CS263: Runtime Systems
Lecture: High-level language virtual machines

Part 1 of 2

Chandra Krintz
UCSB Computer Science Department

Mobile, OO Execution Model

- Execution model embodied by recent PL Implementations
  - Java, all .Net languages, scripting languages
  - Object-oriented
    - Encapsulation and data hiding, inheritance, polymorphism
    - Static methods – static dispatch; virtual methods – dynamic dispatch
      - When same name is used (polymorphism): hidden vs overridden
    - Static fields and instance fields
      - When same name is used (polymorphism): hidden

Mobile, OO Execution Model

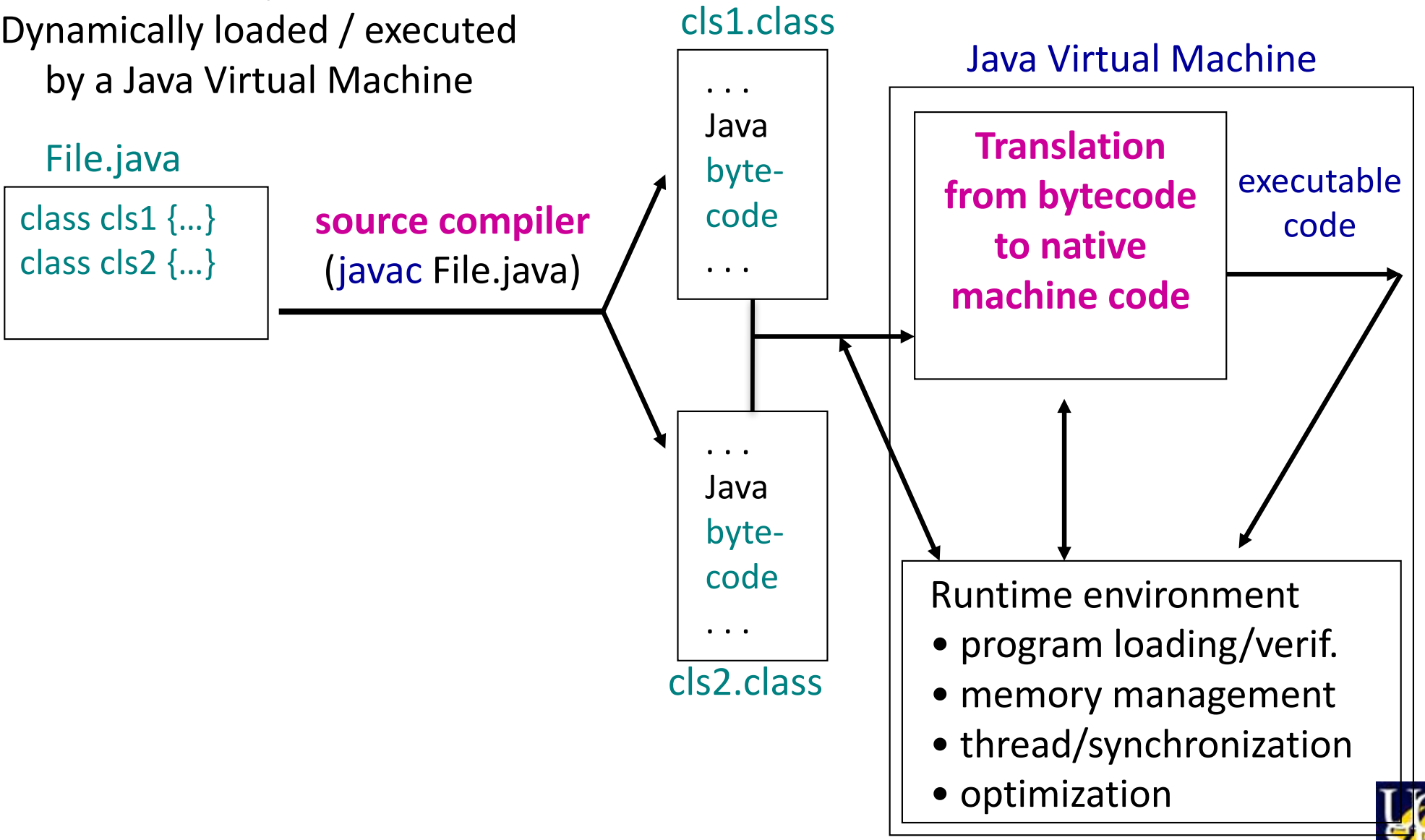
- Execution model embodied by recent PL Implementations
  - Java, all .Net languages, scripting languages
  - Object-oriented
    - Encapsulation and data hiding, inheritance, polymorphism
    - Static methods – static dispatch; virtual methods – dynamic dispatch
      - When same name is used (polymorphism): hidden vs overridden
    - Static fields and instance fields
      - When same name is used (polymorphism): hidden
  - Program is portable and assumed to be mobile
    - **Architecture-independent representation of code and data**
    - Incrementally loaded (transfer unit = class, assembly, archive (jar))
      - Data translated from file/class to VM's internal representation
        - » Class model and object model
    - Code is translated to native code interleaved with execution
      - Instruction level (interpretation), method level (JIT compilation), or path level (trace compilation)

Priciple of Laziness!

Java Classfiles

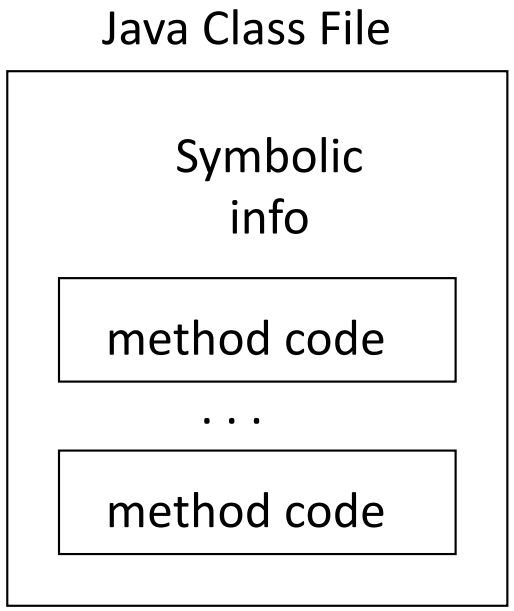
- architecture-independent
- architecture-dependent

- Architecture-independent
- Format called bytecode
- Dynamically loaded / executed by a Java Virtual Machine



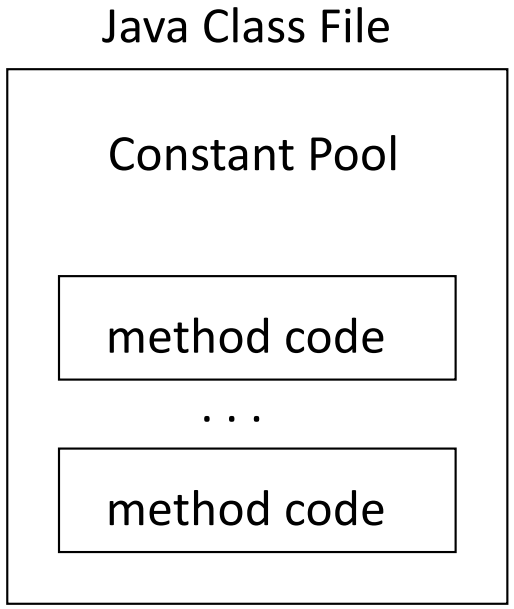
A Java Program (Compiled Java Source)

- Class files (generated from source using a source compiler)
- Each class file is in a binary bytecode format
- Each class file contains – in **a fixed, well-defined format**
  - Symbolic information (Metadata in MSIL/.Net binary files)
    - All of the necessary data to execute the class file
  - Instructions (code) in the JVM ISA format



A Java Program (Compiled Java Source)

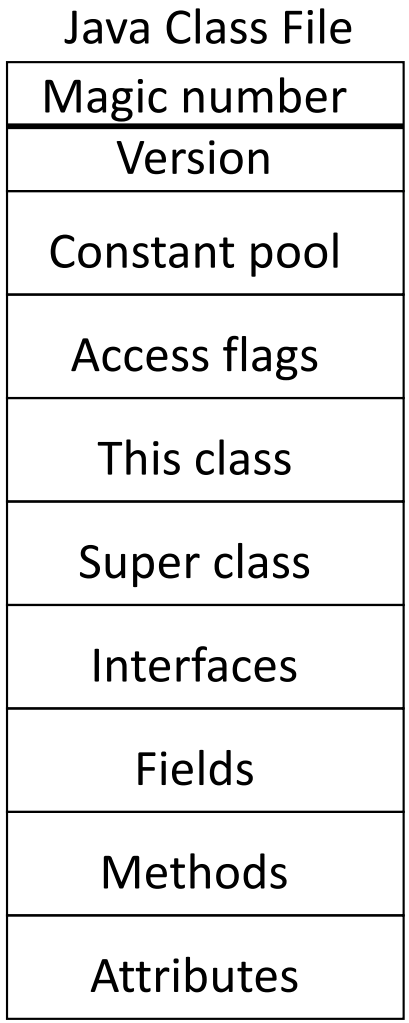
- Class files (generated from source using a source compiler)
- Each class file is in a binary bytecode format
- Each class file contains – in **a fixed, well-defined format**
  - Symbolic information (Metadata in MSIL/.Net binary files)
    - All of the necessary data to execute the class file
  - Instructions (code) in the JVM ISA format



