Memory
Code Area
Static Data
Internal Data


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Memory
Code Area
Static Data
Internal Data
Class Loader

Interpretation

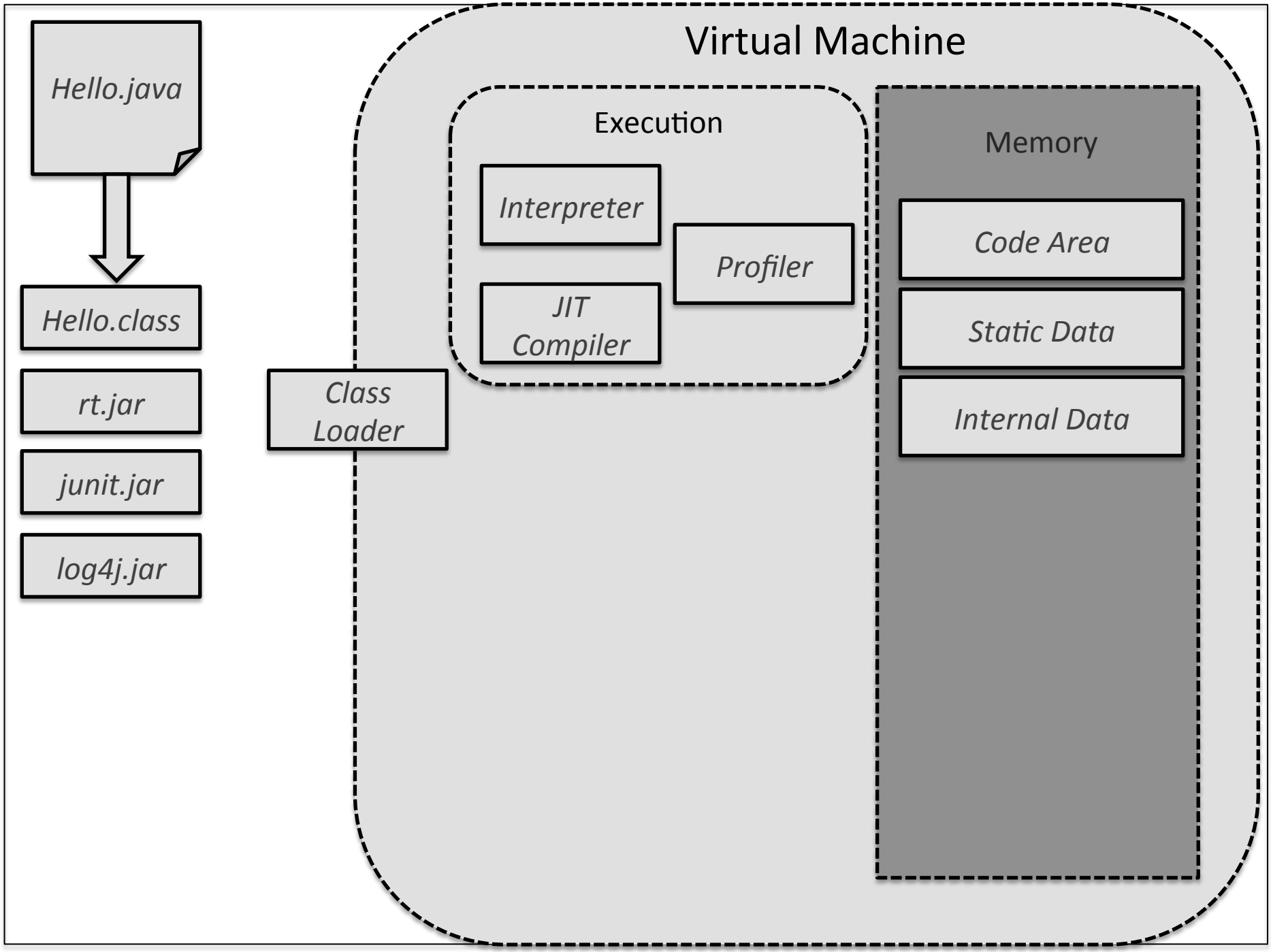
- Take each bytecode and simulate a machine
- Downsides
  - Slow and dumb
- So why bother?
  - Simple
  - Tradeoff speed vs. compilation time
  - Consider how many times `main` is called
    - What would the compilation overhead be?

