Memory Management in

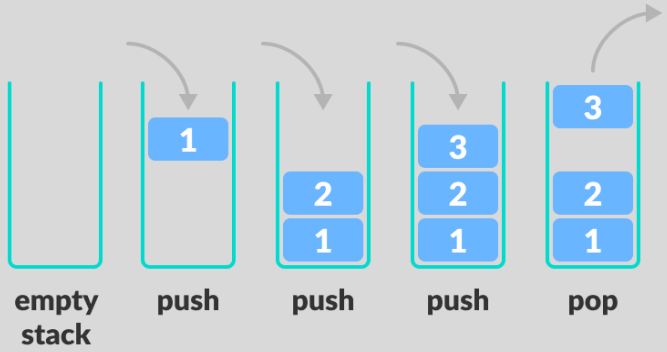
Kotlin

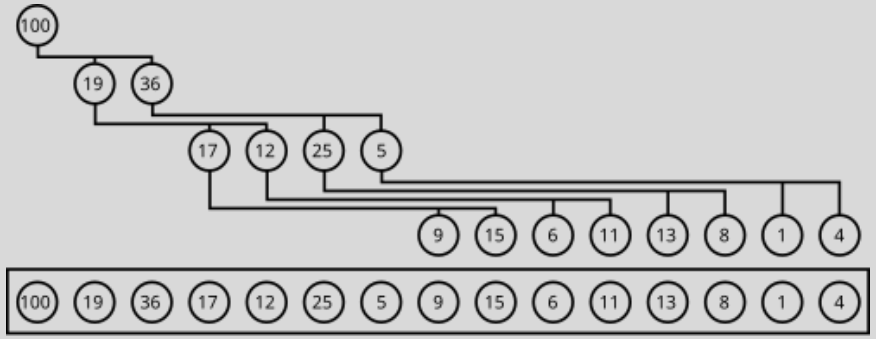
Isaac Hennessey | Daniel Abbott
Dean Maynard | Isaac Letcher

OVERVIEW

1. **Stack | Heap | Buffer**
2. Java Virtual Machine (JVM)
3. Manipulating the Stack
4. Garbage Collection / Heap
5. OS -> JVM Communication
6. IntelliJ IDEA Tools

