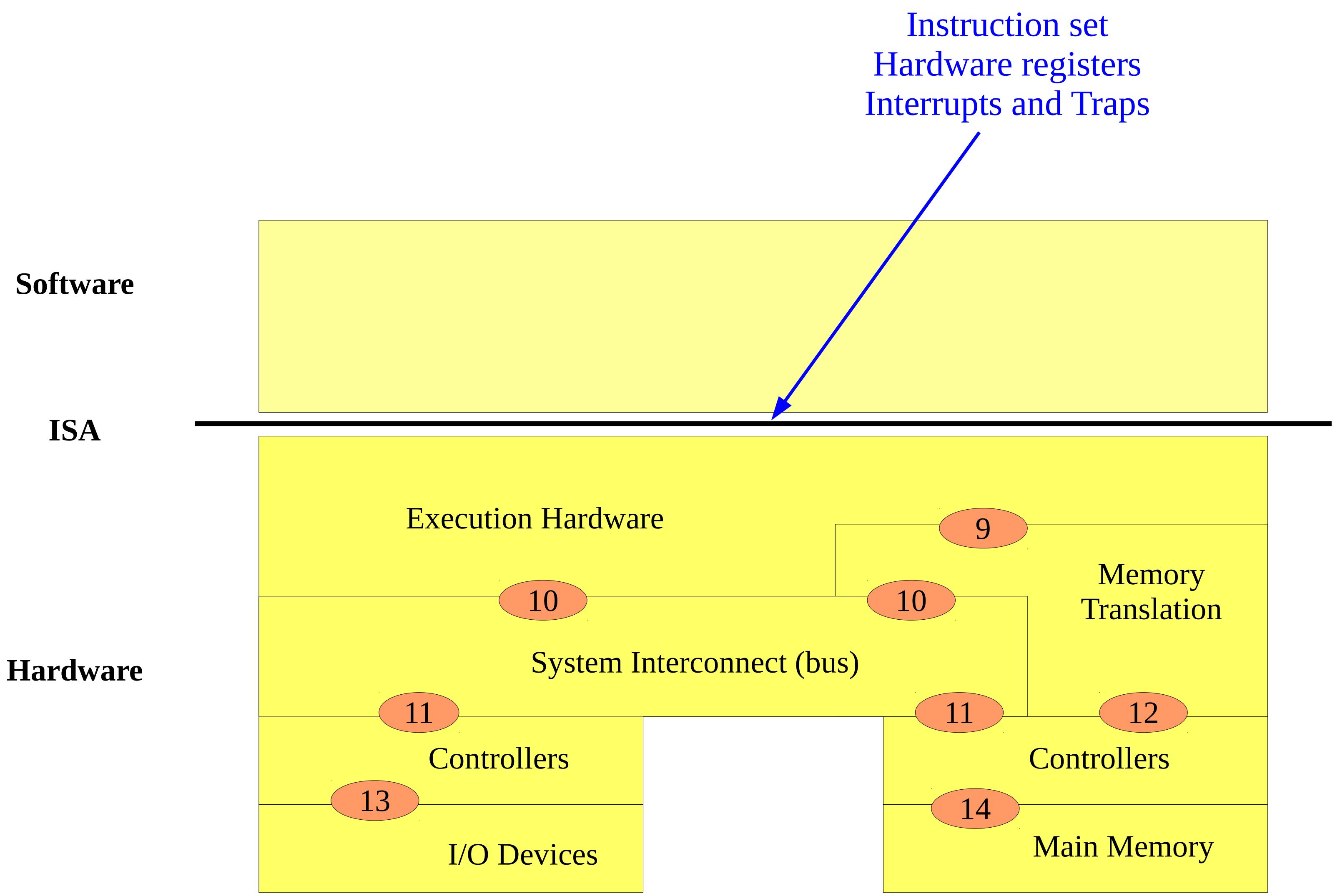
Virtual Machines
For
High-Level Programming Languages

Pr. Olivier Gruber

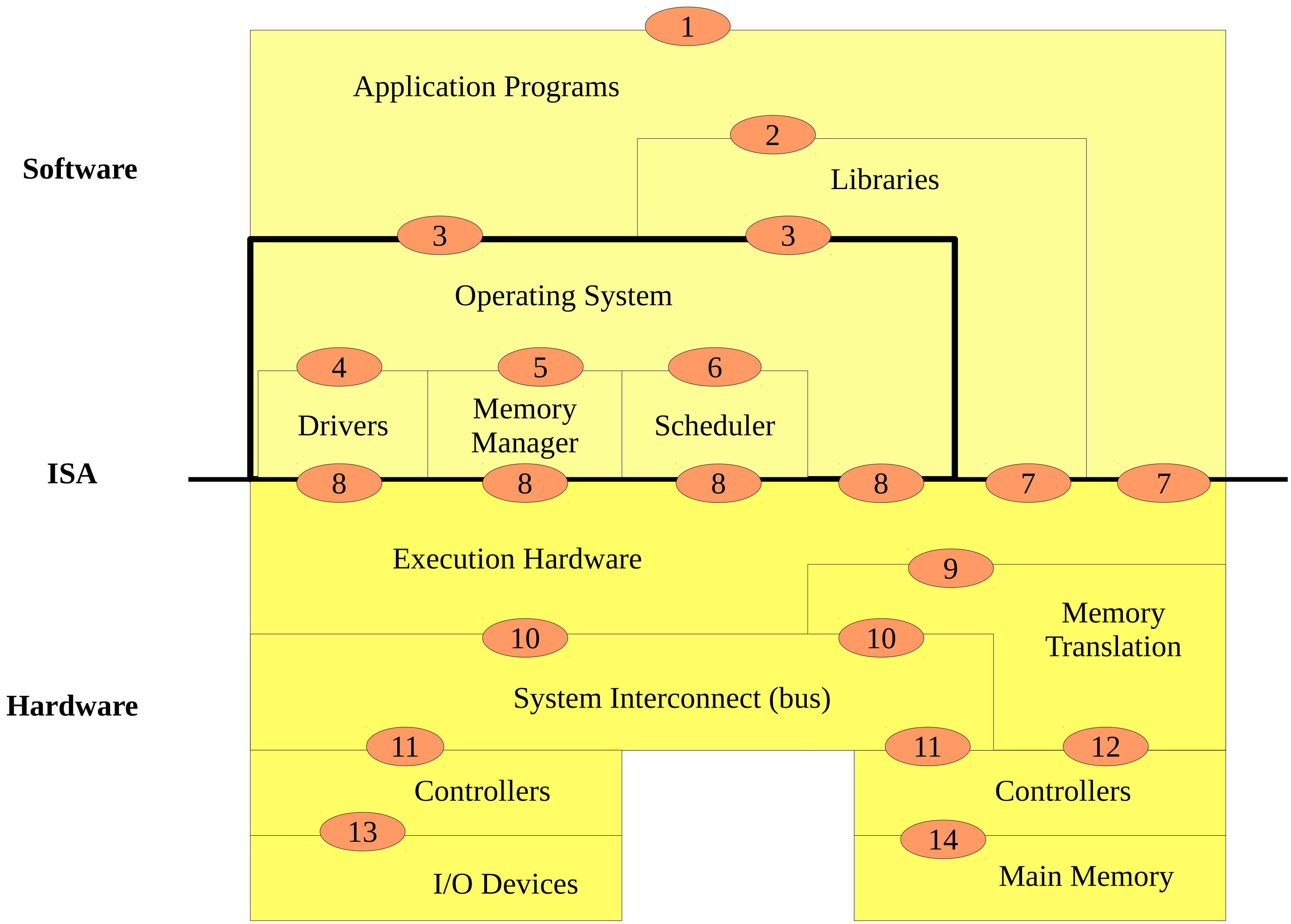
Université of Grenoble

Olivier.Gruber@imag.fr

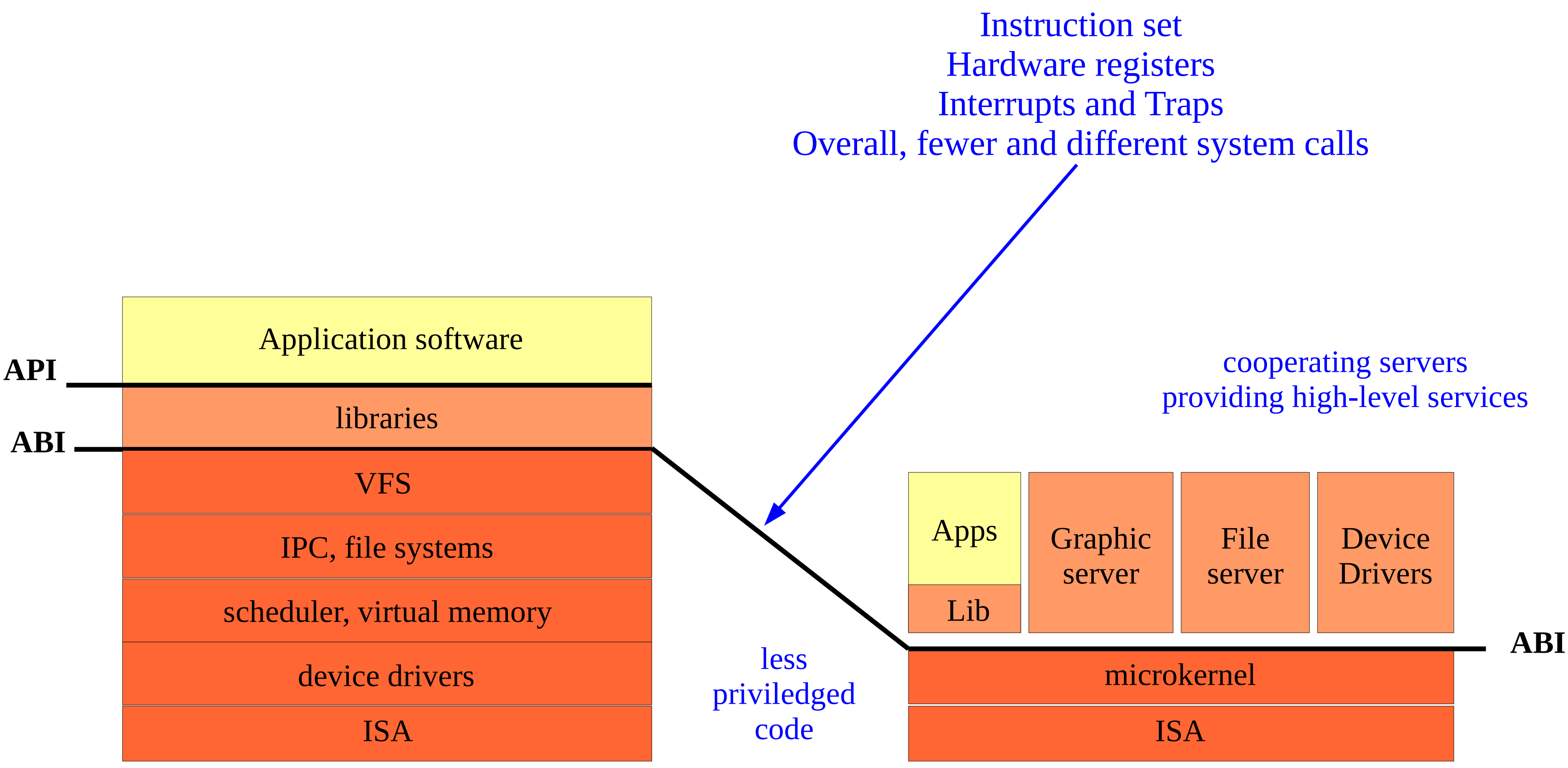
Stepping Back...



Stepping Back...

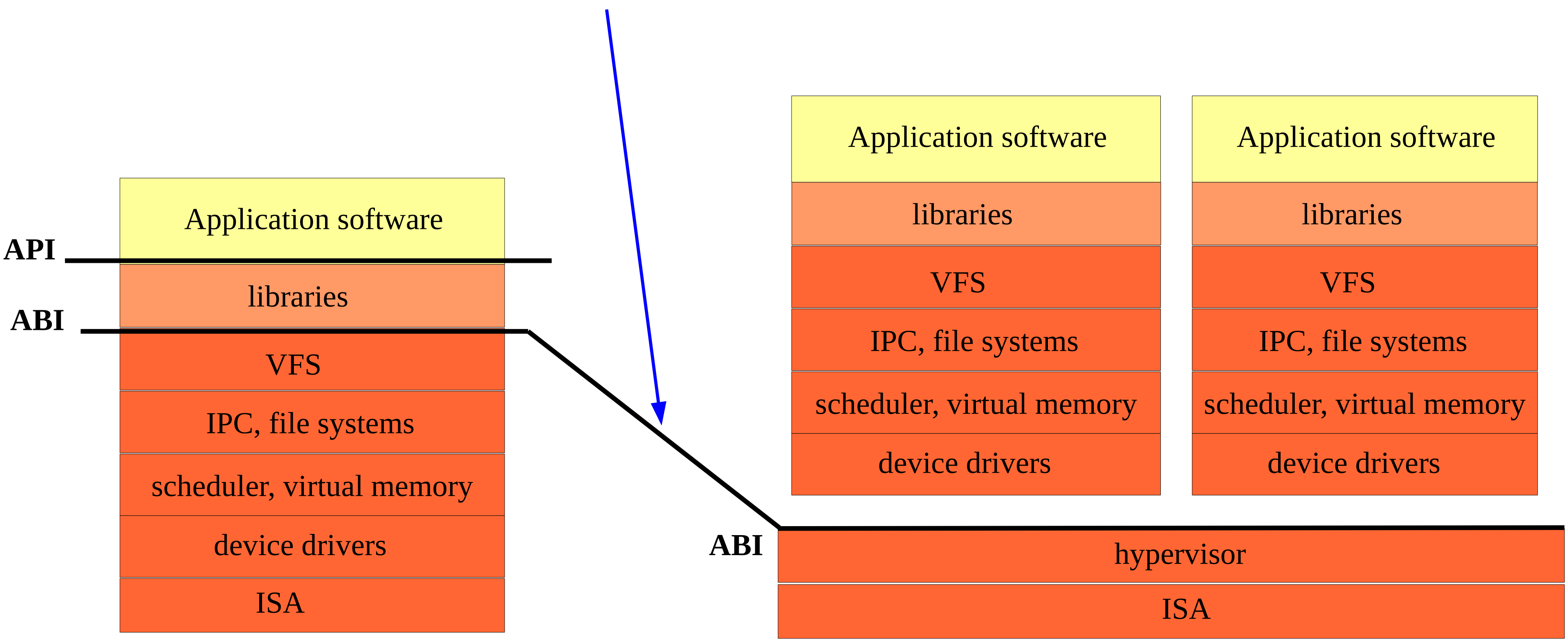


Stepping Back...



Stepping Back...

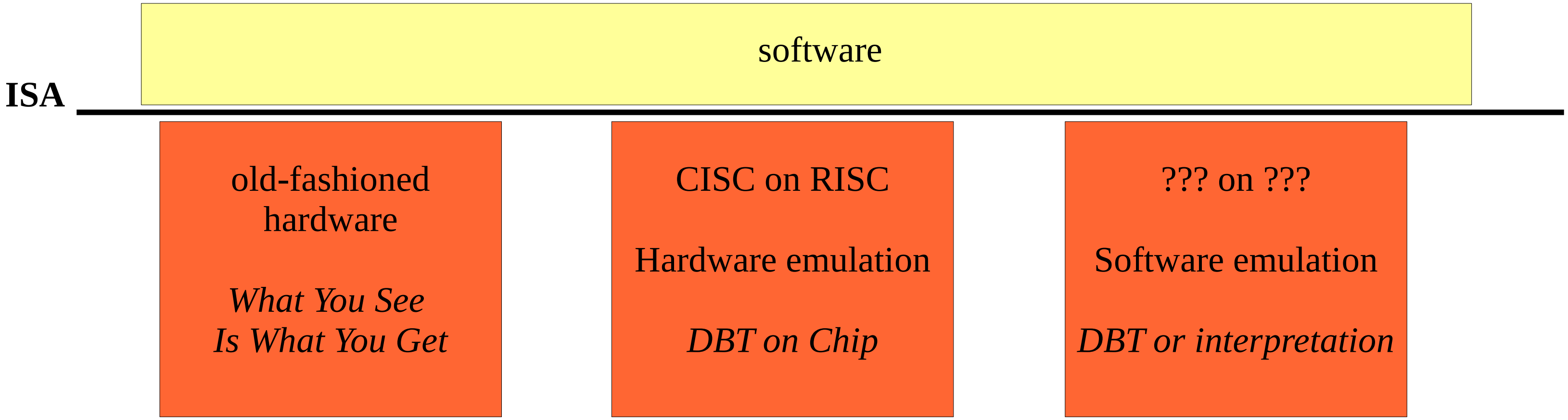
Instruction set
Hardware registers, Interrupts and Traps
Overall, fewer and different system calls with para-virtualization



Stepping Back

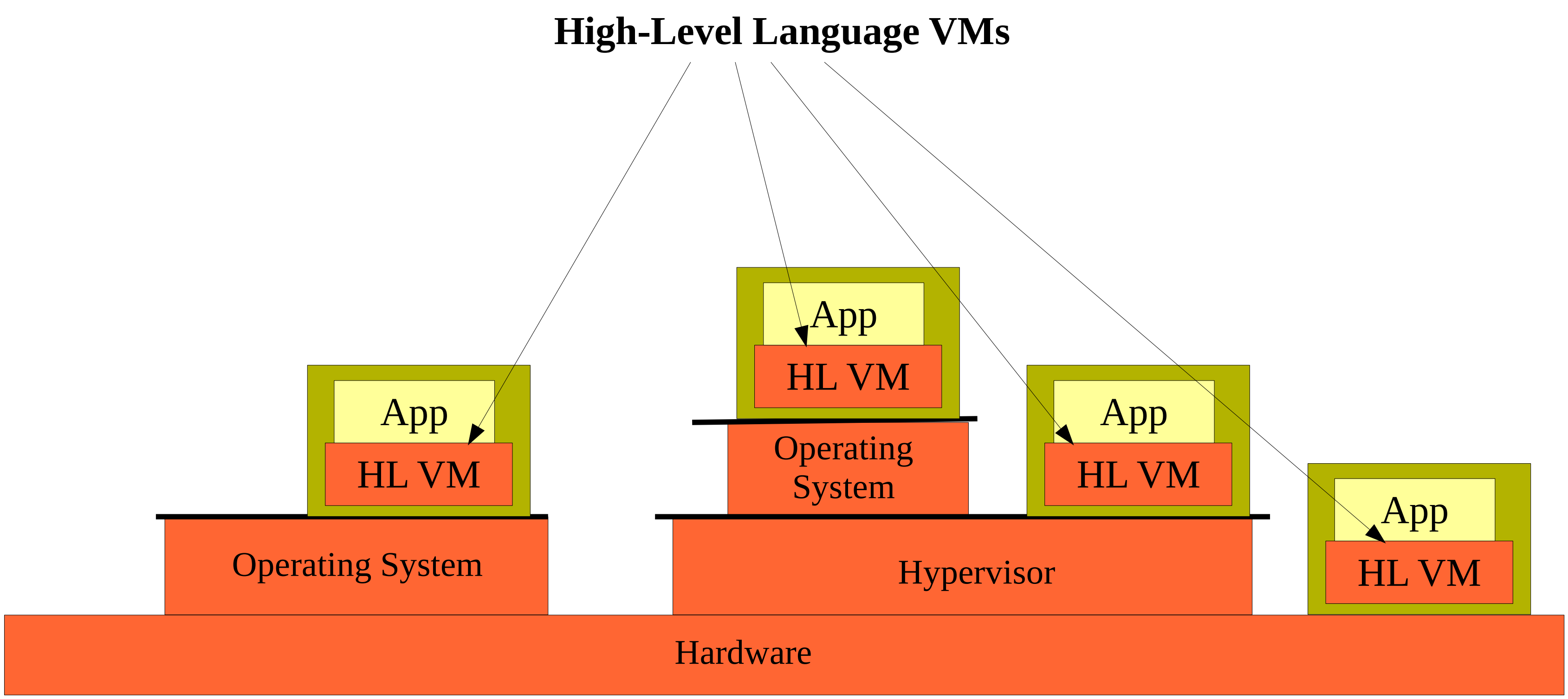
So, what about that instruction set?