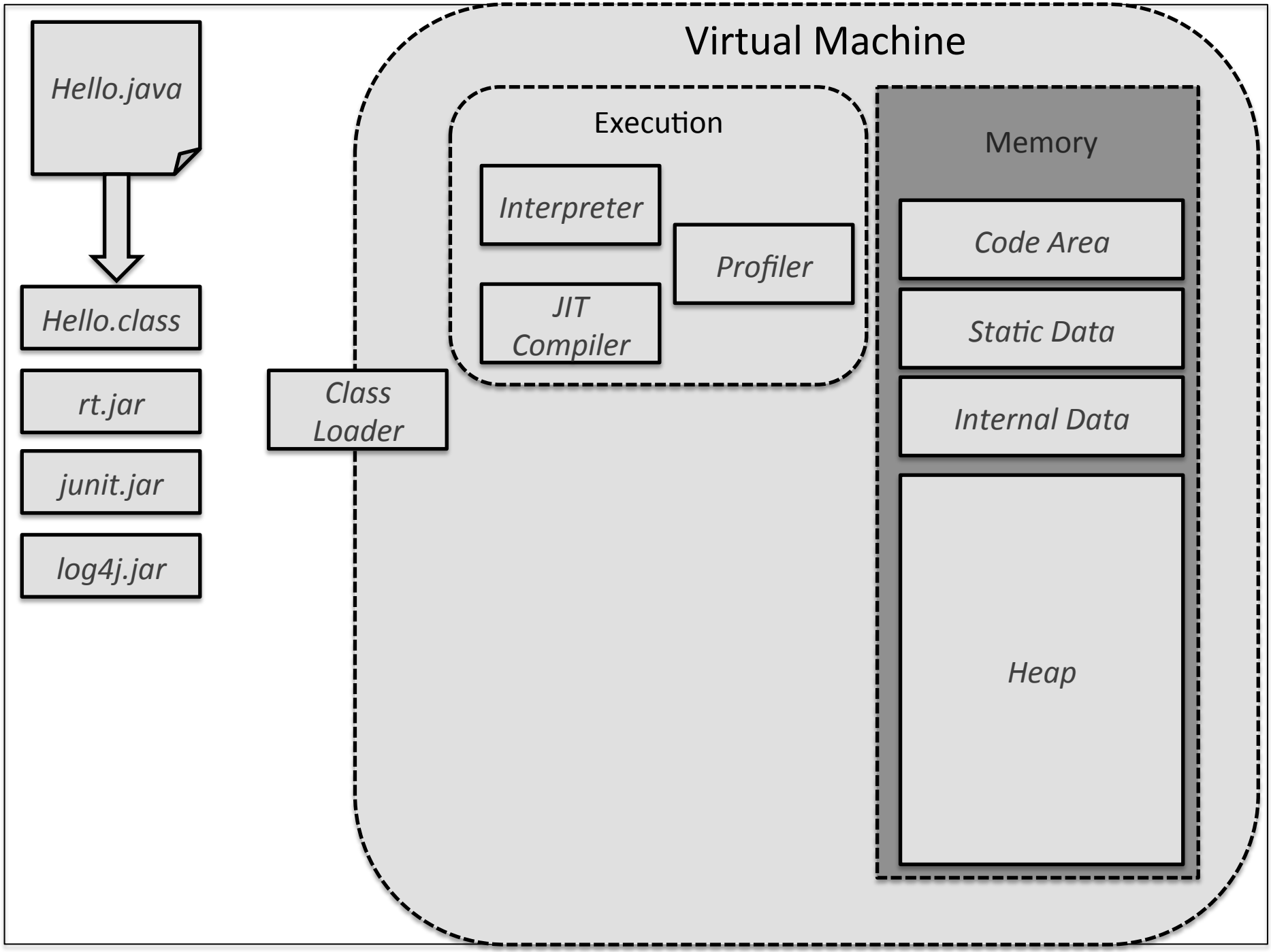
Profiling

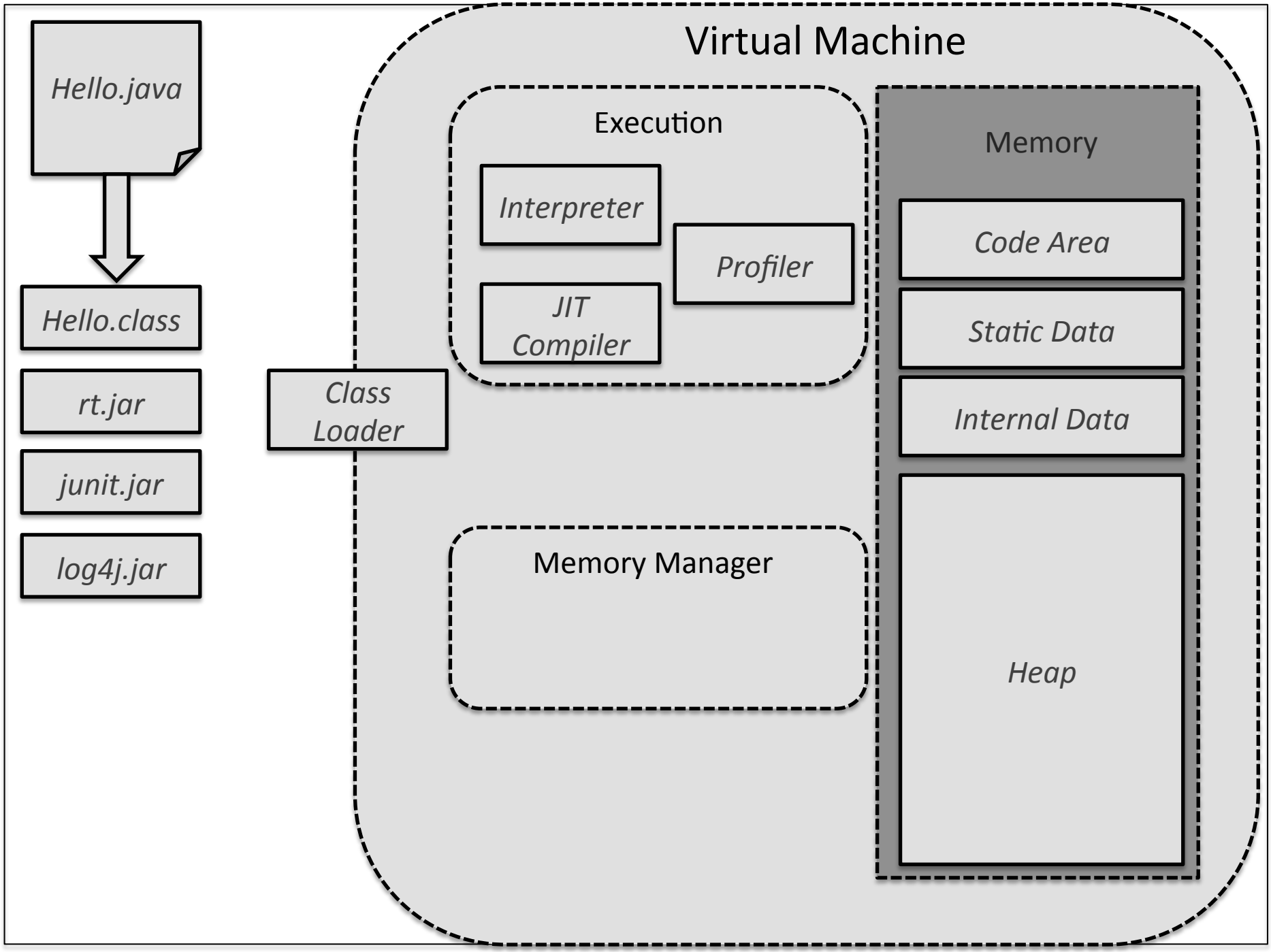
- The interpreter can track program execution
  - Tell what methods are hot
  - Identify loops
  - Identify frequently-used data
  - Phase behavior
- Targeted optimizations
  - Ahmdal's Law
  - Don't bother optimizing code that's not used

JIT Compilation

- Identify what code to optimize
  - Compile that using an optimizing compiler
  - Leave the rest to be interpreted
- JIT knows what optimizations are allowed
  - Depends on the classes that are loaded
  - Can (will) change over time
  - OK to be optimistic, so long as there's a fallback
- Gives the runtime flexibility to add extra features
- Optimize across libraries


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Memory
Code Area
Static Data
Internal Data


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Memory
Code Area
Static Data
Internal Data
Heap


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Memory Manager
Memory
Code Area
Static Data
Internal Data
Heap

Memory Management

- In the Good Old Days:
  - Memory could be stack or heap allocated
    - Stack allocation required fixed sizes
    - Heap allocation had to be tracked
  - Lots of bugs came from memory management
    - Memory leaks
    - Double frees
    - Buffer overflows clobbering metadata
- Allocation and Free are always linked.