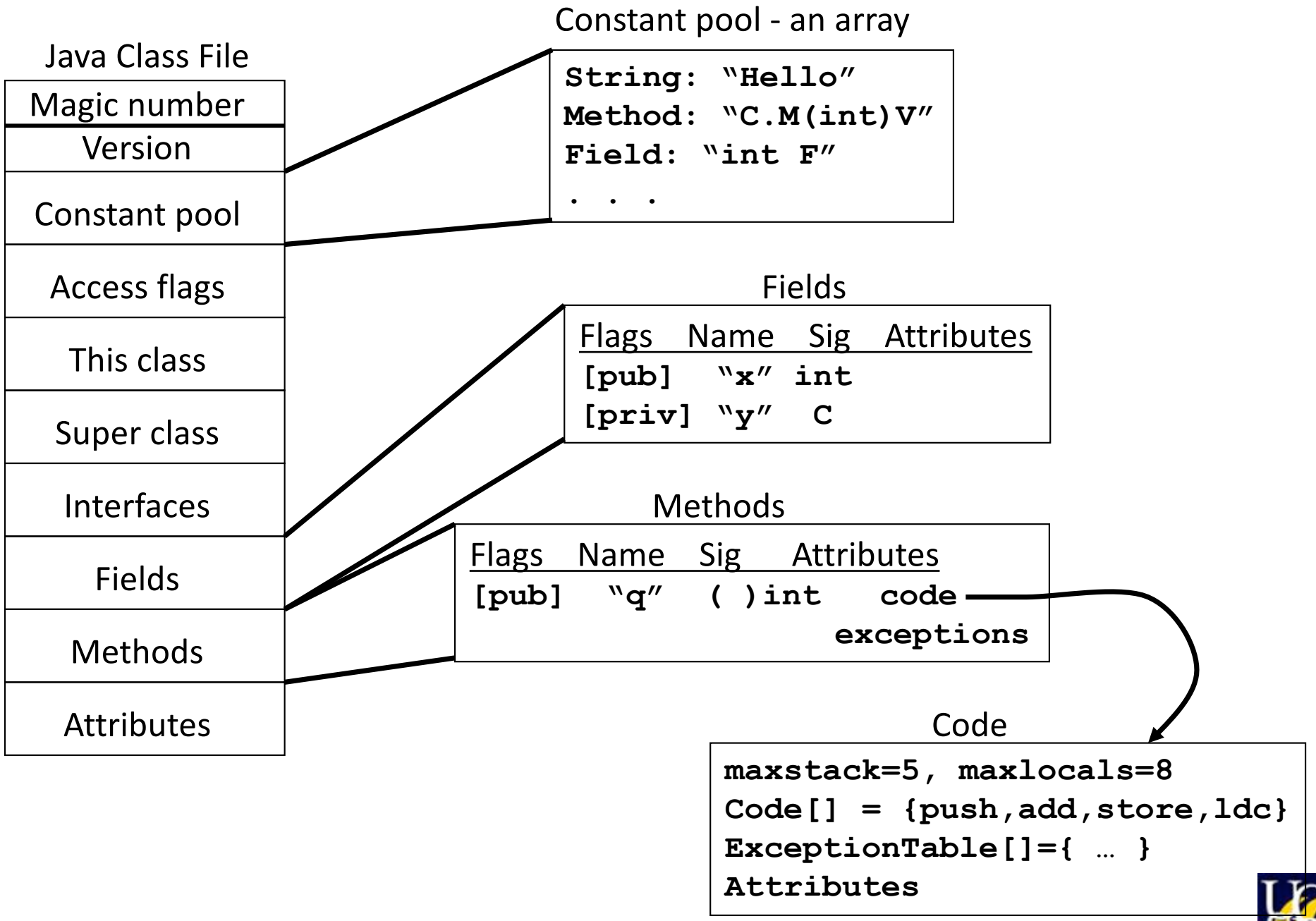
Instead of repeating all of the symbol names throughout, we store all of the symbols in a table (**the constant pool**) and then have everything that uses the symbol (code, other data structures) use a reference into the table instead

Types (packages, classes, primitives), names, and also method and field data structures

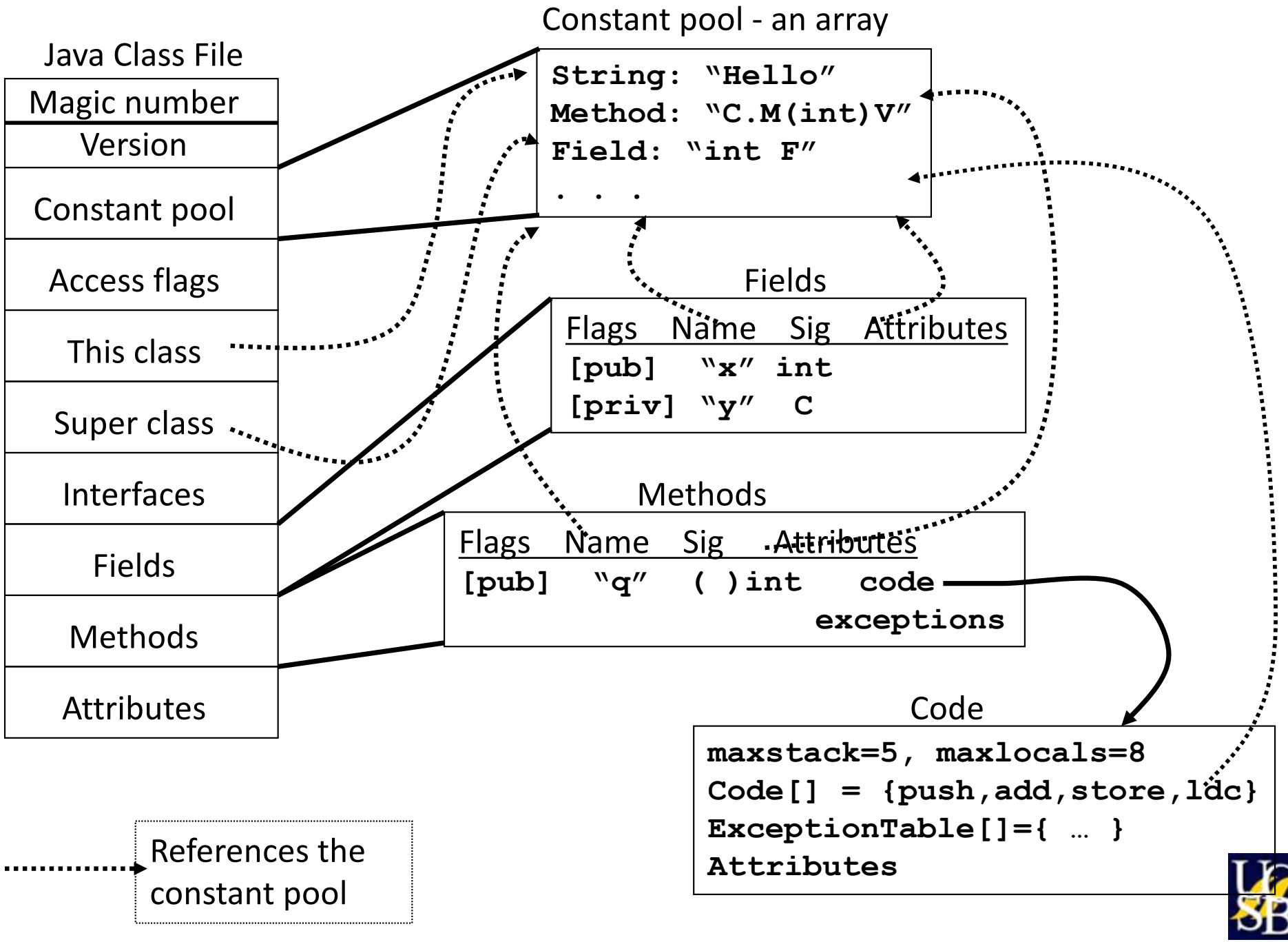
The Java Class File Format (Symbolic Info)



The Java Class File Format (Symbolic Info)

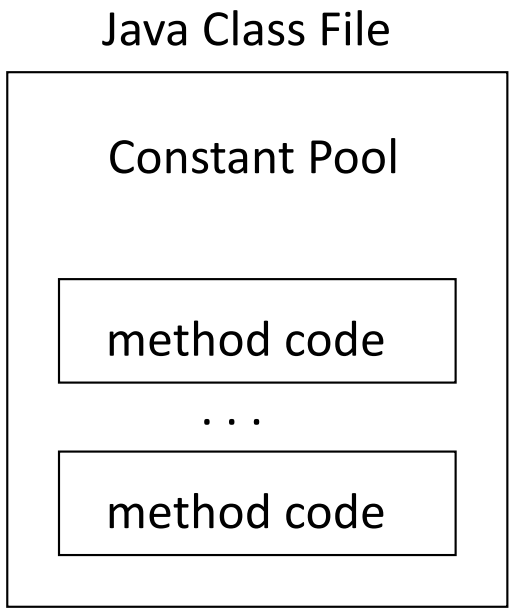


The Java Class File Format (Symbolic Info)



A Java Program (Compiled Java Source)

- Class files (generated from source using a source compiler)
- Each class file is in a binary bytecode format
- Each class file contains – in **a fixed, well-defined format**
  - Symbolic information (Metadata in MSIL/.Net binary files)
    - All of the necessary data to execute the class file
  - Instructions (code) in the JVM ISA format



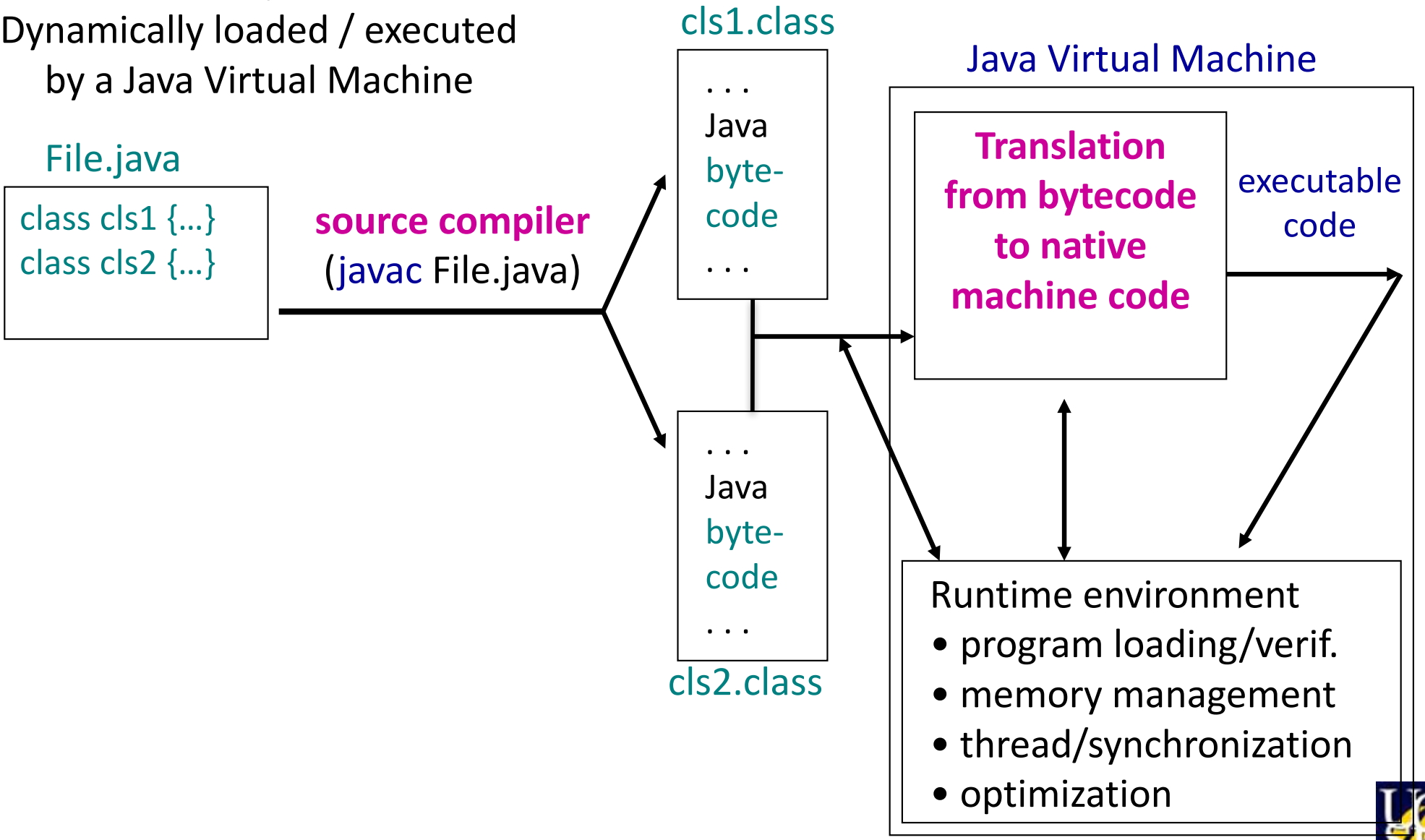
When a class file is read into memory by the runtime, it is stored in a representation internal to the runtime.

