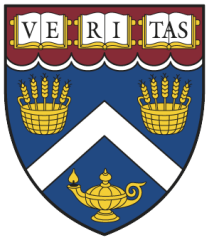
Over the Last Week

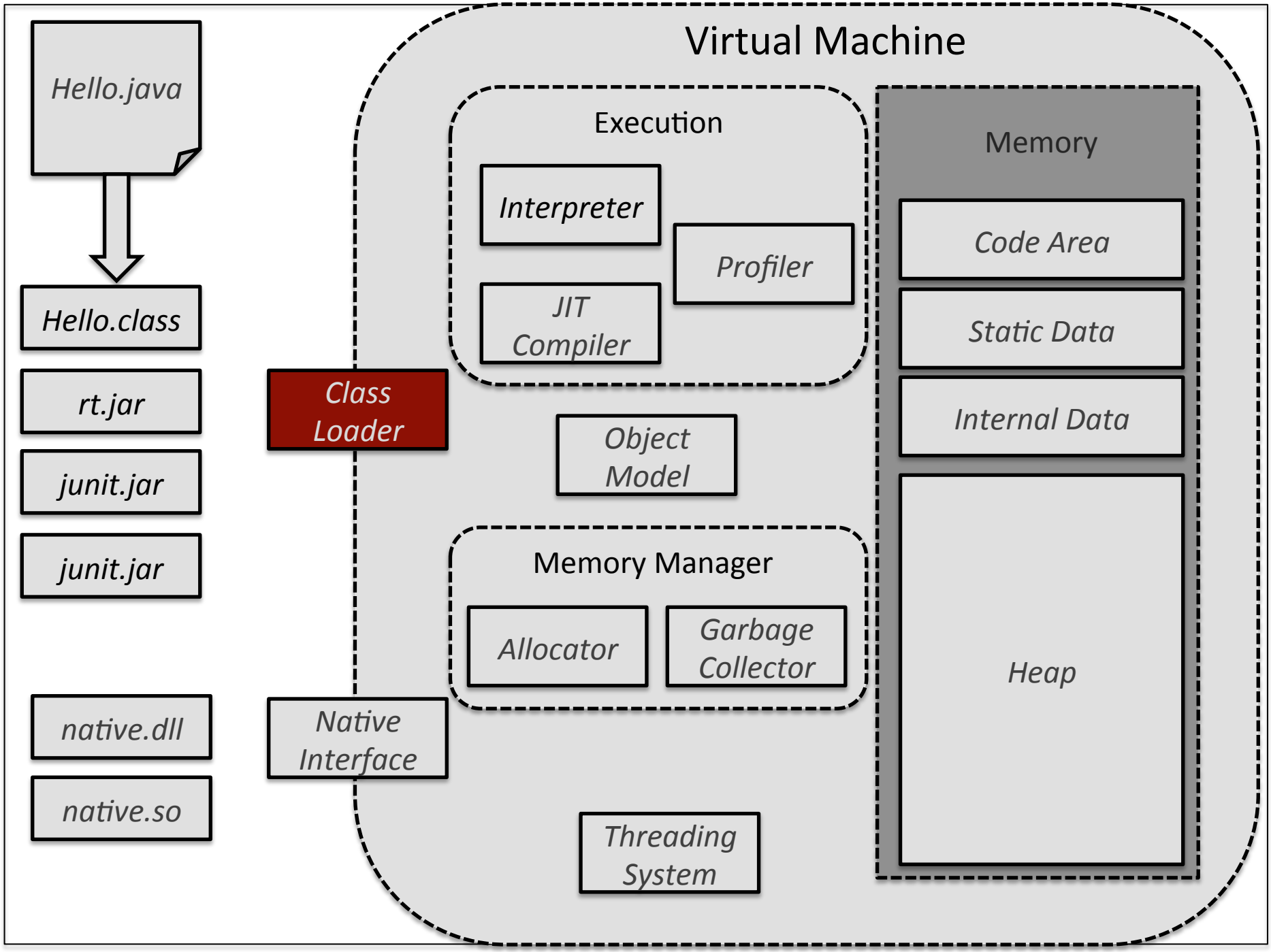
- A little confusion over calling conventions
  - Pass by value vs. pass by reference
- Good discussion on the forums
  - Magic numbers
  - Running languages other than Java on the JVM