Allocation

- Lots of possible allocation algorithms
  - Best fit
  - Worst fit
  - First fit
  - Free list
  - Stack or Region allocation
- All have their tradeoffs
  - Performance
  - Fragmentation

Garbage Collection

- Save developers from managing memory
  - Safer, fewer bugs
  - Simplify interfaces
  - Often more efficient (memory freed earlier)
- Present the abstraction of an infinite heap
- A handful of basic approaches
  - With infinite variations on the themes

Reference Counting

- Keep track of references to a memory location
  - When that number hits zero, free the object
- Implemented in C++ as smart pointers
  - Or shared/unique/weak pointers
- Good if nothing else is available
  - Susceptible to circular references
  - Fallback mechanism or new semantics required.

Tracing

- Start with a known set of live references (roots)
  - Static variables
  - Pointers on the execution stack
  - Various others
- Mark the objects that those point to
- Scan the marked objects for more references

Mark Sweep

- Also called scavenging collectors
  - Figure out what objects are alive
  - Delete everything else
- Small conceptual step from reference counting
- Makes use of the full heap
  - But can lead to fragmentation

Copying

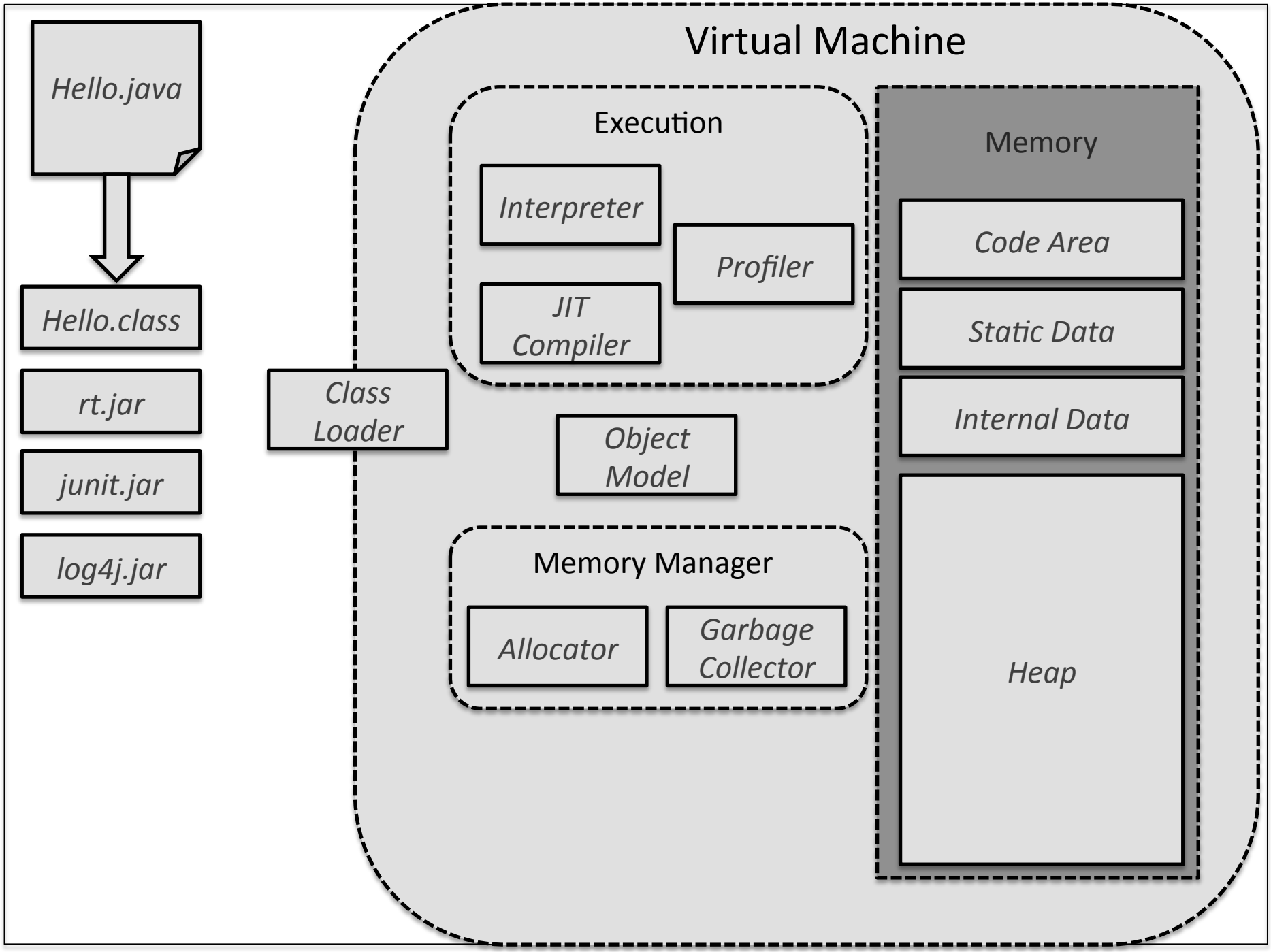
- Takes the opposite approach to Mark Sweep
  - Figure out what objects are alive
  - Copy those objects to some other place
  - Clear the entire allocation area
- Allocation is trivially simple
  - You'll see in Assignment Three
- Adds overhead in time and space
  - Need to set aside an area to copy live objects
  - Although that can be partially mitigated