The runtime keeps a global table which Refers to all these structures, and a map that records which indexes hold which structure. E.g. all internal class representations, static fields, and static methods, are stored in this global table (sometimes called the statics table).

Java Classfiles

- architecture-independent
- architecture-dependent

- Architecture-independent
- Format called bytecode
- Dynamically loaded / executed by a Java Virtual Machine



Java Bytecode Instructions

Code

```
maxstack=5, maxlocals=8
Code[] = {push,add,store,ldc}
ExceptionTable[]={ … }
Attributes
```

- Load/store
- Arithmetic
- Type conversions (int to long, etc)
- Object creation (objects/arrays/multi-dim arrays)
- Object manipulation (get/put fields, check properties)
- Operand stack manipulation
- Control transfer (conditionals/unconditionals)
- Method invocation and return
- Exception and finally implementation
- Synchronization

- http://docs.oracle.com/javase/specs/jvms/se7/html/index.html
- …/html/jvms-4.html#jvms-4.10.1.4

Java Bytecode Instructions

Code

```
maxstack=5, maxlocals=8
Code[] = {push,add,store,ldc}
ExceptionTable[]={ … }
Attributes
```

- Load/store
- Arithmetic
- Type conversions (int to long, etc)
- Object creation (objects/arrays/multi-dim arrays)
- Object manipulation (get/put fields, check properties)
- **Operand stack manipulation**
- Control transfer (conditionals/unconditionals)
- Method invocation and return
- Exception and finally implementation
- Synchronization

- http://docs.oracle.com/javase/specs/jvms/se7/html/index.html
- …/html/jvms-4.html#jvms-4.10.1.4

Operand Stack

- Do not confuse this with the user/runtime/system stack!
  - User stack: manages local variable scope for functions/methods at runtime
  - **Operand** stack: holds **operands** so that instructions can be executed
  - Both are last-in-first-out (LIFO) queue data structures

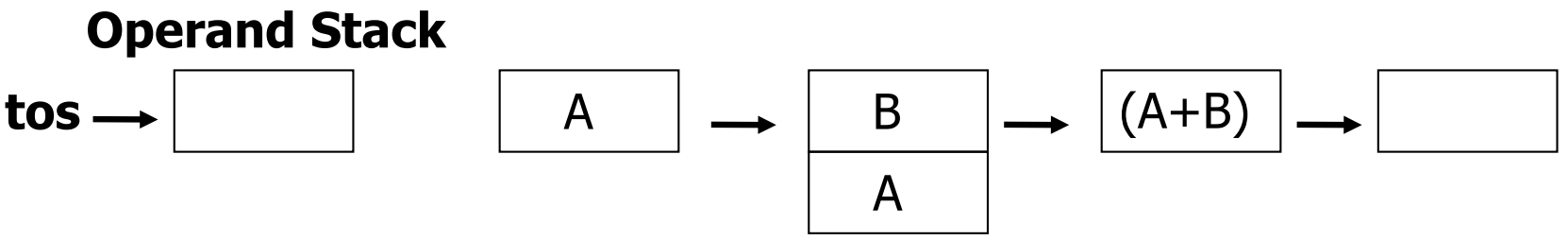
- Most modern VM languages execute a stream of instructions using an operand stack
  - Requires that the virtual instruction set architecture (ISA) implements a "stack machine"

Stack-based ISA

- Operands are implicit - on the top of stack (tos)
- Compiler simplification (no register allocation)
- Compact instruction encoding

C = A + B

```
push A          //push A on tos
push B          //push B on tos
add             //add the to elements at the tos
pop C           //pop the tos and put the value in C
```



Used in hardware in old HP calculators
Use in most modern language runtimes to enable program portability
JavaCard implements Java bytecode ISA in hardware; as does Azul Vega processor

The Alternative: Register-based ISA

- Operands are explicit (ie specified *in* the instruction)
- **Register-register** (aka load-store)
  - Memory accesses via load & store instructions only

C = A + B **(RISC)**

```
load R1, A              //load memory at &A into reg1
load  R2, B             //load memory at &B into reg2
add R3,R1,R2            //add reg1+reg2 and store result in reg3
store C, R3             //store value in reg1 into memory at &C
```

- **Register-memory**
  - Operands can be taken from registers or memory directly

C = A + B **(CISC)**

```
add  C, A, B          // load memory from each address
                // perform operation
                // store result in memory
```

Java Bytecode Instructions

- Can manipulate data (operands) in following formats: **typed instrs**

	instr. starts with
– Integers (32-bits)	i
– Longs (64-bits)	l
– Floats (32-bits)	f
– Doubles (64-bits)	d
– Shorts (16-bits)	s
– Bytes (8-bits)	b
– Object references (32-bits)	a
– Char (16-bits)	c
– Arrays	

- Opcode + data - variable length instructions
  - Opcode: 1-byte wide (256 possible)
  - Data: zero or more values to be operated on (**operands**)

Operands are 1-byte each so
For a 2-byte index you need:
(operand1 << 8) | operand2

Java Bytecode Instructions

- Refer to method parameters and local variables by number
  - 256 is the maximum
  - **wide** version of instruction extends this to 65536
  - EX: ***istore 2*** pops tos and puts value in local variable number 2
- Java stack (operand stack)
  - Holds the operands during operations
  - All (virtual) computation performed here
  - 32-bits wide; longs/doubles take 2 stack entries

```
maxstack=5, maxlocals=8
Code[] = {push,add,store,ldc}
ExceptionTable[]={ … }
Attributes
```

Two per method data structures:

Local variable array (max index is 64K-1):

Index: 0 1 2 3 4 5

Holds params followed by local vars

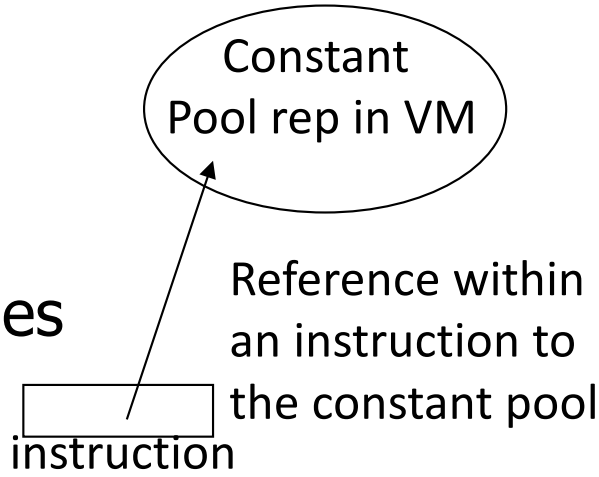
Operand stack

Grows/shrinks with instructions executed

Each method gets its own

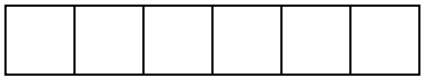
Java Bytecode Instructions

- Refer to method parameters and local variables by number
  - 256 is the maximum
  - **wide** version of instruction extends this to 65536
  - EX: ***istore 2*** pops tos and puts value in local variable number 2
- Java stack (operand stack)
  - Holds the operands during operations
  - All (virtual) computation performed here
  - 32-bits wide; longs/doubles take 2 stack entries



Two per method data structures:

Local variable array (max index is 64K-1):



Index: 0 1 2 3 4 5

Holds params followed by local vars

Operand stack

Grows/shrinks
with instructions
executed

Each method gets its own

http://www.cs.ucsb.edu/~cs263/lectures/constantpoolexample.pdf

```
class Simple {
  static int field1;
  static Simple field2;
  static int field3;
  public static void foo() {}

  public static void
    main(String args[]) {
      int i = args.length;
      int j = 1000027;
      field1 = i+j;
      field2 = null;
      foo();
  }
}
```

```
public static void
main(java.lang.String[]);
  Code:
   0: aload_0
   1: arraylength
   2: istore_1
   3: ldc   #2; //int 1000027
   5: istore_2
   6: iload_1
   7: iload_2
   8: iadd
   9: putstatic  #3; //Field
                     //field1:I

   12: aconst_null

   13: putstatic #4; //Field
                     //field2:LSimple;

   16: invokestatic #5;
                     //Method foo:()V
   19: return

}
```

http://www.cs.ucsb.edu/~cs263/lectures/constantpoolexample.pdf

```
class Simple {
  static int field1;
  static Simple field2;
  static int field3;
  public static void
     foo(int k) {}

  public static void
    main(String args[]) {
      int i = args.length;
      int j = 1000027;
      Simple.field1 = i+j;
      Simple.field2 = null;
      Simple.foo(7);
  }
}
```

When a method call is made (using line-by-line interpretation): Runtime moves arguments to correct location in caller's register (locals) array

```
public static void
main(java.lang.String[]);
  Code:
    0: aload_0
    1: arraylength
    2: istore_1
    3: ldc   #2; //int 1000027
    5: istore_2
    6: iload_1
    7: iload_2
    8: iadd
    9: putstatic  #3; //Field
                      //field1:I

   12: aconst_null

   13: putstatic #4; //Field
                     //field2:LSimple;

   16: bipush 7
   17: invokestatic #5;
                     //Method foo:(i)V
   20: return

}
```

http://www.cs.ucsb.edu/~cs263/lectures/constantpoolexample.pdf

```
class Simple {
  static int field1;
  static Simple field2;
  static int field3;
  public static void
     foo(int k) {}

  public static void
    main(String args[]) {
      int i = args.length;
      int j = 1000027;
      Simple.field1 = i+j;
      Simple.field2 = null;
      Simple.foo(i);
  }
}
```

When a method call is made (using line-by-line interpretation): Runtime moves arguments to correct location in caller's register (locals) array

```
public static void
main(java.lang.String[]);
  Code:
    0: aload_0
    1: arraylength
    2: istore_1
    3: ldc   #2; //int 1000027
    5: istore_2
    6: iload_1
    7: iload_2
    8: iadd
    9: putstatic  #3; //Field
                      //field1:I

   12: aconst_null

   13: putstatic #4; //Field
                     //field2:LSimple;

   16: ???
   17: invokestatic #5;
                     //Method foo:(i)V
   20: return

}
```

http://www.cs.ucsb.edu/~cs263/lectures/constantpoolexample.pdf

```
class Simple {
  static int field1;
  static Simple field2;
  static int field3;
  public static void
     foo(int k) {}

  public static void
    main(String args[]) {
      int i = args.length;
      int j = 1000027;
      Simple.field1 = i+j;
      Simple.field2 = null;
      Simple.foo(i);
  }
}
```