Does it have to be an assembly language of a "real machine"?

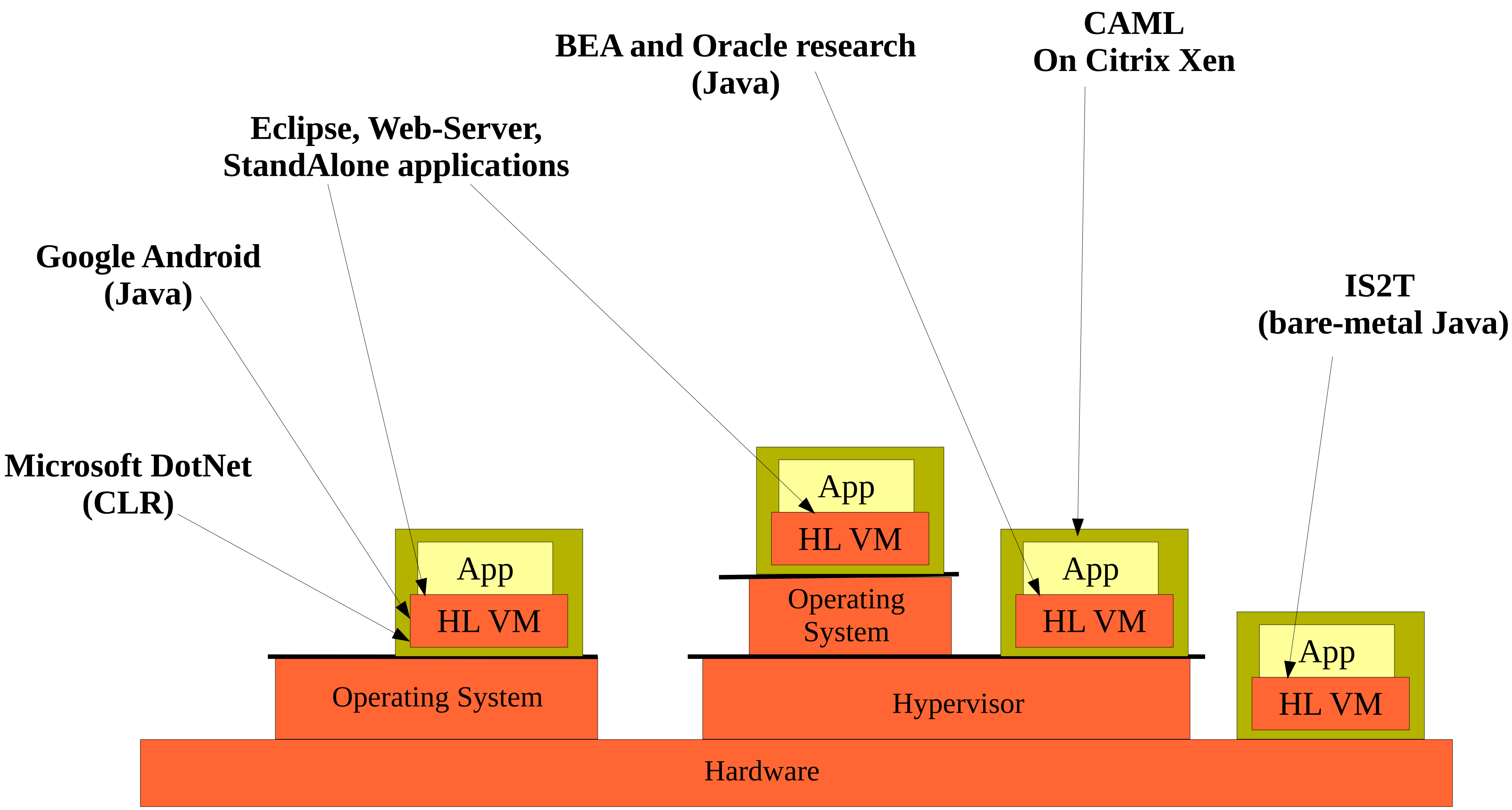


Global View

- High-Level Language Virtual Machines (HLL-VMs)
  - At many different layers...
  - With different goals...

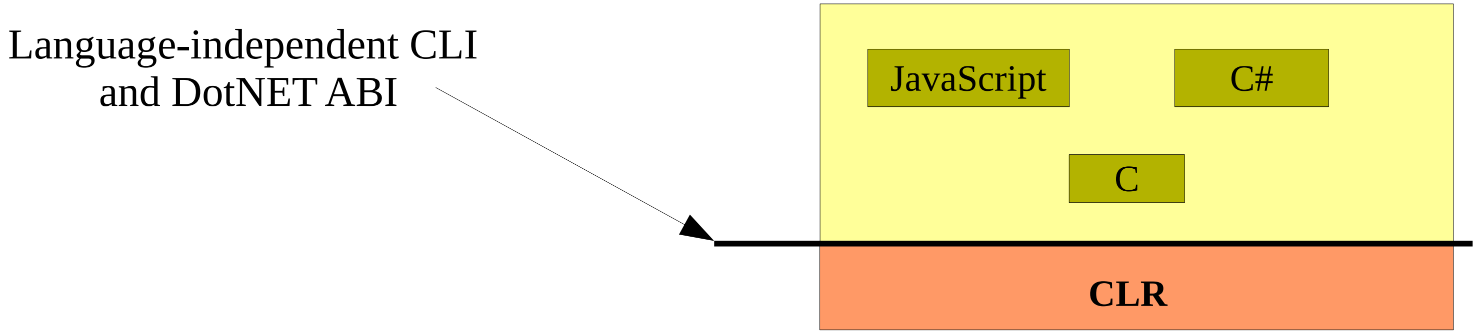


Global View – Examples



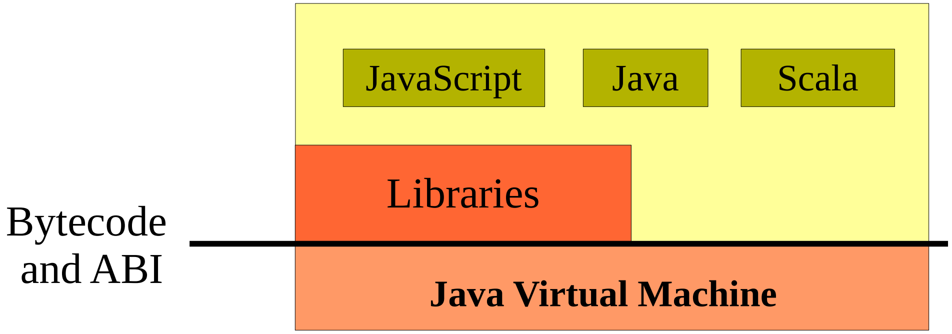
Global View (cont.)

- Focus on the virtual machine
  - They define software machines as opposed to hardware machines
- Microsoft.NET
  - Based on Common Language Runtime (CLR)
    - For all Microsoft language (C, C#, JavaScript, etc.)
    - No compiler generates real assembly language...
  - Common Language Instructions (CLI)
    - Object-oriented bytecode
    - Later compiled to the assembly language of some real machine...



Global View (cont.)

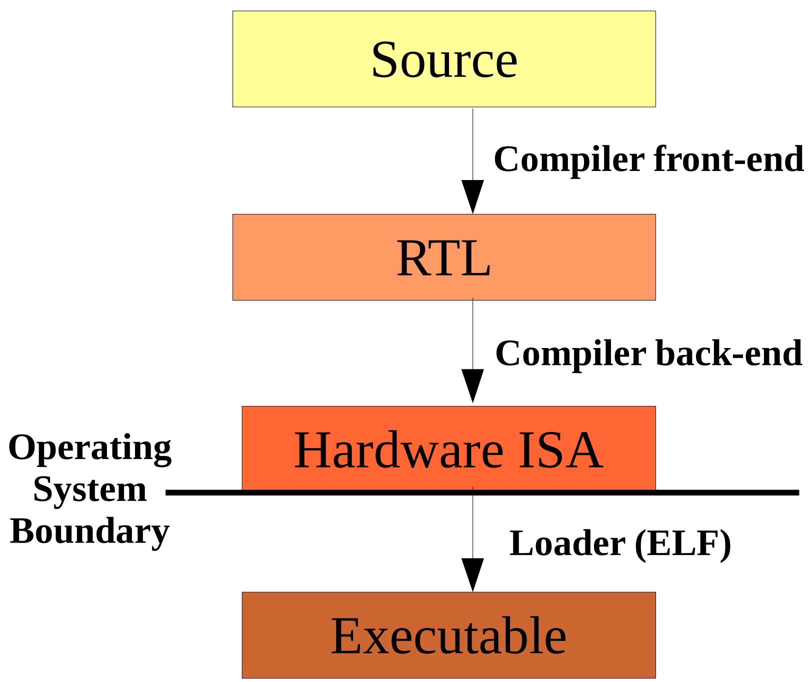
- Java Platform
  - Based on Java ByteCode
    - Object-oriented bytecode
    - Can be used as a target for different languages (Python, Scala, JavaScript, etc.)
  - Java Runtime
    - Java libraries
    - Application Binary Interface (ABI)
  - Multi-application plateform
    - ClassLoaders (security control)
    - Isolates (sofware-isolated processes)
    - Linux processes (Google Android)



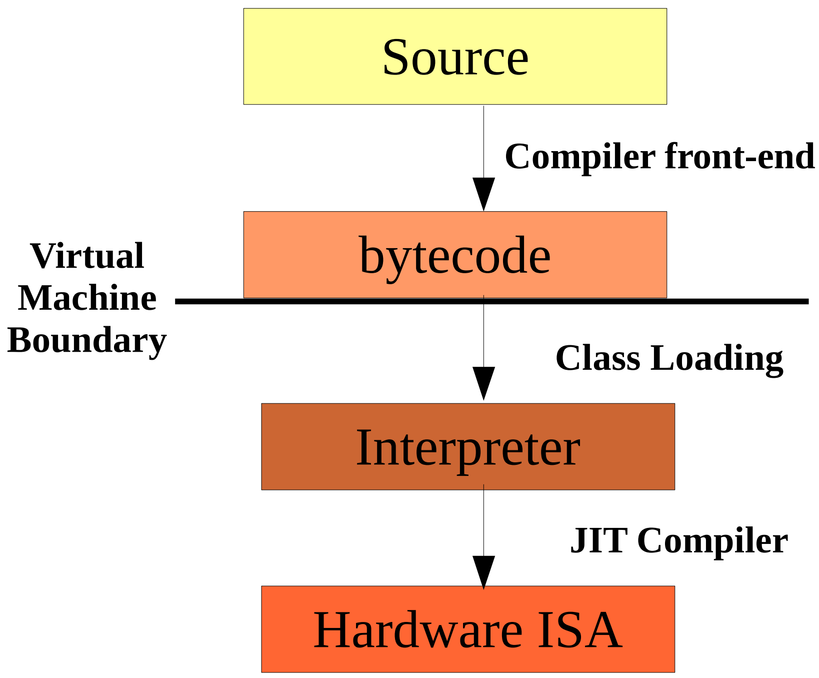
Global View – Compilation

- A shift of responsabilities...

Traditional Language Compilation
& Loading Chain



High-Level Language Compilation
& Loading Chain



Global View – Loading

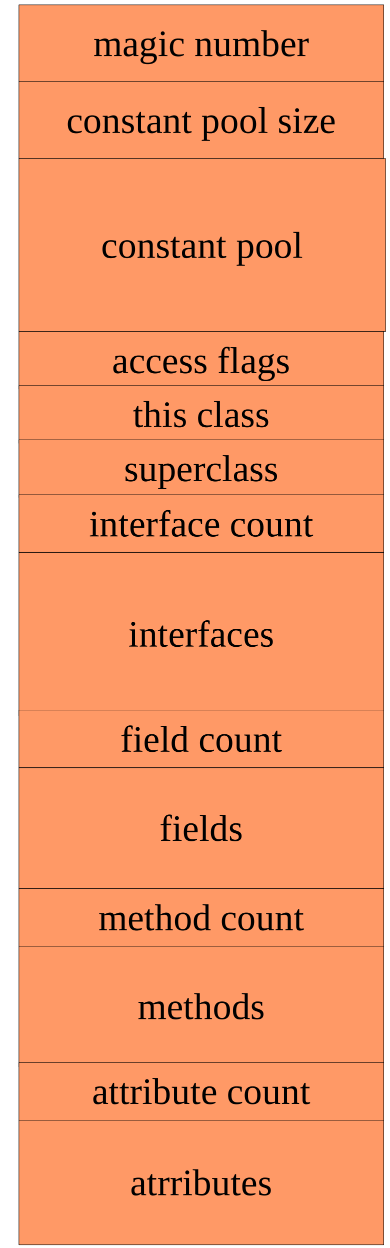
- Traditional Process Loading
  - Exec syscall invokes the loader
  - Loader creates the process regions
  - Pre-main code executes (mostly to carry out initializations)
  - Your main executes
  - As need, dynamic libraries are loaded
  - Libraries are the interface to the kernel
- High-Level Language Loading
  - The VM initializes memory regions
  - The VM initializes data structures
  - Loads classes on demand and performs class initializations
  - Executes your main, often with JIT compilation
  - Libraries are the interface to the kernel

Object-Oriented ISA

- Instruction Set
  - Very much like a regular processor
  - An object-oriented view of memory though
  - Method invocations, not only branch instructions
- Other concepts
  - Memory management such as object allocation and garbage collection
  - Need ways to achieve I/Os, from using POSIX down to hardware registers and interrupts
  - Other concepts like threads, synchronization, etc.
- ClassFile Format
  - Very much like the ELF format
  - Describe a piece of executable code
  - But the core difference:
    - ELF is rather concrete, ClassFile is rather abstract
    - For e.g. field offsets in C-struct and memory addresses are known in an executable ELF

Class File Format

- Meta-data part
  - A Java type description
    - A class name and flags
    - Its superclass and implemented interfaces
    - Its fields and methods
  - **All linking information is expressed through names**
    - Naming types (classes, interfaces)
    - Naming members (fields and methods)
- Constant pool
  - Contains the linking names
  - But also some constant values
    - Primitive types and strings
- Code part
  - Bytecode sequences
  - As attributes on methods



Class File Format


```
public class Line {
  int a;
  int b;
  Line(int a, int b) {
    this.a = a; this.b = b;
  }
  int equation(int x) {
    return a*x+b;
  }
  public String toString() {
    return “a line“;
  }
}
```

```
magic number
constant pool size
constant pool:
  “a line“
  java.lang.Object

access flags: public
this class: Line
superclass:    idx
interface count: 0
interfaces:

field count: 2
  int a;
  int b;
method count: 3
  <init>(int a, int b)
  int equation(int x)
  public String toString()

attribute count: 3
  bytecode arrays
```

Class File Format


```
package org.xyz;

public class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) {...}

}
```

```
package org.pqr;

import org.xyz.Foo;

public class Bar extends Foo
  implements  IBar {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) {... }
  void foo(int x, int y) {... }

  int bar(int x, int y) { ... }
}
```

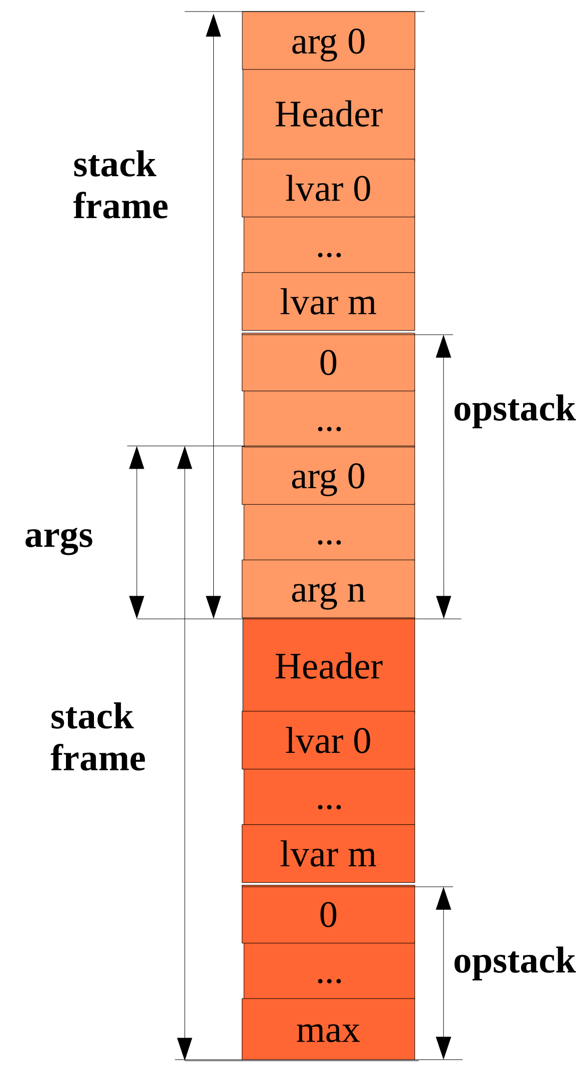
```
magic number
constant pool size
constant pool:
  java.lang.String
  org.pqr.IBar
  org.xyz.Foo
access flags: public
this class: Bar
superclass:  idx
interface count: 0
interfaces:  idx

field count: 2
  int a;
  String c;

method count: 3
  <init>(String c, int b)
  int foo(int x)
  void foo(int x, int y)
  int bar(int x, int y)

attribute count: 4
  bytecode arrays
```

Global View – Object-Oriented ISA

- Java Instruction Set
  - Common instructions
    - Arithmetic instructions, branch instructions, etc.
  - Object-related instructions
    - Allocation:
      - ***new***, ***anewarray*** and ***multinewarray***
    - Type cheching
      - ***checkcast*** and ***instanceof***
    - Field access
      - ***getfield***, ***getstatic***, ***putfield*** and ***putstatic***
    - Array access
      - ***aload and astore***
    - Method invocation
      - ***invokesuper, invokestatic, invokeinterface***, and ***invokevirtual***

Global View – Object-Oriented ISA

- Architected Stack
  - **Stack Frames**, one per method invocation
  - Per stack frame:
    - **Frame header**
      - Return address, and corresponding method
    - **Arguments and local variables**
    - **Operand stack**
- Instruction operands
  - From the operand stack or the class constant pool



putfield (8bit) field-index (16bit)

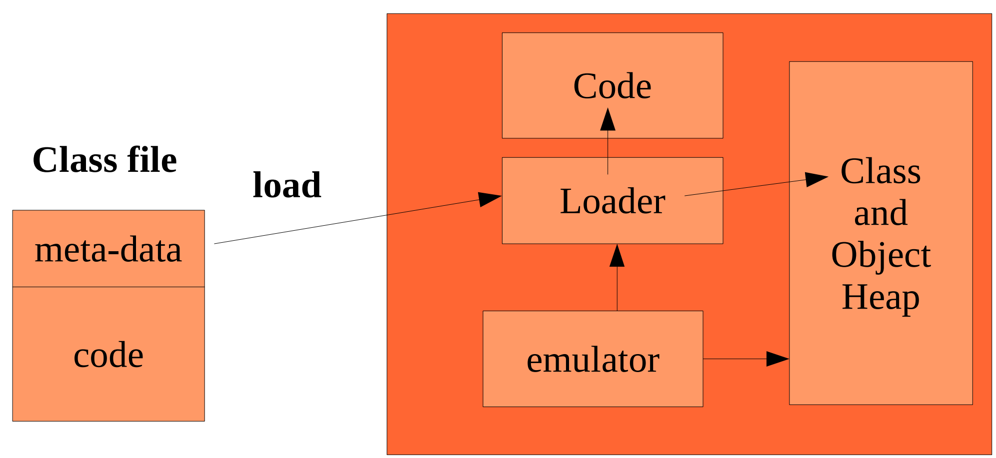
Opstack: ..., objectref, value
=> ...,

invokeinterface (8bit) method-index (16bit)

Opstack: ..., objectref, [arg1, [arg2, ...]]
=> … [value]

Internal Architecture

- Memory management
  - Allocate and reclaim objects
- Emulator and Loader
  - Interpreter, with or without Just-In-Time (JIT) compilation
  - Single-threaded or multi-threaded execution



Global View (cont.)