Assignment 1 Now Available

- Due two weeks from yesterday
- Some of you have started already
  - I'd suggest that you don't leave it to the last minute
- Let me know of any problems with the code
- Remember that grading will be done by script
  - Tests will be more thorough than the sample code

Class Loading




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

Class Loading

- Java doesn't require the whole program up front
- New types can be added at any point
  - Allows flexibility
  - Makes optimization harder
- Dynamic class loading enables this

Dynamic Class Loading

- Very powerful mechanism
  - Fetch new classes during execution
  - Generate new classes on the fly
  - Modify existing code (aspects)
  - Run multiple applications on one VM instance
  - Pass implementations between VMs (RMI)

Java Class Loading

- Only objects are loaded
  - Primitives built into the VM
  - Arrays are extensions of objects
- Three phases
  - Loading
  - Linking
  - Initializing
- Eagerness is implementation specific
  - Normally lazy

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

Which Class Loader?

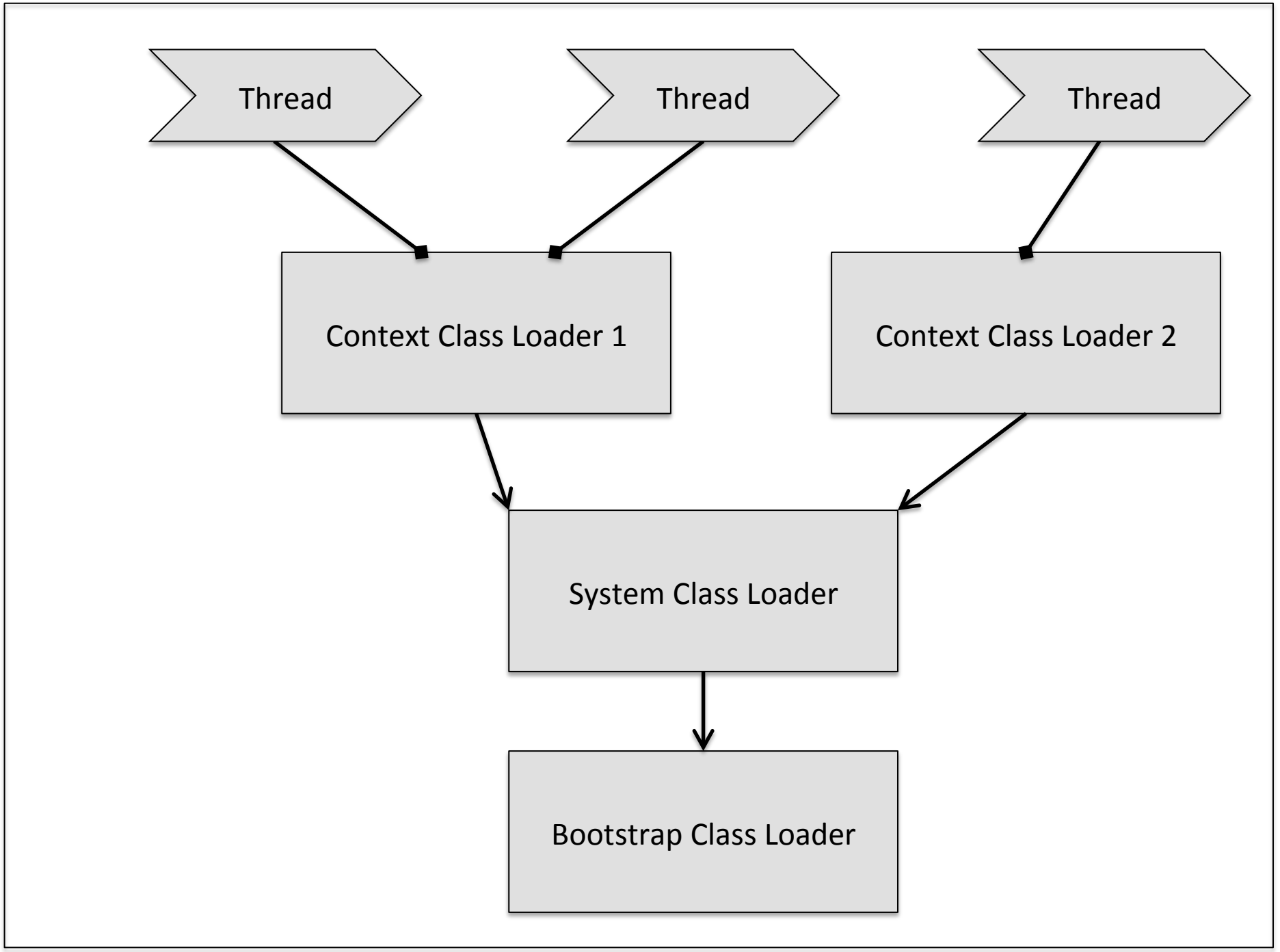
- The VM doesn't have a single Class Loader
  - Always at least two
  - Applications can define their own
- Objects hold a reference to their class loader
  - `myObject.getClass().getClassLoader()`
  - `ClassA.class.getClassLoader`
- By default, a class uses its thread's class loader