Stack | Heap | Buffer





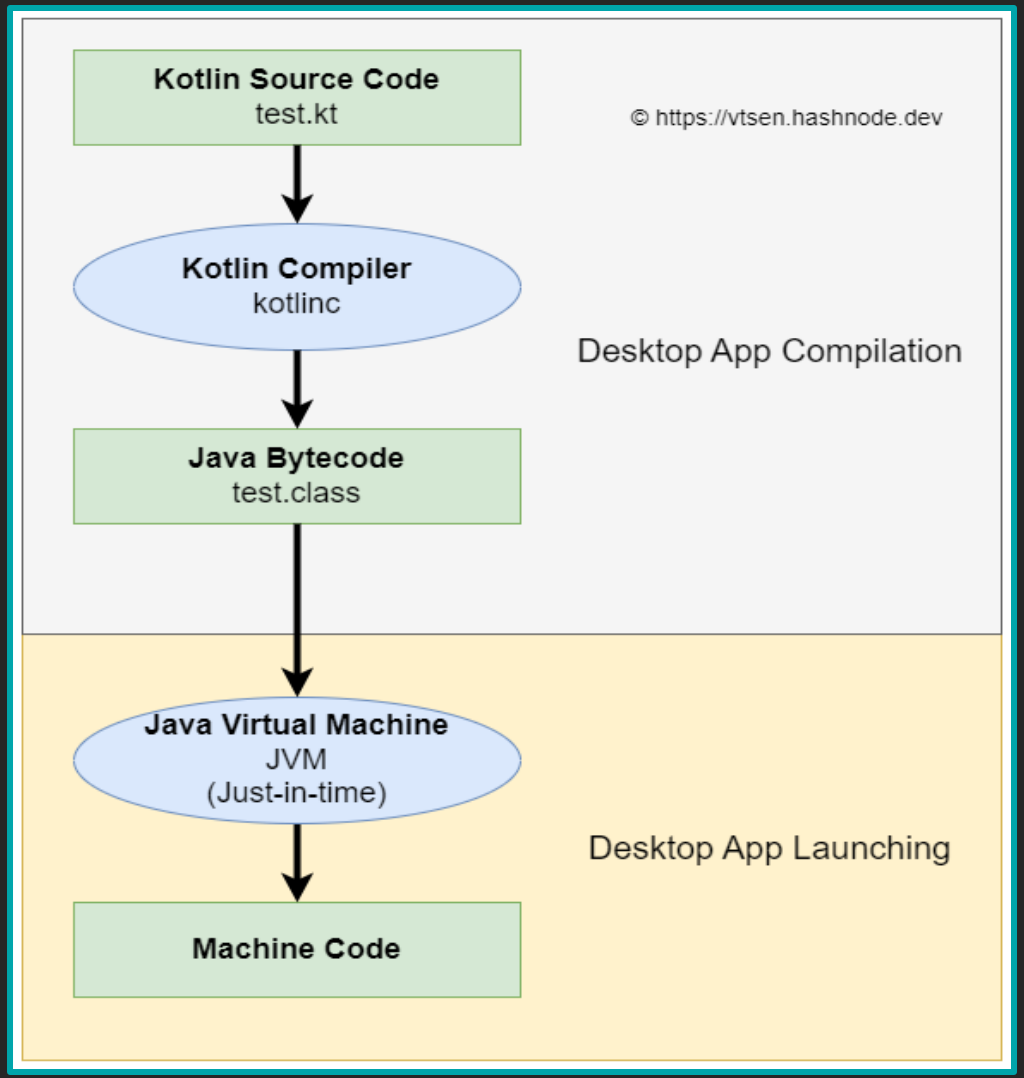
1. Stack

2. Heap

3. Buffer

Java Virtual Machine (JVM)

- The JVM divides memory into stack and heap, pc register, Metaspace, and native method stack
- Kotlin compiler transforms IR code into bytecode
- the Java virtual machine divides memory into two distinct areas, java memory and native memory



Manipulating the Stack

```
fun recursive(depth: Int, maxDepth: Int): Int {  2 Usages
    println(">>> PUSH Frame for depth $depth >>>")
    val localInt = depth // local variable so its stored on the stack
    val localString = "Depth $depth" // local variable reference . localString is a
    // reference to the Object type String . The actual string is stored on the heap .
    // so localString is stored on the stack and holds the memory address of the Object
    // which is stored on the heap . String literals are stored in the String pool on the heap

    println("Stack frame $depth") // println statement
    try {
        if (depth == maxDepth){
            return depth
        }
        // helps with stack overflow . if depth reaches
    // its maxDepth it should immediatly return . this pops the current frame off of the
    // call stack . The return tells the JVM: "I'm done—pass back any value (if not void)
    // and clean up this frame."

    return recursive(depth + 1, maxDepth) // recursively calls itself until depth == maxDepth (5)
    }
    finally{
        println("<<< POP Frame for depth $depth <<<")
    }

}
```

"main"@1 ...: RUNNING

recursive:22, MemTestKt
recursive:19, MemTestKt
recursive:19, MemTestKt
recursive:19, MemTestKt
recursive:19, MemTestKt
main:45, MemTestKt

Bytecode Example

```
    if (depth == maxDepth){
        return depth
    }
return recursive(depth + 1, maxDepth)
```

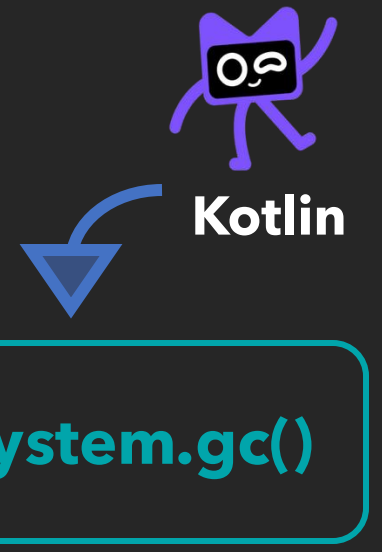
```
0: iload_1                // load 'depth' from local var #1
 1: iload_2                // load 'maxDepth' from local var #2
 2: if_icmpne 12           // if depth != maxDepth → jump to 12
 5: iload_1                // load 'depth'
 6: ireturn                // return it

12: iload_1                // load 'depth'
13: iconst_1               // push constant 1
14: iadd                   // add: depth + 1
15: iload_2                // load 'maxDepth'
16: invokevirtual #7       // call: recursive(depth+1, maxDepth)
19: ireturn                // return the result
```