- What is Java? (or any high-level language for that matter)
  - A programming language
    - Syntax, type system, etc.
  - A platform (Java Runtime Environment)
    - JRE (Java Runtime Environment)
    - Defines concepts such as threads, files, or sockets
    - Defines dynamic class loading, security model, etc.
  - A virtual machine
    - An instruction set
    - An Application Binary Interface

Global View (cont.)

- Java Runtime Environment
  - Different profiles: J2EE, J2SE, J2ME, JavaCard
    - From almost everything (J2EE) to almost nothing (JavaCard)
  - Google Android
    - Another completely different runtime environment
- Virtual Machines
  - From large-scale servers
    - Thousands of threads on 150GB heap on 64bit multi-cores
  - To client platforms
    - A few threads on 500MB to 1GB heap on 32bit or 64bit processors
  - To embedded platforms
    - Often a single thread on as little as 64KB on 8bit or 32bit microcontrollers
  - To Smart Cards
    - Almost nothing at all... on a smart-card System-On-Chip (SoC)

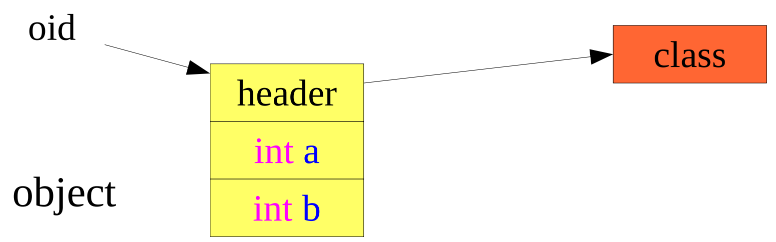
Global View (cont.)

- Focus on the virtual machine
  - They define software machines as opposed to hardware machines
- Object-oriented ISA
  - Both CLI and Java bytecode are object-oriented
- Instruction Set
  - All the regular instructions
  - Stack-oriented instructions
  - Object-oriented calling convention
- Application Binary Interface
  - Object-oriented interfaces ``*replace system calls*``
  - In other words, some objects are gates to the outside world
  - Either to a different language, the operating system, or the hardware

Object-Oriented ISA – Object-Oriented Model

- Object-oriented model
  - An object is a triplet
    - An identity, a state, and a behavior
  - An object is an instance of a class
    - A class is a factory for its instances
    - Instances of a class form its extent
  - Classes reify types
    - Define a structure (fields)
    - Define a behavior (methods)
    - Define constructors

```
class Line {
  int a;
  int b;
  Line(int a, int b) {
    this.a = a; this.b = b;
  }
  int equation(int x) {
    return a*x+b;
  }
}

int x,y;
Line line = new Line(2,3);
x = 5;
y = line.equation(x);
```



Object-Oriented ISA

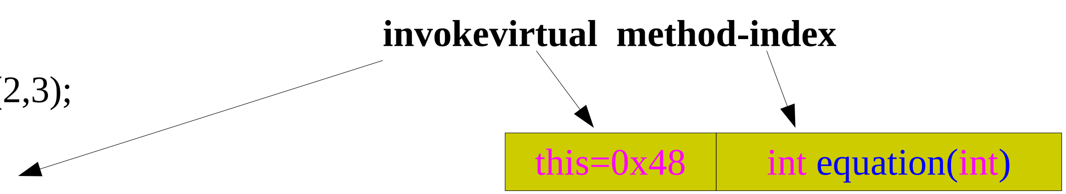
- Object-oriented model
  - Arrays are objects in Java
    - **The synthetic field *length***
    - Special builtin operator **[]**
  - Array classes also automatically created
    - Array classes have the access modifiers of their element type
    - An array of private classes is private
    - Arrays are cloneable and serializable
  - ***Classes are objects too!***
    - *We will come back to that later...*

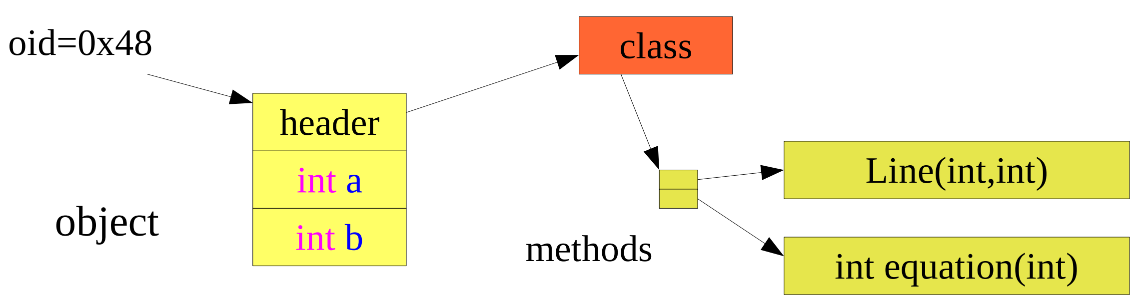
Object-Oriented ISA

- Object-oriented model
  - Method invocation
    - Sending a message to an object
    - The object is called the receiver
  - The class dispatches the message
    - This is called late binding (finding the code)
    - Matching the method signature to the method declared in the class

```
class Line {
  int a;
  int b;
  Line(int a, int b) {
    this.a = a; this.b = b;
  }
  int equation(int x) {
    return a*x+b;
  }
}
```

```
int x,y;
Line line = new Line(2,3);
x = 5;
y = line.equation(x);
```





Object-Oriented ISA

- Object-oriented model
  - Classes are organized in sub-typing hierarchy
    - Subtypes inherit both the structure and behavior of super types
    - Do not confuse with aggregation
  - Method inheritance
    - Method overloading
      - Same name, but different signatures
    - Method overriding
      - Same signature
  - Structural inheritance
    - All fields are inherited
    - No matter the names or types

```
class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) {...}
}
```

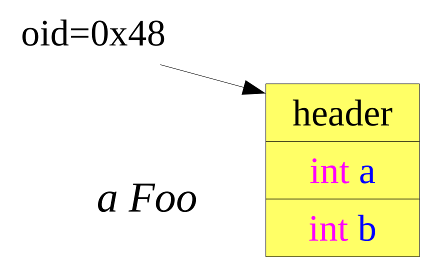
overriding

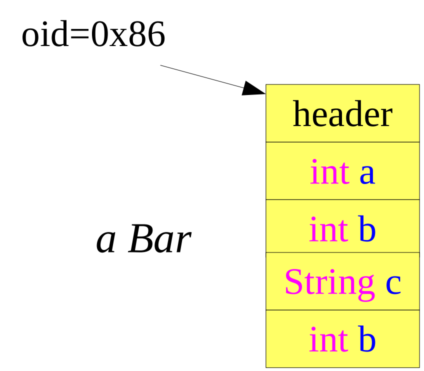
overloading

```
class Bar extends Foo {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) {... }

  void foo(int x, int y) {... }

  int bar(int x, int y) { ... }
}
```

Object-Oriented ISA

- Object-oriented model – Structural inheritance
  - All fields are inherited
  - No matter the names or types






```
class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) {...}

}
```

```
class Bar extends Foo {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) {... }
  void foo(int x, int y) {... }

  int bar(int x, int y) { ... }
}
```

Object-Oriented ISA – Reified Classes

- Object-oriented model – Structural inheritance
  - Computing the memory layout of a class C

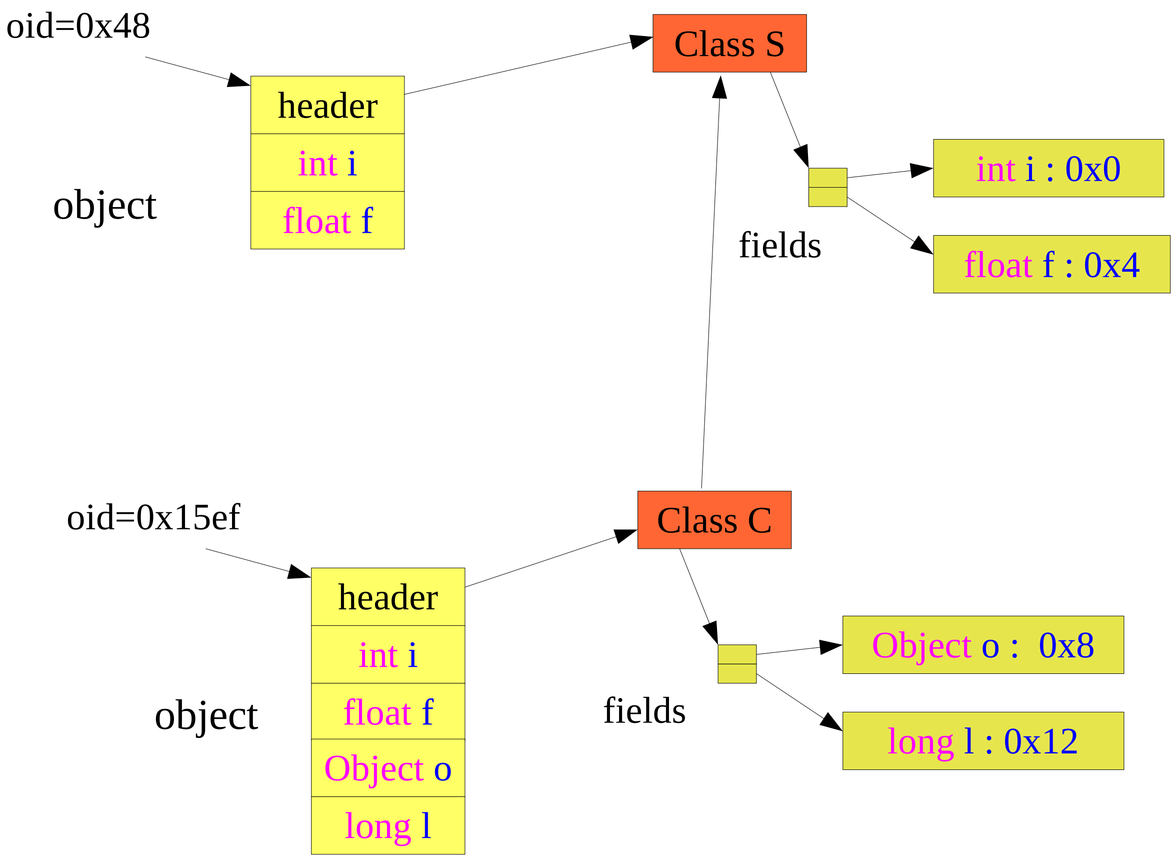
Instructions:

Putfield (8bit) index (16bit)

Opstack: ..., objectref, value
=> ...,

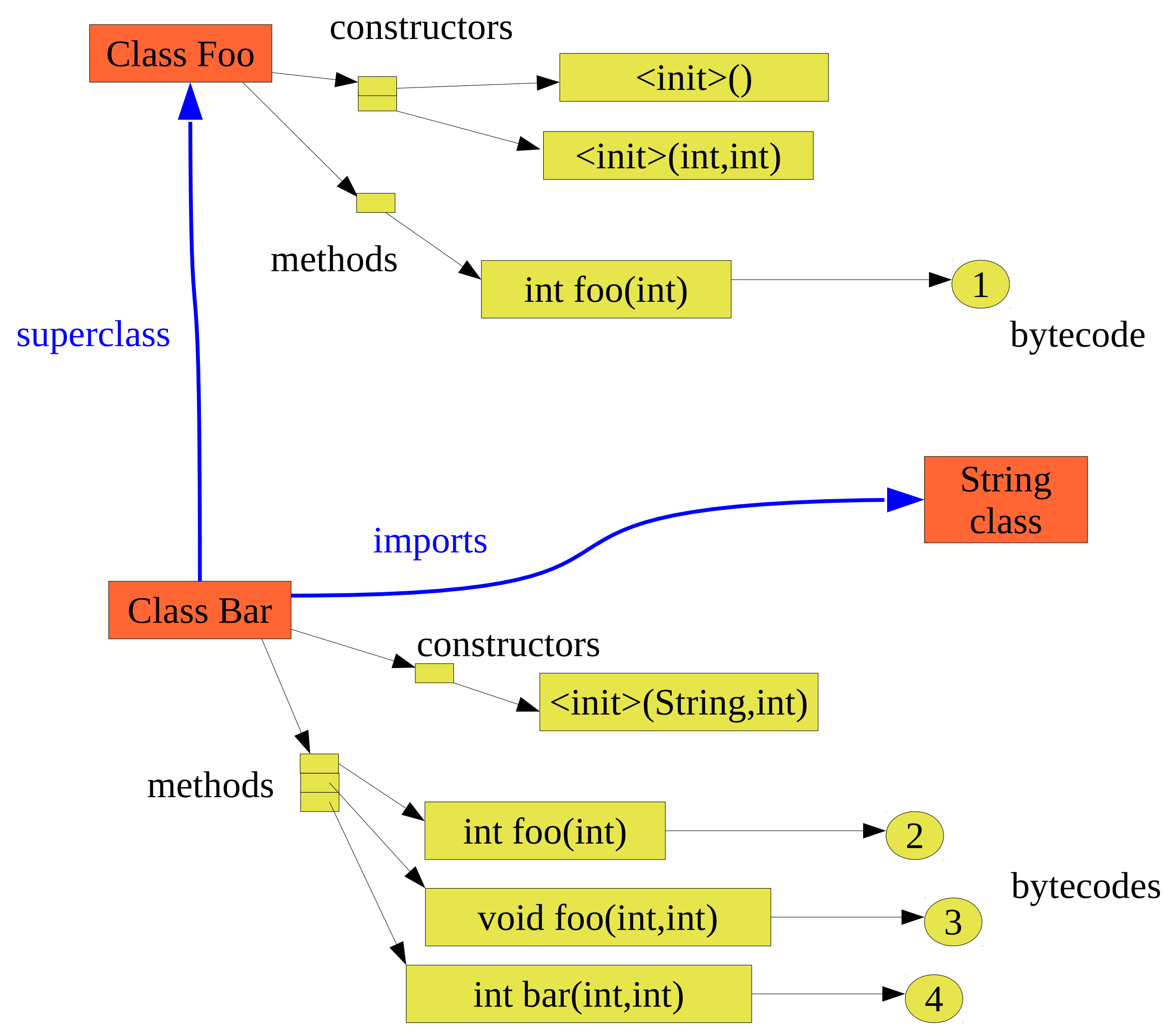
getfield (8bit) index (16bit)

Opstack: ..., objectref
=> ..., value



Object-Oriented ISA – Reified Classes


```
class Foo {
  int a;
  int b;
  Foo() {...}
  Foo(int a, int b) {...}

  int foo(int x) { 1 }
}
```



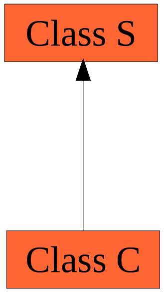

```
class Bar extends Foo {
  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) { 2 }

  void foo(int x, int y) { 3 }

  int bar(int x, int y) { 4 }
}
```

Object-Oriented ISA – Method Invocations

- Method invocations
  - ***invokevirtual index***
    - opstack: ..., this, [arg1, [arg2,...]] => ...
    - Index is to a method symbolic reference in the constant pool
    - Will be translated to a vtbl-indirect jump at runtime
  - ***invokestatic index***
    - opstack: ..., [arg1, [arg2,...]] => ...
    - Index is to a method symbolic reference in the constant pool
    - Will be translated to a direct jump address at runtime
  - ***invokeinterface index***
    - opstack: ..., this, [arg1, [arg2,...]] => ...
    - Index is to a method symbolic reference in the constant pool
    - Will require a dynamic lookup for the method signature in order to locate the code to execute

Object-Oriented ISA

- Object-oriented model – Virtual Method Invocation
  - Optimizing method invocation through virtual tables
    - Compute recursively the virtual table of the superclass S of C
    - Make a copy of the vtbl of class S
    - Use it as a starting point for the vtbl of the class C
    - Skip static methods, private, and constructor methods
  - For each method M declared in class C
    - If M signature is new
      - That is, no method in the virtual table has it
      - Add an entry, with the code pointer of M
      - Remember the vtbl index in the reified description of M
    - Else if M overrides an existing method M'
      - Replace the M' entry with the code pointer of M



Instruction:
invokeinterface (8bit)
index (16bit)