Core Libraries and User Classes

- VM must know about some implementations
  - `java.lang.Object`
  - `java.lang.String`
- Core and application classes treated differently
  - Based on the package name
- User-defined loaders can't load core classes

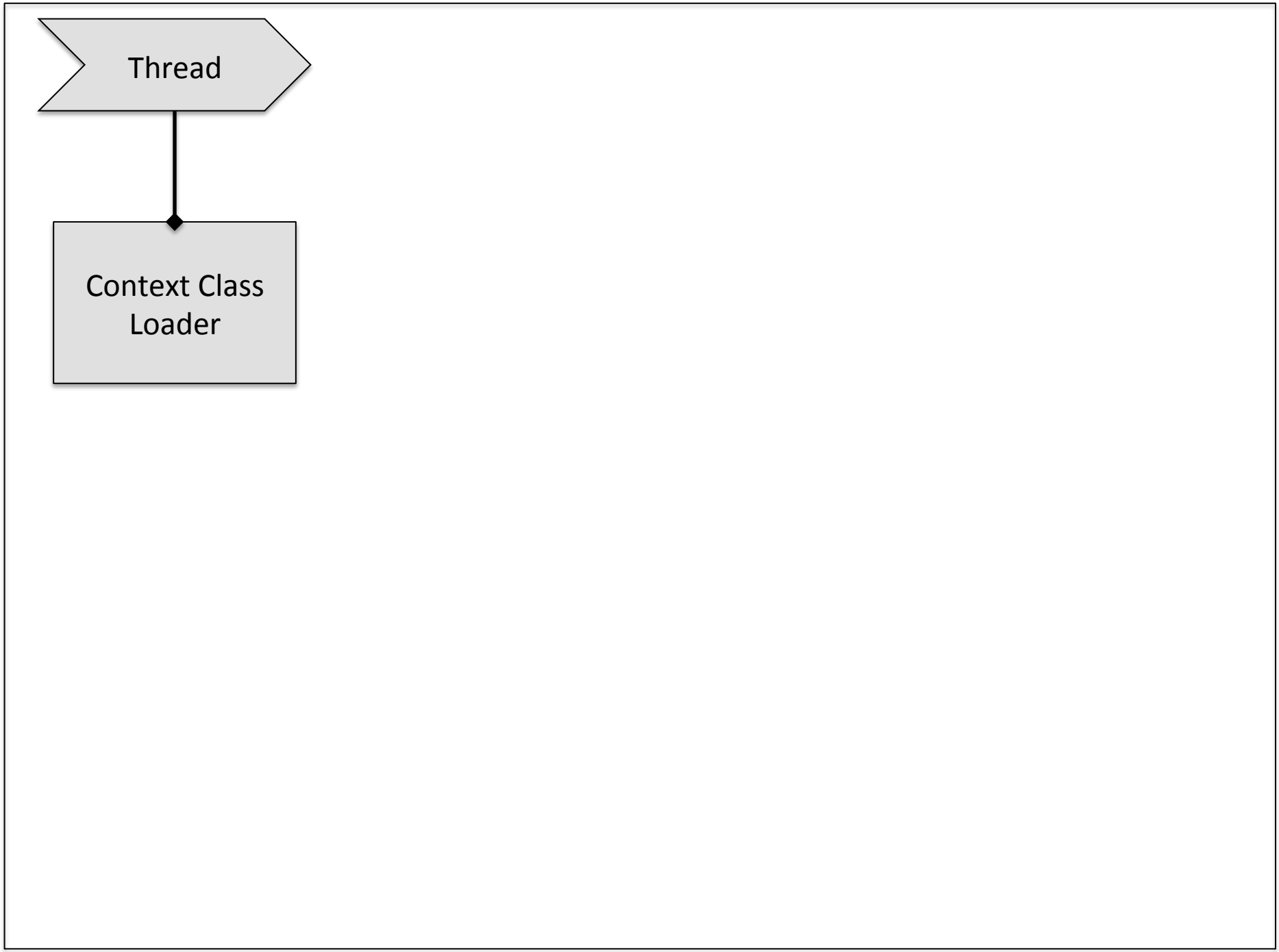