Garbage Collection

Systematically goes through the heap

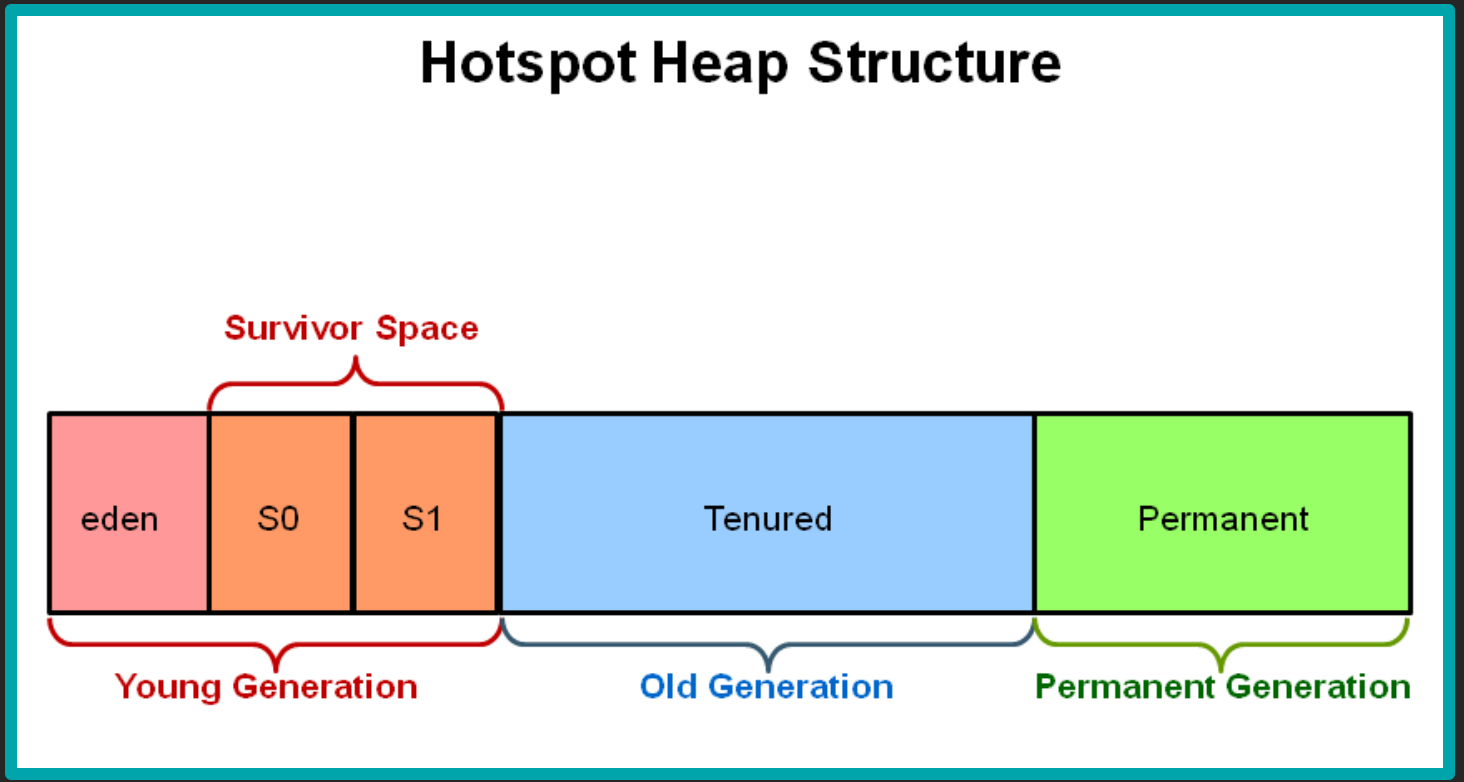
Looks for objects with references and keeps them

Deletes (allocates memory as free) unreferenced objects



Observing the Heap

- Anything stored in the heap needs a reference to prevent Garbage Collection
- When an object survives a Minor GC its age increases
- Major GC happens in the old generation not as often as Minor GC




```
println("Heap Demo:") // header
repeat( times = 100) { addBox( id = it + 1,  size = 1024 * 1024) } // repeat 3 times , add a new box
// with an id and a size for the payload . each allocates 1MB of memory onto the heap
```

Managing the Buffer

```
fun byteBufferExample() {  1 Usage
    // A ByteBuffer manages a memory buffer internally
    val buffer = ByteBuffer.allocate( capacity = 10000)

    // Write to buffer (moves position forward automatically)
    for (i in 0 ≤ .. ≤ 9999) {
        buffer.put( b = i.toByte())
    }
    // Prepare buffer for reading
    buffer.flip()

    // Read data in order
    while (buffer.hasRemaining()) {
        println(buffer.get())
    }
```

- **Cannot access the actual memory buffer**
- **ByteBuffer class manages memory buffer**
- **Provides a safe and secure environment to manipulate the memory buffer**