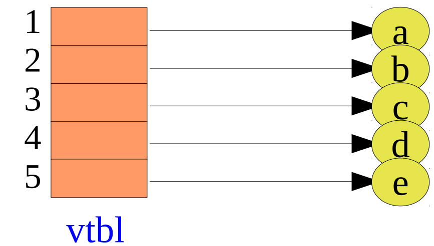
Opstack: ..., objectref, [arg1, [arg2, ...]]
=> … [value]

Object-Oriented ISA


```
class Foo {
  int a;
  int b;

  Foo(int a, int b) {...}

  int foo(int x) { 1 }

  private _foo(int x) {     }
}



class Bar extends Foo
  implements  IBar {

  int b;
  String c;

  Bar(String c, int b) { ... }

  int foo(int x) { 2 }

  void foo(int x, int y) { 3 }

  int bar(int x, int y) { 4 }
}
```

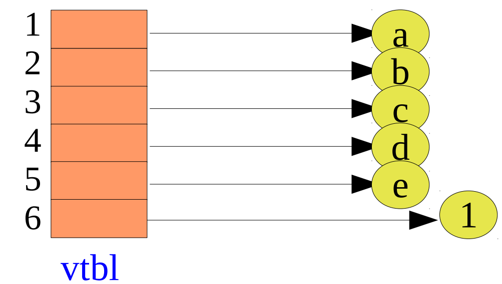
class Object

String toString()
int hashCode()
boolean equals()
void wait();
void wait(long);



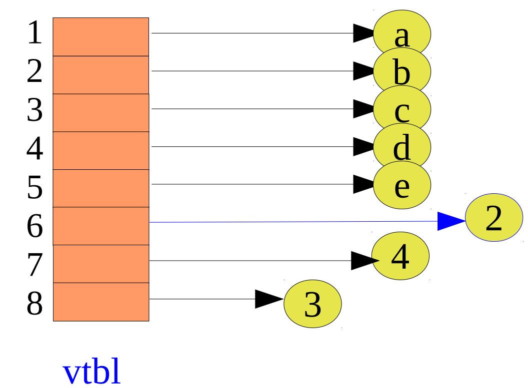
class Foo

String toString()
int hashCode()
boolean equals()
void wait();
void wait(long);
int foo(int);



class Bar

String toString()
int hashCode()
boolean equals()
void wait();
void wait(long);
int foo(int);
int bar(int,int);
int foo(int, int);

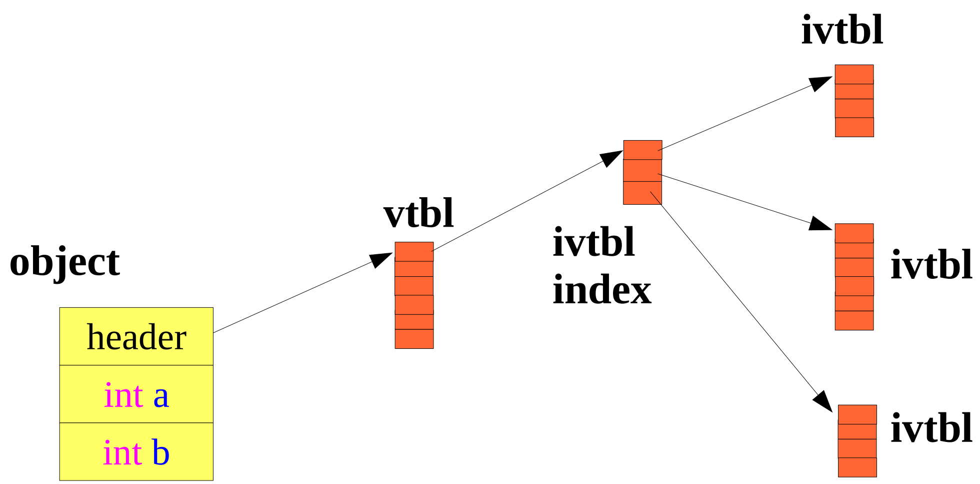


Object-Oriented ISA

- Service-oriented programming
  - Java Interfaces
    - Interfaces only define behaviors
    - Interfaces support multiple inheritance
    - A class implements one or more interfaces
  - Abstract classes
    - Classes that cannot be instantiated
    - Interfaces are always abstract
- Invocation overheads
  - Abstract classes retain the virtual-table invocation
  - Interfaces introduce more overhead
    - One vtbl is necessary for the virtual invocations (the one for the class)
    - One vtbl is necessary per implemented interface

Object-Oriented ISA

- Object-oriented model – Interface Method Invocation
  - Caller only knows the interface type, not the actual receiver type
  - We need a mechanism to select the right ivtbl on the actual receiver
  - Use the 16-bit index as the index in the ivtbl
  - Use the 16-bit unused to store the unique interface id



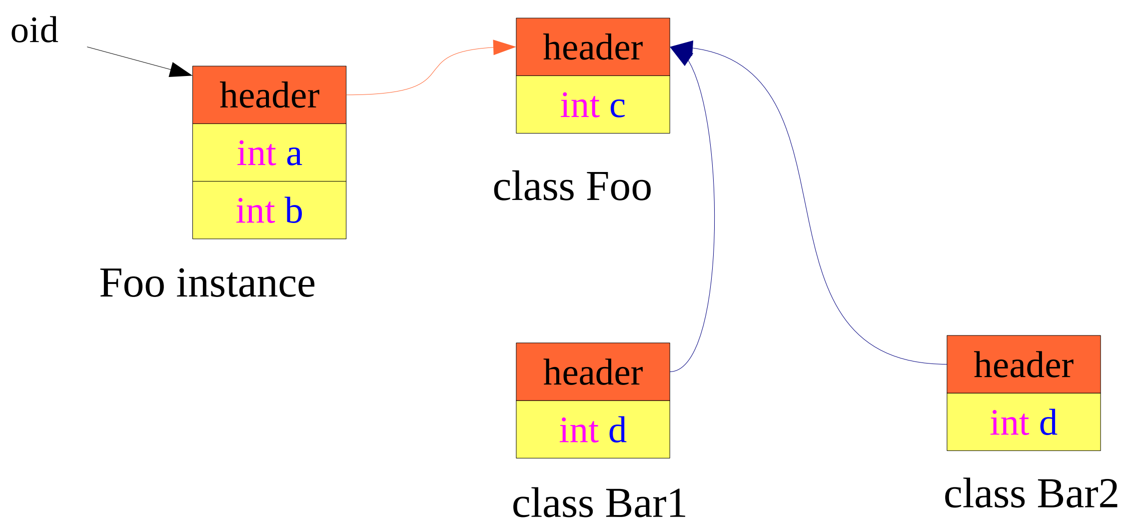
Instruction:
    invokeinterface (8bit)
    index (16bit)
    unused (16bit)

Opstack: ..., objectref, [arg1, [arg2, ...]]
=> … [value]

Object-Oriented ISA

- Object-oriented model
  - Static fields
    - As constants, both in interfaces or classes
    - As non-constant fields, only in classes
  - Statics are named global variables
    - They are not class fields, in the proper sense

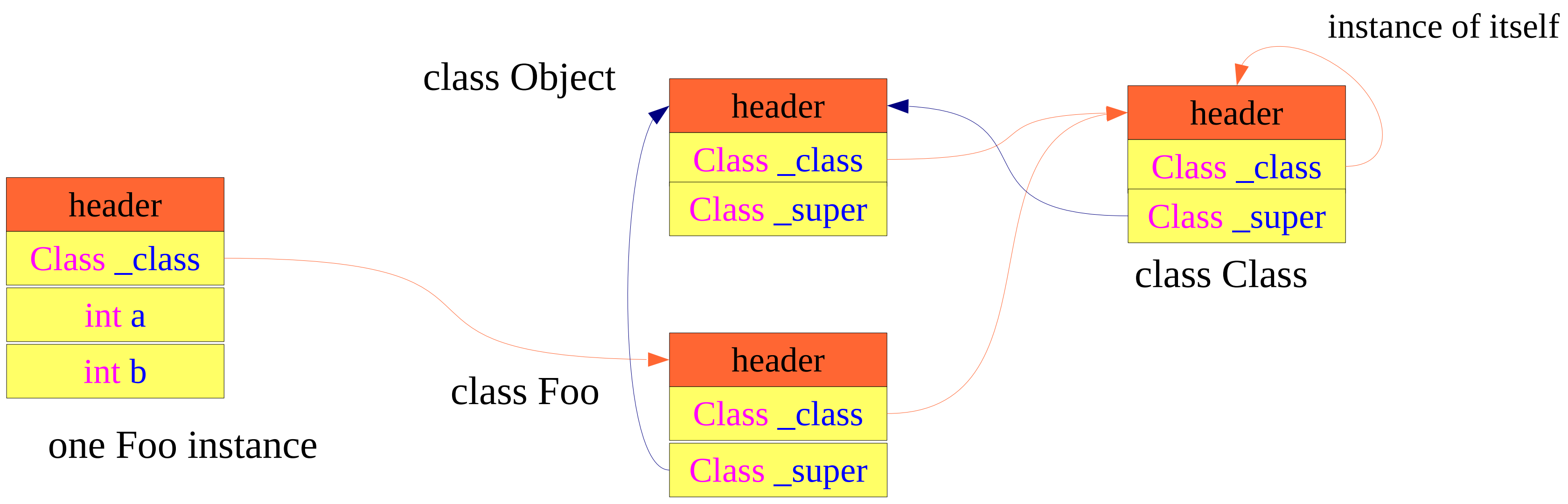
```
class Foo {
  int a,b;
  static int c;
}

class Bar1 extends Foo {
  int e;
  static int d;
}

class Bar2 extends Foo {
  int e;
  static int d;
}
```



Object-Oriented ISA

- Object-oriented model – **Classes are objects**
  - So they also have a class
  - The metaclass, called the class lass
  - Keep classes alive as long as they have an instance

```
class Object {
  Class getClass();
  ...
}

class Foo extends Object {
  int a,b;
  ...
}
```



Java Platform – Class Loaders

- Started Simple
  - As a sandbox for applets
  - Wanted a complete isolation of downloaded code
- Essentials
  - Its own copy of classes
    - Avoid sharing statics
    - Avoid name and version conflicts between loaded classes
  - Works hand-in-hand with Java security
    - Controls accesses to resources
- Evolved Poorly – Mixing several concepts
  - A scoping mechanism for types
  - A dynamic and lazy linker for classes
  - A mechanism to define (load) types

Java Platform – Class Loaders

- Class Loading
  - Only through the class file format
    - This is quite unfortunate
    - Only the JVM can create types programmatically
  - Special native method in the JVM
    - The native method **ClassLoader.define(...)**
    - Passing the byte array of a class file to define the described type
  - The class file is an exchange format
    - Could have been in XML, just using a more efficient binary representation
    - **Produced by Java compilers and consumed by class loaders**

Java Platform - Class Loaders

- Class loaders
  - A scope for Java types
    - Two class loaders defining the same type yields two runtime types
    - Even when using the same class file
  - Beware of equivalent names
    - Name equivalence does not mean a thing between class loaders
    - **Same type name** does not mean **the same type**
  - Structural equivalence does not mean the same type
    - Two types are the same only if the two class objects are the same class object

Rule 1: two classes are the same if they are the same class object

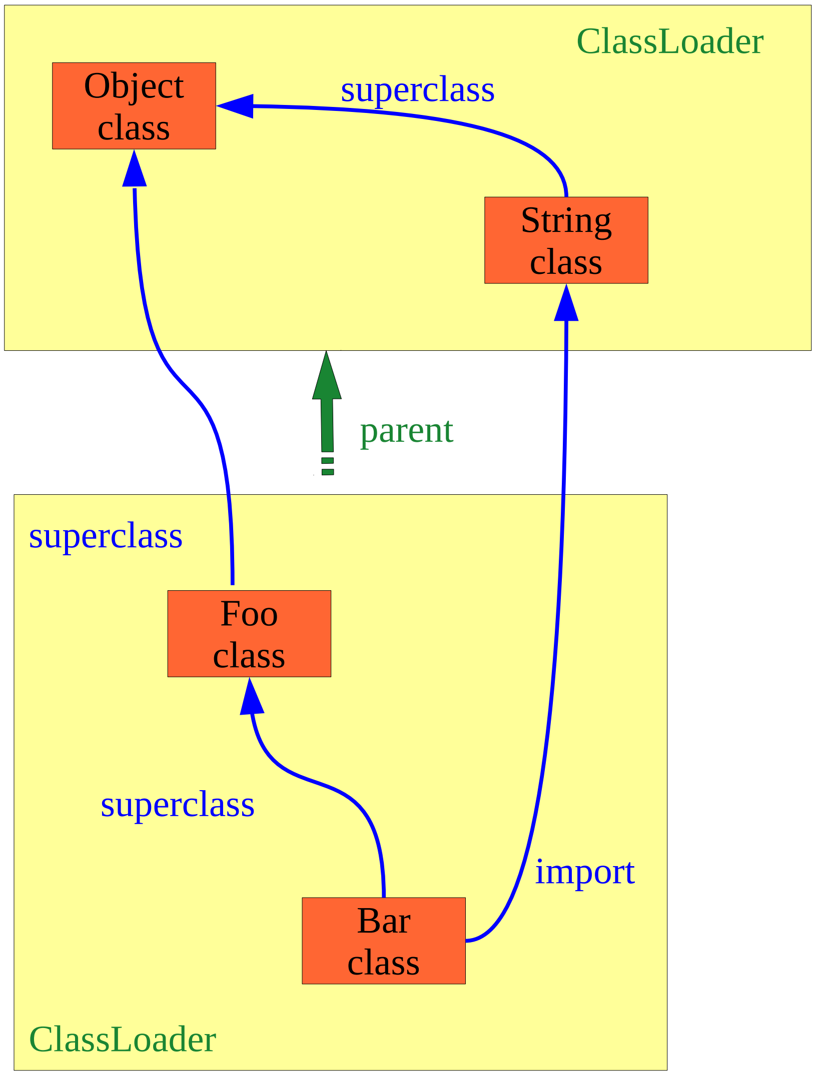
Rule 2: one class object belongs to one and only one classloader

Java Platform – Class Loaders

- Hierarchy of scopes
  - A single tree of class loaders per JVM
  - A class loader has a parent class loader
  - Types in the parent class loader are visible
- Bootstrap class loader
  - The root of all class loaders
  - Created at bootstrap by the JVM to load core classes
    - java.lang.Object, java.lang.Class
    - java.lang.String, java.lang.Throwable, java.lang.Exception
    - Etc.

Java Platform – Class Loaders

- Class loading
  - A tree of class loaders
  - A complex graph of types across all class loaders
- Reminder
  - Could have redundant loading!

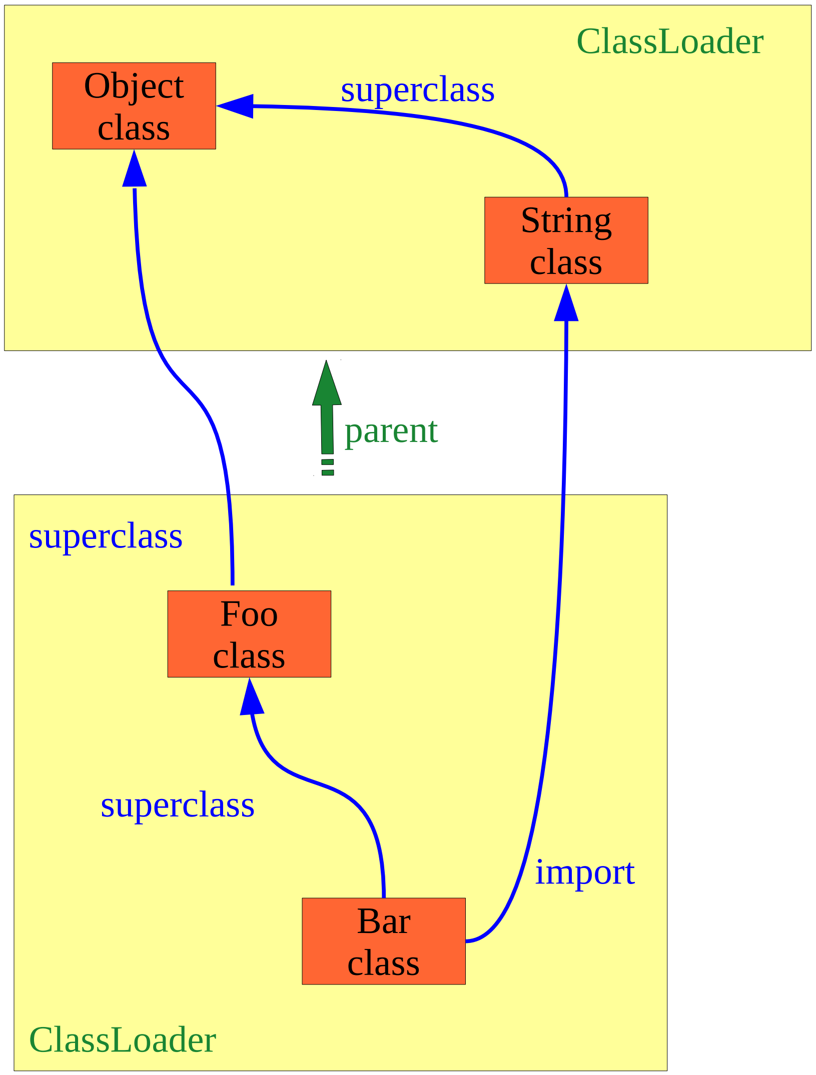


If the same class file is loaded
in different class loaders...

