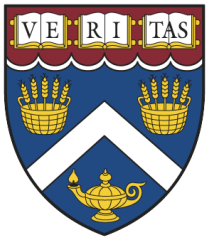
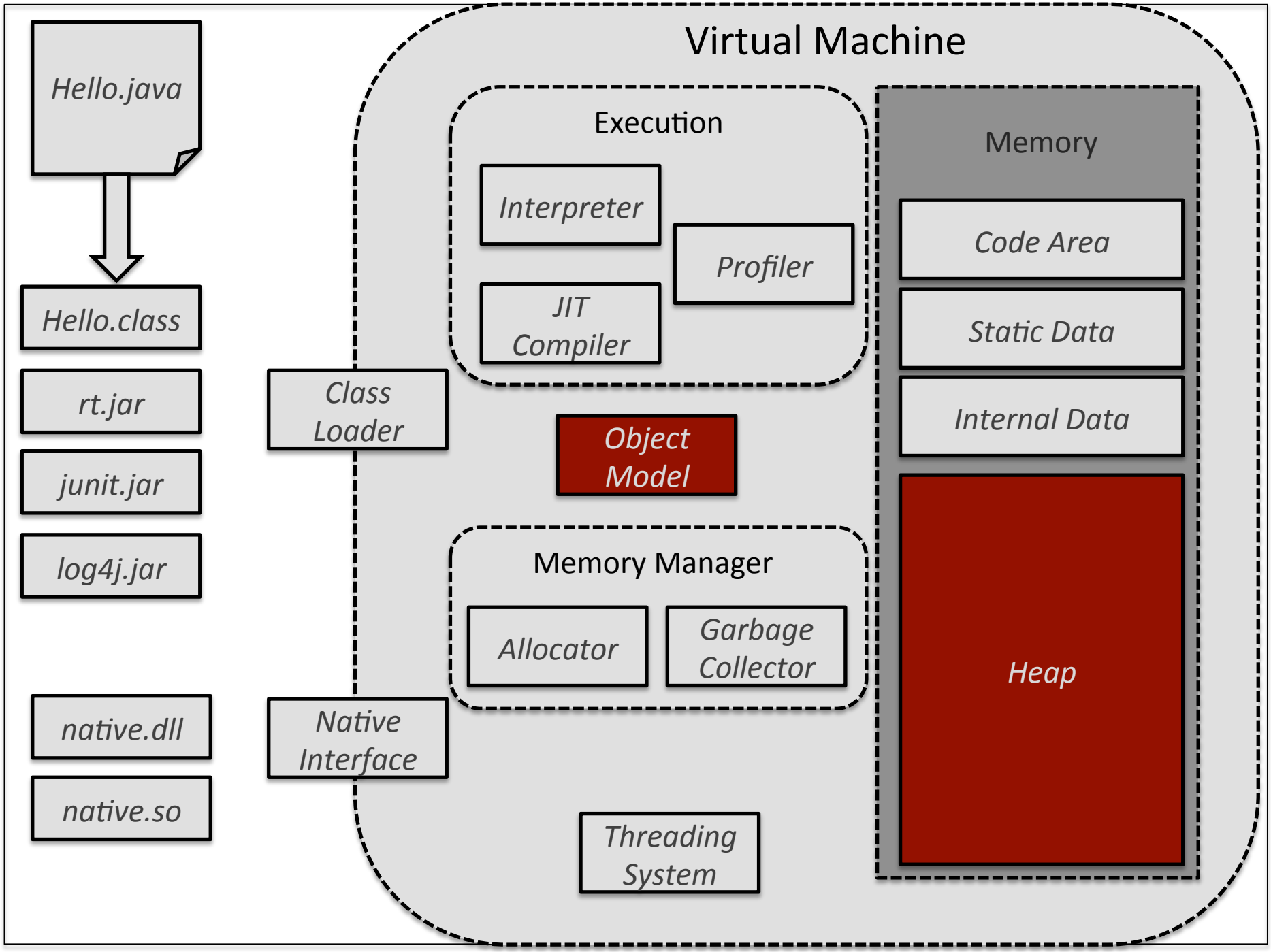
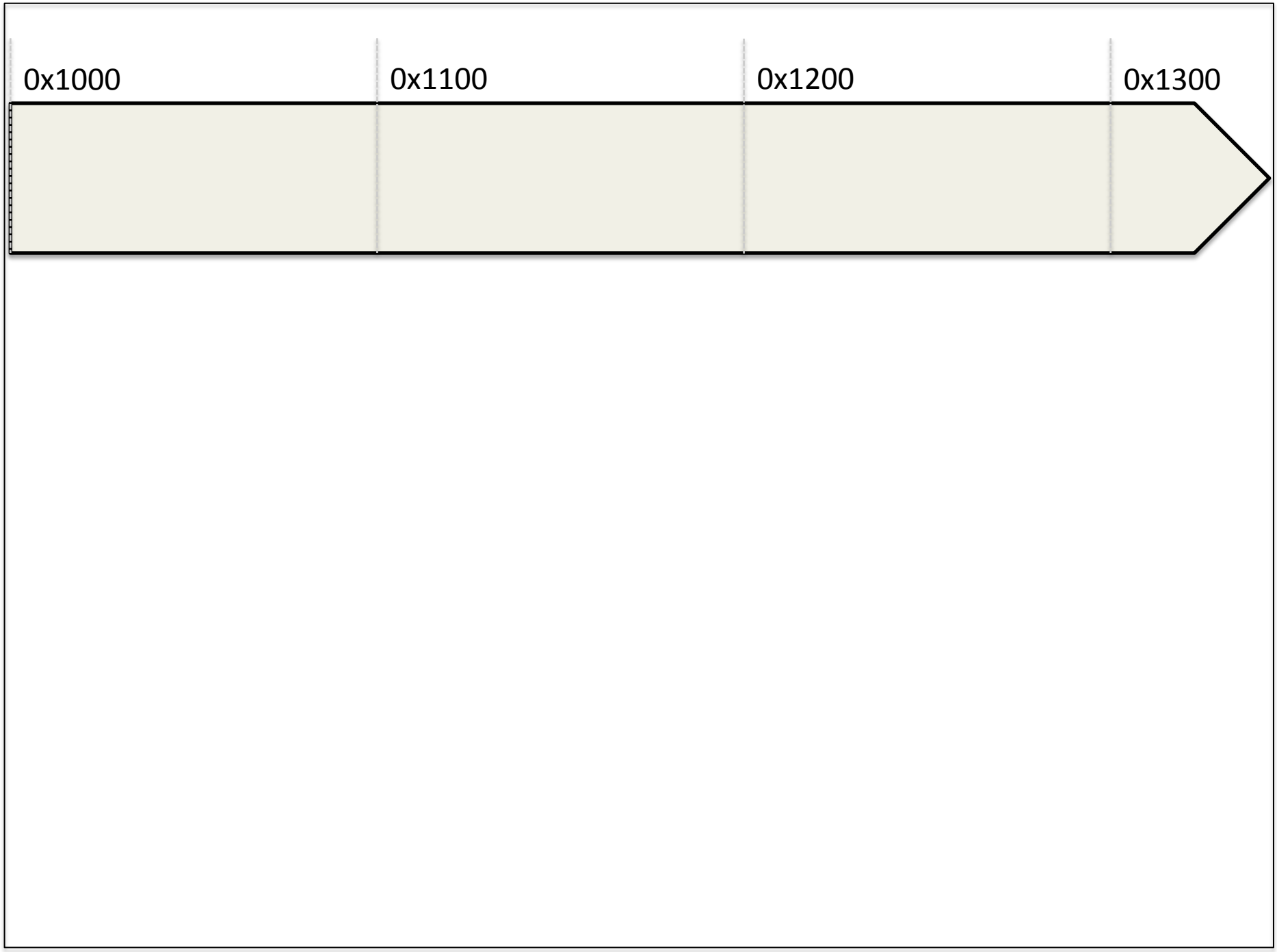
Object Representations

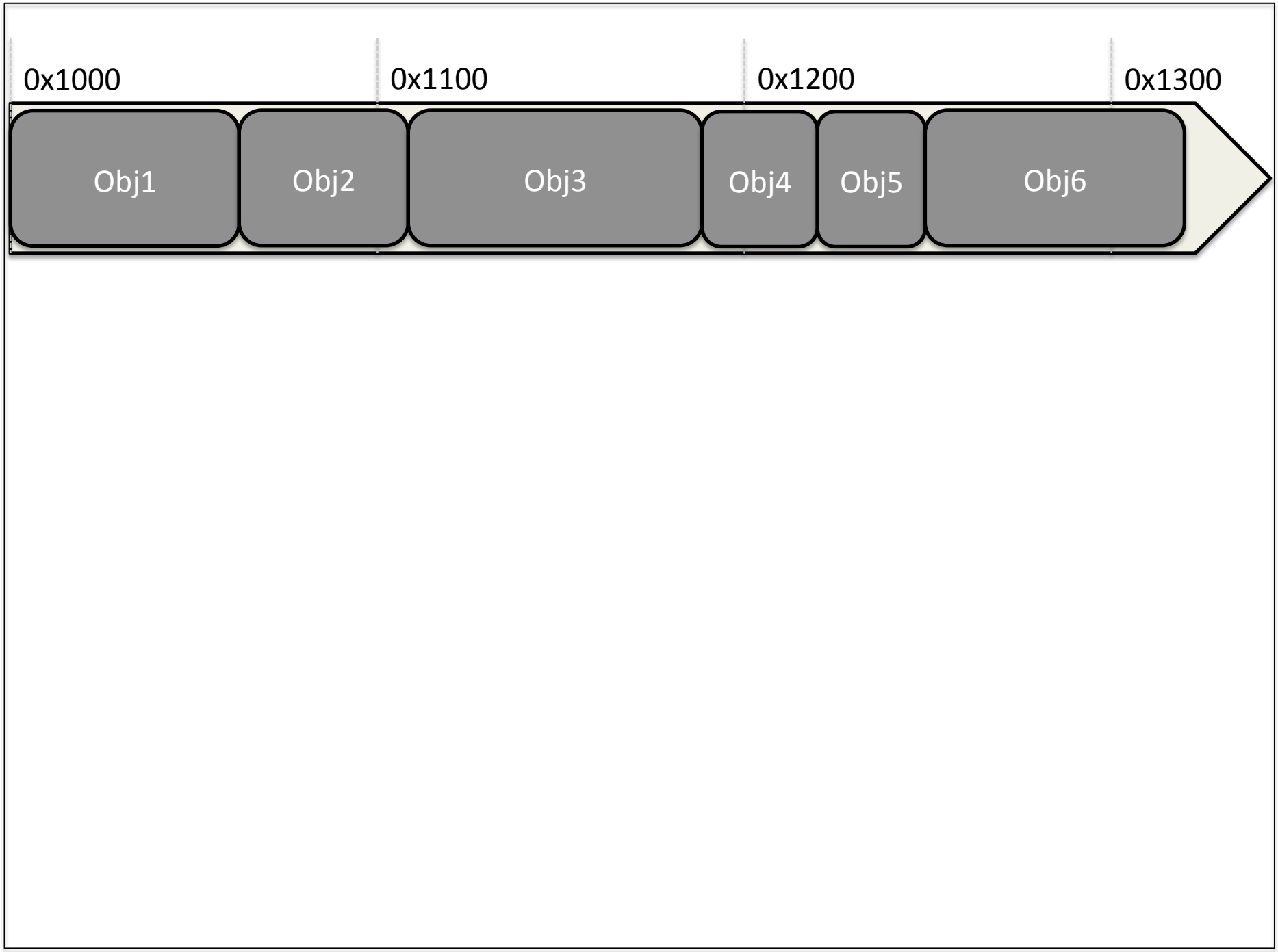


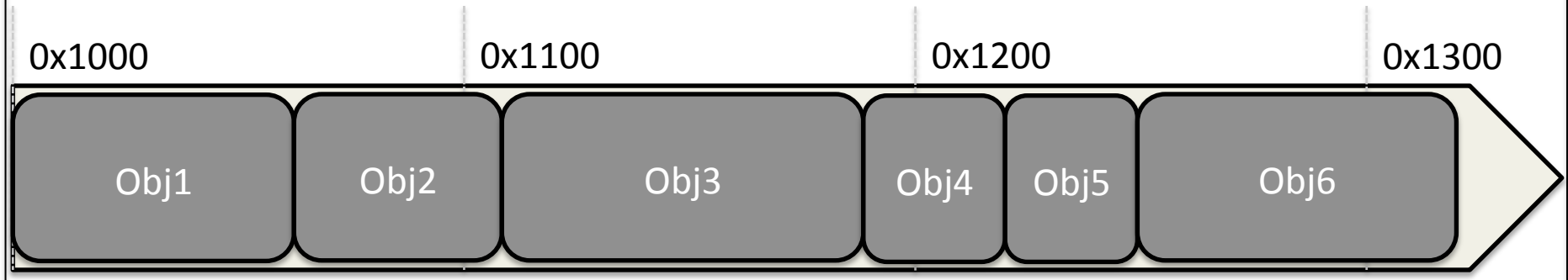

Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Primitives and Objects

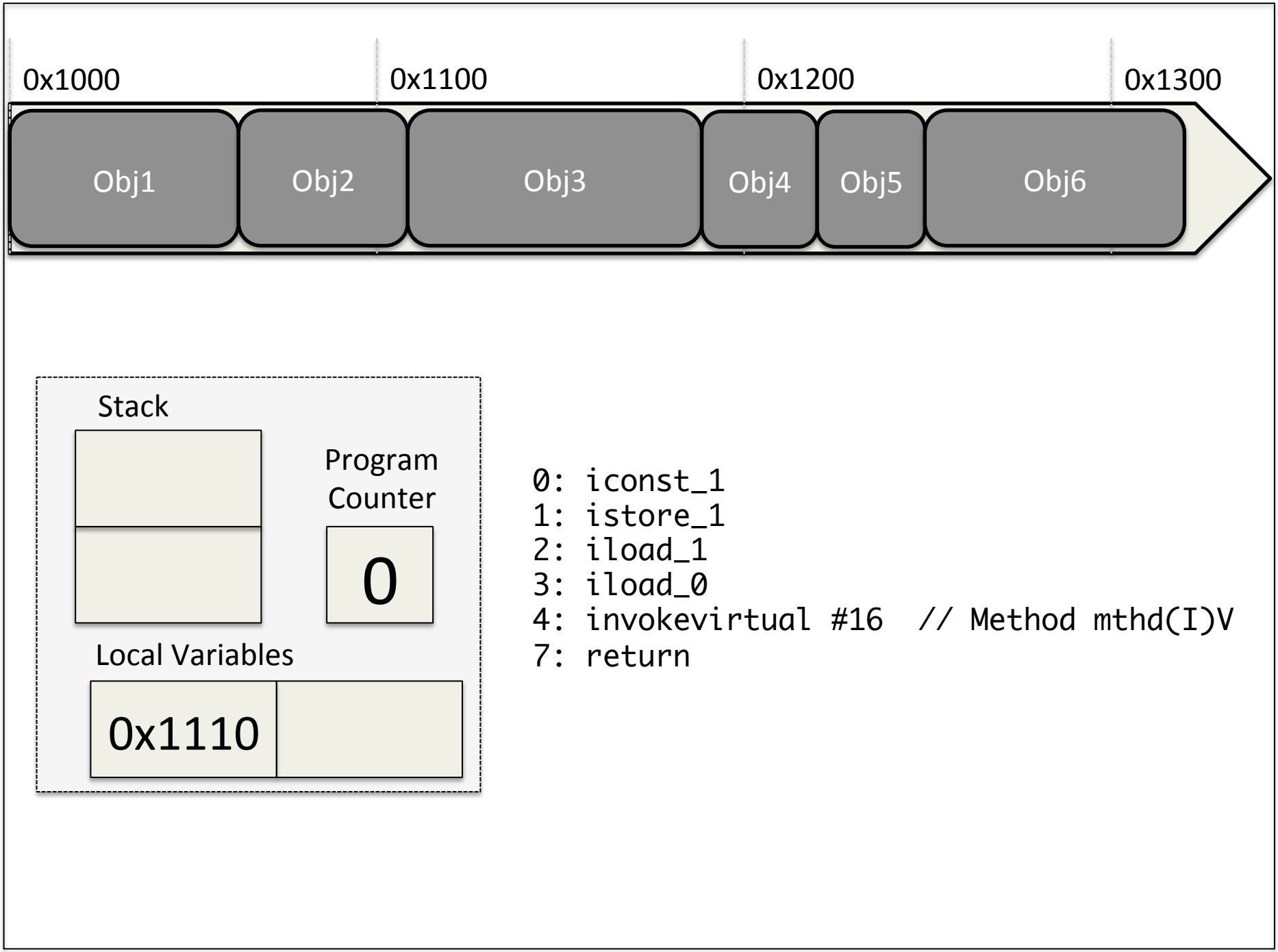
- Primitives
  - Built-in
  - Immutable
  - Stored on the Java stack or in fields
- Objects
  - Classes defined by application or library
  - Mutable
  - Stored on the heap


0x1000
0x1100
0x1200
0x1300

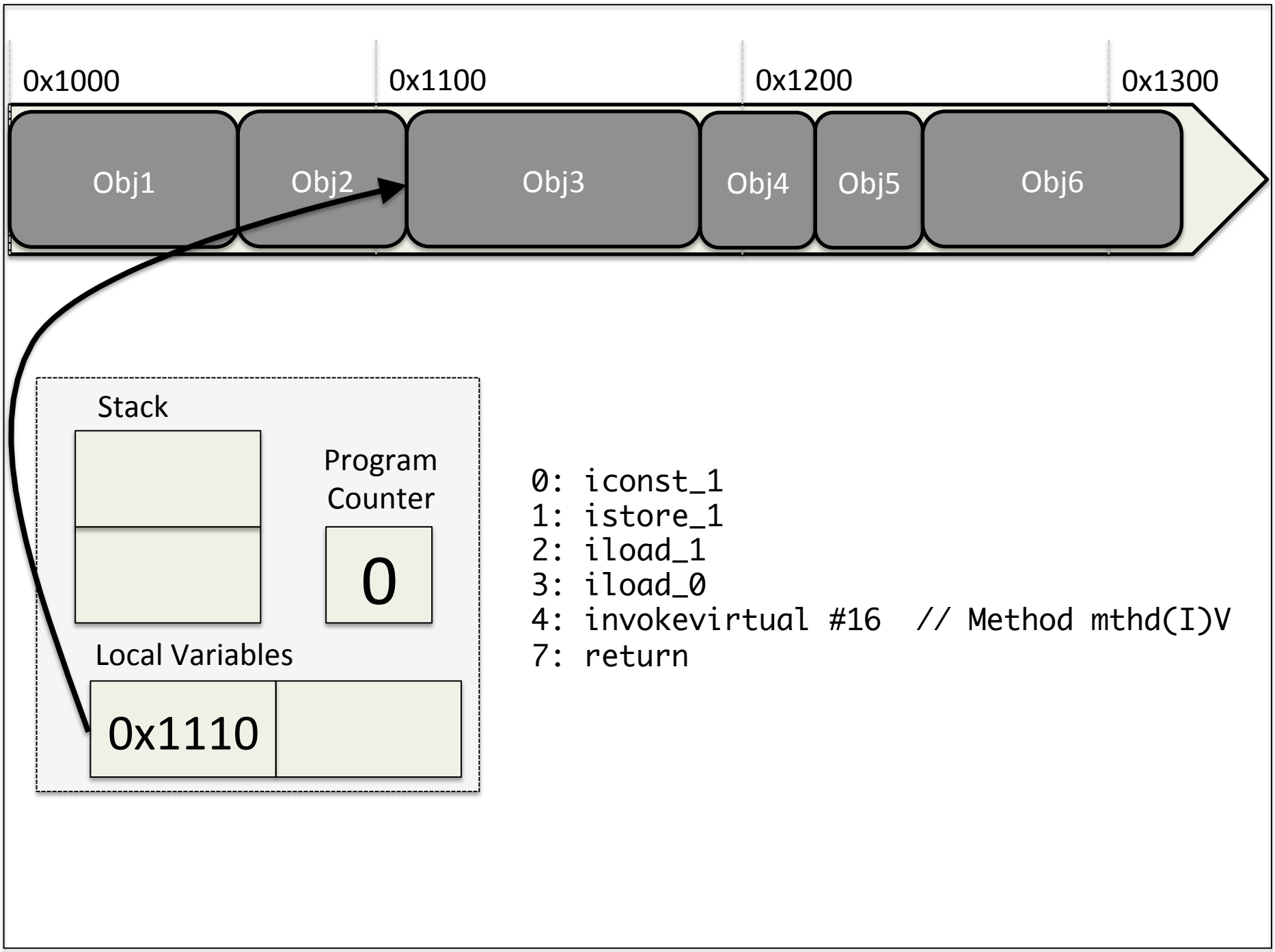

0x1000
0x1100
0x1200
0x1300
Obj1
Obj2
Obj3
Obj4
Obj5
Obj6



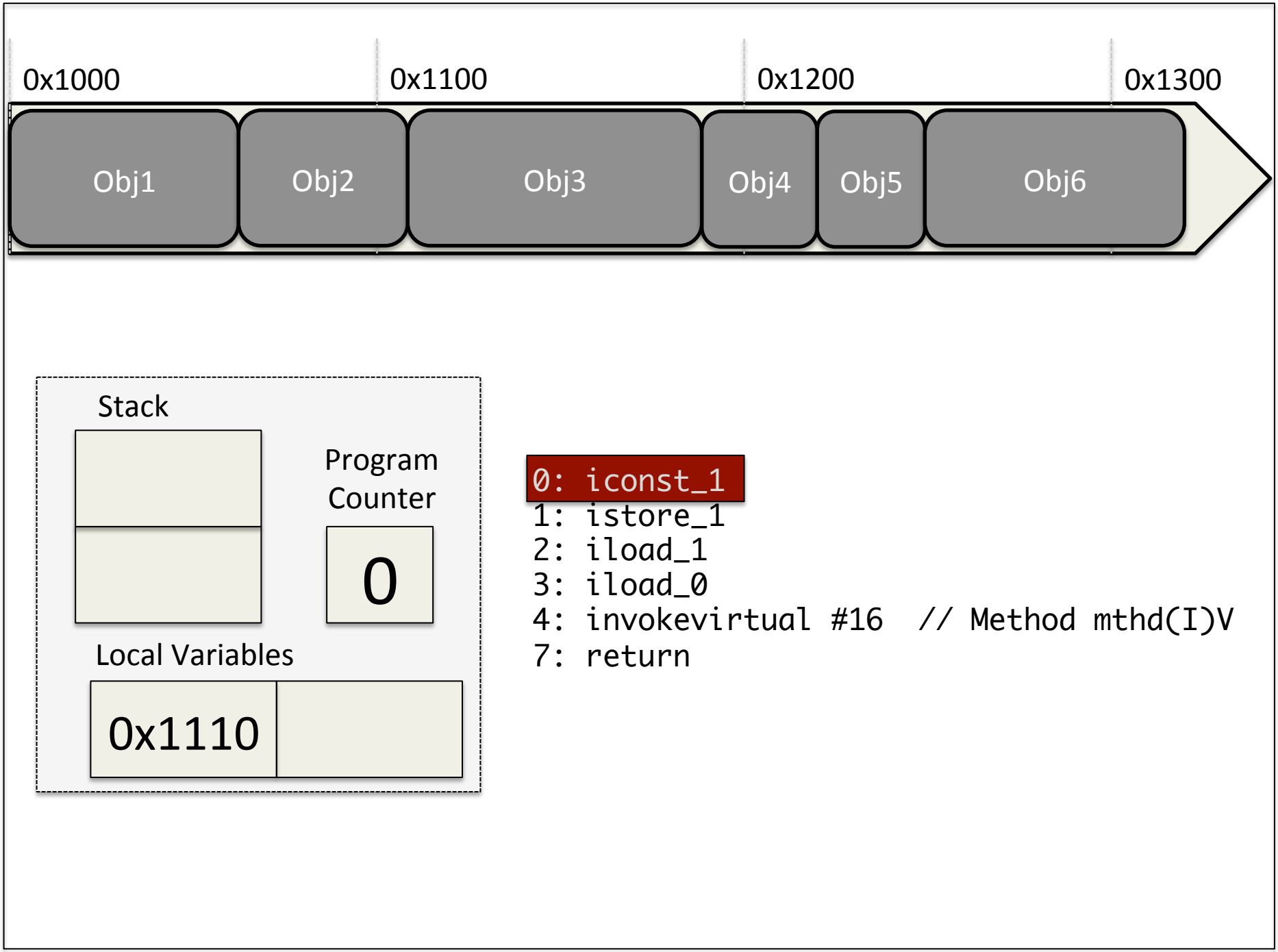

```
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16  // Method mthd(I)V
7: return
```



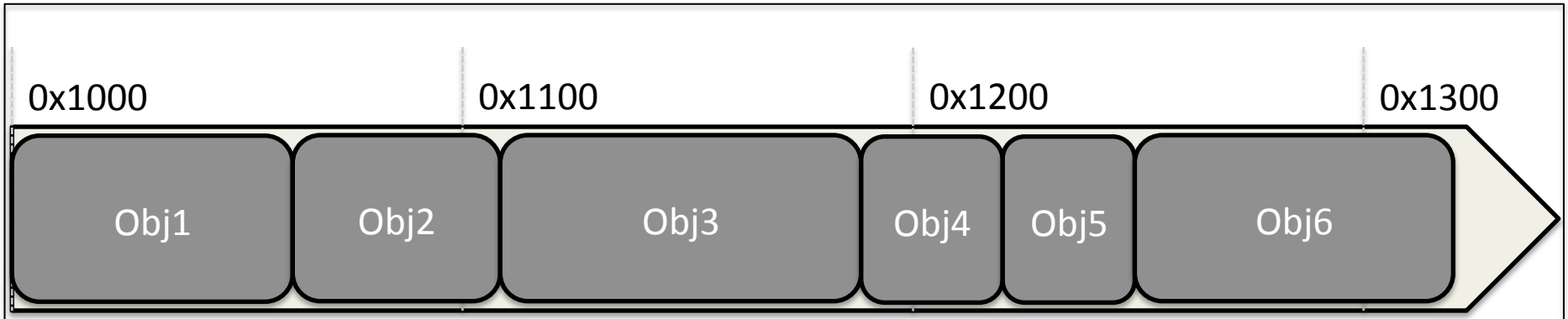

```
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16  // Method mthd(I)V
7: return
```

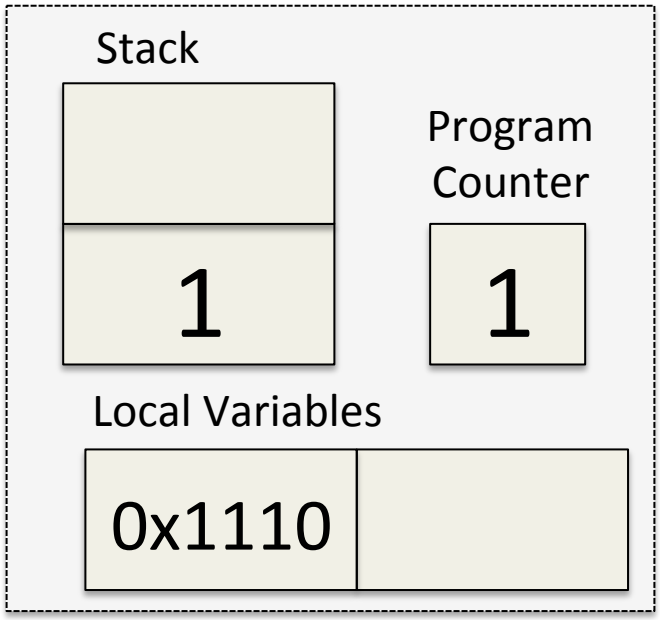



```
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16  // Method mthd(I)V
7: return
```

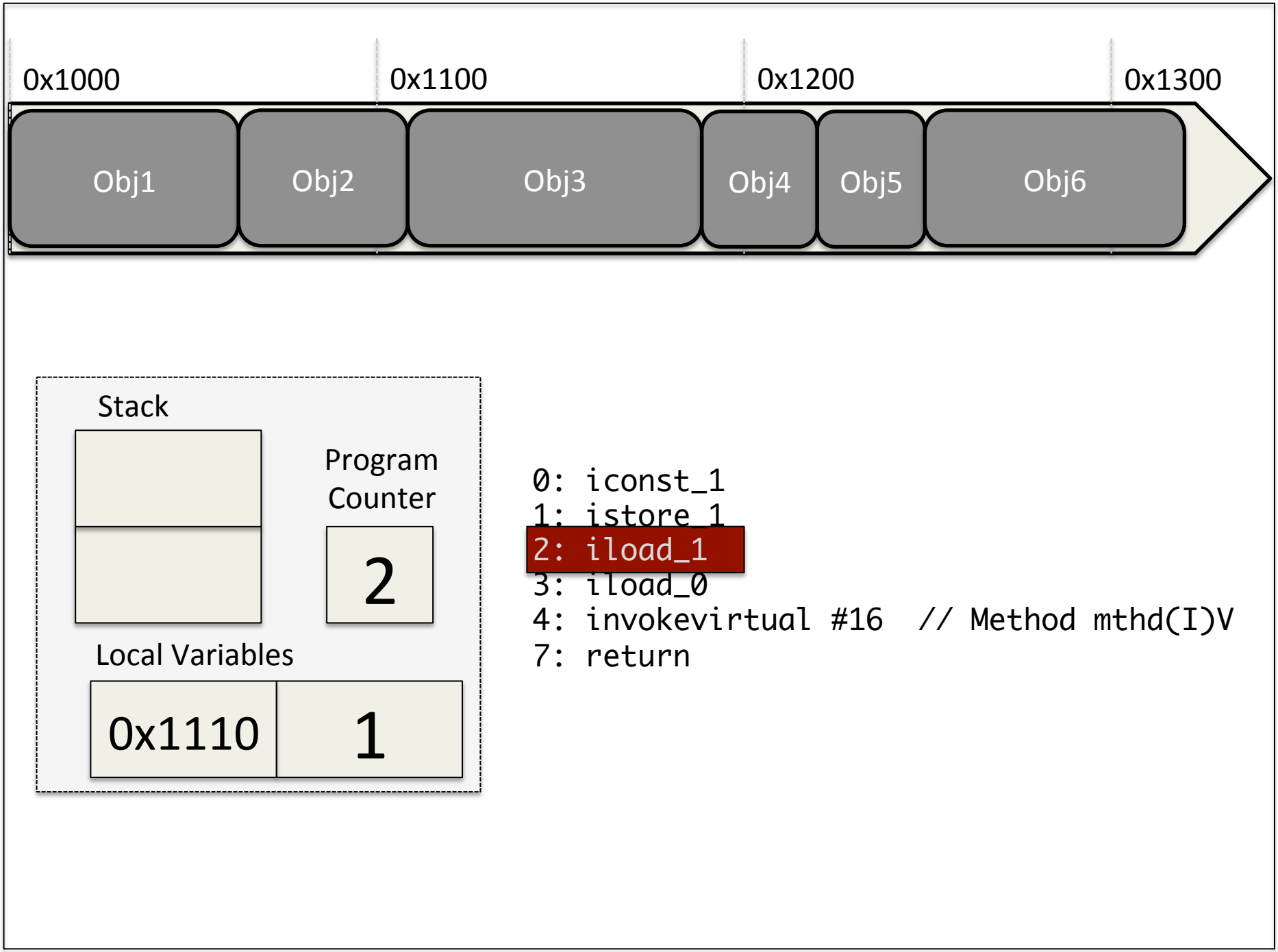


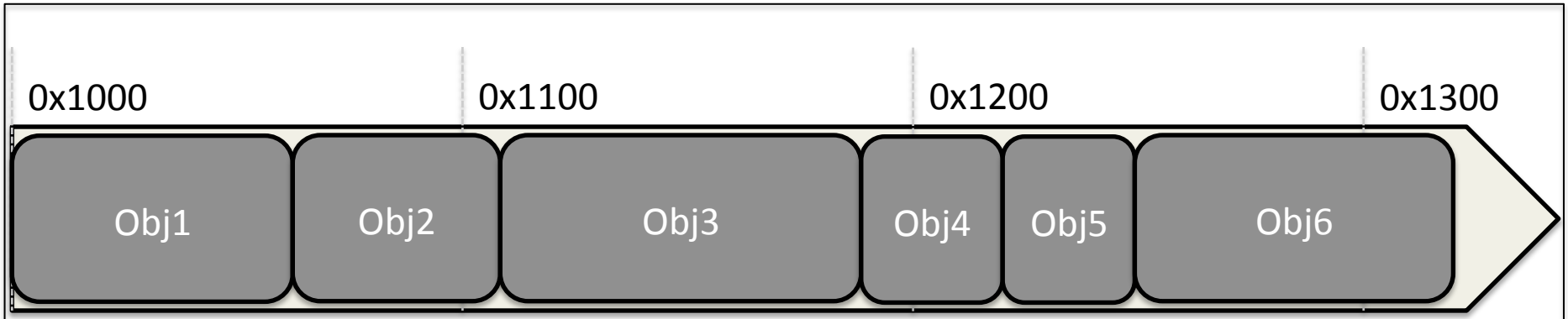

```
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16  // Method mthd(I)V
7: return
```

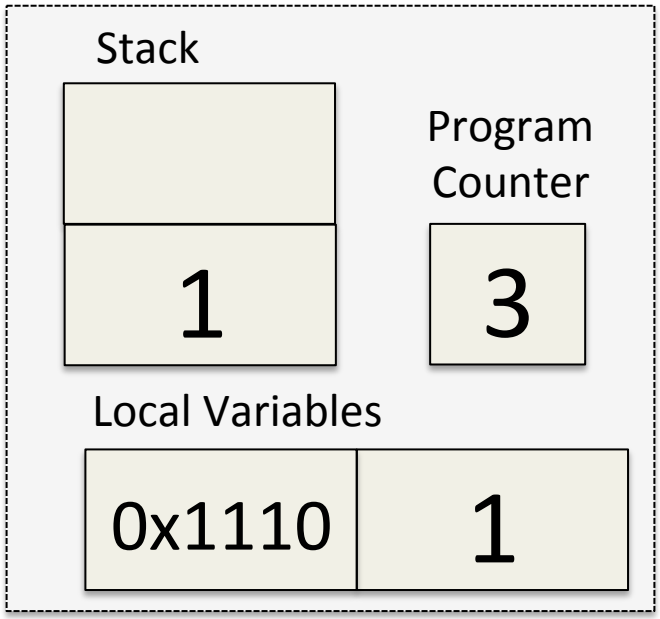




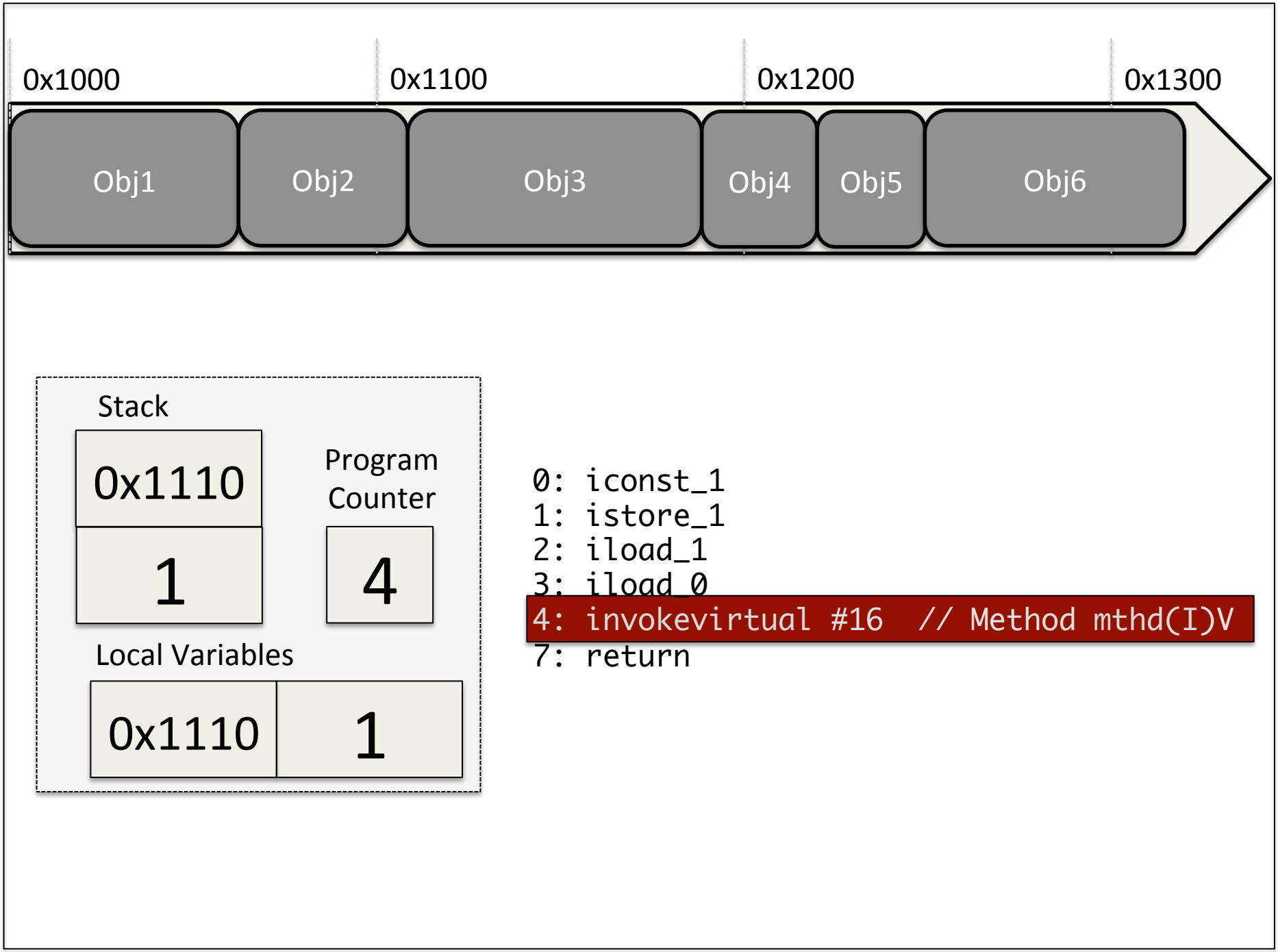

```
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16  // Method mthd(I)V
7: return
```

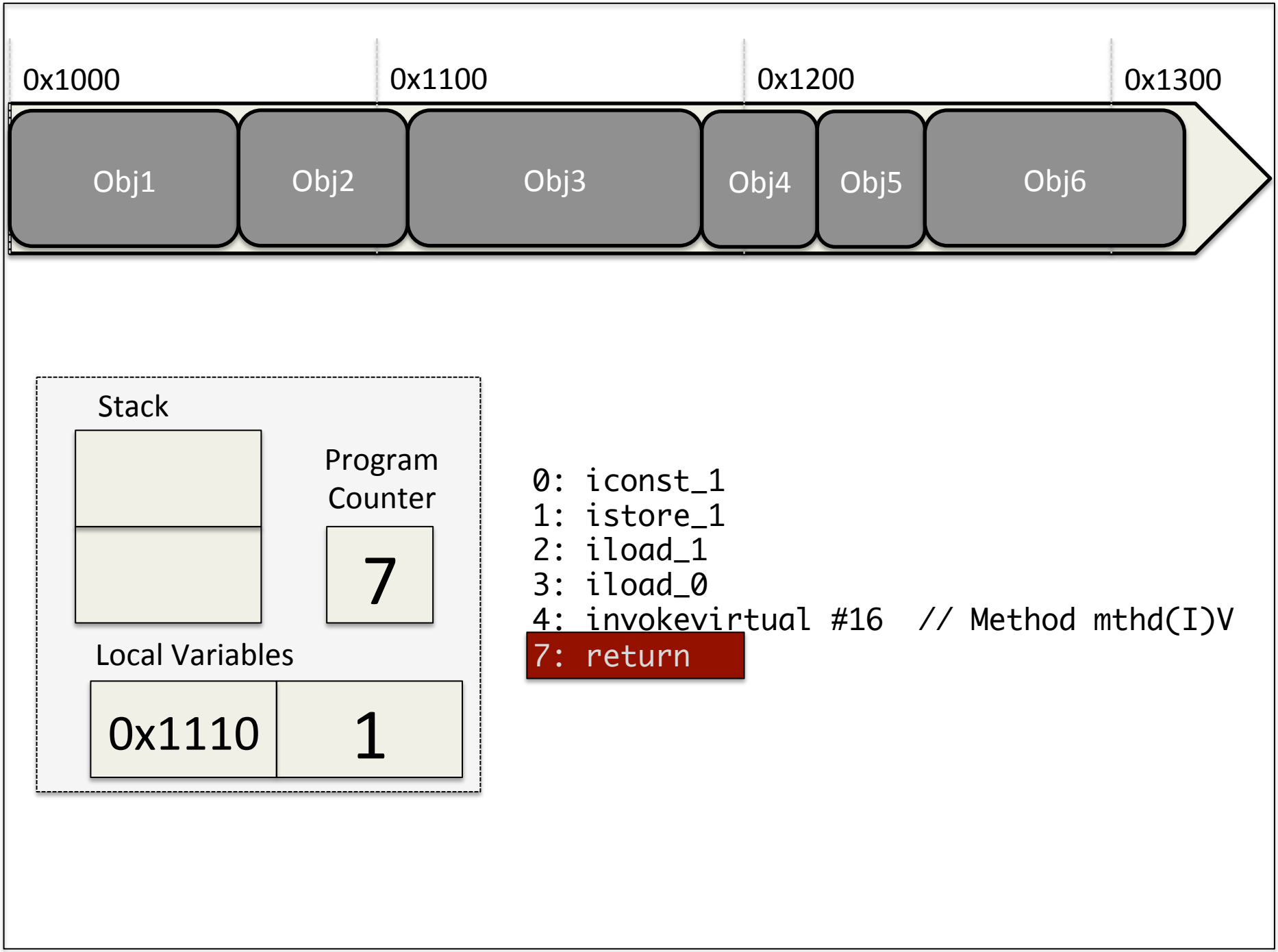

0x1000
0x1100
0x1200
0x1300
Obj1
Obj2
Obj3
Obj4
Obj5
Obj6
Stack
Program Counter
2
Local Variables
0x1110
1
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16 // Method mthd(I)V
7: return






```
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16  // Method mthd(I)V
7: return
```


0x1000
0x1100
0x1200
0x1300
Obj1
Obj2
Obj3
Obj4
Obj5
Obj6
Stack
0x1110
1
Program Counter
4
Local Variables
0x1110
1
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16 // Method mthd(I)V
7: return


0x1000
0x1100
0x1200
0x1300
Obj1
Obj2
Obj3
Obj4
Obj5
Obj6
Stack
Program Counter
7
Local Variables
0x1110
1
0: iconst_1
1: istore_1
2: iload_1
3: iload_0
4: invokevirtual #16 // Method mthd(I)V
7: return

References in Java

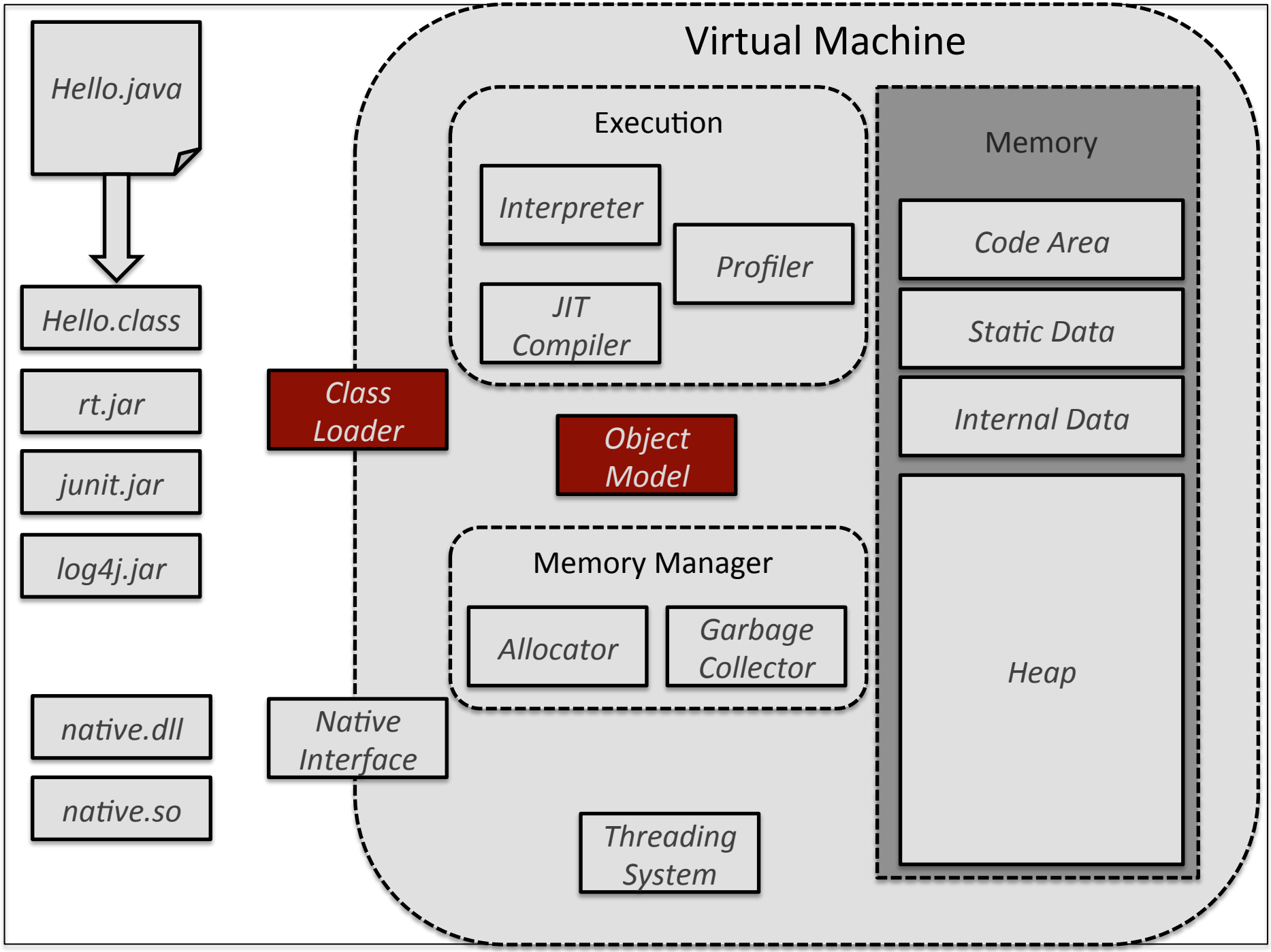
- Java hides references from developers
  - All a developer knows is that they have an object
  - As VM developers, we need two different concepts
- References can be treated similarly to primitives
  - Exist on the stack and local variables
    - Category 1 value
  - They can be stored in object fields
  - They are passed by value during method calls

Pointers vs References

- References
  - Guaranteed to point to a valid heap object
  - Dereference a reference to get an object
- Pointers
  - Arbitrary location in the heap
  - May point to the middle of an object
  - Not possible in Java

Arrays

- Largely similar to Objects
  - Stored on the heap
  - Mutable
- Some differences
  - All fields are the same type
  - VM needs to track array size
  - Accessed using different bytecodes


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

Object Representation in C

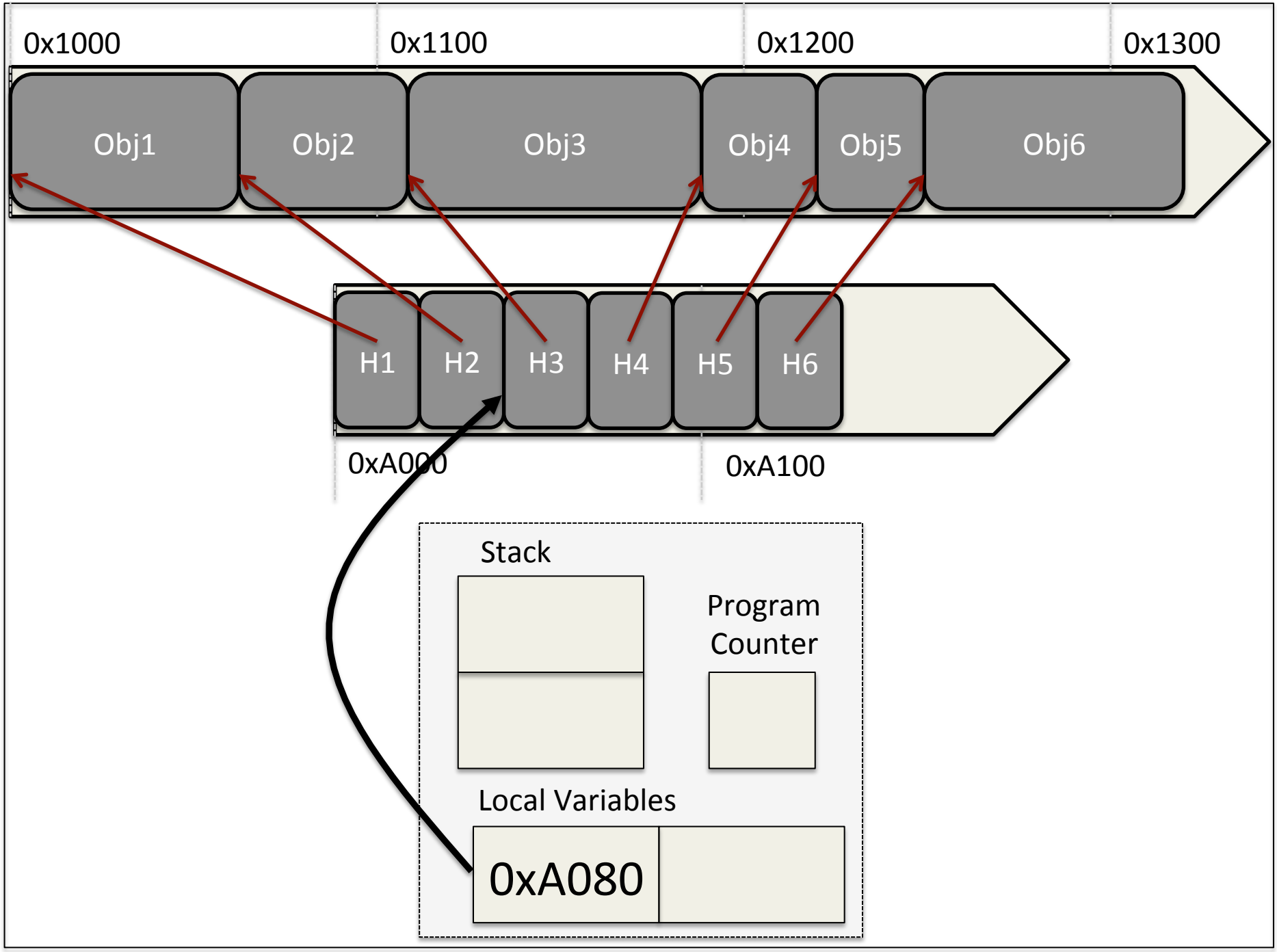
- Bits is bits
  - Object is a series of bits on the heap
- The meaning of the object comes from its use
  - Unions
  - Casting char array as a struct
- Developer or compiler tracks types

Managed Objects

- Runtime needs to know more about the object
  - Type of object
  - Class hierarchy for dynamic dispatch
  - Layout of fields, location of references
  - Lock state
  - GC state
  - Hash code
- Two kinds of information
  - Per-Class
  - Per-Object

Handles

- Object references point to metadata block
  - Metadata is stored off to the side
  - Keeps track of where the object is on the heap

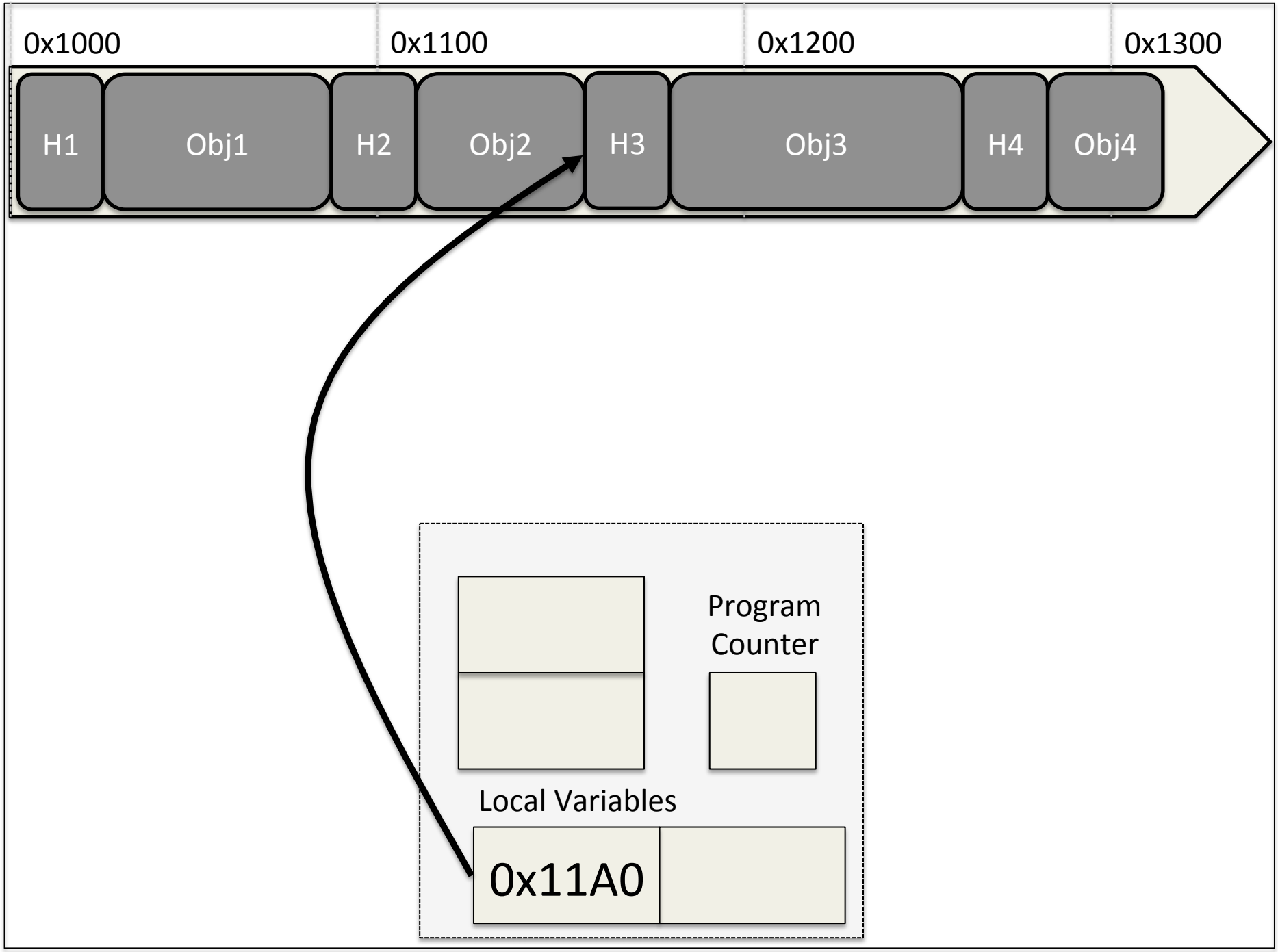

0x1000
0x1100
0x1200
0x1300
Obj1
Obj2
Obj3
Obj4
Obj5
Obj6
H1
H2
H3
H4
H5
H6
0xA000
0xA100
Stack
Program Counter
Local Variables
0xA080

Handles

- Object references point to metadata block
  - Metadata is stored off to the side
  - Keeps track of where the object is on the heap
- All object dereferences go through indirection
  - Two memory references instead of one
- Handle locations independent of heap layout
  - Useful for copying garbage collection

Object Headers

- Every object has a header block


0x1000
0x1100
0x1200
0x1300
H1
Obj1
H2
Obj2
H3
Obj3
H4
Obj4
Program
Counter
Local Variables
0x11A0

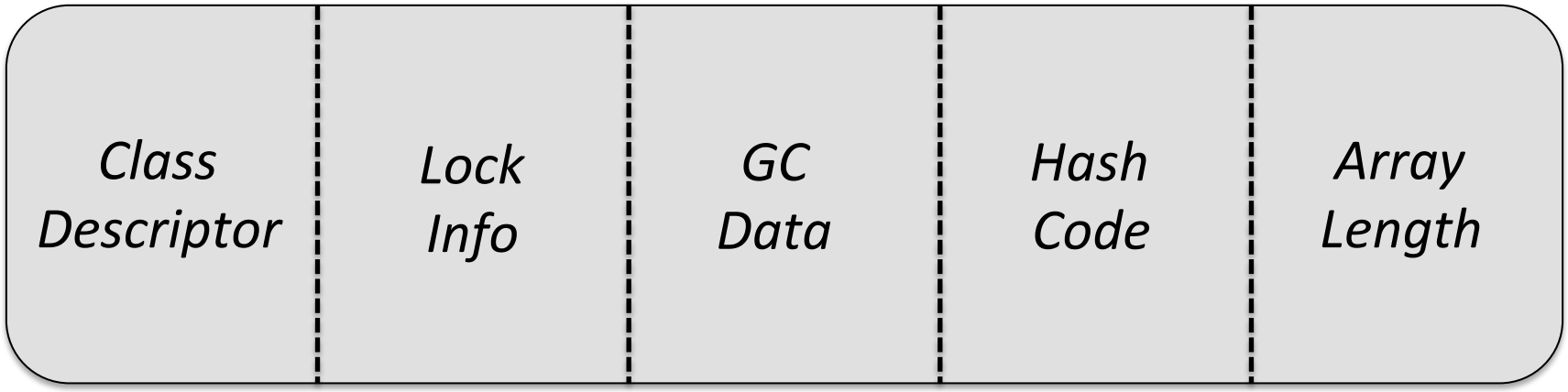
Object Headers

- Every object has a header block
- Keep metadata close to the object
  - Only one access per dereference
  - Makes better use of the cache
- Takes up more space on the heap

Header Format

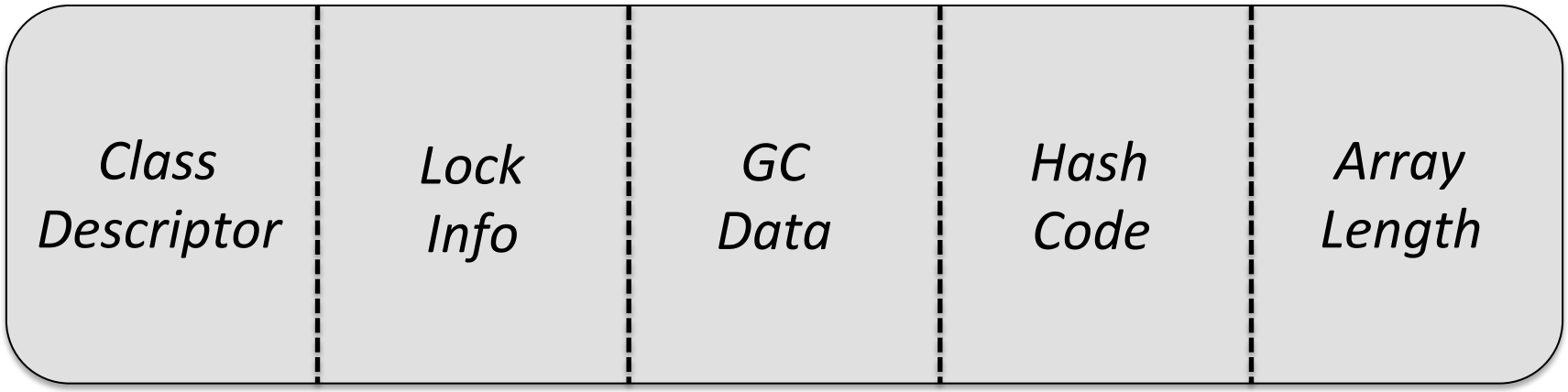
- Fixed header format
  - Implementation specific
  - Simplifies generated access code
- Slight difference between arrays and objects
  - Array length field
  - Different bytecodes to handle each

Header Data



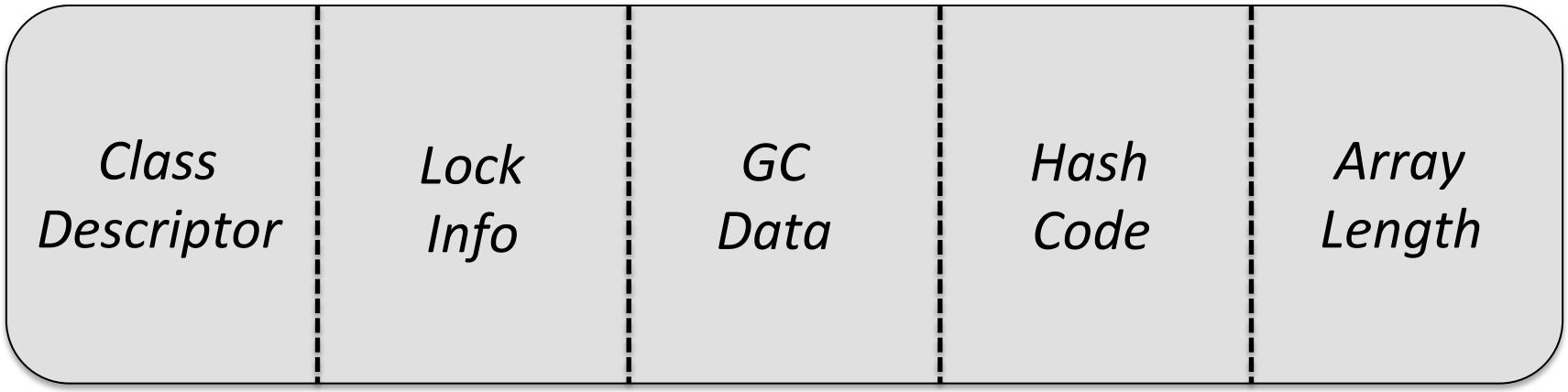
Class Descriptor

- One instance per class
- Identifies the runtime type of the object
  - Includes the class hierarchy
- Gives access to static data



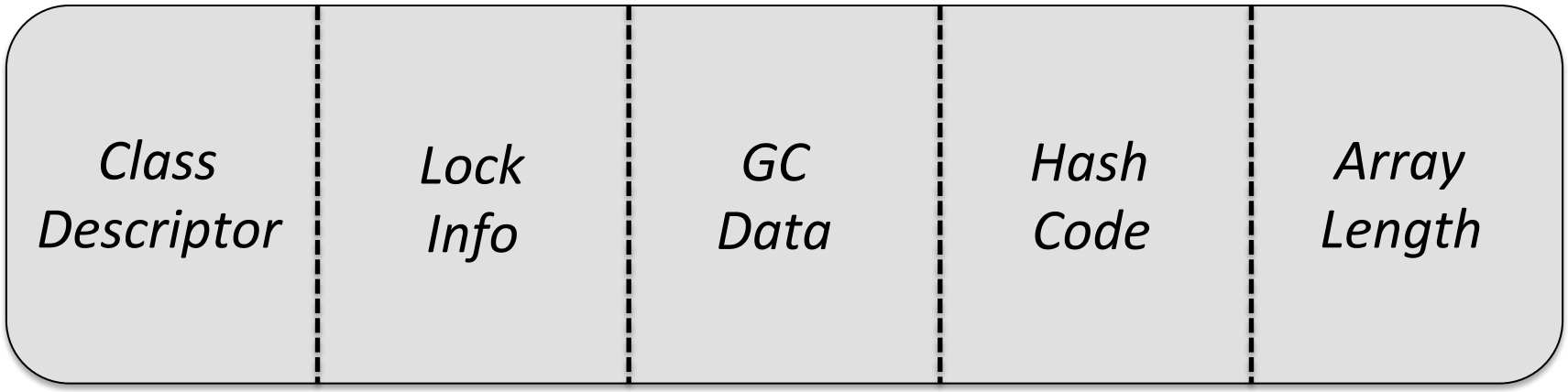
Lock Word

- Any object can be used as a monitor
- Java locks are re-entrant
  - You can take the a lock twice with the same thread
  - We'll talk more about this later in the semester



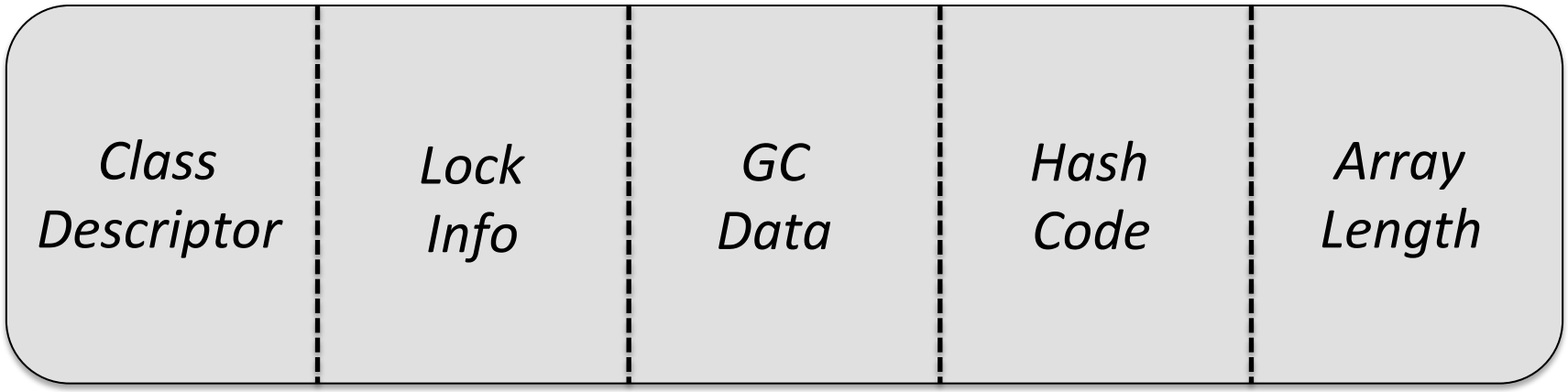
Garbage Collection Data

- Most GC algorithms need some per-object data
  - Mark bits
  - Forwarding pointers
  - We'll talk a lot more about this later



Hash Code

- Any object can be hashed
  - Used when stored in maps or sets
- Need to track hash code when assigned
  - Can't just use the object address



The SimpleJava Header

- SimpleJava has a two-word header
  - Single header structure for objects or arrays
  - No locking or hashing
- Both words contain `java.lang.Object` type
  - Word 0 is the class descriptor
  - Word 1 is the array size
- `Object` type lets us overwrite words during GC
- See `ObjectHeader` interface for more details

Header Compression

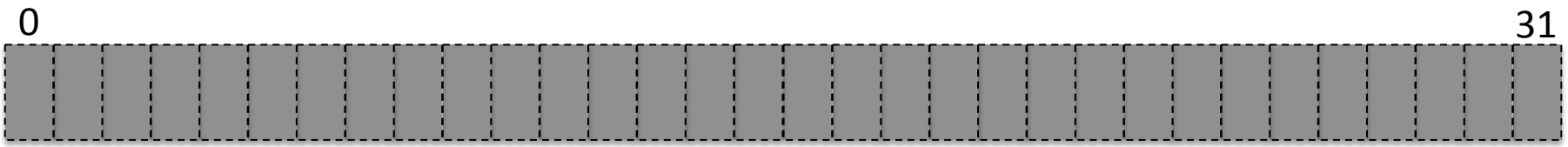
- Five words per object is excessive
  - More header than object data
- A few observations let us reduce the overhead
  - Most objects aren't locked or hashed
  - Most of the time the GC isn't running
- Objects can be aligned on byte boundaries

Addressing

- Hardware is generally word-addressed
  - 32-bit memory can address 4GB of memory
- The VM controls allocation
  - It can force objects to align on particular boundaries

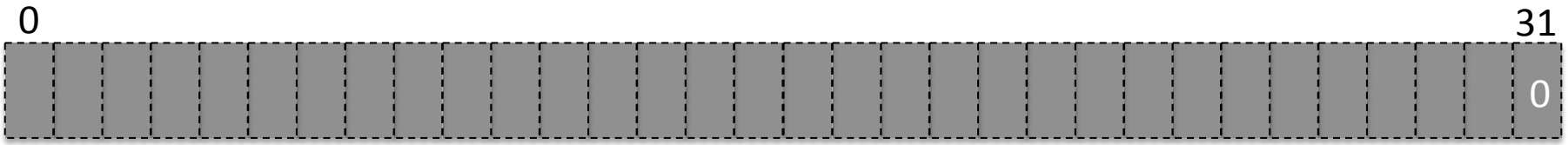
Addressing

- Word aligned
  - Requires all 32 bits to address



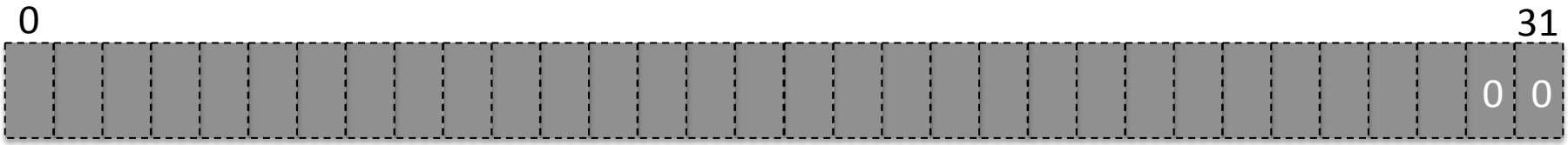
Addressing

- Word aligned
  - Requires all 32 bits to address
- Double word aligned
  - We know that the least significant bit will be a zero



Addressing

- Word aligned
  - Requires all 32 bits to address
- Double word aligned
  - We know that the least significant bit will be a zero
- Four-word aligned
  - The two least significant bits will always be zero



Bit Stealing

- We can align on any power of two value
  - Guarantees a number of bits to be zero
- Those bits can then be re-used for other things
  - Known as bit stealing
- Need to mask the stolen bits on read
  - Normally a cheap operation
  - Sometimes done in hardware

Compressed Headers

- Hotspot uses two words per object
  - Three words for arrays
  - Could go smaller, but with higher overhead
- One word is reserved for the class descriptor
  - Frequently used
- Second word is the mark word
  - Encodes all the other information

Mark Word

- Bottom two or three bits are used as a tag
  - Two bits in most cases, third for a corner case
  - Remainder of the word determined by their value
- Five separate states
  - Unlocked
  - Lightweight locked
  - Heavyweight locked
  - Marked by GC
  - Biased Locks
- We'll talk about each state later in the semester

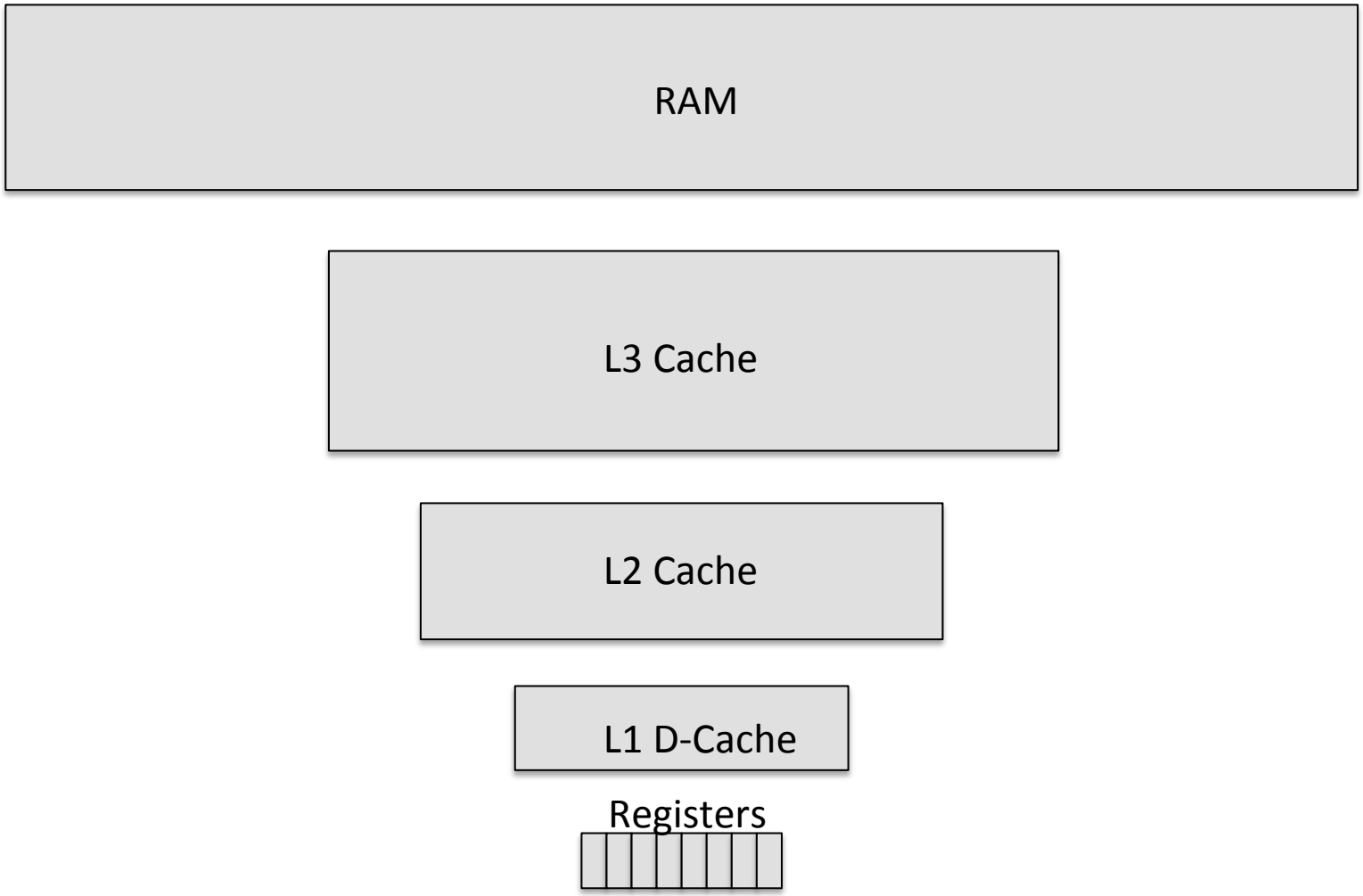
Compression Trade-offs

- We can save a lot of space
- A lot depends on the assumptions
  - We win if uncommon cases don't arise
  - If they do, we have extra memory accesses
- Code sequences can be complicated
  - Instructions have to check the header's mode

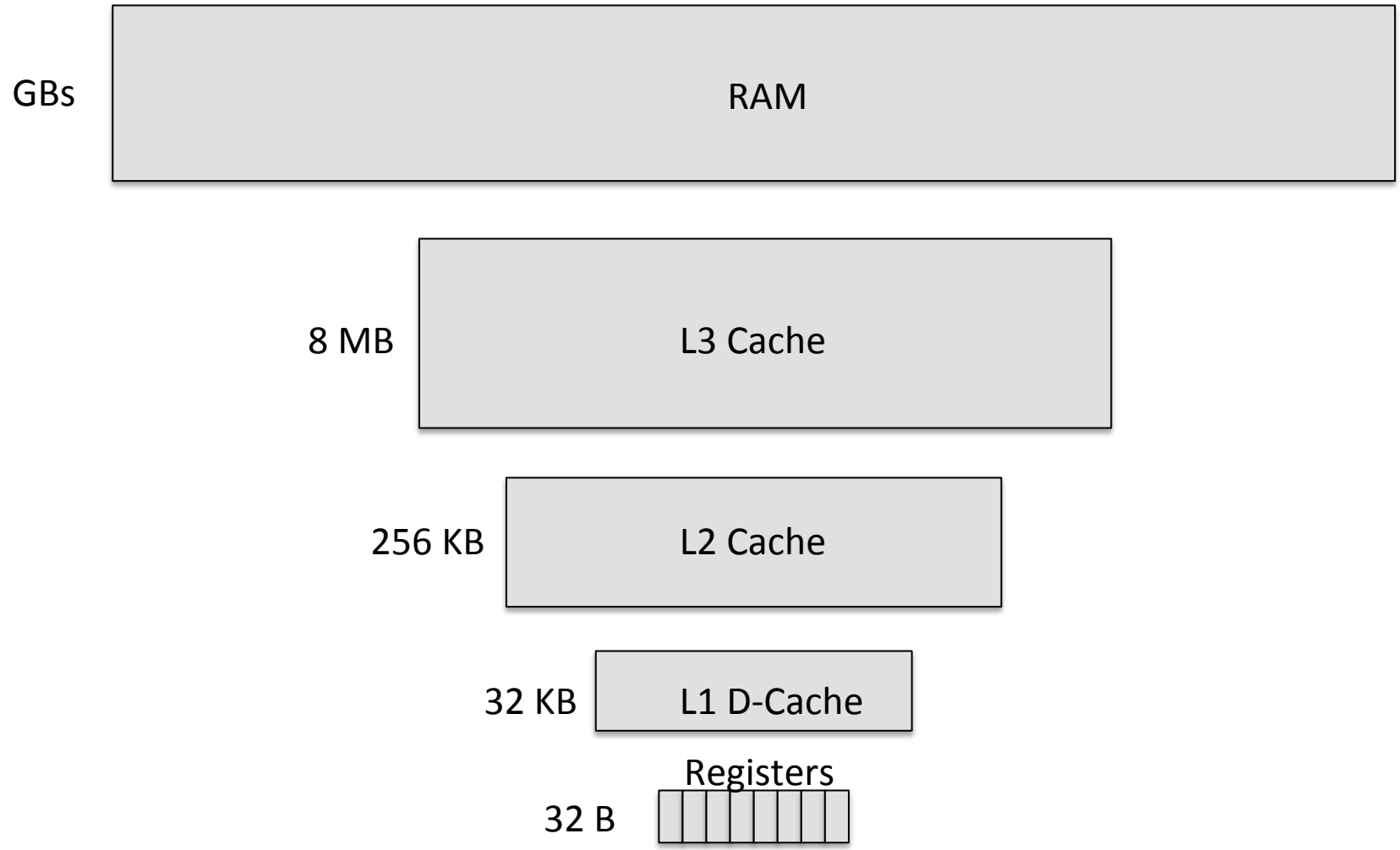
Cache Digression

- Why do we care about large headers?
  - Memory is cheap
- Larger objects require more garbage collection
  - GC can have a per-byte overhead
  - Large objects harder to fit in a fragmented heap
- Large objects take up more space in the cache

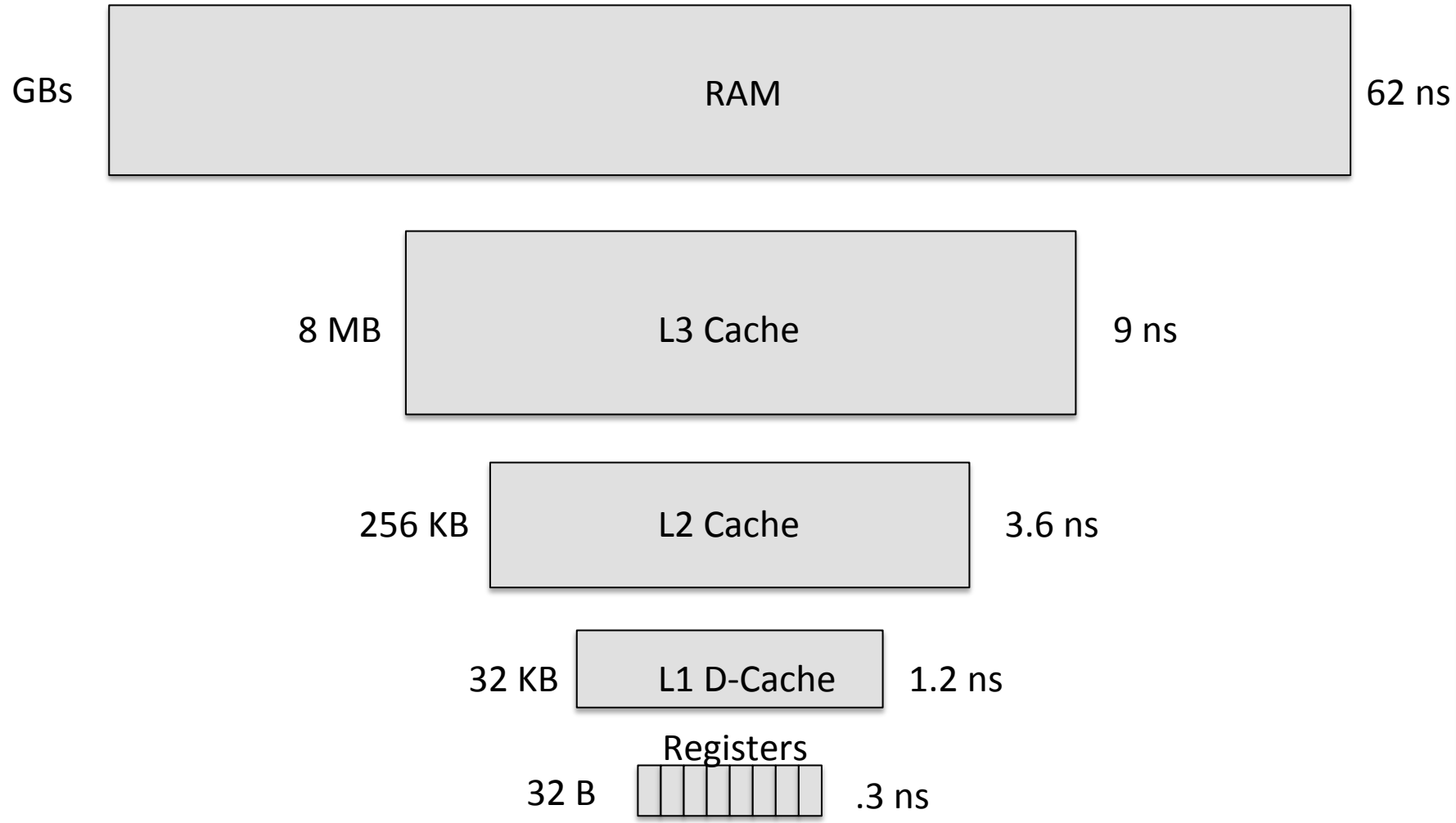
Intel I-7 Memory Hierarchy



Intel I-7 Memory Hierarchy



Intel I-7 Memory Hierarchy



Caching 101

- Caches are made up of lines
  - 64 bytes per line, in Intel Ivy Bridge L1 D-Cache
- Main memory access is by cache line
  - You don't just fetch a single object
  - Recall that objects are word-aligned
- The cache line includes header + data
  - Smaller header means more data
  - More data means fewer memory accesses

Caching 101

- Caching will be increasingly important to us
  - A big part of memory management
  - Implications on JIT compilation strategies
- We'll be covering it on-demand
- Caching is a fascinating topic
  - Its effects can be massive and counter-intuitive
  - Consider taking CSCI E-93

Object Layout

- Objects contain non-static field data
  - Values can be different for every instance
- Includes all fields defined by super classes
  - Remember that fields do not override
- Method data is not stored with the object
  - Loaded once per class

Field Layout

- Fields start at a fixed offset from the header
  - Remember that the header is a known size
- Object header can be at the end of the object
  - Fields laid out backwards
  - Pushes null pointer checks into hardware
- Same effect by guarding first few memory pages

Field Layout

- Layout strategy is implementation dependent
  - Ordered by defining class
  - Ordered by definition in the class file
  - Ordered by predicted access patterns
  - Ordered by type, keeping references together
- All have different cache characteristics