Three Types of Class Loader

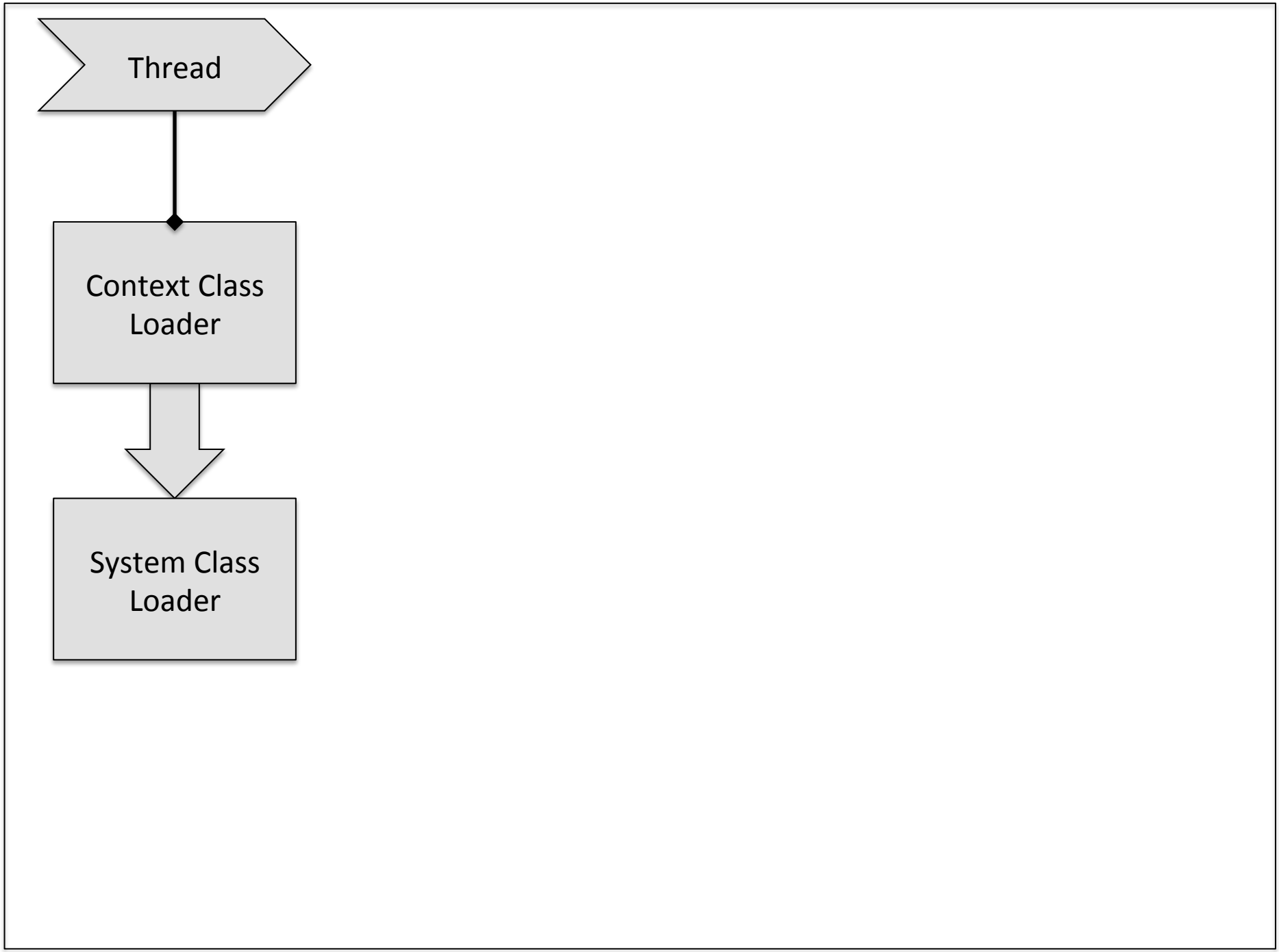
- Bootstrap class loader
  - Loads core library code
- System class loader
  - Loads application code
  - Can be overridden
- Context class loader
  - User defined
  - Set at the thread level

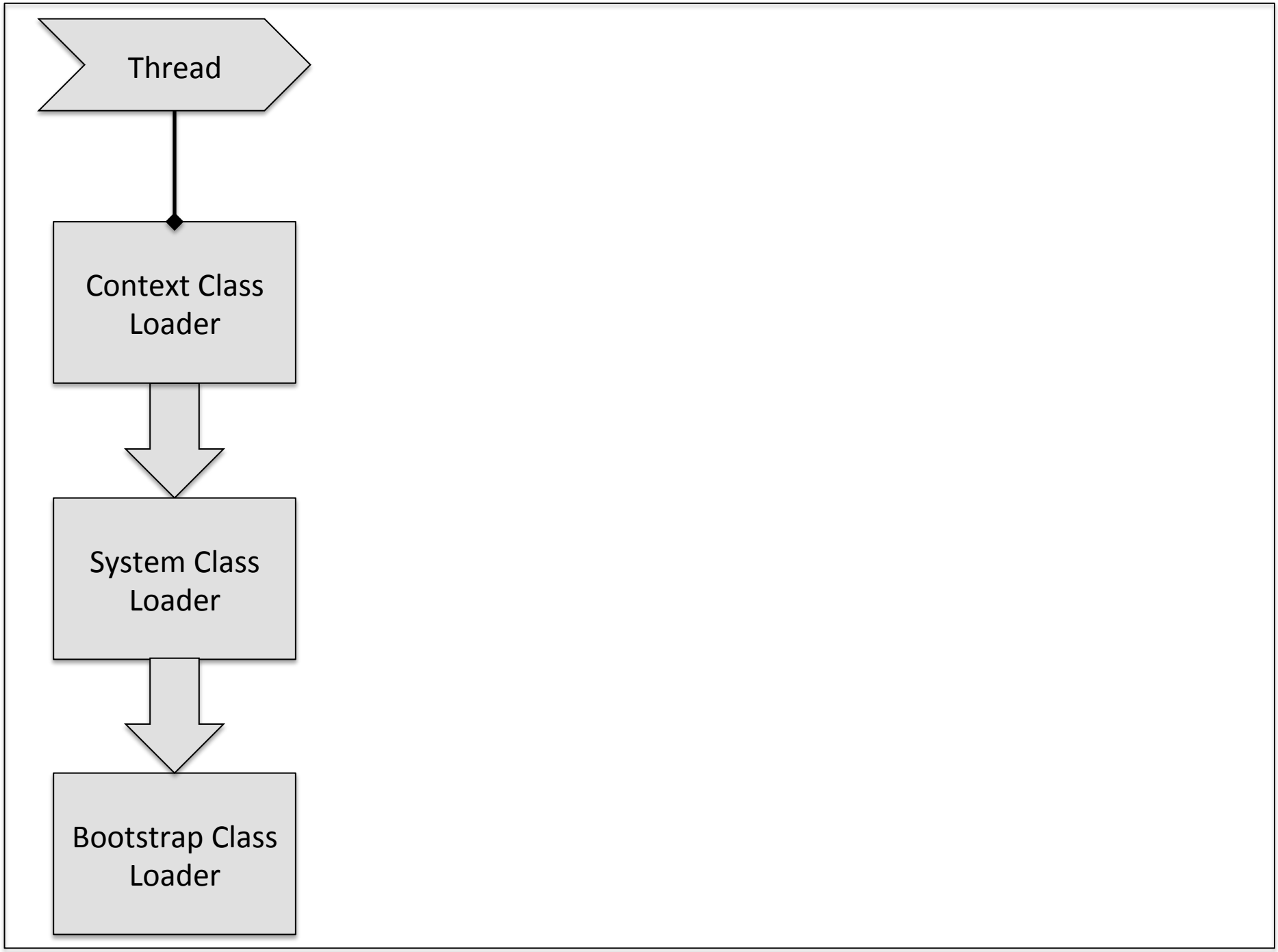

Thread
Thread
Thread
Context Class Loader 1
Context Class Loader 2
System Class Loader
Bootstrap Class Loader

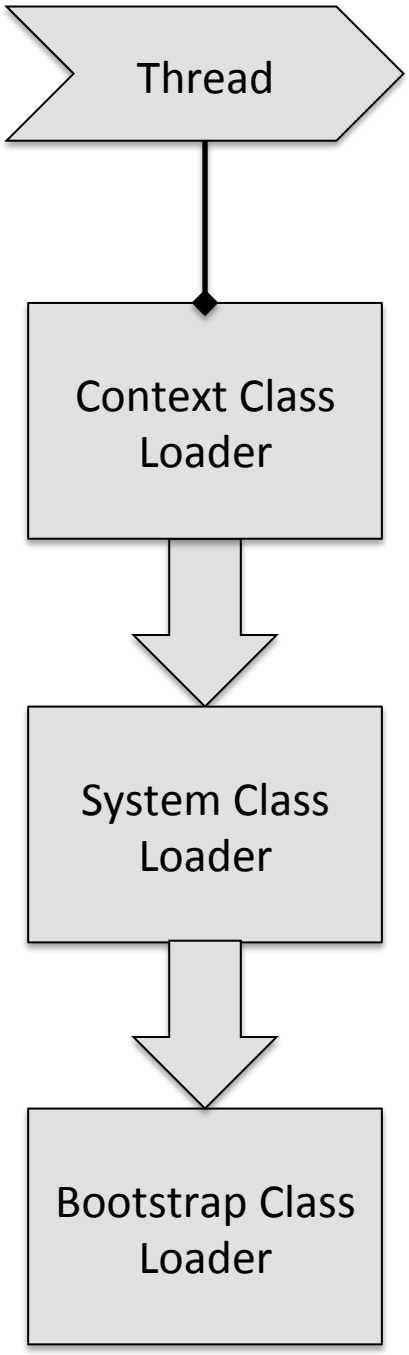
Delegation Model

- Class loaders form a hierarchy
  - Each class loader has a parent
- A load request first goes to the context loader
- A class loader should delegate to its parent
  - This works recursively
  - Recommended only, not required
- Look for the class if the parent doesn't find it