When a method call is made (using line-by-line interpretation): Runtime moves arguments to correct location in caller's register (locals) array

```
public static void
main(java.lang.String[]);
  Code:
    0: aload_0
    1: arraylength
    2: istore_1
    3: ldc   #2; //int 1000027
    5: istore_2
    6: iload_1
    7: iload_2
    8: iadd
    9: putstatic  #3; //Field
                      //field1:I

   12: aconst_null

   13: putstatic #4; //Field
                     //field2:LSimple;

   16: iload_1
   17: invokestatic #5;
                     //Method foo:(i)V
   20: return

}
```

Instance fields and methods

```
public class Hello {
  int field1;
  int getField() { return field1; }
  public static void main(String args[]) {
    System.err.println("Hello World!");
    Hello obj = new Hello();
    obj.field1 = 7;
    int i = obj.getField();
  }
}
```

When a method call is made (using line-by-line interpretation): Runtime moves arguments to correct location in caller's register (locals) array

```
public static void main(java.lang.String[]);
  Code:
    0: getstatic     #2       // Field java/lang/
                              // System.err:Ljava/io/
                              // PrintStream;
    3: ldc           #3       // String Hello World!
    5: invokevirtual #4       // Method java/io/
                              // PrintStream.println:
                              //(Ljava/lang/String;)V
    8: new           #5       // class Hello
   11: dup
   12: invokespecial #6       // Method "<init>":()V
   15: astore_1
   16: aload_1
   17: bipush        7
   19: putfield      #7       // Field field1:I
   22: aload_1
   23: invokevirtual #8       // Method getField:()I
   26: istore_2
   27: return
}
```

Interesting bytecodes and method dispatch specializations

```
class A {
   static int field1= 4;
   int field2 = 3;
   int field3 = 7;
   static int field4;
   static void m1() {}
   void m2() {}
   void m3() {}
   A() {}
   static void main(String args[]) {
      B.foo();  }
}
```

```
class B extends A {
   static int fielda= 4;
   int field2;
   int fieldc = 2;
   void m2() {…}
   static void m4() {…}
   final void m5() {…}
   B() {…}
   static void foo() {
      A tmpA1 = new A();
      A tmpA2 = new B();
      tmpA2.m2();
   }
}
```

```
javac A.java
java A          //what is the output?
```

Interesting bytecodes and method dispatch specializations

```
static void foo();
  Code:
    0: new           #9  // class A
    3: dup
    4: invokespecial #6 // Method A."<init>":()V
    7: astore_0
    8: new           #10    // class B
   11: dup
   12: invokespecial #11  // Method "<init>":()V
   15: astore_1
   16: aload_1
   17: invokevirtual #12 // Method A.m2:()V
   20: return
```

```
class B extends A {
   static int fielda= 4;
   int field2;
   int fieldc = 2;
   void m2() {...}
   static void m4() {...}
   final void m5() {...}
   B() {...}
   static void foo() {
      A tmpA1 = new A();
      A tmpA2 = new B();
      tmpA2.m2();
   }
}
```

java A – output:

```
class A default constructor
class A default constructor
class B default constructor
class B instance method m2()
```

Bytecode ISA

- JVM
  - Typed instructions
  - Opcode: 1-byte wide (253 are used)
  - Data: zero or more values to be operated on (**operands**)
- MSIL
  - Typed instructions
  - Opcode is 2-bytes (64K possible)
- Python
  - 113 opcodes (42 with arguments and 71 without)

- All use an operand stack for one or more of their operands
- Translator must translate this ISA to native code
  - Interpreter (line-by-line translation): calls functions for each instr
  - Compiler: translates methods or code blocks to native all at once

Constant Pool Handout and Exercise

- http://www.cs.ucsb.edu/~cs263/lectures/constantpoolexample.pdf
- http://www.cs.ucsb.edu/~cs263/lectures/hw1.pdf

```
//dump the bytecode (shows constant pool entries resolved)
javap -c Simple          //Simple must be in your classpath

//http://www.cs.ucsb.edu/~cs263/showme.tar.gz
//cs263 dir must be in your class path
//dump constant pool (so you can resolve the entries yourself)
java cs263.ShowMeTheStructure Simple.class
```

JVM instructions:
http://cs.au.dk/~mis/dOvs/vmspec/VMSpecIX.fm.html

CS263: Runtime Systems
Lecture: High-level language virtual machines

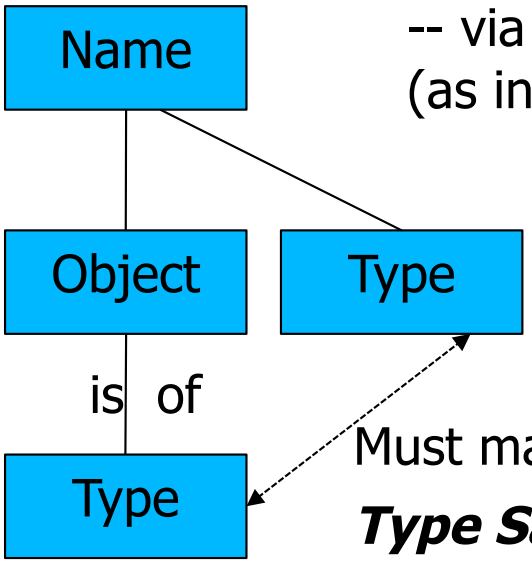
Today: Part 2 of 2

Chandra Krintz
UCSB Computer Science Department

Static vs Dynamic Typing

STATIC

Variable



Type is associated with variable and known at compile time
-- via explict specification or ***type inference***
(as in Scala, C# v3+, ML, Swift, Go)

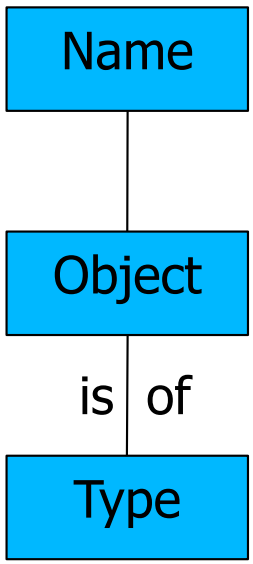
Must match (be compatible)

Type Safe if this and operations on types are checked (statically if possible, dynamically required for some operations) and legal according to the type system.

A **type system** generally tries to guarantee that operations are not used with values for which that operation does not make sense.

DYNAMIC

Variable



Type is associated with value and not known until **runtime when the code executes**

Checked at runtime

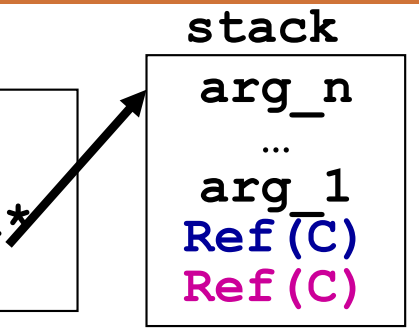
Java Object Allocation and Construction

- Object initialization - 2 step process
  - **new C** - creates a new object instance of class C
    - Instance fields in the object are set to default values
  - **invokespecial C:<init>...** invokes one of C's constructors
    - Can initialize fields to non-default values or perform any arbitrary computation
  - JVM specification requires that this protocol be respected
    - The object returned by new is uninitialized and cannot be used until the initialization method is invoked on it **& returns normally**
    - Premature use is type-safe since default values are used
    - However, inconsistencies can arise if an object is accessible before the init method has completed
  - Added difficulty, <init> methods **don't return an object**
    - However, instructor signature (type) shows a void: <init>()V

Object Initialization and Verification

```
x = new C();
```

```
new C
dup
*compute arguments (if any) & put them on the stack*
invokespecial C.<init>(arg_1,...,arg_n)V
```



- invokespecial doesn't return an object
  - Operates by **side effect**, updates object pointed to by reference

- Two references (hence the dup instruction) are needed
  - The topmost is "consumed" by invokespecial
  - The second reference
    - Holds the reference to the initialized object (after invokespecial)
    - This object goes from uninitialized (before the invokespecial) to initialized (after it)
      - Initializer needs not return an object

Homework 1

- Different source compilers implement the same semantics for the same thing

```
aload_1          aload_1
dup              aload_1
```

- Constructor methods are instant methods that are statically dispatched!
- You need not add super() to your constructors (it happens for you)

```
Par(){
  super();  // only add this if you want a different super, e.g. super(int);
   field3 = 7;
}
```

- The "this" object for instance methods becomes the first argument

```
Simple obj = new Simple();
int i = obj.instMethod(72, "cat");      //args: Simple, int, String
                                        //signature: instMethod(int,LString;)I
//you know its an instance method bc of invokevirtual & attributes
```

Mobile, OO Execution Model

- Execution model embodied by recent PL Implementations
  - Java, all .Net languages, scripting languages
  - Object-oriented
  - Program is portable and assumed to be mobile
    - Architecture-independent representation of code and data
    - **Incrementally loaded (transfer unit: class, assembly, jar)**
      - Data translated from file/class to VM's internal representation
        - » **Class model and object model**
    - Code is translated to native code interleaved with execution
      - Instruction level (interpretation), method level (JIT compilation), or path level (trace compilation)

Priciple of laziness!

Dynamic Class File Loading

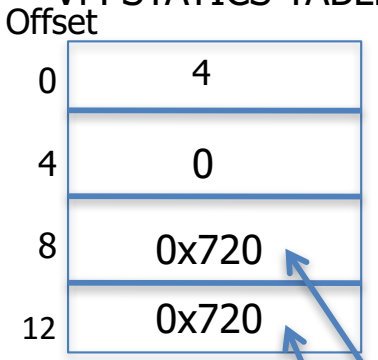
- Lazy: load incrementally as classes are used by the program
- Convert data to internal representation in memory
  - Upon access, read in the class file or assembly
    - Check that its structure is valid
      - Check whether code is type-safe if language requires it
    - Convert symbols (Java = constant pool) to internal data structures
  - Linking: **give target symbols memory locations**

Dynamic Class File Loading

- Lazy: load incrementally as classes are used by the program
- Convert data to internal representation in memory
  - Upon access, read in the class file or assembly
    - Check that its structure is valid
      - Check whether code is type-safe if language requires it
    - Convert symbols (Java = constant pool) to internal data structures
  - Linking: **give target symbols memory locations**
  - Table of all static fields and methods of all classes (statics table)
    - Including internal class representations and their maps
  - Keep a map of where you put everything
    - Statics table map
  - Symbol table for names/constants