Then, it will be different class objects
and therefore different types

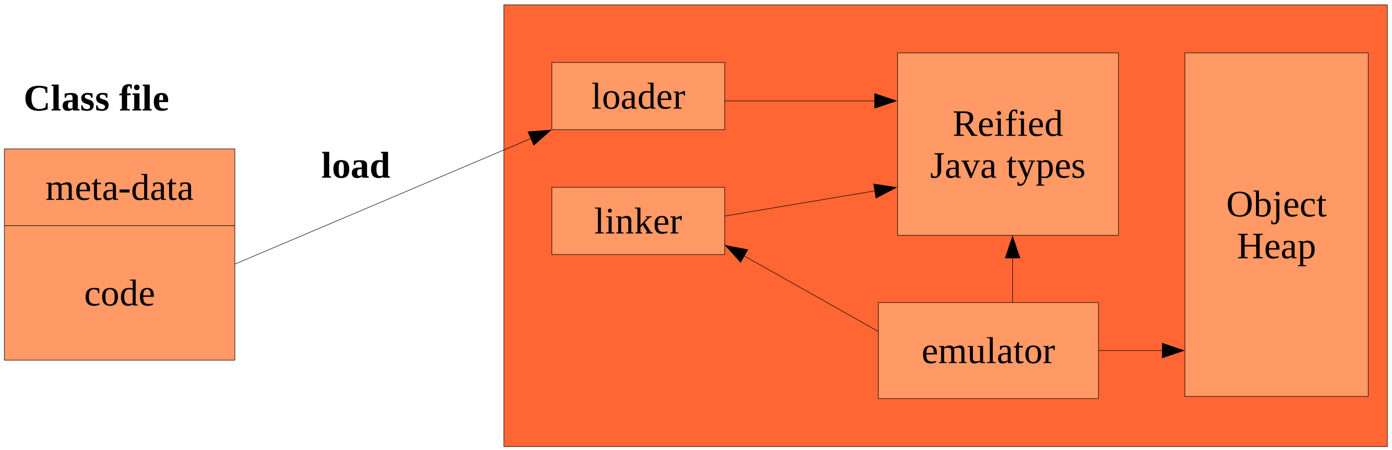
Java Platform – Class Loaders

- Dynamic and lazy class linker
  - Multi-stage linking
    - Loading
    - Prepared
    - Resolved
    - Initialized (static initializer)
  - Warning
    - Loading may succeed but resolving or initializing may fail much later



Java Platform – Reification

- Different approaches are possible
  - Original Sun's JVM
    - All C structures to represent Java types, **no reflection in Java**
  - Mix-mode
    - A mix of internal C structures and Java objects
    - This is the current approach for Oracle's JVM
  - Pure Java approach
    - Uniform representation using only objects
    - The emulator uses directly this representation or some derivative of it

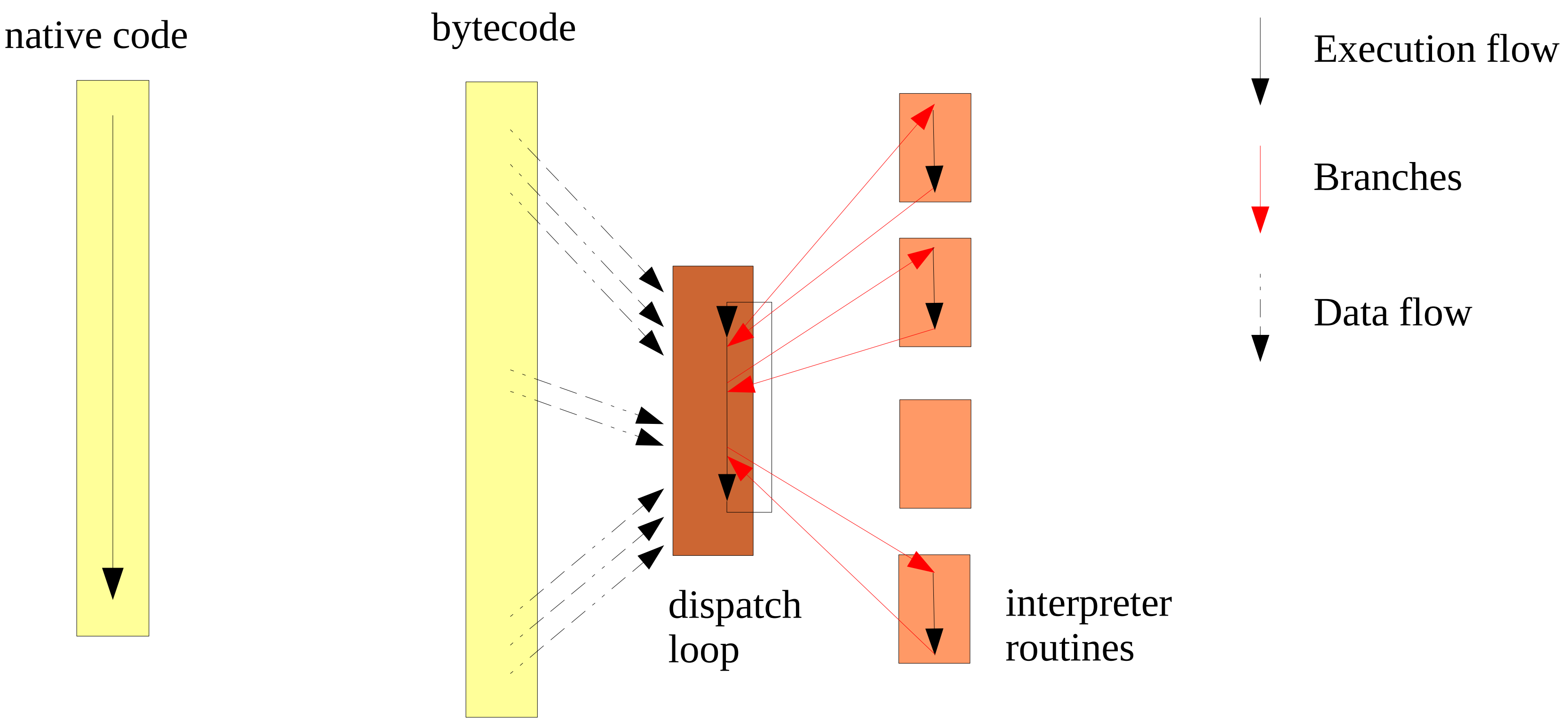


Java Platform – Emulator

- Emulator
  - It is the execution engine
  - It can be either an interpreter or a binary translator
- Binary translator
  - Can be a Just-In-Time compiler (JIT)
    - Most JVM have a JIT approach
  - Can be a Ahead-Of-Time compiler (AOT)
    - GCJ can be used as an AOT
- Interpreter
  - Use a traditional fetch-decode-issue cycle

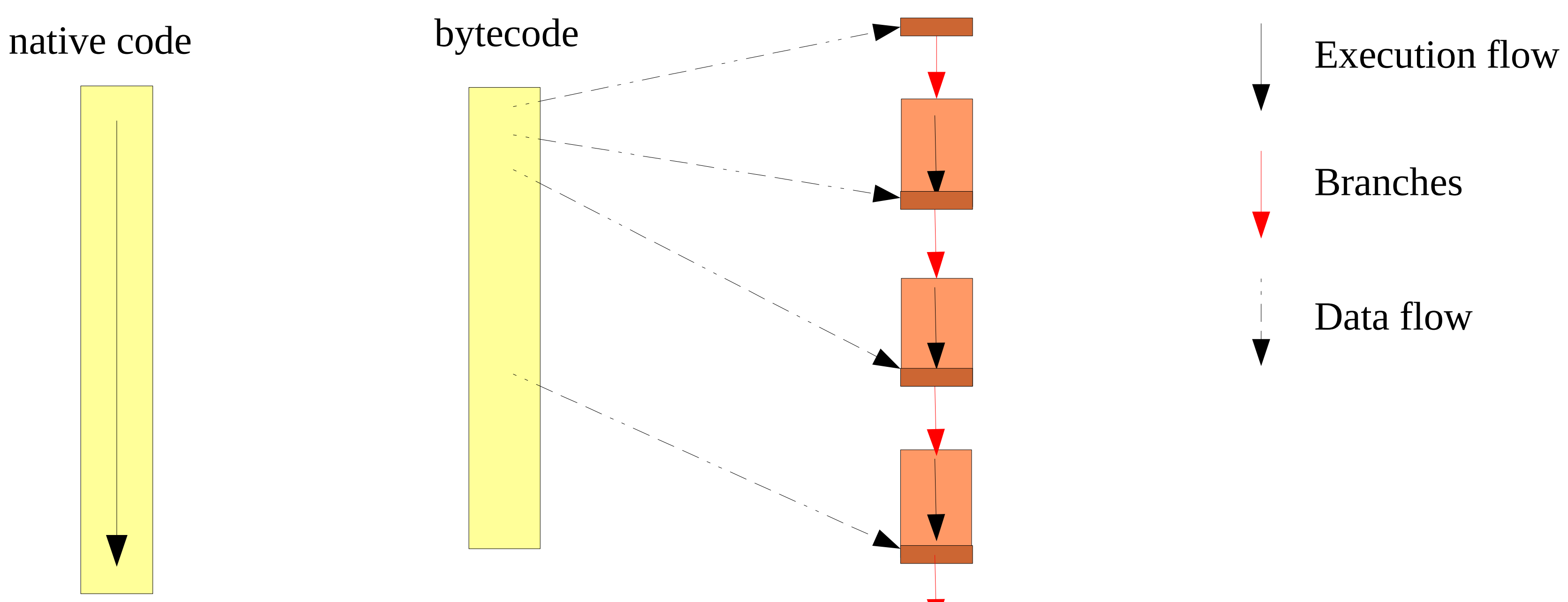
Java Platform – Emulator

- Interpreter
  - Use a traditional fetch-decode-issue cycle
  - Native code, the processor is the interpreter
  - Bytecode, the interpreter is a dispatch loop



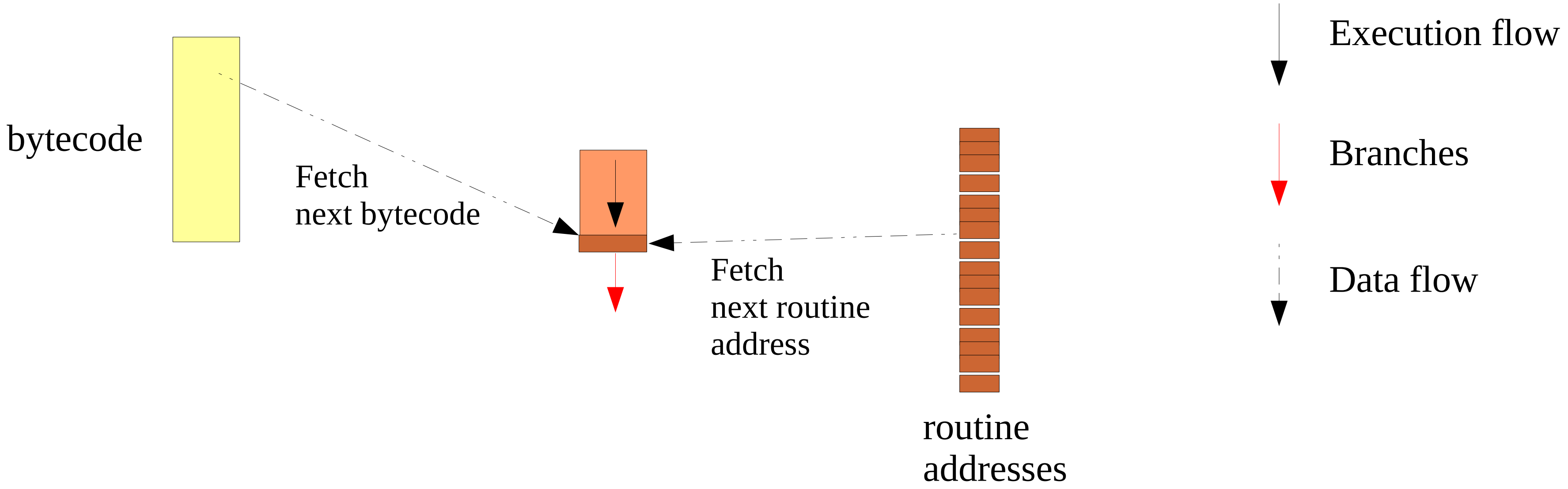
Java Platform – Emulator

- Indirect threaded interpreter
  - Use a traditional fetch-decode-issue cycle
    - But save 2 out 3 branches...
    - Huge gain in performance...
  - But avoid the loop and switch; threads in routines the dispatch
    - Use an array of routines, indexed by bytecode
    - Fetch the next bytecode, use it to index the array to find the next routine to jump to

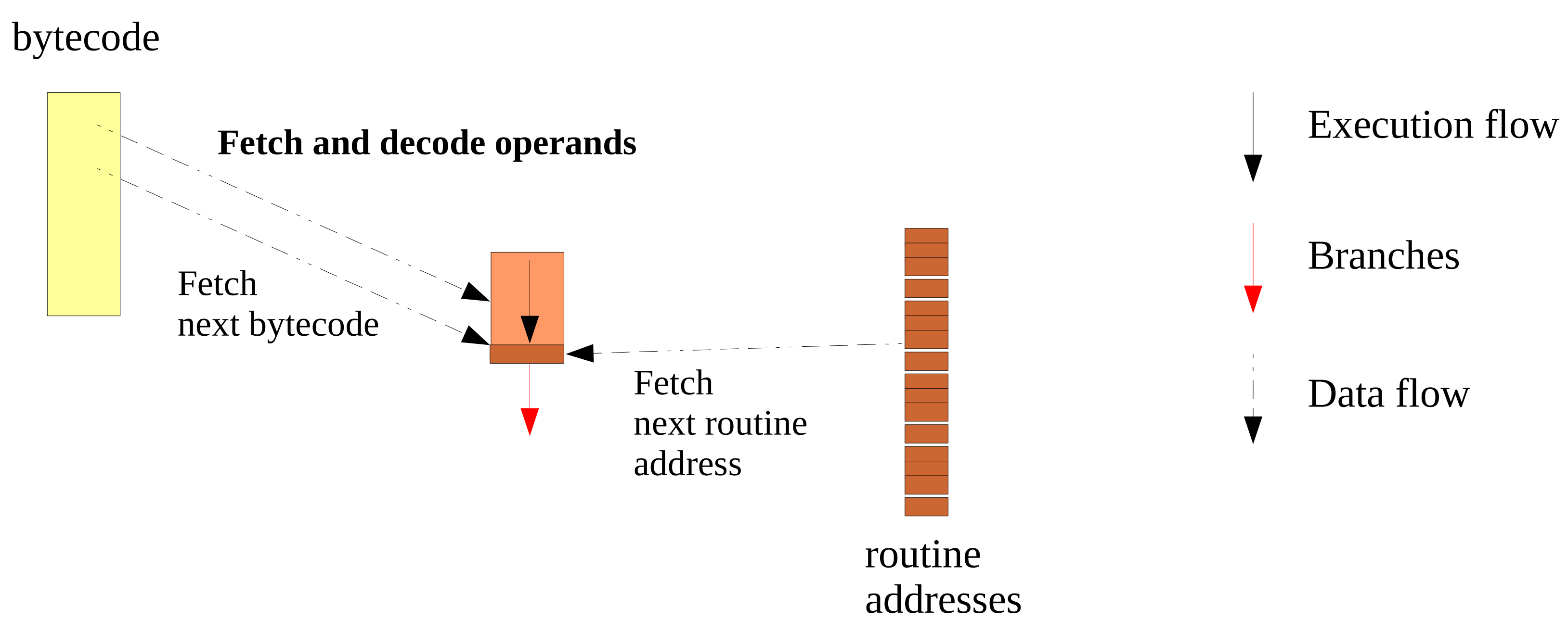


Java Platform – Emulator

- Indirect threaded interpreter - Analysis
  - Memory access
    - We still have a memory access to fetch the next bytecode
    - Another register-indexed memory access to read the corresponding routine address
  - Branch
    - We still have a register-indirect branch
    - Target address is known right just before doing the jump
    - So out-of-order and pipeline execution cannot help...
    - Neither can speculative execution...

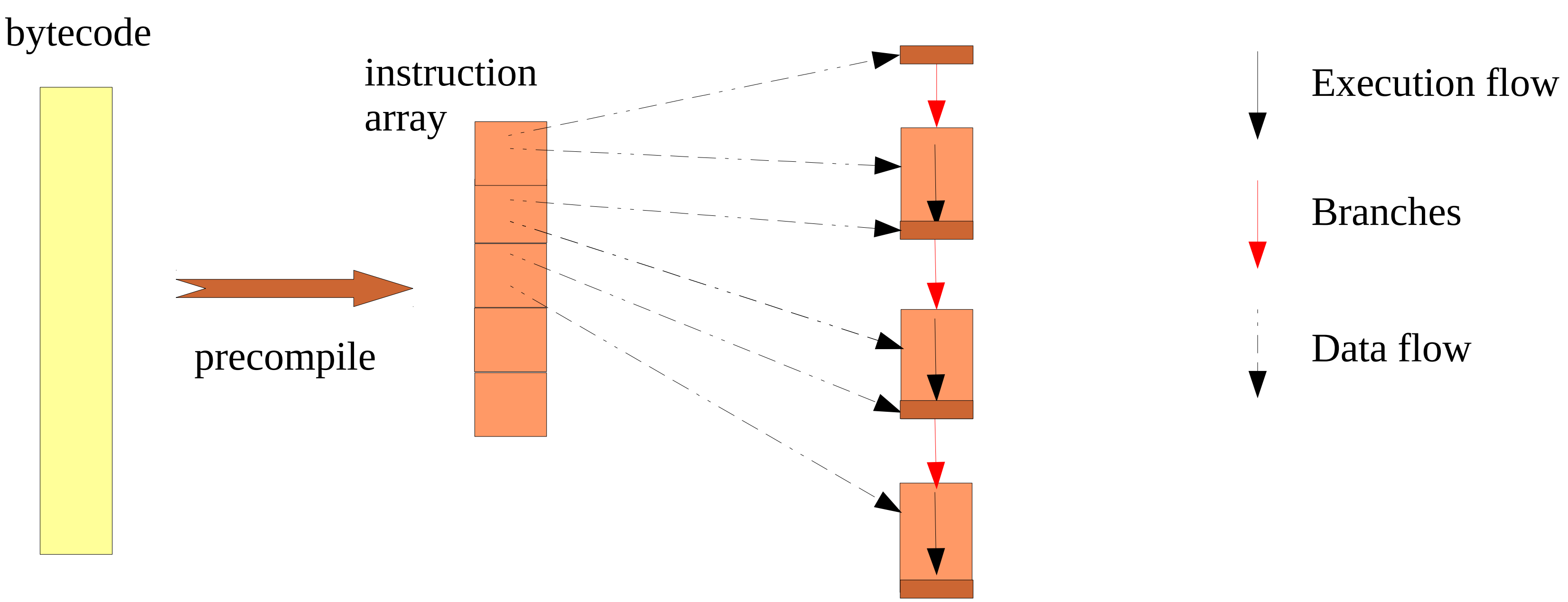


- Indirect threaded interpreter - Analysis
  - Repeated decoding
    - Still have to decode operands for every bytecode
    - Examples:
      - Extract constant values from the constant pool
      - Field offsets (indexed access through the constant pool)
      - Vtbl indices when invoking methods (indexed access through the constant pool)



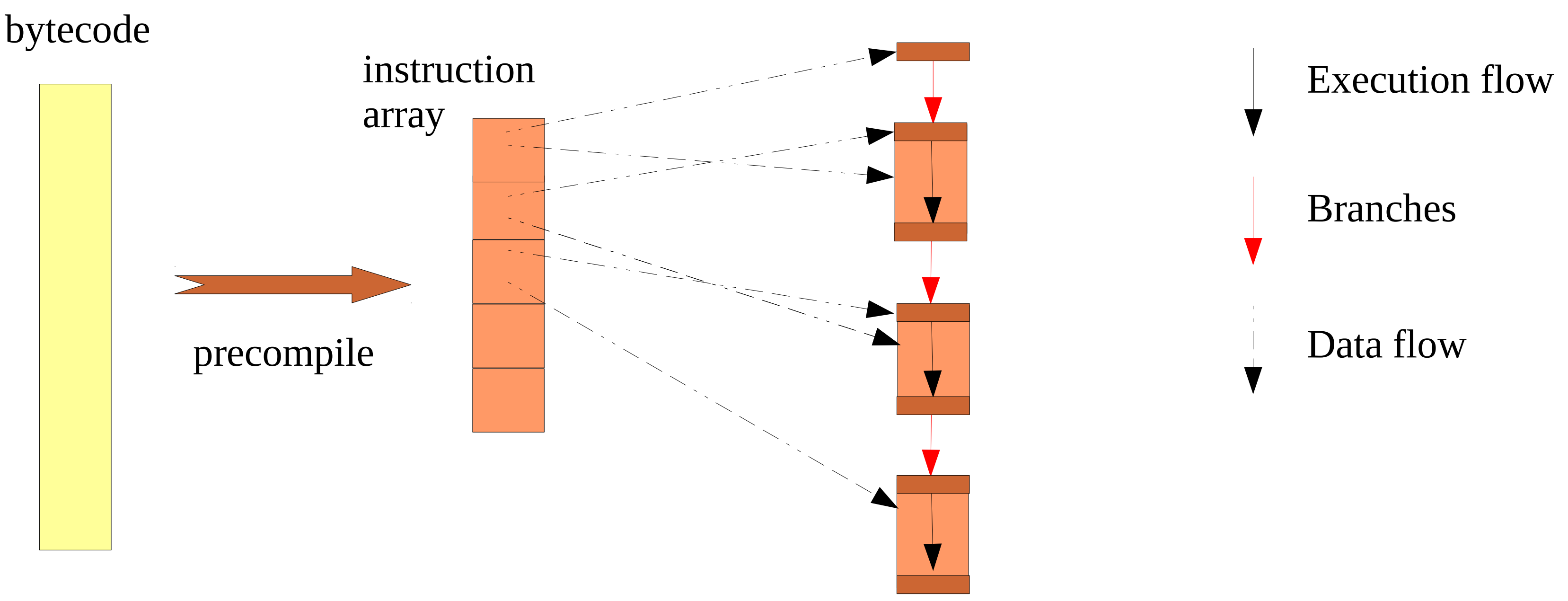
Java Platform – Emulator

- Direct threaded interpreter
  - Pre-compile bytecode sequence into **instruction sequence**
  - Instructions are made easier to interpret
    - Usually a struct in memory
      - Have the address of the routine and one or two extracted operands
    - Allocated as an array, contiguous in memory
      - Execution flows in sequence through instructions in memory, but for branches



Java Platform – Emulator

- Direct threaded interpreter – Prefetching
  - Using superscalar ability to reorder instructions
    - Prefetch the next handler before executing the current one
  - Expected gains
    - Expected gain on memory access delays
    - Expected to help keep the pipeline from stalling if target address is known soon enough



Java Platform – Emulator

- Just-In-Time Compilation
  - Produce assembly instructions from bytecode
- A specific field of Dynamic Binary Translation (DBT)
  - Must be fast, as DBT in hypervisors
  - Simpler since it is translating well-formed bytecode
- Key optimizations
  - Making the interpreter disappear...
  - Code relayout
  - Inlining
  - Dynamic decisions

Java Platform – Emulator

- Making the interpreter disappear...
  - We are executing native assembly instructions
- What is the difference with the assembly produced from the sources of a C program?
  - Only the semantics of the language introduce differences
  - Null pointer checks, array index checks,
  - Method polymorphism, dynamic type checks
  - Object monitors
- Code relayout
  - Same as for all statically compiled languages
  - Lifting invariants from loops
  - Efficient use of registers as the ultimate cache level of the memory hierarchy
  - Ordering instructions to help reduce the memory barrier

Java Platform – Emulator

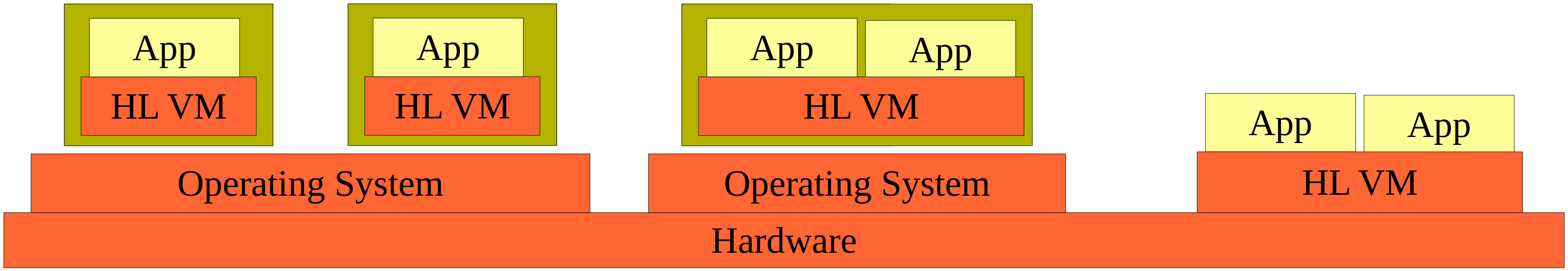
- Inlining
  - Essential but hard because of bounded polymorphism in object-oriented programs
  - Easy on static methods and constructors
  - But often requires to be able to de-virtualize
- Dynamic decisions
  - Monitor programs' behavior and adapt the produced code
  - Optimize harder the hot spots
    - Example: allocate more time for register allocation (more than 50% of compile time in JITs)
  - Produce slow and fast paths for common cases
  - Often rely on inserting barriers in the instruction stream
  - Often requires On-Stack Replacement
  - Helps with debugging, OSR of optimized methods

Java Platform – Execution

- Traditional approach based on threads
  - Define a Thread class, instances map to kernel threads
  - For each thread, we have an invocation stack
  - Synchronization based on monitors with an exit consistency
  - Added Java locks later on...
- But other approaches exist
  - Single-threaded Java, no monitor
  - Event-oriented execution, usually single-threaded

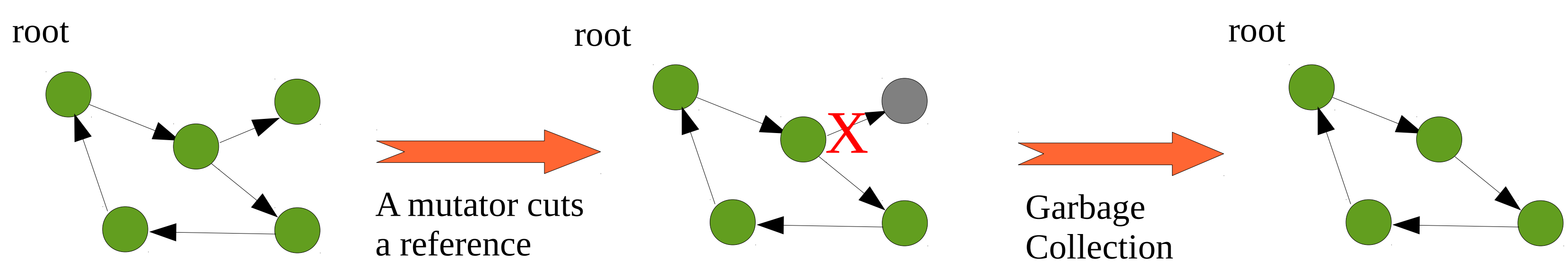
Java Platform – Execution

- Memory isolation
  - Traditionally done through processes, leveraging virtual memory
  - Can be done almost for free in HLL Vms
    - Isolate in J2ME
    - AppDomains in CLR
- The principle
  - Since object references cannot be forged...
  - Isolation can be achieved by controlling how references are passed
  - Enabling code sharing (as regular operating systems do)



Java Platform – Garbage Collection

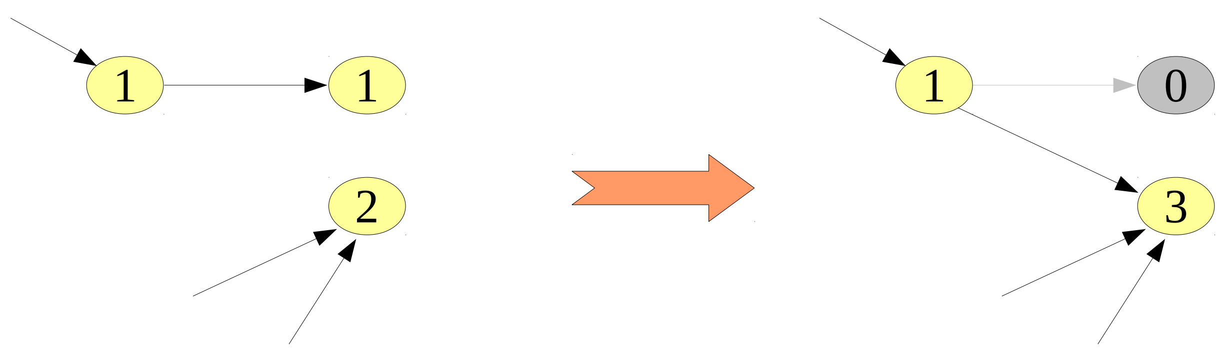
- Java is garbage collected
  - **Live objects are kept**
  - Live objects are reachable from roots of persistence
  - Roots are traditionally thread stacks and static fields in loaded classes
- **Being garbage is a stable property**
  - I.e. once an object is garbage, it remains garbage



Java Platform – Garbage Collection

- Garbage Collector
  - Garbage collection is about detection and reclaimation of garbage objects
  - Different approaches are possible
    - Scavenger, mark&sweep, generational, etc.
- **Performance**
  - Limit the overhead, so run the GC rarely
  - Avoid growing the heap, so run the GC often enough
- **Correctness**
  - **Never detect and reclaim a live object**
- **Liveness**
  - **Detect and reclaim garbage faster than objects are allocated**

- Reference Counting
  - Each object is associated a counter
    - Counts the number of references on that object
  - Counter management
    - Happens on assigning reference
      - Decrement the count of the previously referenced object (if any)
      - Increment the counter of the newly referenced object
    - Applies to
      - Reference fields in objects as well as local variables and parameters
    - When a counter reaches zero
      - The object owning that counter is garbage

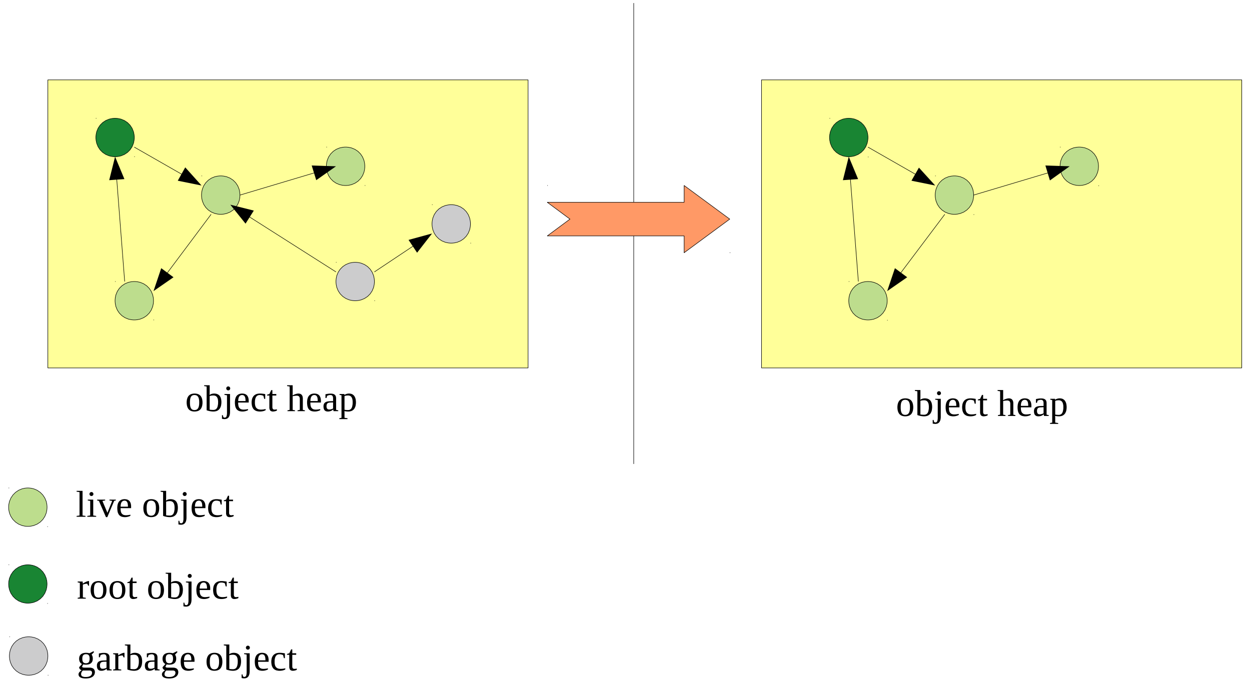


Java Platform – Garbage Collection

- Discussing Reference Counting
  - Problematic on multi-processors
    - Inherently incremental: impossible to run concurrently
    - Incrementing and decrementing require a critical section
  - Does not require to scan thread stacks
    - But requires to account for local variables and arguments
    - Introduces a high overhead (increment/decrement)
  - Extra paging
    - Accesses objects even if only references are manipulated
    - Dirties memory pages, potentially increasing the overhead of virtual memory paging
  - **Does not reclaim cycles**

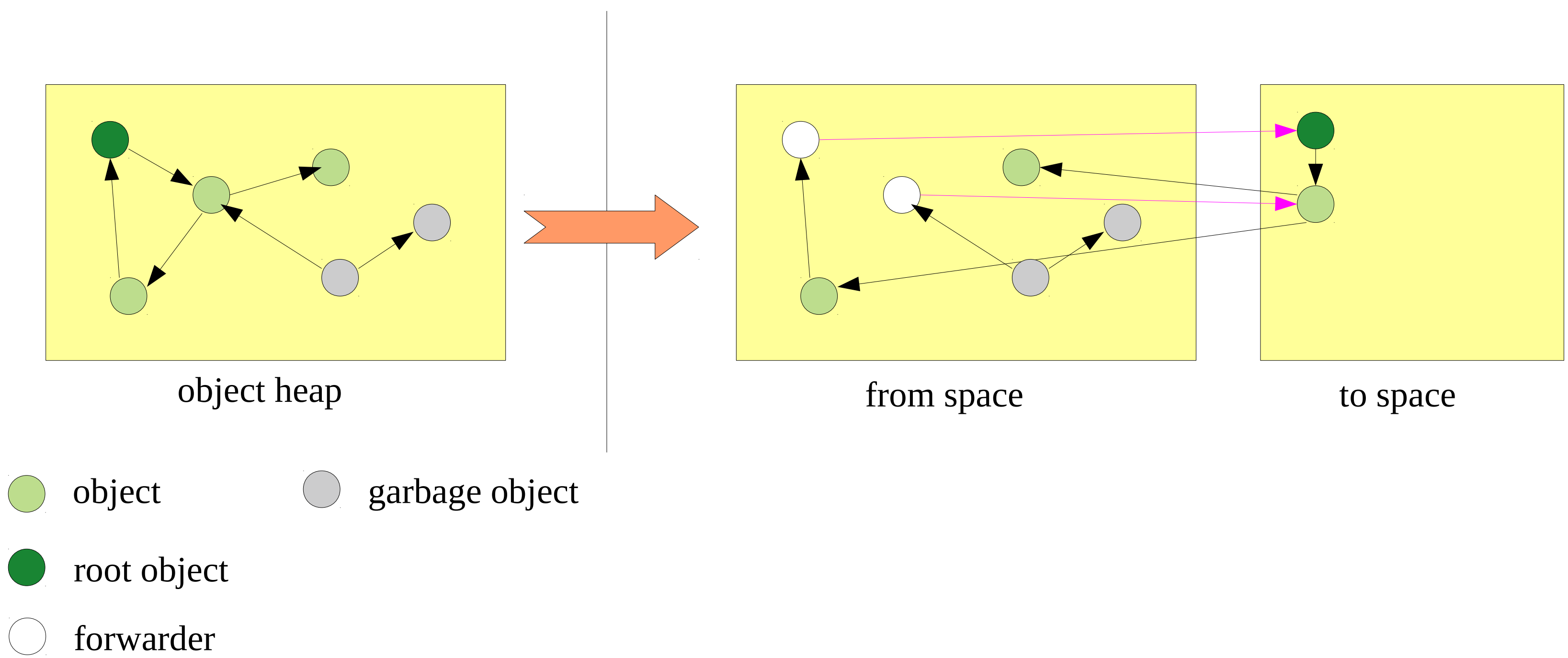
Java Platform – Garbage Collection

- Scavenger
  - Copying collector, using two spaces
    - Copy live objects from the old space to the new one
    - Discard the old space



Java Platform – Garbage Collection

- Scavenger details
  - Live objects are reachable from roots (thread stacks and class statics)
  - Leave a forwarder in-place of copied objects
    - Allows to detect cycles (correctness when copying)
    - As well as treat correctly shared objects
  - Use to-space as a recursion stack

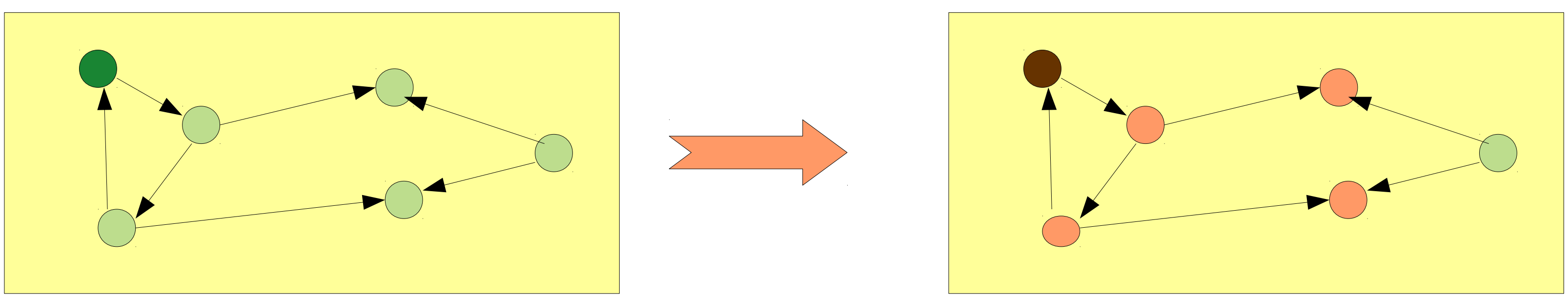


Java Platform – Garbage Collection

- Discussing Scavenger
  - Simple when designed as stop the world
    - A simple depth-first recursive walk of an object graph
    - Cycles are easily detected through forwarders
    - Require to scan thread stacks
  - Clustering objects
    - Depth-first scavenging produces efficient in-memory clustering of objects
  - Efficiency
    - Depends on the ratio of live versus garbage objects
    - Also depends on the cumulative size of live objects
    - The fewer live objects, the more effective
    - May lead to allocate twice the heap size

Java Platform – Garbage Collection

- Mark & Sweep
  - A two-phase garbage collection
    - A marking phase, coloring live objects
    - A sweeping phase reclaiming garbage objects (not colored)
  - Marking phase
    - Walks the refer-to graph from roots (thread stacks and class statics)
    - Carry the current color



object heap

object heap

object

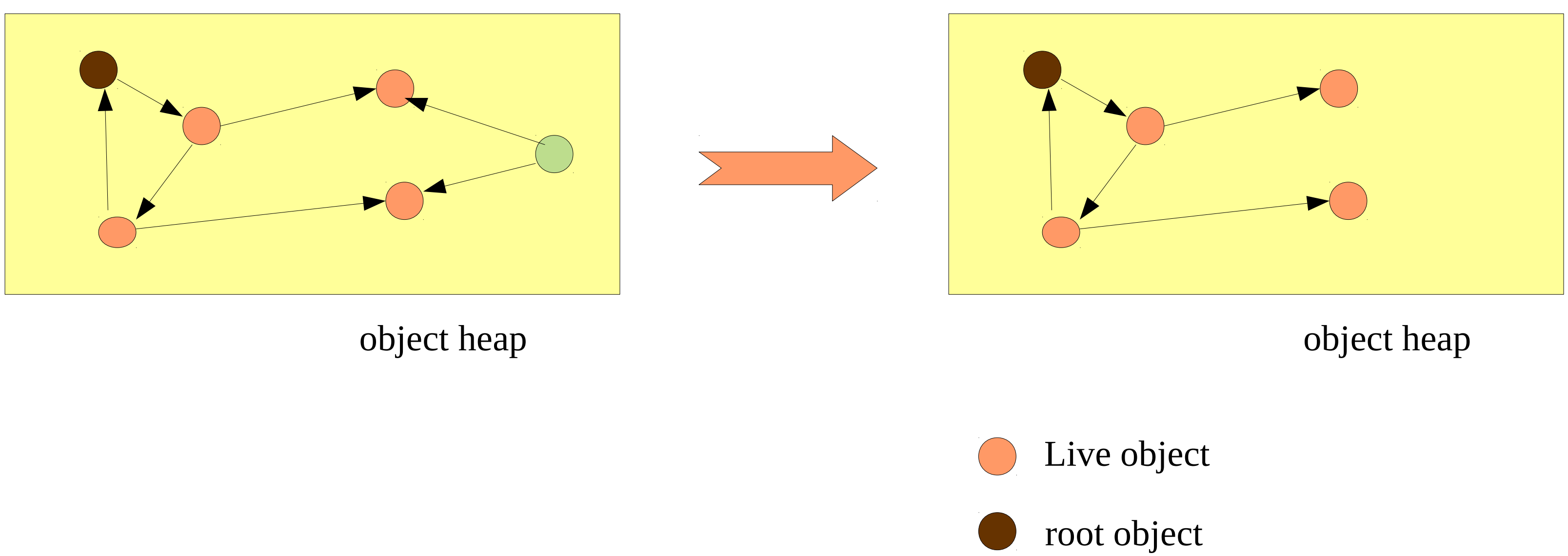
root object

Live object

root object

Java Platform – Garbage Collection

- Mark & Sweep
  - Sweep phase
    - Sweeps sequentially the object heap to discover garbage objects
  - Reclaiming garbage
    - Using free lists (non-compacting sweeping)
    - Compact as sweeping (challenging to maintain references)

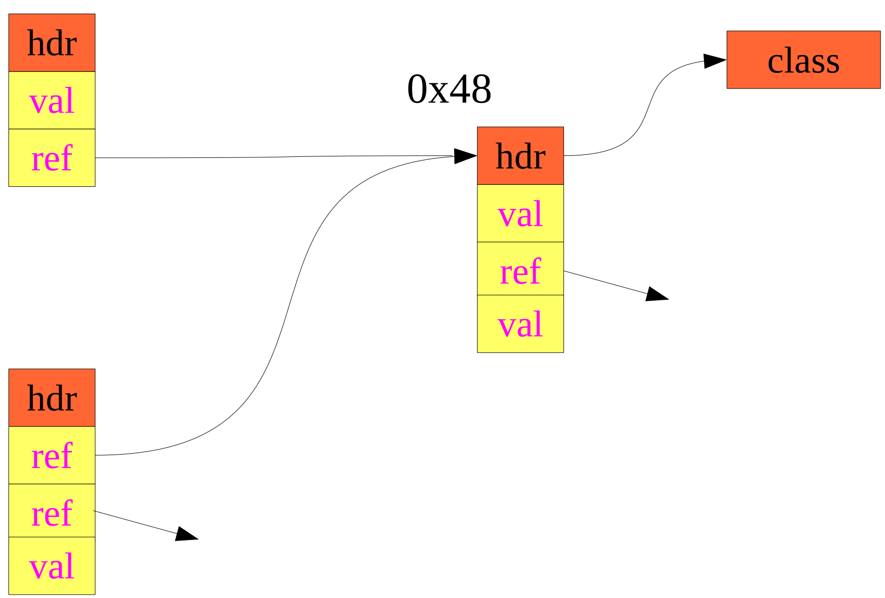


Java Platform – Garbage Collection

- Discussing Mark & Sweep
  - Not too sensitive to the live/garbage ratio
  - Requires to scan thread stacks
  - Caveats of free-list memory management
    - Can lead to traditional fragmentation
    - Costly allocation (different algorightms such as first-fit, best-fit, etc.)
  - Two scans of the object heap
    - One through references and the other sequentially
    - May lead to heavy paging activity if heap larger than main memory
    - It defeats the LRU policy of most virtual memory systems
- Compacting Mark&Sweep
  - Some mark&sweep do compact the heap during the sweep phase
  - Usually done by slidding objects, does not improve locality

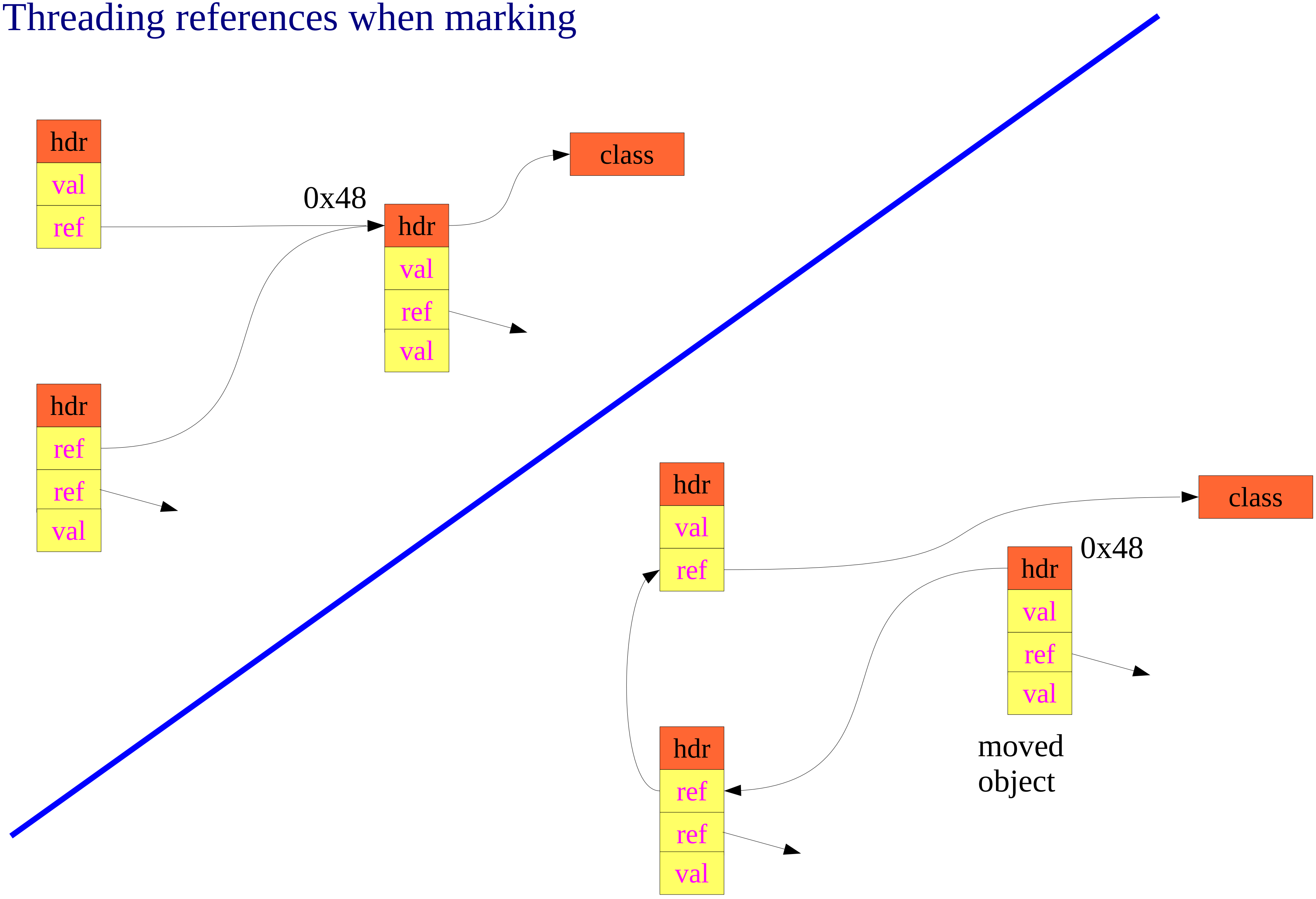
Java Platform – Garbage Collection

- Compacting Mark&Sweep
  - Usually done by sliding objects
  - Does not improve locality, but eliminates fragmentation
  - But how do we update references when sliding objects?
  - A possible design: threading references...



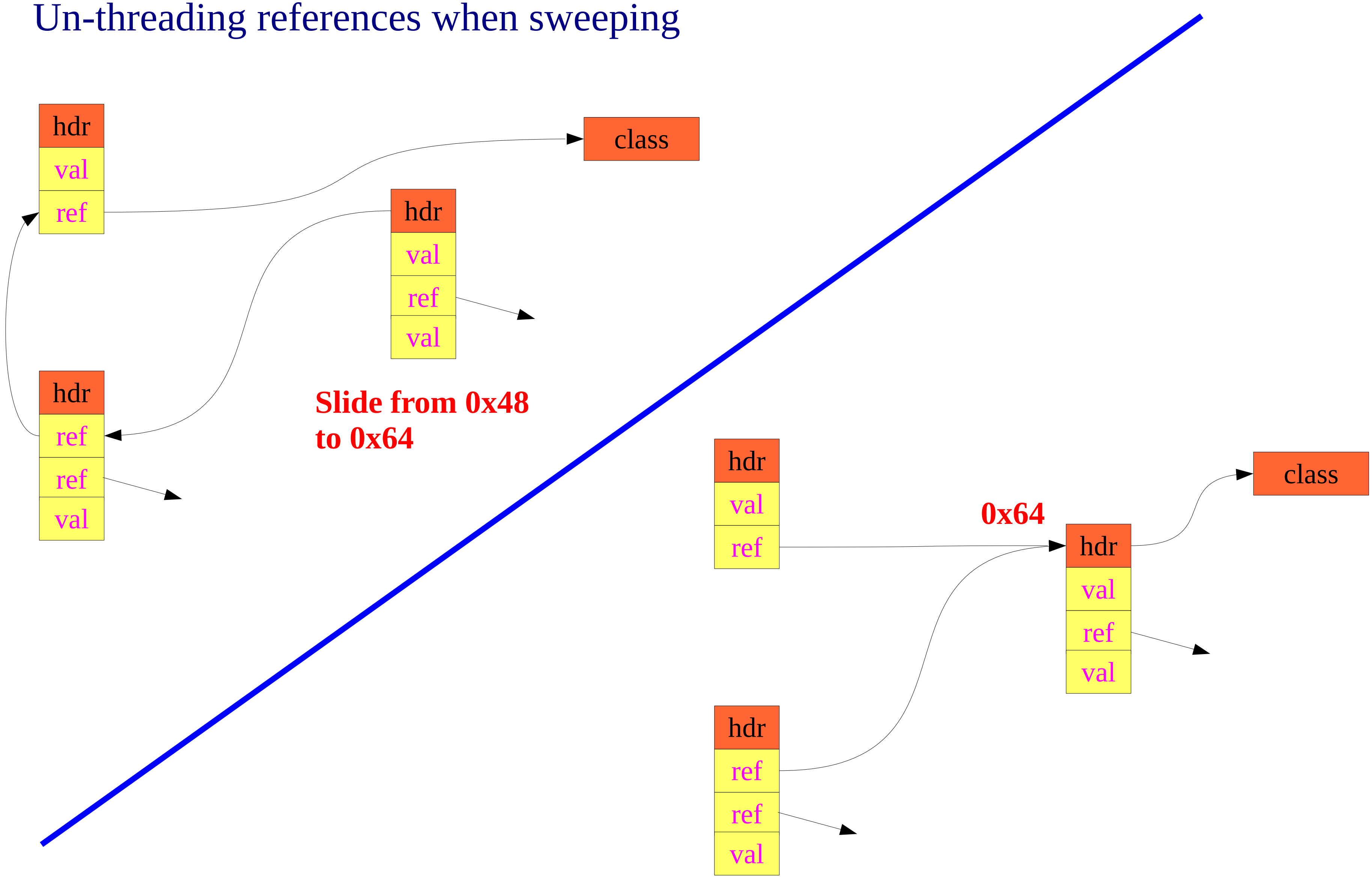
Java Platform – Garbage Collection

- Threading references when marking



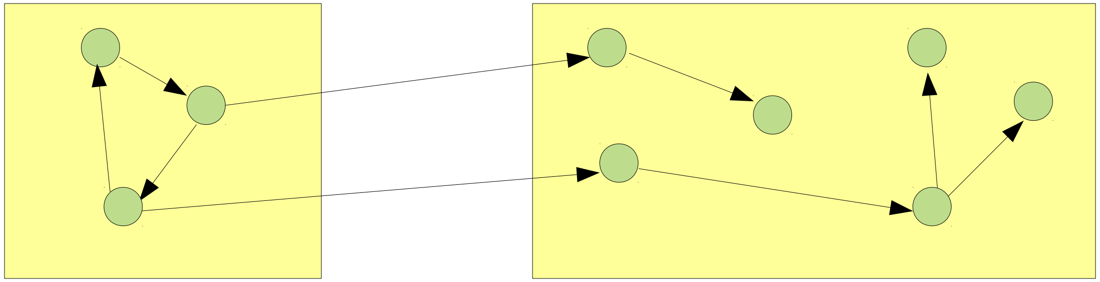
Java Platform – Garbage Collection

- Un-threading references when sweeping



Java Platform – Garbage Collection

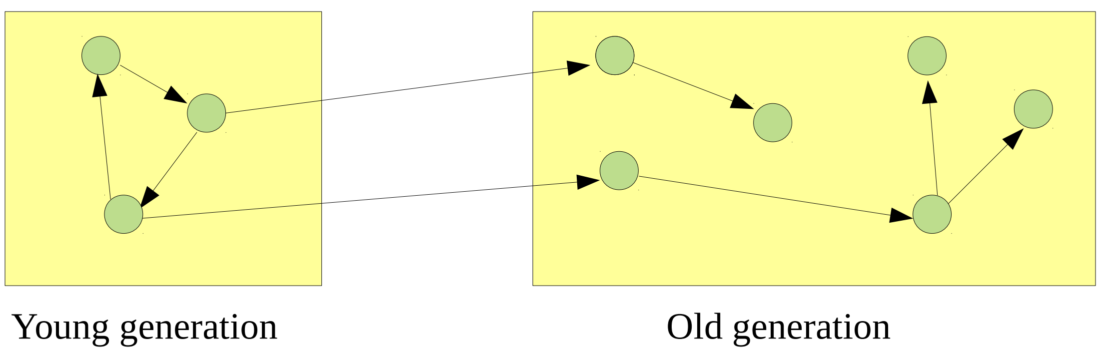
- Generational
  - Observation
    - Young objects die young
    - The older the object, the more likely it will survice a GC
  - Group objects per generation (old and new)
    - All objects are allocated in the new generation
    - Promote objects to older generations as they survive enough GC cycles
  - Goal
    - Garbage old generations less frequently...



Young generation   Old generation

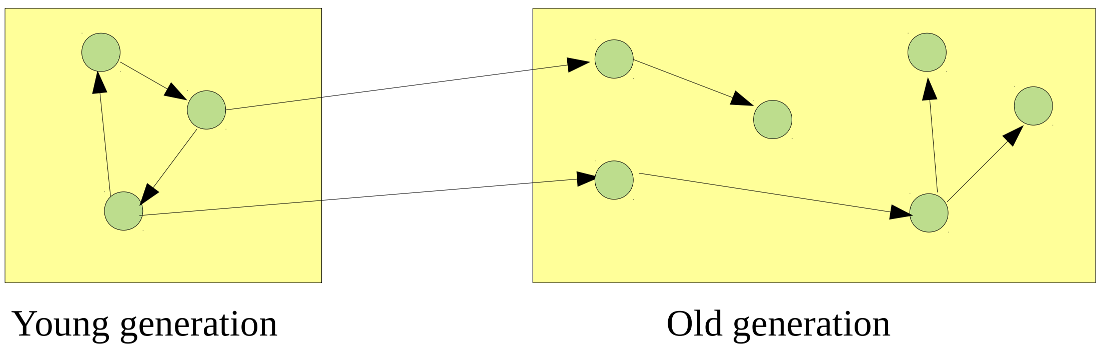
Java Platform – Garbage Collection

- Generational rules
  - Rule 1: all objects are allocated in the new generation
  - Rule 2: promote objects to older generations as they survive enough GC cycles
  - Rule 3: never allow references from the old generation to the young generation



Java Platform – Garbage Collection

- Generational
  - Garbage collect the young generation more frequently
  - Collect both every once in a while



Java Platform – Garbage Collection

- Discussing Generational
  - Local garbage collection can be scavenging or Mark&Sweep
    - Hence, requires to scan the thread stacks
    - Has the same pros and cons
  - Trade-off
    - Exploit that young objects die young
    - But promotion may be expensive
  - Promotion is challenging
    - Requires to leave forwarders for copied objects
    - Mutators have to check for forwarders (performance overhead)
    - Forwarders can be short-circuited through later garbage collections
    - Quite difficult in multi-threaded programs to ensure the atomic deep copy

Java Platform – Garbage Collection

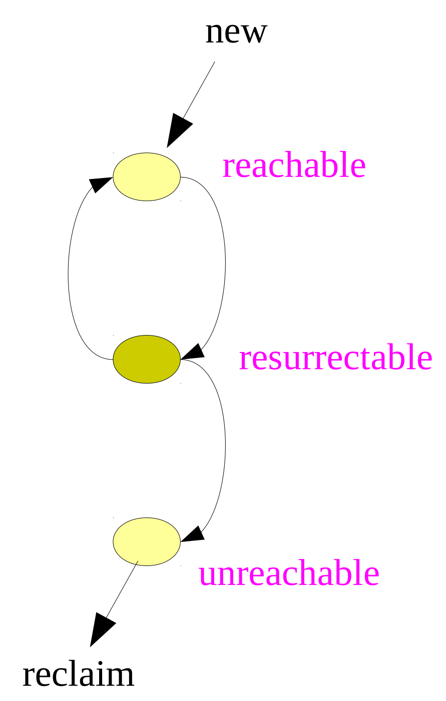
- Challenges
  - Scalability
    - Up to 150GB of heap space...
    - Even worse if we consider I/Os
  - Real-time behavior
    - Stop-the-world is an easier design for garbage collectors
    - Incremental garbage collectors are possible
  - **Memory leaks exist even with a garbage collector...**
    - In C++, leaks occur because developers forget to free objects
    - In Java, leaks occur because developers forget to forget references
    - The object cache nightmare...
  - Native resources...

Java Platform - Finalizers

- The problem
  - Java depends on a lot of native resources represented by objects
  - How does one free those resources?
- The finalize method
  - The object class defines a method *finalize()*
    - Any class may redefine this finalize method
    - A class that redefines its finalize method is said to have a ***finalizer***
  - When is it called?
    - The finalize method is called when the object is detected as being garbage
    - If the finalize method is not redefined, it is not called
    - However, the finalize method is called only once
  - Threads?
    - There is no guarantee about which thread is used to call finalize methods
    - But that thread does not hold any user-level Java monitor

Java Platform - Finalizers

- Finalizers introduces resurrection
  - **It is legal for a finalize method to make a garbage object live again**
  - Reminder: finalizers are called only once per object
  - Require to detect twice that an object is garbage
- Impacts garbage collection
  - Introduce a new state:
    - Reachable (live)
      - There is a path from roots to the object
    - Resurrectable
      - The object is not reachable
      - The object may be resurrected
      - All objects go through that state
    - Unreachable (garbage)
      - The object is not reachable
      - The object cannot be resurrected



Java Platform - Finalizers

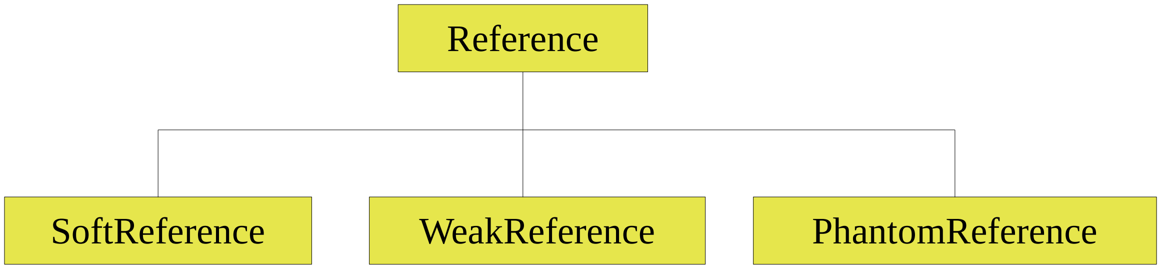
- Compatibility with GC algorithms
  - Compatible with reference counting
    - Easy to call the finalizer when the counts drop to zero
    - Easy to know that the object remained garbage
      - Counter still at zero after the finalizer run
    - But reference counting is rarely used in practice
  - Incompatible with scavenging
    - Reintroduces a sweep to find garbage objects with a finalizer
    - Never know when to free the from-space because of resurection
  - Mark&Sweep is well-suited
    - Easy to extend the sweeping phase to find objects with finalizers
      - But delays the actual reclamation of garbage objects
    - Still requires two marking phase to really know if an object is garbage

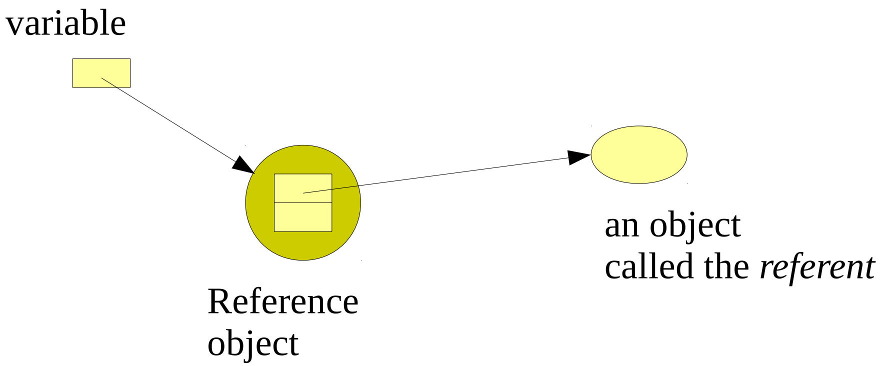
Java Platform - Objects

- Java Finalizers – complex and not enough
  - Native resources are often really scarce
  - Garbage collection is too asynchronous
  - So native resources are not freed fast enough
- Raising the GC frequency is difficult
  - Because it is most often stop-the-world
  - Because it represents an overhead
    - Marking the object graph
    - Sweeping the object heap
- Introduce explicit close/dispose operations
  - On Sockets, files
  - On Widget toolkits
  - Etc.

Java Platform - References

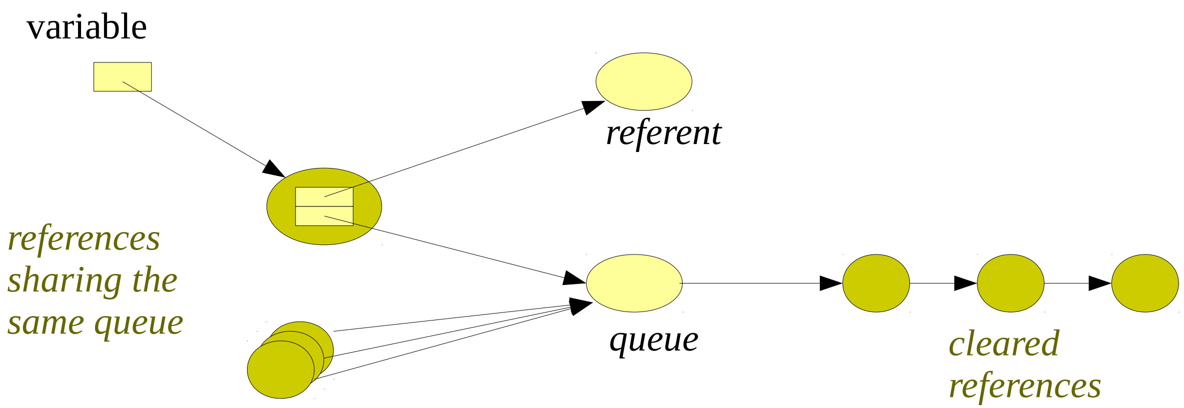
- The Problem
  - How does one write a cache in Java?
- Introducing different semantics for Java references
  - Strong references
    - The usual object references in the Java language
  - Weaker references in *java.lang.ref*
    - SoftReference and WeakReference
    - PhantomReference





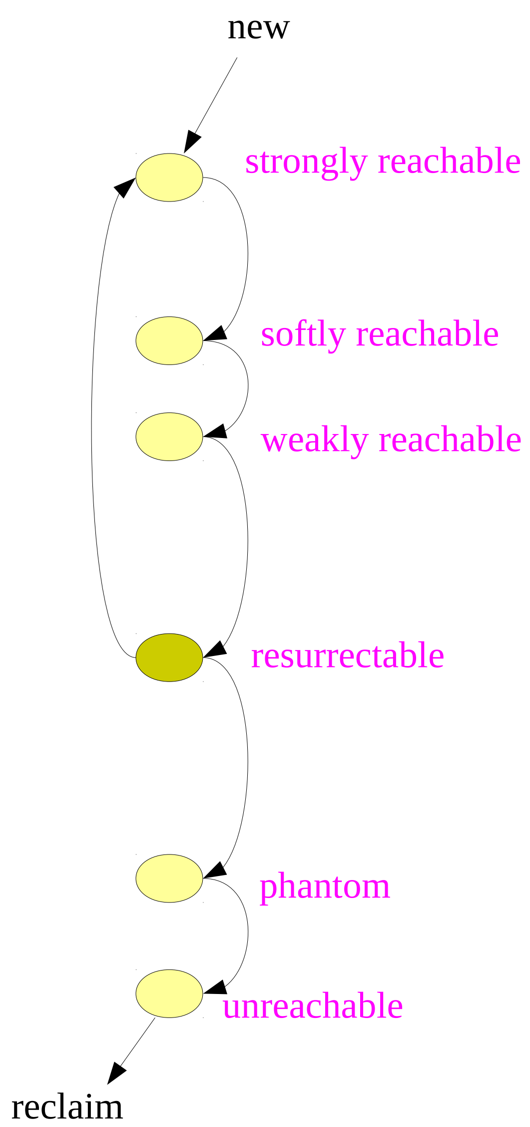
Java Platform - References

- Java References
  - Normal semantics for objects that are strongly reachable
    - If you do not use weaker references, nothing is different than usual Java
  - Weaker references are managed by the GC
    - When an object is no longer strongly reachable
    - The GC may **clear** weaker references to that object at any time
  - Notification
    - A reference may be associated to a reference queue (*ReferenceQueue* class)
    - Once the GC cuts a reference, it push that reference on its associated queue



Java Platform - References

- State changes
  - Reachable is detailed into
    - Strongly reachable
      - Reachable through strong references
    - Softly reachable
      - Not strongly reachable
      - Reachable through soft references
    - Weakly reachable
      - Neither strongly nor softly reachable
      - Reachable through weak references
  - Resurrectable
    - Only resurrectable through a finalizer as before
  - Unreachable
    - Phantom reachable
      - Not reachable but through phantom references
      - Such objects are not resurrectable
    - Unreachable
      - Entirely unreachable
      - Ready to be reclaimed



Java Platform - References

- Discussing soft versus weak references
  - Weak references
    - Weak references must be cleared by the GC as soon as the referenced object is weakly reachable (neither strongly or softly reachable)
    - Used for canonical mappings
      - Keep a mapping key to value
      - Clean the mapping as soon as the key is no longer used (reachable)
  - Soft references
    - Soft references must only be cleared by the GC before it raises an out-of-memory exception, but it may sooner
    - It is suggested that clearing soft references follows the policy:
      - Keep recently created and recently used soft references
    - Used for caching objects
      - A service provides an object
      - Clients keep a reference as long as they need to use the object
      - The GC only reclaims the object and cuts your soft reference if it needs memory

Java Platform - References

- Discussing phantom references
  - More powerful than just finalizers
    - Finalizers are called only once
    - So if objects are resurrected, finalizers can no longer be used for cleanups
  - Phantom references introduce post-mortem resource management
    - An object that is phantom-reachable can no longer be resurrected
    - It is therefore the absolute last moment to do some cleanup
- Cleared Reference
  - Sub-class the appropriate reference class (soft, weak, or phantom)
  - Add the info you need to the cleanup as fields in your reference subclass

Java Platform - Performance

- Let's discuss performance...
  - This is a complex subject because it is heavily related to the workload characteristics...
  - Macro or micro benchmarks? Neither is perfect
  - Beware of the tyranny of micro-benchmarks. Think in terms of 2s to open a window...
- Java macro characteristics
  - Footprint: from Java cards to huge servers (150GB of object heap)
  - Performance: from within 10% of hand-crafted C to dozens of times slower
  - Do not confuse Java semantics and the design of some specific virtual machine...
- Expressive power
  - Some say lower than C, some say way higher...
  - It is a matter of perspective, higher from a software engineering perspective

Java Platform - Performance

- What is the cost of emulation?
  - High with an interpreter, obviously
  - What about Just-In-Time or Ahead-Of-Time?
- Comparing C and Java
  - Can the generated code be as efficient?
  - Are we comparing apples and oranges?

Java Platform - Performance

- A lot of instructions compile the same
  - Arithmetic expression, branch, loops, …
  - Invoking static methods (equivalent to a function call)
  - Allocating objects (not so different than malloc)
  - Threads and monitors, usually implemented using pthreads
- Some instructions have more semantics
  - Field access includes NPE checks, many can be eliminated (JIT)
  - Array access includes NPE checks and bound checks, many can be eliminated (JIT)
  - Virtual method invocations, some may eliminated (JIT)
  - Runtime check-casts, hard to eliminate

Java Platform - Performance

- So, what if...
  - You trust your code, you could remove NPE and bound checks as well as runtime check-casts
  - You don't use polymorphic types, you could devirtualize method calls
- What about programming style?
  - Object-oriented programming promotes encapsulation which promotes small methods
    - Compile-time in-lining can be used, when the invoke can be de-virtualized
    - Polymorphism has been argued to improve the structure and maintainability of programs
  - Object-oriented programming promotes creating a lot of small objects
    - True, but this is also poor programming to abuse it
    - It depends on the application, some have easy to manage data structures, other do not
  - How should we compare malloc/free versus garbage collection?

Java Platform - Performance

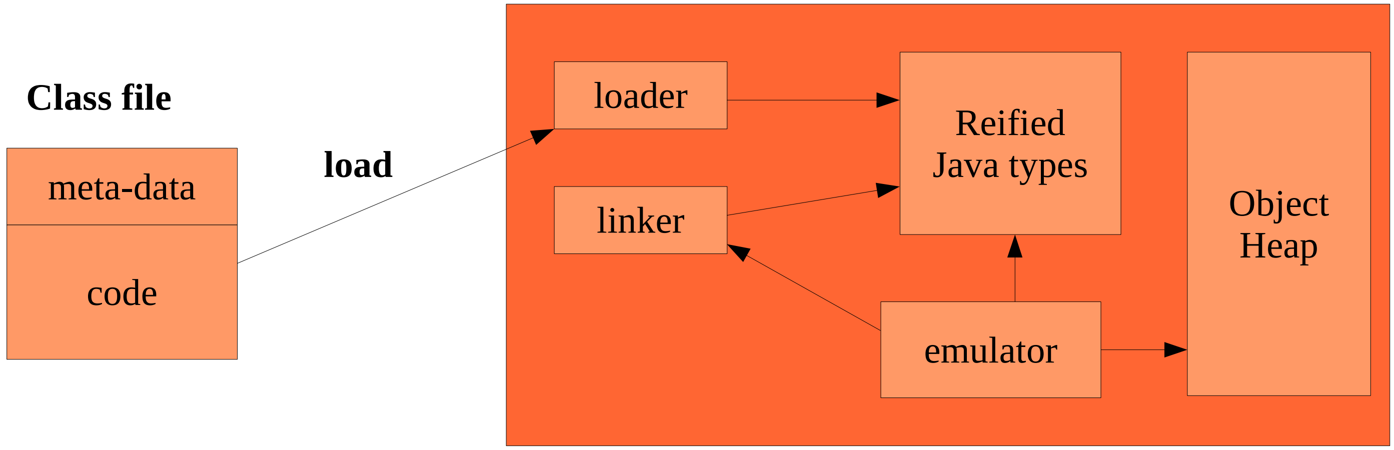
- So, why is Java slow?
  - We need to distinguish the language and the platform...
  - There are hidden costs... not always obvious to see...
- Garbage collection
  - This is not free, of course
  - It can be incremental (very short, frequent pauses)
  - It can be parallelized, could be very interesting on multi-core systems
  - This is hard stuff...
- Class loading
  - This is not free either, this is dynamic linking, bytecode verification, and JIT compilation
  - Verification can be turned off if you trust the source of your code
  - JIT compilation can be avoided by AOT compilation
  - Dynamic linking can be reduced using pre-linked formats (close to shared libraries)
  - Watch for the spaghetti plate effect in your libraries... leading to lazy class loading

Java Platform - Performance

- Are you sure it is Java that is slow?
  - Or could it be because of middleware frameworks we run above the JRE?
  - Or could it be because of sloppy programs written in Java?
  - Or could it be because so many Java code is automatically generated by tools?
- A little bit of all the above points...
  - But most importantly, because it can...
  - In reality, because it could...
  - Slower improvements in hardware and tighter energy budget are game changers...
  - New JVM implementations and cleaner JREs are appearing for embedded devices...
  - Java can even be found in hard and soft real-time environments...

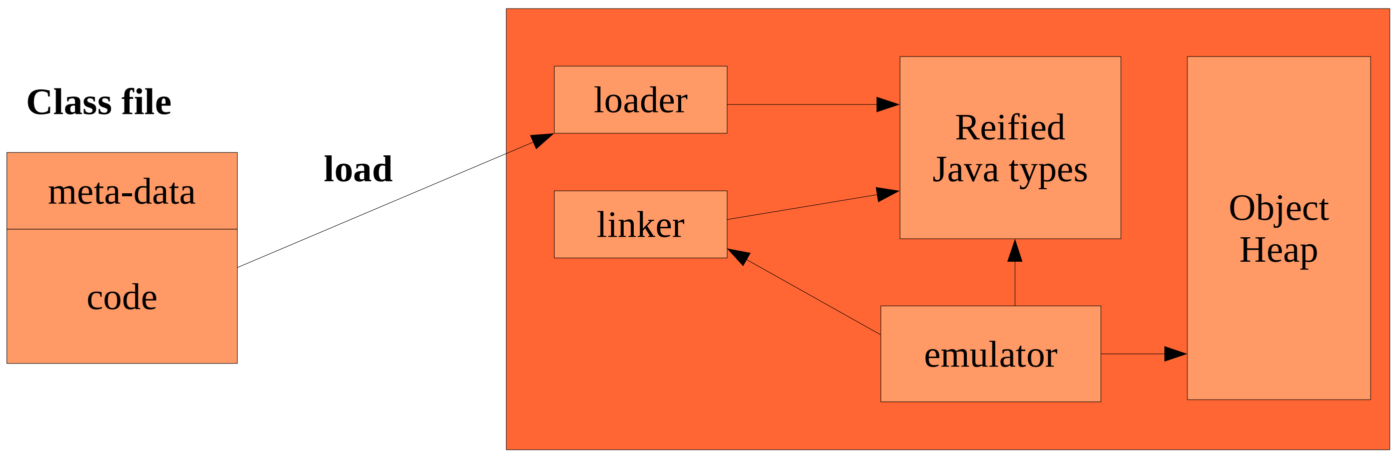
Carte Blanche Study

- Embedded world
  - Java bytecode, on single core, no MMU
  - How would you emulate?
  - Discuss your expected performance
  - How do you load and link code?
  - Would you provide threads or events?
  - Would you rely on cooperative scheduling or pre-emptive?



Carte Blanche Study

- Embedded world
  - Java bytecode, on single core, no MMU
  - Discuss the differences between an object heap and regular memory?
  - How would you design your object-oriented memory?
  - Discuss the performance of object-oriented memory
  - Do you need an MMU?
  - How would/could you use an MMU?



Carte Blanche Study

- Embedded world
  - Java bytecode, **multiple core**, **with** MMU
  - How would/could you use multiple cores?
  - Revisit your execution model to account for multiple cores