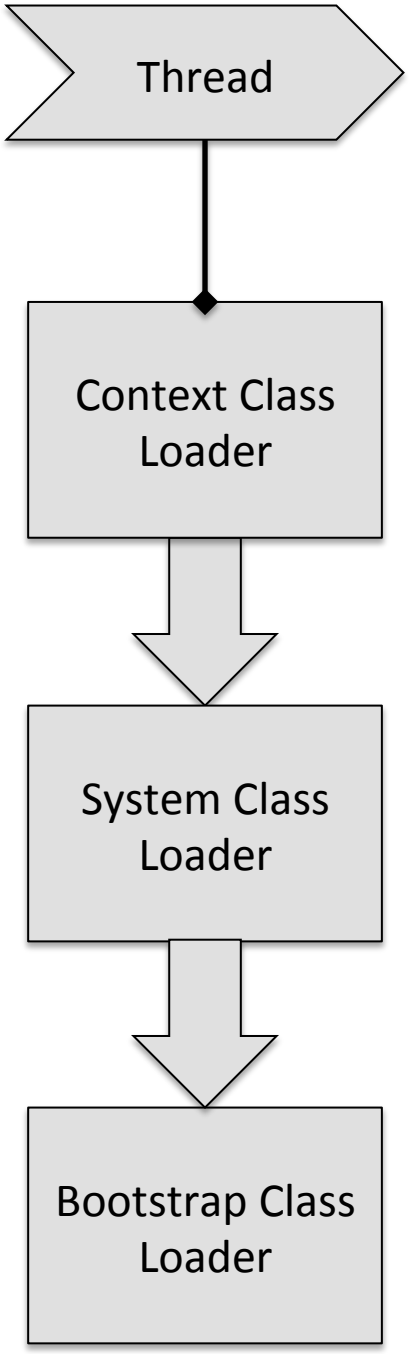

Thread
Context Class Loader


Thread
Context Class Loader
System Class Loader


Thread
Context Class Loader
System Class Loader
Bootstrap Class Loader

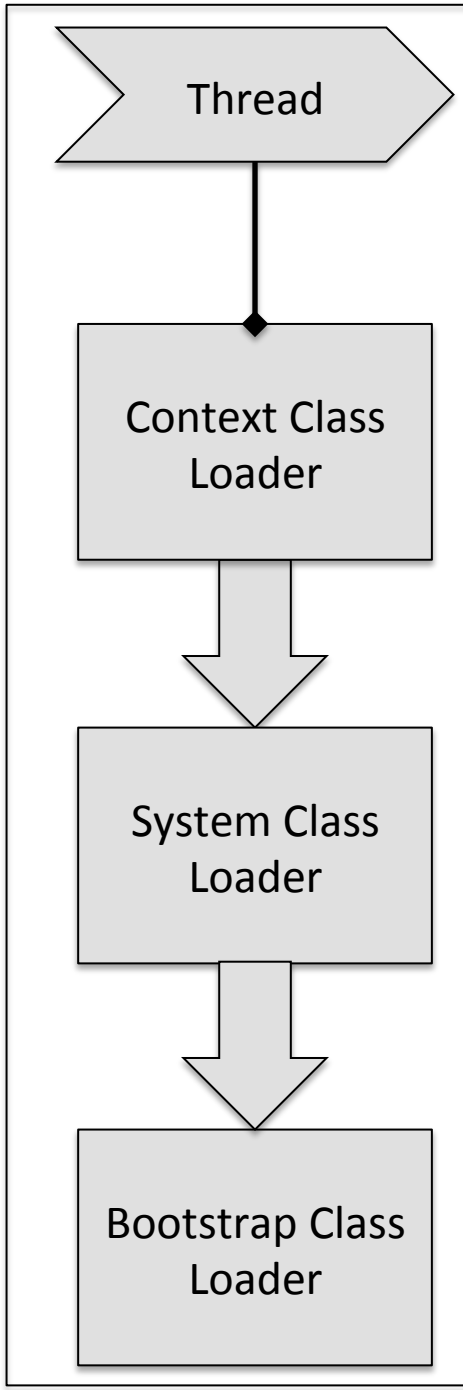


Looks in system jar files



Searches the class path

Looks in system jar files



Custom search mechanism

Searches the class path

Looks in system jar files

Class Loading Steps

- Load
  - Get the bytes
- Link
  - Verify the class
- Initialize
  - Set up data structures and statics