OS -> JVM communication

```
val payload = ByteArray(size)
```

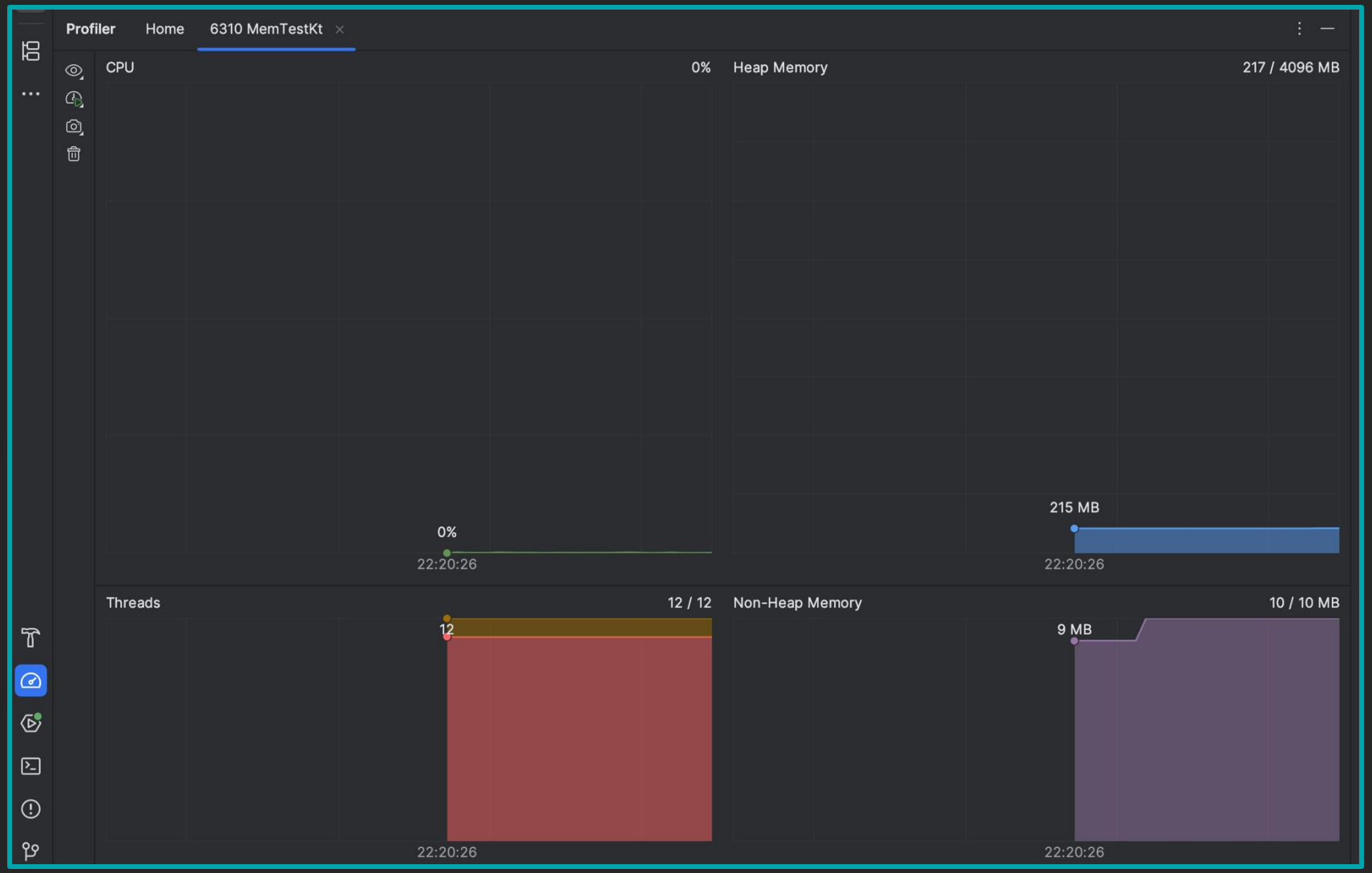
```
println("Heap Demo:") // header
repeat( times = 100) { addBox( id = it + 1,  size = 1024 * 1024) }
```

- The JVM asks the OS for 1mb of virtual memory
- The OS doesn't give it real RAM quite yet
- The JVM fills in the array with 0's (default)
- The OS sees the first write and a Page Fault happens which gives the JVM real RAM

- Default page size for MacOS is 4kb (4096 bytes)
- payload = (1024 * 1024) = 1,048,576 bytes
- Payload / page size = 256 pages per array
- Total pages = 25,600 pages

IntelliJ IDEA Tools

IntelliJ Profiler



RECAP

1. **Stack | Heap | Buffer**
2. Java Virtual Machine (JVM)
3. Manipulating the Stack
4. Garbage Collection / Heap
5. OS -> JVM Communication
6. IntelliJ IDEA Tools

Questions?

Isaac Hennessey | Daniel Abbott
Dean Maynard | Isaac Letcher