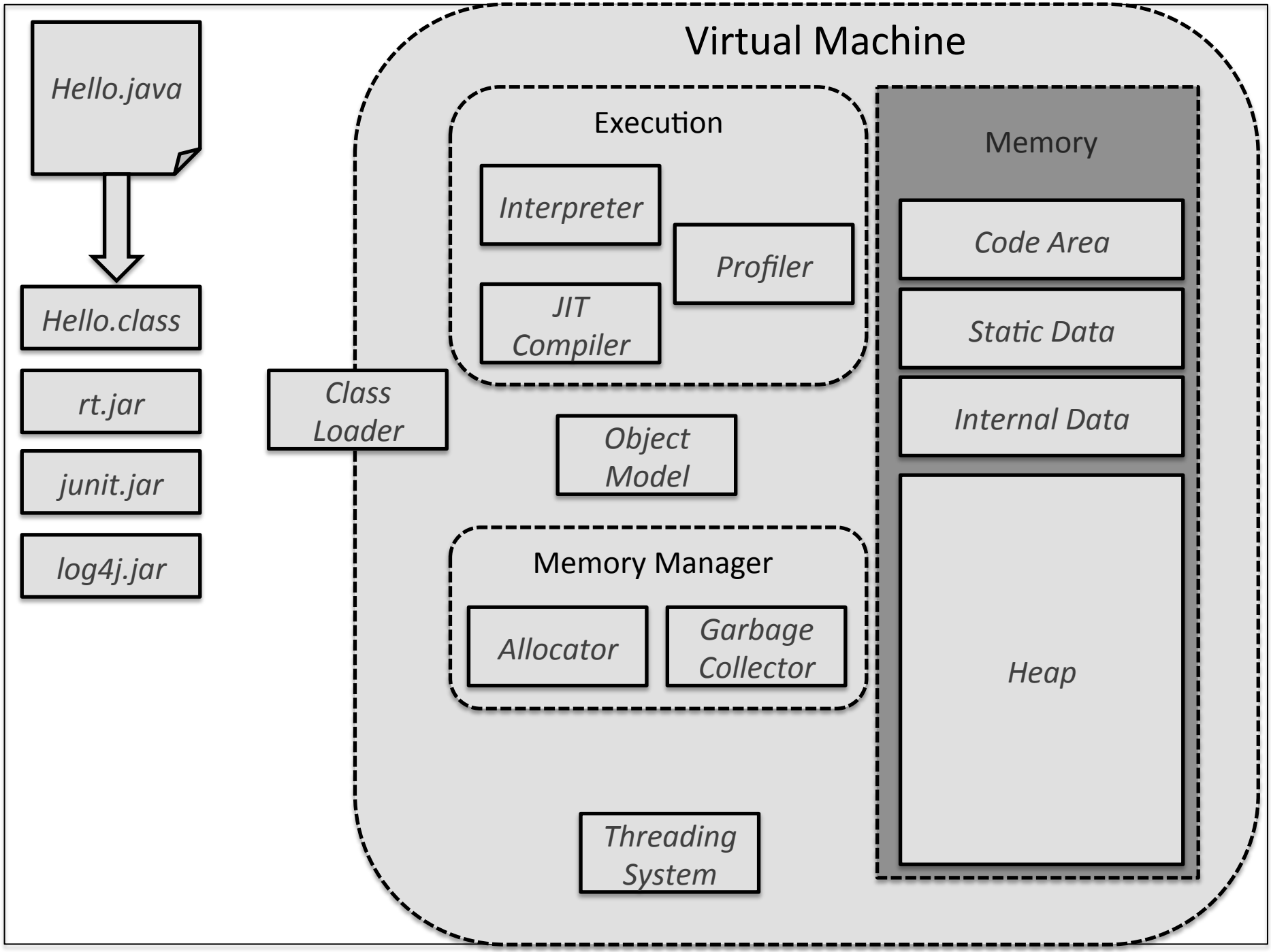
Conservative GC

- Less common alternative to reference counting
  - Useful when we don't have perfect information
- If it looks like a pointer, assume it is a pointer
  - In case we guess wrong and don't mark a live object
- Can't do copying in case we're wrong
- Lots of complexity for little improvement over RC.