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
}
```

VM STATICS TABLE (Address stored in R7)



VM STATICS MAP (stored in known location too, say 0x1234):

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12

Temporary place-holder address for all method bodies before they are *compiled* -- called a **stub**

For interpreter-only this will ahe address of the bytecode array of the method in memory

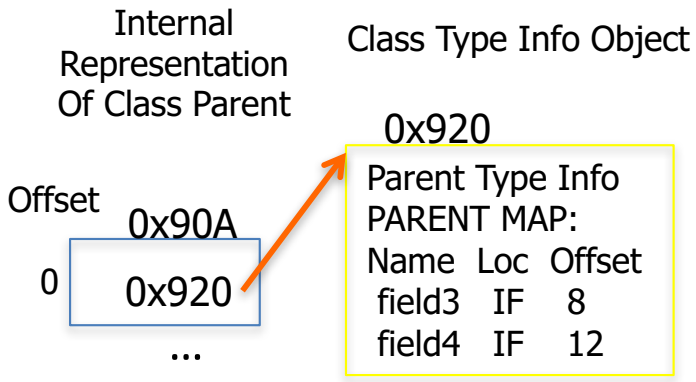
```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	0x920
20	0x90A

VM STATICS MAP (at 0x1234):

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Parent.Type		16
Parent.Rep		20



```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
      int i = 999999;
   }
   static void test2() {
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	0x920
20	0x90A
24	999999

VM STATICS MAP (at 0x1234):

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Parent.Type	Type	16
Parent.Rep	Rep	20
Parent.const1	O	24

All other constants from the constant pool of a class go here too!

Internal Representation Of Class Parent

Offset	0x90A
0	0x920
...	

Class Type Info Object

0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12

Review: Static (Class) Methods & Fields

- Most simple solution: One big table (statics/symbol table) in the runtime
  - Typically, a runtime/class loader populates this as classes are loaded incrementally
    - Keeps a map of symbols → offsets
  - As the translator translates instructions, it **uses the statics table map** (symbol table) of an accessed class to obtain the offset of the field or method that the code accesses
    - The addresses of which are stored in the statics table
  - For statically or dynamically typed languages

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
      Parent.test2(); //invokestatic
   }
   static void test2() {
      Parent.field2=7; //putstatic
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	0x920
20	0x90A
24	999999

VM STATICS MAP (at 0x1234):

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Parent.Type	Type	16
Parent.Rep	Rep	20
Parent.const1	O	24

Internal Representation Of Class Parent

Offset	0x90A
0	0x920
	...

Class Type Info Object

0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12

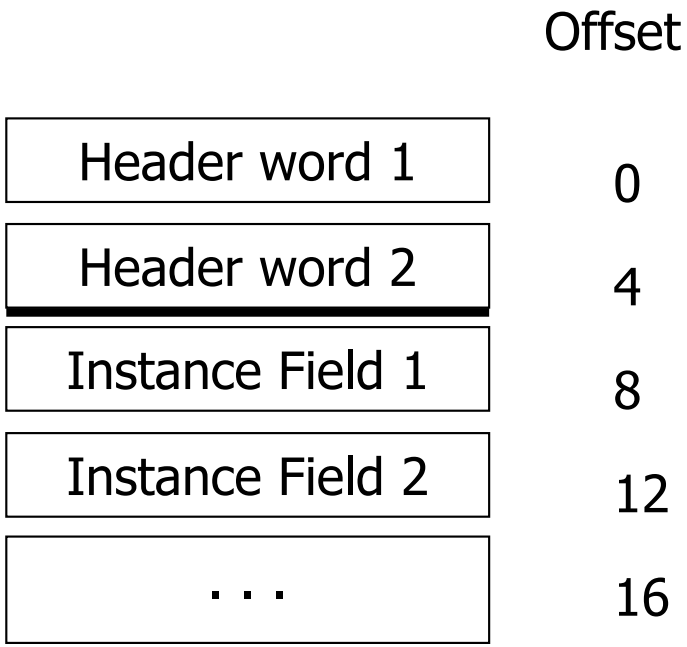
Review: Instance Fields

- If the language is statically typed
  - Use the static type or cast to figure out which field it is
    - If there is a Parent-Child relationship (OO hierarchy)
    - And there is a field of the same name in both
      - Both are available in the child object
      - Use different variable/types (or use casts) to get to the one you want
- If the language is dynamically typed
  - Do a hashtable lookup **by name** at runtime
    - Hashtable per object

- **Lookup gives an index into the object**
  - All instance fields stored in the object

Object Layout (Java/C#)

- The first 2 words of an object is the header



VM Support of Static OO PLs

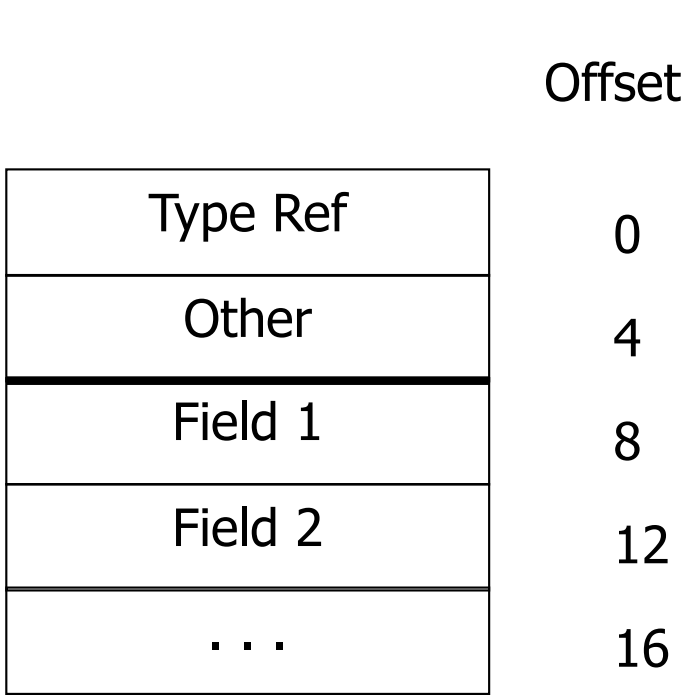
- Object model
  - Specifies the layout of the object in memory
  - Header: holds information for the runtime
    - For lookup of class (type)
      - For access to lookup tables (dispatch tables, hash tables (python/ruby))
    - For garbage collection (if any)
    - For locking
      - Store info about whether there is a lock on the object, whether it is contended for, etc.
    - For hashcodes
      - Unique identifier per object in the system
      - The object's identity

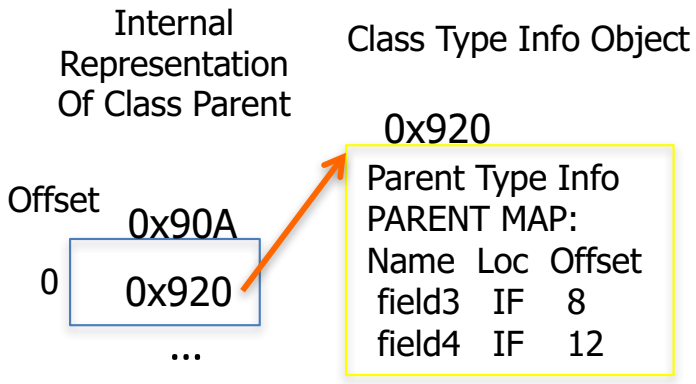
VM Support of Static OO PLs

- Object model
  - **Specifies the layout of the object in memory**
    = Internal Class Representation
  - Header: holds information for the runtime
    - For lookup of class (type)
      - For access to lookup tables (dispatch tables, hash tables (python/ruby))
    - For garbage collection (if any)
    - For locking
      - Store info about whether there is a lock on the object, whether it is contended for, etc.
    - For hashcodes
      - Unique identifier per object in the system
      - The object's identity

Object Layout (Java/C#)

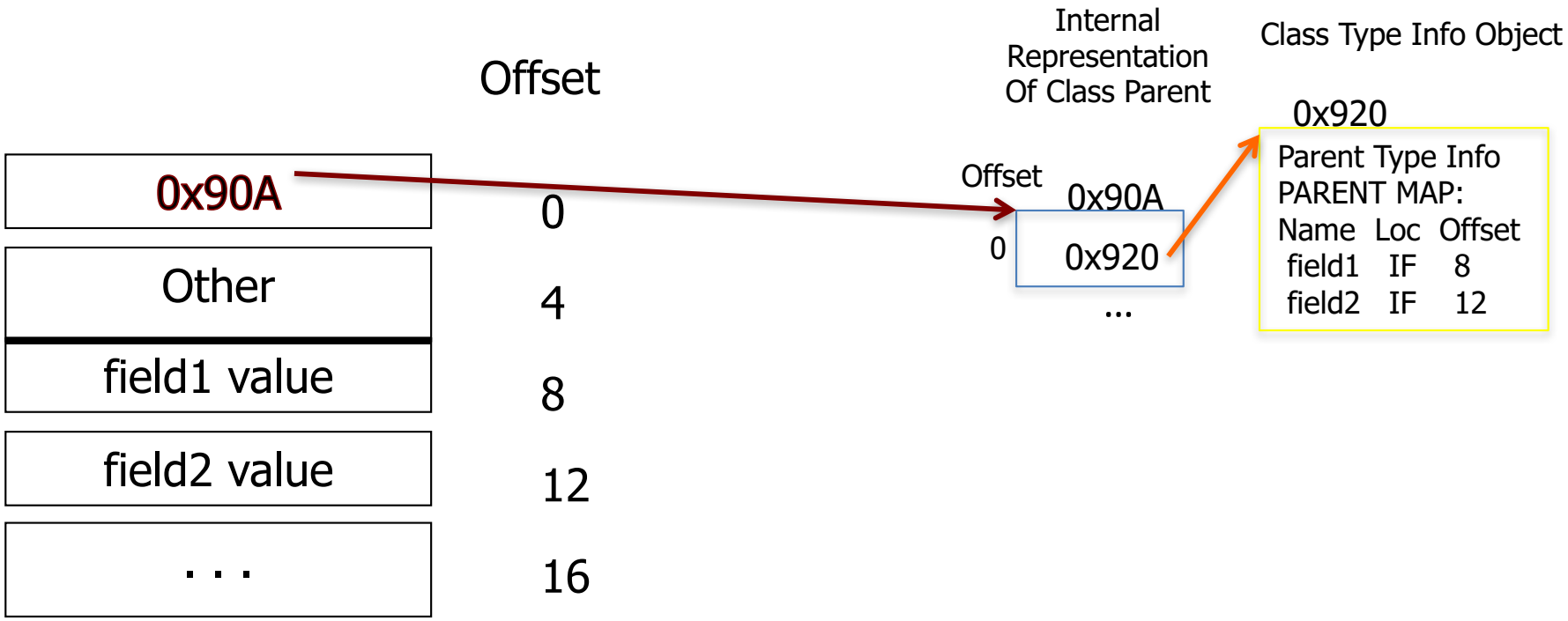
- The first 2 words of an object is the header
- Type Ref - type information block (1word ptr to class)
  - Entry in static array that holds a reference to **class object**
    - Reference to the internal representation of the class