Loading

- Check if the class has been loaded already
  - Return the same Class instance
- Find the `ClassInfo` structure
  - Read from file, network, generate
- Check compatibility
  - Parse the structure
  - Check class version
  - Basic file layout checks

Linking the class

- Load superclass and interfaces
- Verify the class file
- Set up internal data structures
  - Method table
  - Static data
  - Runtime constant pool
  - Internal representation is implementation specific
- Loading may stop at this step
  - Lazily loading super class or interface

Verification

- The VM doesn't trust a class without verification
- Several possible sources of errors
  - Compiler bugs
  - Corrupted files
  - Generated code
  - Malicious code
  - Version skew

Explicit Verification

- Two options for verification
  - On-demand: verify bytecode when executed
  - Ahead-of-time: verify when class is loaded
- AOT verification lets the interpreter run faster
  - But makes class loading slower
  - One-time vs. ongoing costs

Class-Level Verification Checks

- Check that the ClassFile has a valid structure
  - Must start with 0xCAFEBABE
  - Must have a valid version number
  - Attributes must have recognizable format
- Enforce final semantics
- Check super class hierarchy
- Check structure of the constant pool
  - Recall relationships between some constant types
  - Check that all descriptors are well-formatted

Method-Level Verification

- Type checking by simulated execution
  - Track the possible stack states at every bytecode
  - Must follow all possible code paths
- Use the information to verify bytecode types
  - Opcodes must operate over the correct types
  - Loads and stores must be consistent
  - Must account for Category 1 or 2 variables
  - Method parameters
  - Field accesses

Method-Level Verification

- Check that jump targets are valid
- Access controls are respected
- Fields and methods have consistent signatures
- Exception handler blocks are valid
- Many more…
  - See the JVM Spec Section 4.10 if you're interested

Internal Data Structures

- Class object
  - Contains class metadata
  - Can be accessed using reflection
  - Can be stored on the heap
- Static variables
- Runtime constant pool
- Code area
- Mappings for optimization

Interning

- Some immutable data stored by the VM
  - Strings
  - Boxed primitives
- Creates a single copy of a given value
  - Saves on space
  - Faster string comparison algorithms
    - Use identity rather than equality

Method Storage

- Methods code located in the code space
  - Some bytecode, some compiled
- Only one copy of a method is loaded
- Internal mappings speed up method lookup
  - Very common operation
  - Optimize for dynamic dispatch

Initialization

- Preparation phase creates static fields
  - Part of the linking process
  - Sets their default values, not final values
  - Must happen before initialization
- Execute the class initialization method
  - Automatically generated if static values are present
  - Can be manually specified using static{}

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

```
  #15 = Methodref    #16.#18
  #16 = Class        #17
  #17 = Utf8         edu/harvard/cscie98/sample_code/ClassB
  #18 = NameAndType  #19:#20
  #19 = Utf8         methodB
  #20 = Utf8         (I)I
```

```
class ClassA {
  static void methodA() {
    int arg = 3;
    ClassB.methodB(arg);
  }
}
```

```
0: iconst_3
1: istore_1
2: iconst_3
3: invokestatic  #15
6: pop
7: return
```

```
class ClassB {
  static final int CONSTANT;

  static {
    CONSTANT = 2;
  }

  static int methodB(int arg) {
    return arg + CONSTANT;
  }
}
```

```
0: iconst_2
1: putstatic #10 // Field CONSTANT:I
4: return

0: iload_0
1: getstatic #10 // Field CONSTANT:I
4: iadd
5: ireturn
```

Class Unloading

- Uncommon for classes to be unloaded
  - Only if the class loader is garbage collected
  - Otherwise a class could be re-used
    - Static data must be preserved for the life of the VM
- Many data structures stored on the heap
  - Use the GC to clean up on unload
  - Potential overhead depending on GC algorithm