Object Layout

- The runtime system needs information
  - Type of an object on the heap
  - Size of the object or array
  - Location of its references
- May also need to store extra data
  - Garbage collection mark bits
- Every heap object has a standard header
  - Size is important


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Memory
Code Area
Static Data
Internal Data
Heap


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Threading

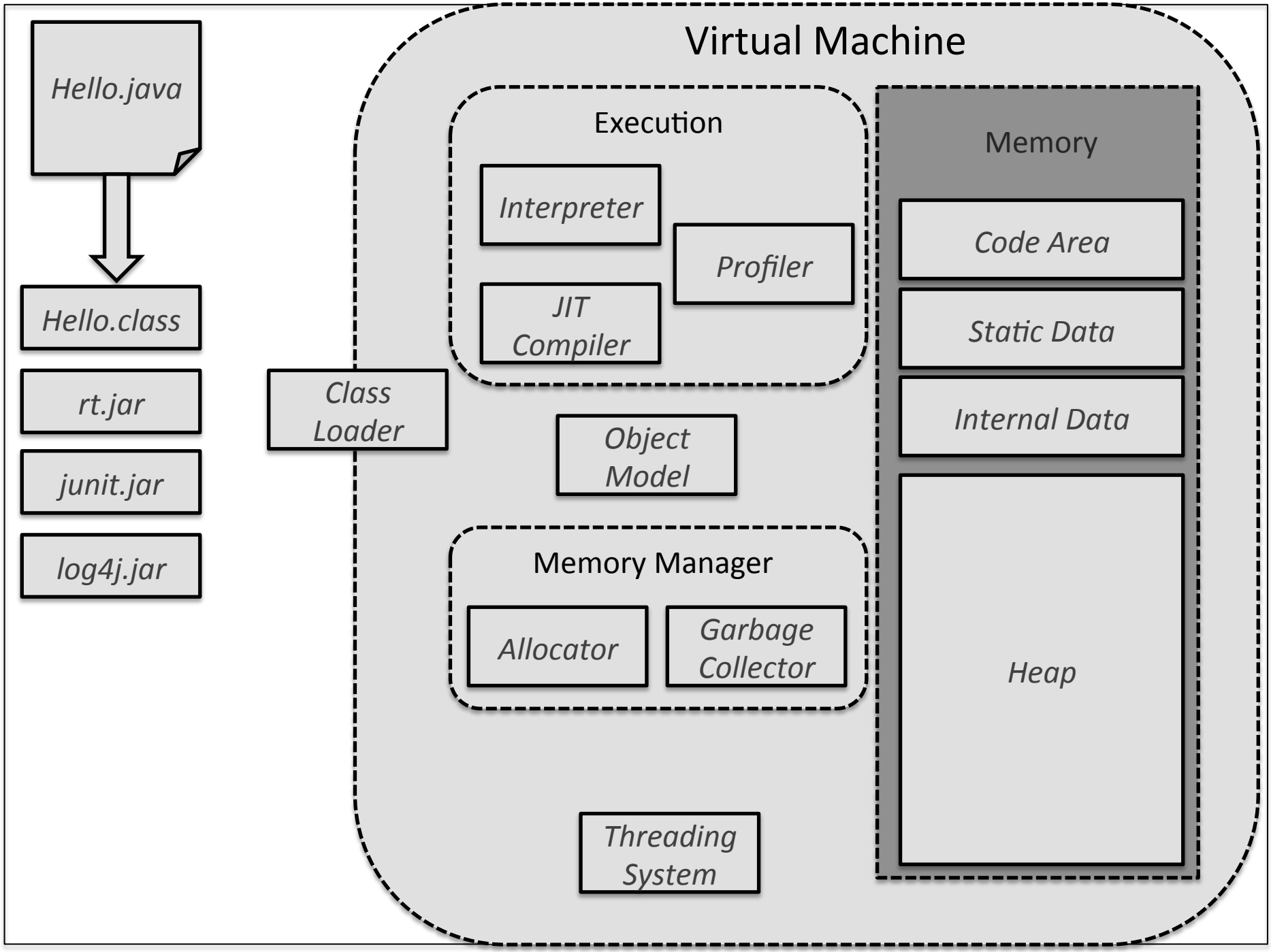
- Threads are vital for performance
- Building a correct concurrent program is hard
  - Building a correct concurrent VM even more so
- Concurrency requires thought in all components