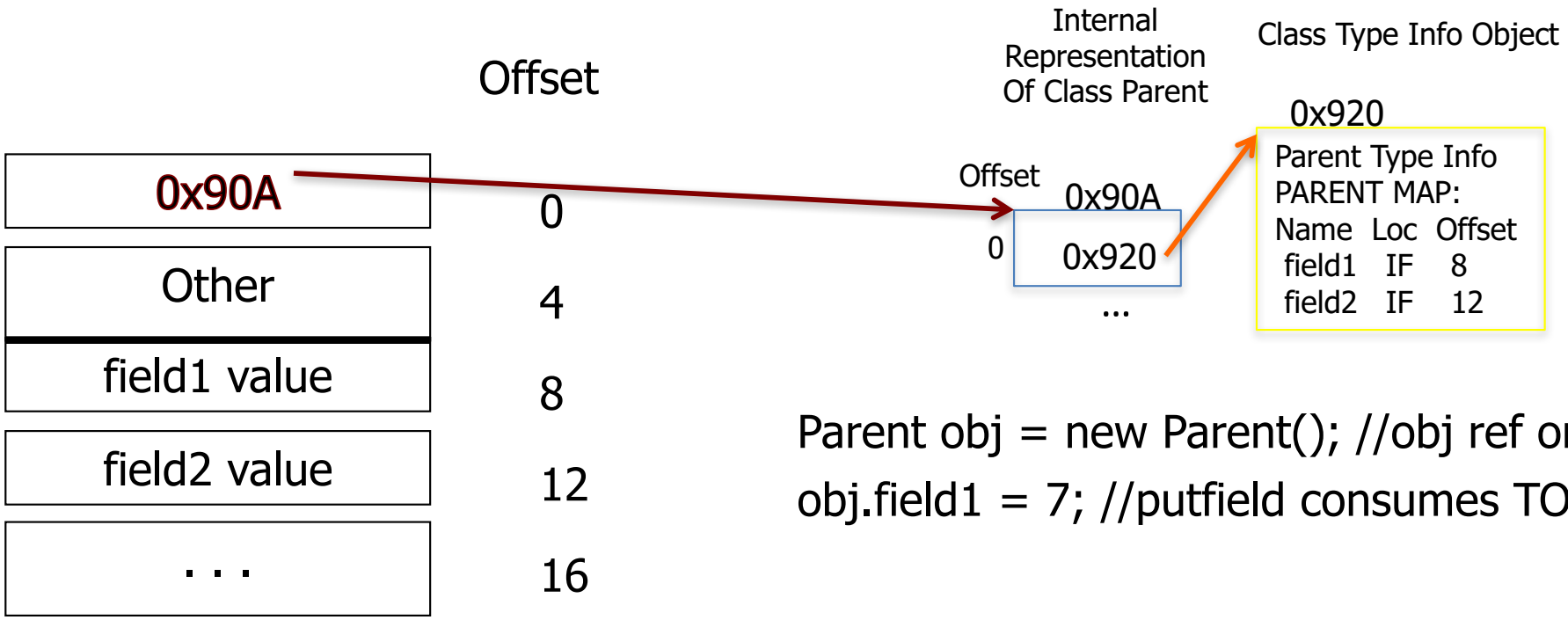
Object Layout (Java/C#)

- The first 2 words of an object is the header
- Type Ref - type information block (1word ptr to class)
  - Entry in static array that holds a reference to **class object**
    - Reference to the internal representation of the class



Object Layout (Java/C#)

- The first 2 words of an object is the header
- Type Ref - type information block (1word ptr to class)
  - Entry in static array that holds a reference to **class object**
    - Reference to the internal representation of the class



Parent obj = new Parent(); //obj ref on TOS

obj.field1 = 7; //putfield consumes TOS

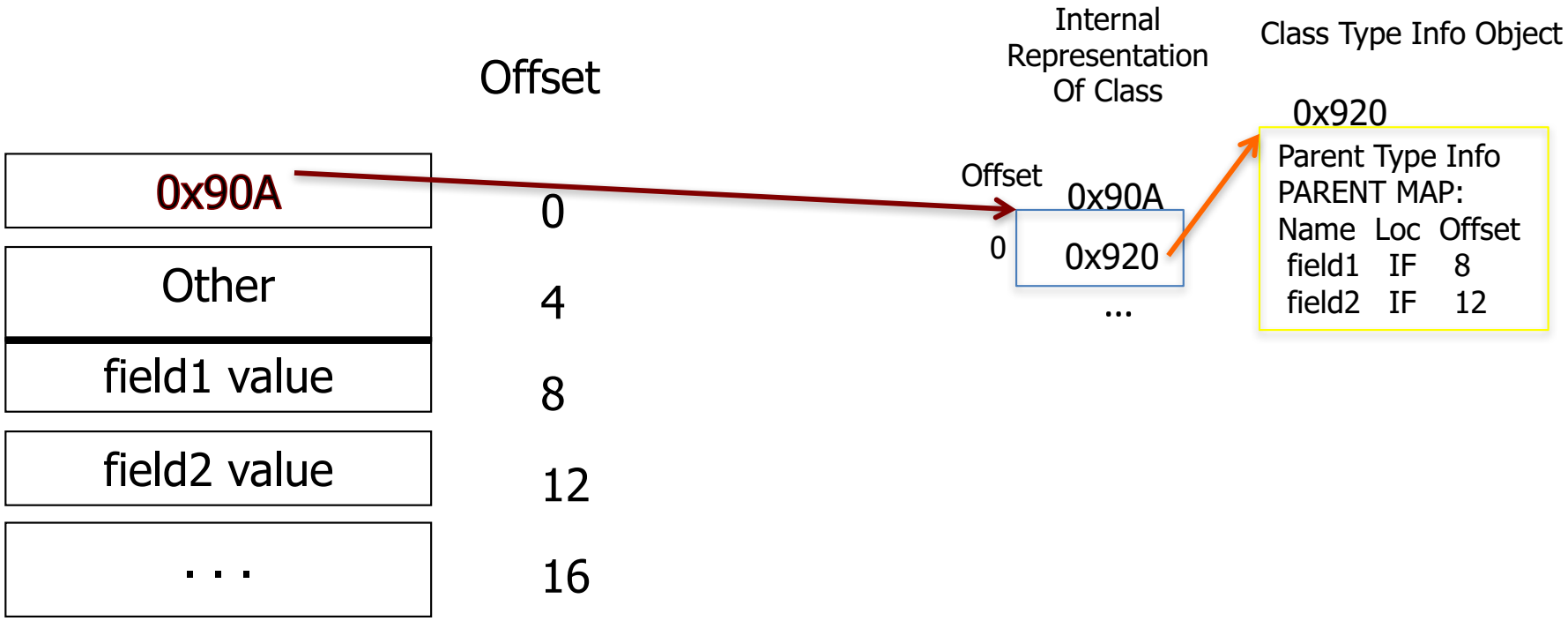
Review: Instance Methods

- If the language is statically typed
  - If the language uses **static dispatch** (C++, C# when the keyword virtual is NOT used)
    - Use the static type or cast to figure out which field it is
      - If there is a Parent-Child relationship (OO hierarchy)
      - And there is a field of the same name in both
        - » Both are available in the child object (static lookup)
        - » Use different variable/types (or use casts) to get to the one you want
    - Put them in the statics table (and statics table map)
    - Look them up just like we do static fields and static methods
    - Constructors in Java are statically dispatched instance methods! Simple:<init>()V

Review: Instance Methods

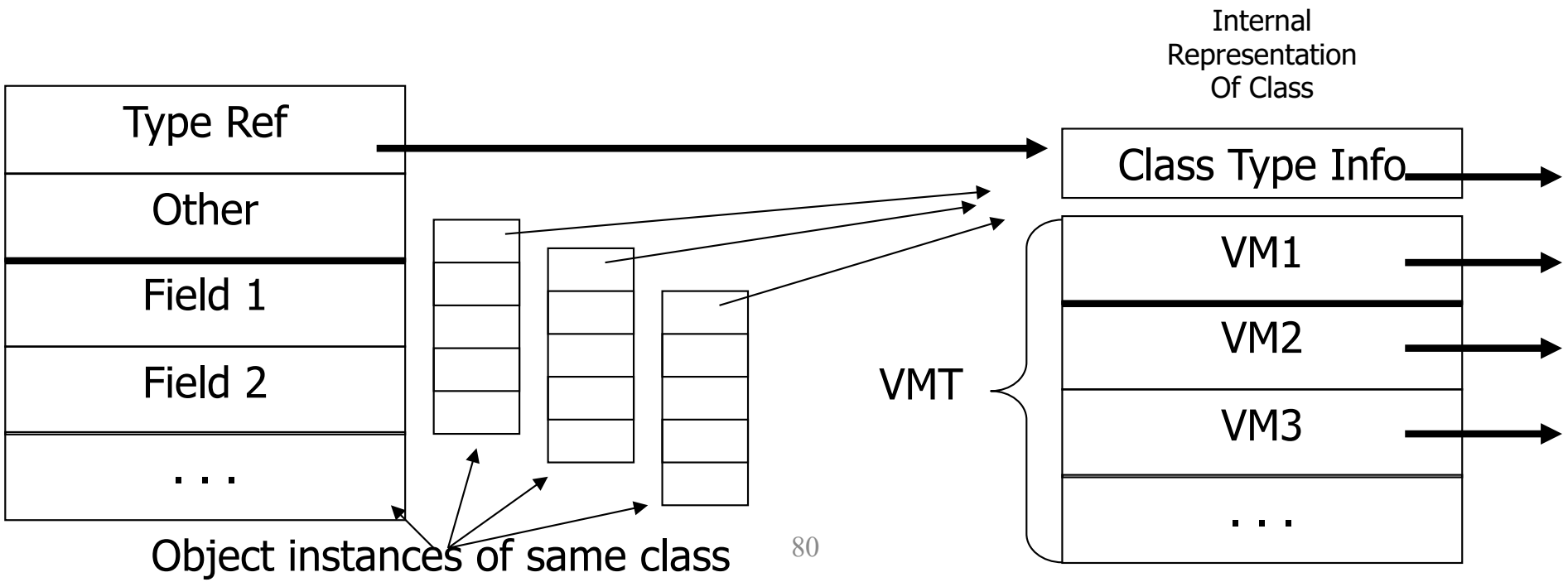
- If the language is statically typed
  - If the language uses **static dispatch** (C++, C# when the keyword virtual is NOT used)
    - Use the static type or cast to figure out which field it is
      - If there is a Parent-Child relationship (OO hierarchy)
      - And there is a field of the same name in both
        - » Both are available in the child object (static lookup)
        - » Use different variable/types (or use casts) to get to the one you want
  - If the language uses **dynamic dispatch** (Java, C++/C# with the virtual keyword)
    - Do the lookup at runtime (typically some sort of table/map lookup)
    - Look at object to which the variable refers, check its type (**TypeRef**)
      - Use its type to figure out which method to invoke (offset of method "in object")

Object Layout (Java/C#) – Instance fields and instance methods



Virtual Method Table in JVMs

- Virtual Method Table (VMT) = Dispatch Table (DT)
- Store it in the Internal class representation instead of individual objects
  - Because method code doesn't change (inst. field values do)



Review: Instance Methods

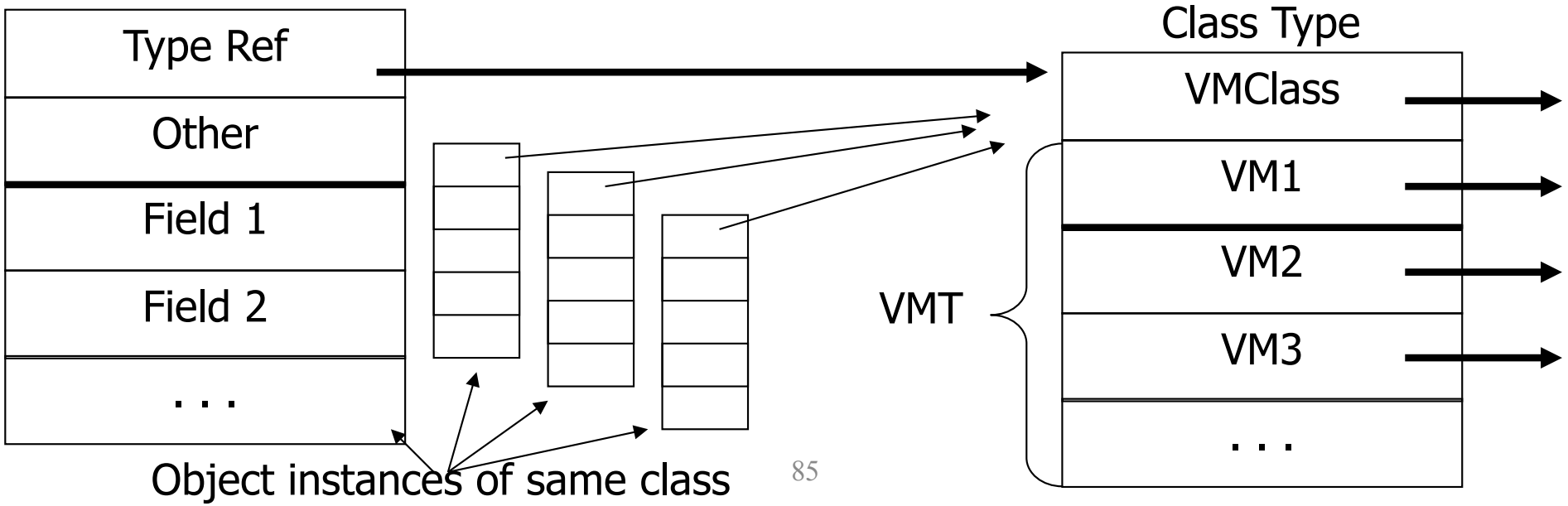
- If the language is statically typed
  - If the language uses **static dispatch** (C++, C# when the keyword virtual is NOT used)
    - Use the static type or cast to figure out which field it is
      - If there is a Parent-Child relationship (OO hierarchy)
      - And there is a field of the same name in both
        - » Both are available in the child object
        - » Use different variable/types (or use casts) to get to the one you want
  - If the language uses **dynamic dispatch** (Java, C++/C# with the virtual keyword)
    - Do the lookup at runtime (typically some sort of table/map lookup)
- If the language is dynamically typed
  - Do a hashtable lookup **by name** at runtime
    - Hashtable per object -- the same as for instance fields

Dispatch Tables

- Every class (in a statically typed language)
  - has a fixed set of virtual instance methods (which includes inherited methods)
- DTs (aka VMTs) hold addresses to methods
  - Block of native code if compiled
  - Block of bytecode if interpreted
- A **dispatch table** indexes these methods
  - AKA virtual method table (VMT)
  - dispatch table = an array of method addresses: entry points
  - A method f lives at a fixed offset in the dispatch table for a class and all of its subclasses
    - The internal class representation holds the offset map

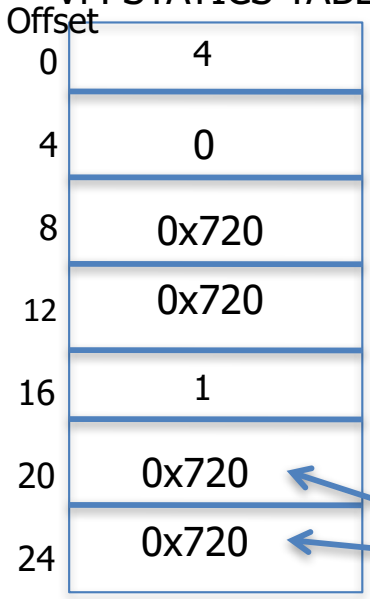
Instance Fields and Methods + Inheritance

- In a statically typed language (Java, C++, C#)
  - Fields and methods cannot be added on the fly
  - Thus we know the layout of all objects
- For fast lookups, assign fields/methods from top of class hierarchy first: *e.g. Great Grandparent, Grand Parent, Parent, then Child*
  - A method f **lives at a fixed offset** in the dispatch table for a class and all of its subclasses



```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

  static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;

   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```

VM STATICS TABLE (Address stored in R7)



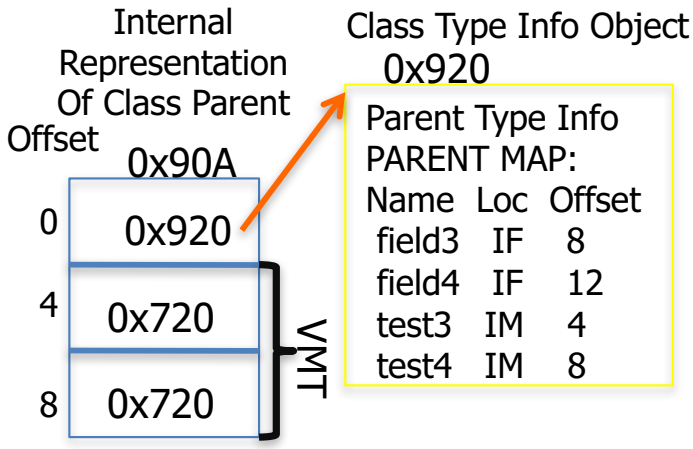
VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24

Method stubs in
a compiled runtime

Loaded first b/c it has main(…) in it
Causes Parent to be loaded right away because Child inherits from Parent
And we cannot create the Child's internal rep without its parent's layout

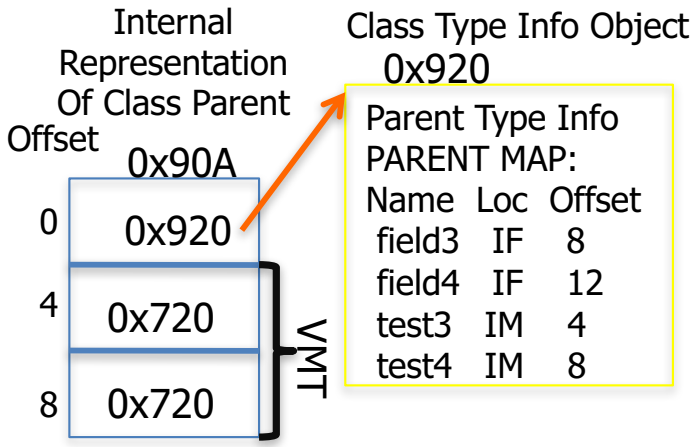
```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```

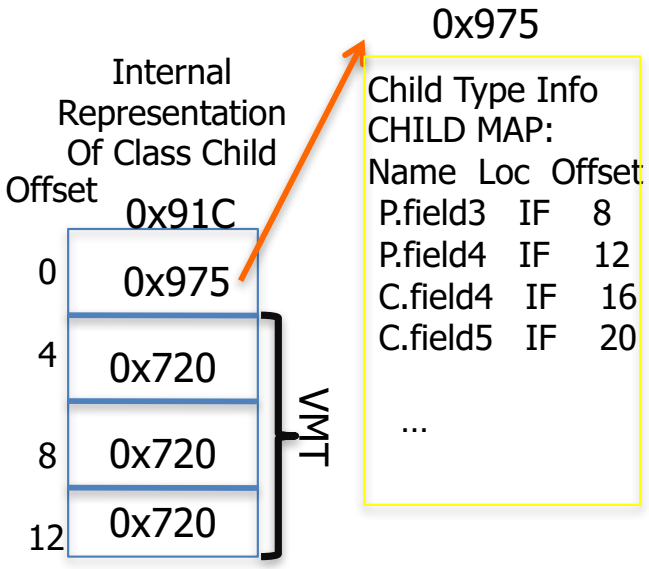


Type Info Map contains offsets in all objects where the instance fields are stored.

It ALSO contains offsets into the VMT in the class representation where the virtual instance method addresses are stored.

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```

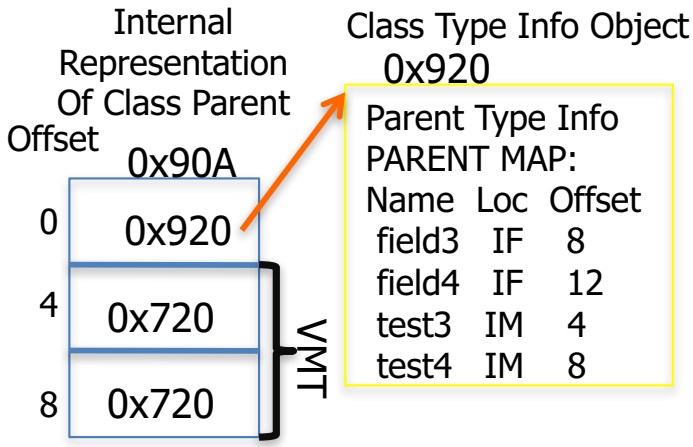


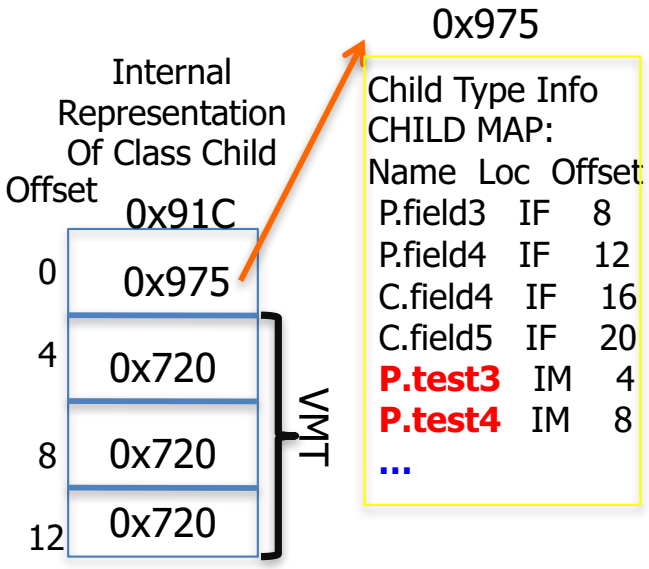


With inheritance, map of child is first loaded with all of the parent parts (fields first)

-- fields are hidden not overridden!

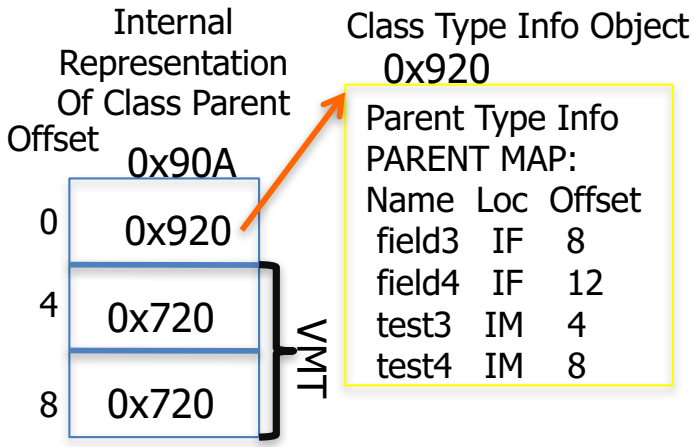
```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```

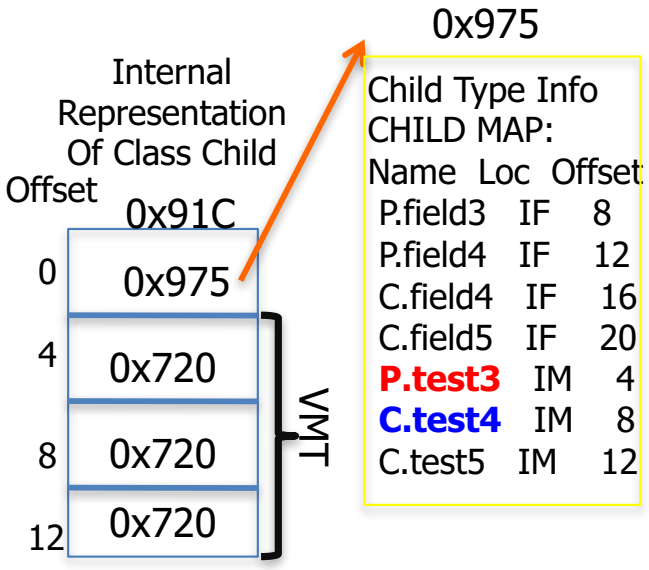




…then instance methods…
- Non-virtual (static dispatch) go in global statics table
-Virtual (dynamic dispatch) go in VMT
and Type Map (all instance methods in Java are virtual)

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```





…then virtual instance methods
-- which child can override

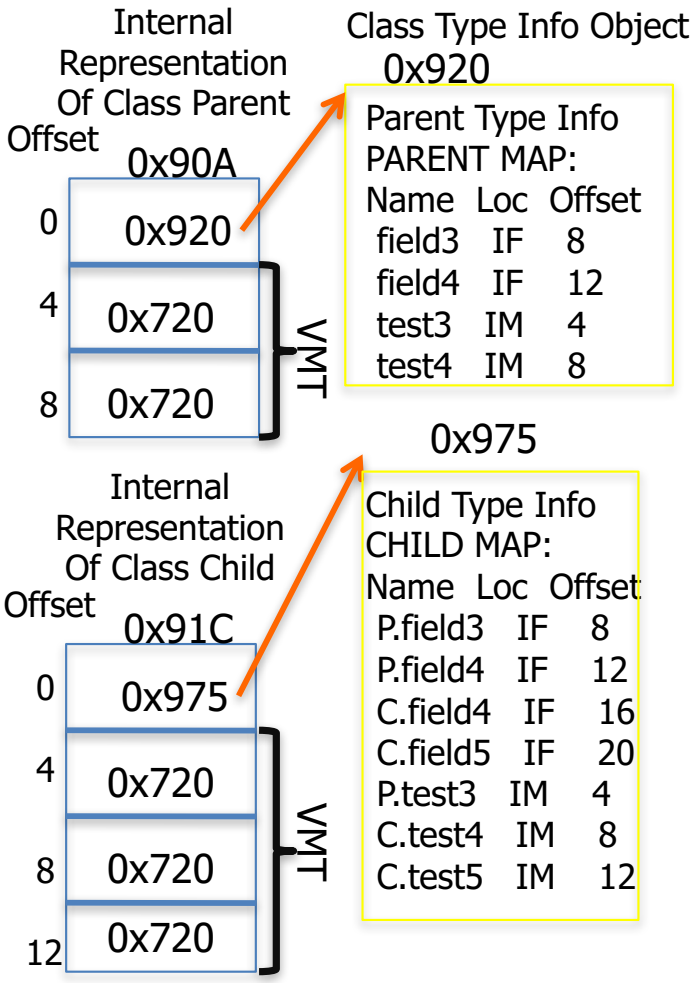
```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x720

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24



Class Info

- HW 1 returned today, grades available via link at
  - http://www.cs.ucsb.edu/~cs263/index.html
    - under announcements
  - All instance methods have a hidden argument (the *this* reference)
    - Including constructors
    - Such methods can be statically dispatched
      - Constructors or those in C# and C++ without virtual and static keywords
  - Methods marked with static key word have no this object passed in and are statically dispatched

- Projects week 1 evaluation – everyone with github setup received 10 points; 100 LOC / commits starting this Friday
  - All approved unless you received an email from me about project overlap with another

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 3;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
        …
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x720
28	0x920
32	0x90A
36	0x975
40	0x91C

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x720
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Constant pool constants are here too!

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x720
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x720
52	0x720

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Parent.<init>()V SIM 48

Child.<init>()V SIM 52

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x720
52	0x720

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Parent.<init>()V SIM 48

Child.<init>()V SIM 52

0x523

Compiled Child.main code

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
          Parent l2 = new Child();
          Child l3 = new Child();
   }
}
```

Program execution point (PC, control)

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x720

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Parent.<init>()V SIM 48

Child.<init>()V SIM 52

0x523

Compiled Child.main code

0x789

Compiled Parent.<init>()V code

Internal Representation Of Class Parent

Offset

0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object

0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset

0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
          Parent l2 = new Child();
          Child l3 = new Child();
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x720

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Parent.<init>()V SIM 48
Child.<init>()V SIM 52

0x523
Compiled Child.main code

0x789
Compiled Parent.<init>()V code

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975
Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

Offset 0x1010

0	0x90A
4	rest of header
8	7
12	0

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
         Parent l2 = new Child();
         Child l3 = new Child();
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x882

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Parent.<init>()V SIM 48

Child.<init>()V SIM 52

0x523
Compiled Child.main code

0x789
Compiled Parent.<init>()V code

0x882
Compiled Child.<init>()V code

Internal Representation Of Class Parent

Offset

0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object

0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset

0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

Offset

0x1010

0	0x90A
4	rest of header
8	7
12	0

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
         Parent l2 = new Child();
         Child l3 = new Child();
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x882

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44

Parent.<init>()V SIM 48
Child.<init>()V SIM 52

0x523
Compiled Child.main code

0x789
Compiled Parent.<init>()V code

0x882
Compiled Child.<init>()V code

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975
Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

Offset 0x1010

0	0x90A
4	rest of header
8	7
12	0

Offset 0x102F

0	0x91C
4	rest of header
8	7
12	0
16	0
20	999999

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
         Parent l2 = new Child();
        Child l3 = new Child();
   }
}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x882

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44
Parent.<init>()V	SIM	48
Child.<init>()V	SIM	52

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	0x720
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

Offset 0x1010

0	0x90A
4	rest of header
8	7
12	0

Offset 0x102F

0	0x91C
4	rest of header
8	7
12	0
16	0
20	999999

Offset 0x104A

0	0x91C
4	rest of header
8	7
12	0
16	0
	999999

Program execution point (PC, control)

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
         Parent l2 = new Child();
        Child l3 = new Child();
         l3.test3();
   }}
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x882

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44
Parent.<init>()V	SIM	48
Child.<init>()V	**SIM**	**52**

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x720
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	**0x720**
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	**IM**	**4**
C.test4	IM	8
C.test5	IM	12

Offset 0x1010

0	0x90A
4	rest of header
8	7
12	0

Offset 0x102F

0	0x91C
4	rest of header
8	7
12	0
16	0
20	999999

Offset 0x104A

0	0x91C
4	rest of header
8	7
12	0
16	0
	999999

Program execution point (PC, control)

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
         Parent l2 = new Child();
        Child l3 = new Child();
         l3.test3();
   }}
```

Program execution point (PC, control)

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x882

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44
Parent.<init>()V	SIM	48
Child.<init>()V	SIM	52

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	**0x618**
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	**0x618**
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

0x618

Compiled **Parent.test3** code

Offset 0x1010

0	0x90A
4	rest of header
8	7
12	0

Offset 0x102F

0	0x91C
4	rest of header
8	7
12	0
16	0
20	999999

Offset 0x104A

0	0x91C
4	rest of header
8	7
12	0
16	0
	999999

```
class Parent {
   static int field1= 4;
   static int field2;
   int field3 = 7;
   int field4;

   static void test1() {
   }
   static void test2() {
   }
   void test3() {
   }
   void test4() {
   }
}
class Child extends Parent {
   static int field2 = 1;
   int field4;
   int field5 = 999999;
   static void test2() {
   }
   void test4() {
   }
   void test5() {
   }
   static void main(String args[]) {
         Parent l1 = new Parent();
          Parent l2 = new Child();
          Child l3 = new Child();
          i3.field5 = 3;
          i3.test3();
          i3.test3();
```

VM STATICS TABLE (Address stored in R7)

Offset	
0	4
4	0
8	0x720
12	0x720
16	1
20	0x720
24	0x523
28	0x920
32	0x90A
36	0x975
40	0x91C
44	999999
48	0x789
52	0x882

VM STATICS MAP at 0x1234:

Name	Loc	Offset
Parent.field1	SF	0
Parent.field2	SF	4
Parent.test1	SM	8
Parent.test2	SM	12
Child.field2	SF	16
Child.test2	SM	20
Child.main	SM	24
Parent.Type	Type	28
Parent.Rep	Rep	32
Child.Type	Type	36
Child.Rep	Rep	40
Child.const1	O	44
Parent.<init>()V	SIM	48
Child.<init>()V	SIM	52

Internal Representation Of Class Parent

Offset 0x90A

0	0x920
4	0x618
8	0x720

VMT

Class Type Info Object 0x920

Parent Type Info
PARENT MAP:

Name	Loc	Offset
field3	IF	8
field4	IF	12
test3	IM	4
test4	IM	8

Internal Representation Of Class Child

Offset 0x91C

0	0x975
4	**0x618**
8	0x720
12	0x720

VMT

0x975

Child Type Info
CHILD MAP:

Name	Loc	Offset
P.field3	IF	8
P.field4	IF	12
C.field4	IF	16
C.field5	IF	20
P.test3	IM	4
C.test4	IM	8
C.test5	IM	12

Offset 0x1010

0	0x90A
4	rest of header
8	7
12	0

Offset 0x102F

0	0x91C
4	rest of header
8	7
12	0
16	0
20	999999

Offset 0x104A

0	0x91C
4	rest of header
8	7
12	0
16	0
	3

Program execution point (PC, control)