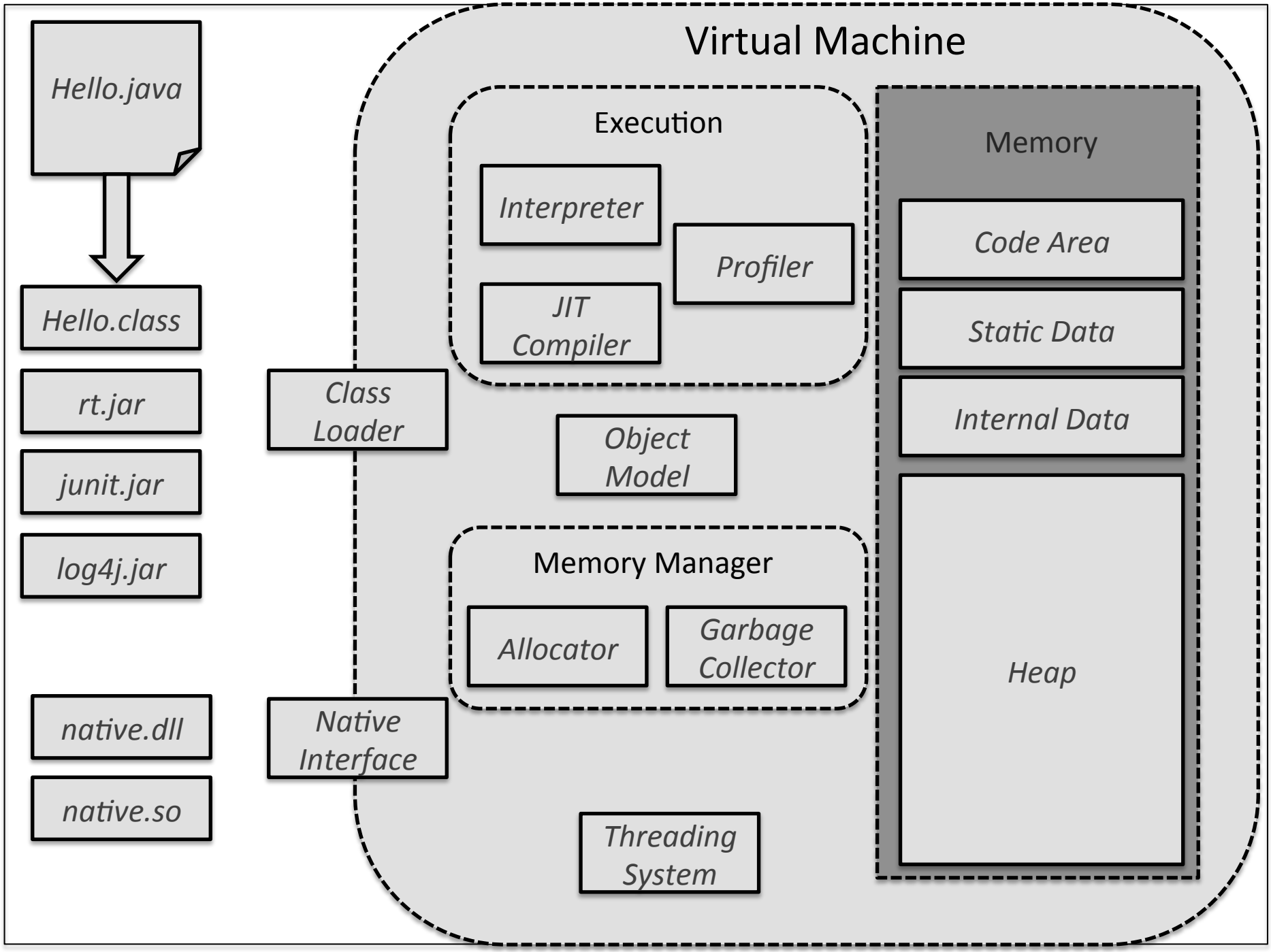
Application Threads

- Threads are available to the developer
  - Require a clear memory model
- Some implementation challenges
  - How to map VM-level threads to OS threads
  - How to implement locks
- Various solutions are available

Background Threads

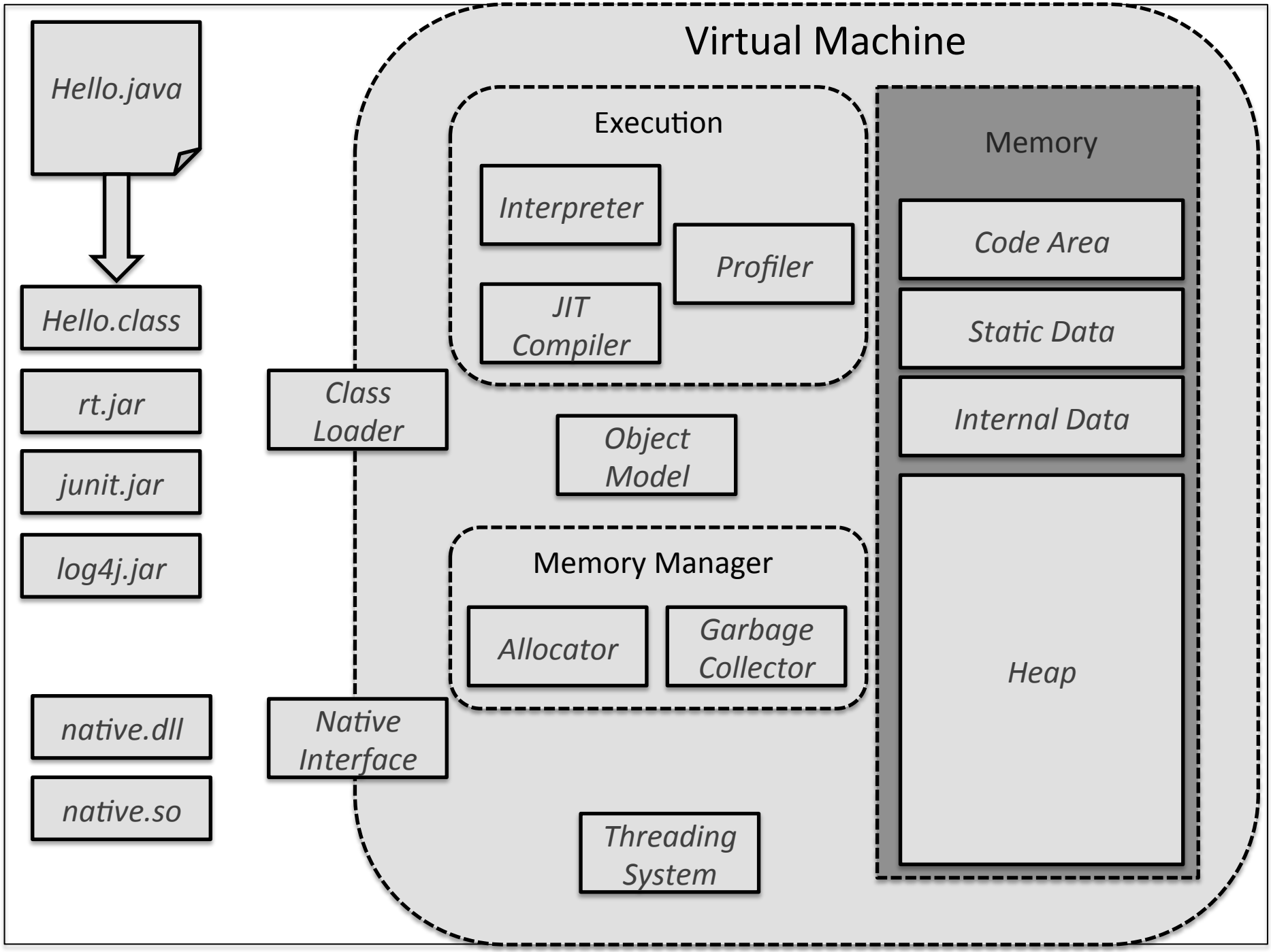
- Concurrency is not just for applications
  - Garbage collection threads
  - JIT compilation threads
  - Profiling threads
- We can't assume that anything is single-threaded
- Huge potential for performance improvement
  - But at the expense of complexity
  - Consider some of the assumptions in a garbage collector


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Threading System
Memory
Code Area
Static Data
Internal Data
Heap


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Native Interfaces

- Many VM languages have a native interface
  - An escape hatch for added performance
  - A way to link to legacy code
- Native code has to be a good citizen
  - Consider how that affects other components
    - Memory management
    - Threading and locking
    - Error handling


Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap