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

Today: Part 4 of 5

Chandra Krintz
UCSB Computer Science Department
### REVIEW: STATIC Method and LVA

#### Code:

```java
static void foo(Simple, int, long);

Code:
0: aload_0
1: bipush 10
3: putfield #2 //Field Simple.ifield:I
6: aload_0
7: invokevirtual #3 //Method Simple.bar:()I
10: pop
11: return
```

- Set up the LVA for this method first:
  - Parameter types left to right, then any remaining local variables top to bottom
  - Start with an empty operand stack

<table>
<thead>
<tr>
<th>LVA</th>
<th>Simple</th>
<th>int.</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>maxlocals:</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- long takes up 2 slots
- long/double second index never accessed directly

---

**Operand stack**
instance methods have object instance passed in as first argument:

```java
class Simple {
    void foo(Simple, int, long);
    
    Code:
    0: aload_0
    1: bipush 10
    3: putfield #2 //Field Simple.ifield:I
    6: aload_0
    7: invokevirtual #3 //Method Simple.bar():
    10: pop
    11: return
}
```

- Set up the LVA for this method first:
  - Parameter types left to right, then any remaining local variables top to bottom
  - Start with an empty operand stack

```
<table>
<thead>
<tr>
<th>LVA</th>
<th>Simple</th>
<th>Simple</th>
<th>int</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><code>maxlocals</code> = 5 entries, long takes up 2 slots but long/double second index never accessed directly</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_operand stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
obj = new Simple();
testobj = new Simple();
obj.foo(testobj, 7, 24L); becomes foo(obj, testobj, 7, 24L);
```
review: questions from last time:

class test extends testp{
    int field1 = 2;
    public static void boo(int var, testp obj, long l) {
        ((test) obj).field1 = 7;
        int i = var;
        float f = i;
        int i2 = (int) 2.3f;
    }
}

public static void boo(int, testp, long);

Code:
0: aload_1
1: checkcast #6. //class test
4: bipush 7
6: putfield #2 //Field test.field1:I
9: iload_0
10: istore
12: iload 4
14: i2f
15: fstore 5
17: iconst_2
18: istore 6
20: return

• Length of LVA?

• Downcast in code turns into checkcast bytecode which checks type of TOS
REVIEW: Questions from last time:

class test extends testp{

    int field1 = 2;

    public static void boo(int var, testp obj, long l) {
        ((test)obj).field1 = 7;
        int i = var;
        float f = i;
        int i2 = (int)2.3f;
    }
}

public static void boo(int, testp, long);

Code:
0: aload_1
1: checkcast #6. //class test
4: bipush 7
6: putfield #2 //Field test.field1:I
9: iload_0
10: istore 4
12: iload 4
14: i2f
15: fstore 5
17: 4
18: istore 6
20: return

• Length of LVA?
  • maxlocals = number of array entries including extra slots for long/double
  • here maxlocals = 7 which is 6 for the params and locals + 1 for the long

• Downcast in code turns into checkcast bytecode which checks type of TOS
Constructors in Java Bytecode

• Operand Stack (FIFO data structure)

_invokespecial #2 //index 2 in constant pool: method Cls.<init>*():()V
  //chase index 2 through multiple constant pool entries to find this full name
  //instance method so object must be on TOS + any arguments (int here)

This instruction invokes constructors (instance methods that perform object initialization). They occur after a new instruction.
Constructors in Java Bytecode

• Operand Stack (FIFO data structure)

invokespecial #2 //index 2 in constant pool: method Cls.<init>::()V
   //chase index 2 through multiple constant pool entries to find this full name
   //instance method so object must be on TOS + any arguments (int here)

This instruction invokes constructors (instance methods that perform object initialization). They occur after a new instruction.

Their implementation is complex so we will leave that until later, so just note that this exists and that constructors are handled as special cases in Java
- And that they all have method name <init>
- And are distinguished by their signature if there are multiple constructors
- Constructors always have void return type in Java
- They are statically dispatched despite being instance methods because we know which method implementation the developer intends (the once specified in the call to new: obj = new Cls();)
Object instances (new)

Java requires that the operand stack be empty when a method returns.

```java
class Simple {
    int ifield;
    static void foo(Simple obj) {
        foo(new Simple());
    }
}
```

```java
public static void main(java.lang.String[]);
Code:
0: new #4 //class Simple
3: dup //<init> is the constructor
4: invokespecial #5 //Method Simple.<init>():V
7: invokestatic #6 //Method foo:(LSimple;)V
10: return
```
Java requires that the operand stack be empty when a method returns.

The code snippet provided demonstrates this requirement.

```java
class Simple {
    int ifield;
    static void foo(Simple obj) {
        foo(new Simple());
    }
}
```

```java
public static void main(java.lang.String[]);
```

The code snippet shows:

- `new` is used to create an instance of `Simple`.
- `dup` is used to duplicate the top element on the stack.
- `invokespecial` and `invokestatic` are used to invoke methods on the new instance.
- The stack is empty before and after the `return` statement.

```
Code:
0: new #4 //class Simple
3: dup //<init> is the constructor
4: invokespecial #5 //Method Simple.<init>:()V
7: invokestatic #6 //Method foo:(LSimple;)V
10: return
```
Object instances (new)

```java
class Simple {
    int ifield;
    static void foo(Simple obj) {
        foo(new Simple());
    }
}
```

```java
public static void main(java.lang.String[]);
```

Code:

```
0: new    #4  //class Simple
3: dup    //<init> is the constructor
4: invokespecial #5  //Method Simple.<init>:()V
7: invokevirtual #6  //Method foo:(LSimple;)V
10: return
```

- Java requires that the operand stack be empty when a method returns
class Simple {
    int ifield;
    static void foo(Simple obj) {
        foo(new Simple());
    }
}

public static void main(java.lang.String[]);

Code:
0:    new   #4 //class Simple
3:    dup //<init> is the constructor
4:    invokespecial #5 //Method Simple.<init>:()V
7:    invokestatic #6 //Method foo:(LSimple;)V
10:   return

• Java requires that the operand stack be empty when a method returns
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

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

- `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
Constructors + Inheritance

public class B extends A {
    public static void main(String args[]) {
        obj = new B();
    }  //same as leaving constructor out:
    public B() {...}
}

Allocation (new) + instantiation (<init>(...)V)
Constructors are statically dispatched
Constructors must return void
    So aliasing (dup) must be used
All constructors run parent constructor first!
    default parent: super()  //added for you
    you can change this to a different super signature via super(...)
Constructors + Inheritance

public class B extends A {
    public static void main(String args[]) {
        obj = new B();
    }  //same as leaving constructor out:
    public B() {...}

    Initialize A parts: super();
    Automatically inserted if omitted
    Can be changed explicitly to another constructor: super(k);

    Allocation (new) + instantiation (<init>(...)V)
    Constructors are statically dispatched
    Constructors must return void
        So aliasing (dup) must be used
    All constructors run parent constructor first!
        default parent: super() //added for you
        you can change this to a different super signature via super(...)
Experience, Practice, Practice!

//dump the bytecode (shows constant pool entries resolved)
javap -c Simple  //Simple must be in your classpath
$ export 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://www.ic.uff.br/~cbraga/comp/vmspec/VMSpecIX.fm.html
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)
Java Classfiles

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

File.java

```java
class cls1 {...} class cls2 {...}
```

source compiler (javac File.java)

Translation from bytecode to native machine code

Runtime environment
- program loading/verif.
- memory management
- thread/synchronization
- optimization

Java Virtual Machine

- architecture-independent
- architecture-dependent

cls1.class

... Java bytecode ...

cls2.class

... Java bytecode ...

executable code
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
        » Insert runtime checks for things that cannot be checked statically
    • 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
```java
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)**

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent.field1</td>
<td>SF</td>
<td>0</td>
</tr>
<tr>
<td>Parent.field2</td>
<td>SF</td>
<td>4</td>
</tr>
<tr>
<td>Parent.test1</td>
<td>SM</td>
<td>8</td>
</tr>
<tr>
<td>Parent.test2</td>
<td>SM</td>
<td>12</td>
</tr>
</tbody>
</table>

```
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  VM STATICS MAP (stored in known location too, say 0x1234):
0      Name  Loc  Offset
4      Parent.field1  SF  0
6      Parent.field2  SF  4
8      Parent.test1   SM  8
10     Parent.test2   SM 12

Temporary place-holder address for all method bodies before they are compiled -- called a stub
For interpreter-only this will be the 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() {
    }
}

Compiled from "Parent.java"
class Parent extends java.lang.Object{
    static int field1;
    static int field2;
    int field3;
    int field4;
    ...
    static {};
    Code:
    0:   bipush 4
    2:   putstatic #10; //Field field1:I
    5:   return
class Parent {
    static int field1 = 4;
    static int field2;
    int field3 = 7;
    int field4;
    
    static void test1() {
    }
    static void test2() {
    }
}
class Parent {
    static int field1 = 4;
    static int field2;
    int field3 = 7;
    int field4;

    static void test1() {
    }
    static void test2() {
    }
}
class Parent {
    static int field1 = 4;
    static int field2;
    int field3 = 7;
    int field4;

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

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

VM STATICS TABLE (Address stored in R7)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0x720</td>
</tr>
<tr>
<td>12</td>
<td>0x720</td>
</tr>
<tr>
<td>16</td>
<td>0x920</td>
</tr>
<tr>
<td>20</td>
<td>0x90A</td>
</tr>
<tr>
<td>24</td>
<td>999999</td>
</tr>
</tbody>
</table>

VM STATICS MAP (at 0x1234):
- Parent.field1: SF 0
- Parent.field2: SF 4
- Parent.test1: SM 8
- Parent.test2: SM 12
- Parent.Map: Map 16
- Parent.Rep: Rep 20
- Parent.const1: O 24

Internal Representation Of Class Parent

Class Type Info
- Parent Type Info
  - PARENT MAP:
    - field3: IF 8
    - field4: IF 12
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(); // invokevirtual
    }

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

VM STATICS TABLE (Address stored in R7)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0x720</td>
</tr>
<tr>
<td>12</td>
<td>0x720</td>
</tr>
<tr>
<td>16</td>
<td>0x920</td>
</tr>
<tr>
<td>20</td>
<td>0x90A</td>
</tr>
<tr>
<td>24</td>
<td>999999</td>
</tr>
</tbody>
</table>

VM STATICS MAP (at 0x1234):

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent.field1</td>
<td>SF</td>
<td>0</td>
</tr>
<tr>
<td>Parent.field2</td>
<td>SF</td>
<td>4</td>
</tr>
<tr>
<td>Parent.test1</td>
<td>SM</td>
<td>8</td>
</tr>
<tr>
<td>Parent.test2</td>
<td>SM</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent.Map</td>
<td>Map</td>
<td>16</td>
</tr>
<tr>
<td>Parent.Rep</td>
<td>Rep</td>
<td>20</td>
</tr>
<tr>
<td>Parent.const1</td>
<td>O</td>
<td>24</td>
</tr>
</tbody>
</table>

Internal Representation Of Class Parent

Class Type Info

Object (Map)

0x920

Parent Type Info

PARENT MAP:

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>field3</td>
<td>IF</td>
<td>8</td>
</tr>
<tr>
<td>field4</td>
<td>IF</td>
<td>12</td>
</tr>
</tbody>
</table>
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
  - Structure is defined by the language’s **object model**

<table>
<thead>
<tr>
<th>Offset</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Header word 1</td>
</tr>
<tr>
<td>4</td>
<td>Header word 2</td>
</tr>
<tr>
<td>8</td>
<td>Instance Field 1</td>
</tr>
<tr>
<td>12</td>
<td>Instance Field 2</td>
</tr>
<tr>
<td>16</td>
<td>...</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Ref</td>
<td>0</td>
</tr>
<tr>
<td>Other</td>
<td>4</td>
</tr>
<tr>
<td>Field 1</td>
<td>8</td>
</tr>
<tr>
<td>Field 2</td>
<td>12</td>
</tr>
<tr>
<td>...</td>
<td>16</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x90A</td>
</tr>
<tr>
<td>4</td>
<td>Other</td>
</tr>
<tr>
<td>8</td>
<td>field1 value</td>
</tr>
<tr>
<td>12</td>
<td>field2 value</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16</td>
<td>...</td>
</tr>
</tbody>
</table>

Parent Type Info

<table>
<thead>
<tr>
<th>Parent Type Info</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x920</td>
<td>0x90A</td>
</tr>
</tbody>
</table>

Internal Representation Of Class Parent

<table>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>field1</td>
<td>IF</td>
<td>8</td>
</tr>
<tr>
<td>field2</td>
<td>IF</td>
<td>12</td>
</tr>
</tbody>
</table>
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

Object instances of same class

Internal Representation Of Class

Class Type Info
Object (Map)

0x920

Parent Type Info
PARENT MAP:
Name Loc Offset
field1 IF 8
field2 IF 12

Other

field1 value
field2 value
...
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)
- VMT/DT entries are references to code bodies
  - Either bytecode that will be interpreted
  - Or arch-dependent binary if compiled
Virtual Method Table in JVMs

- Virtual Method Table (VMT) = Dispatch Table (DT)

\[ R1 = \text{obj}[0] + \text{offset} \]

call\_indirect R1
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

```
<table>
<thead>
<tr>
<th>Type Ref</th>
<th>Class Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>VMClass</td>
</tr>
<tr>
<td>Field 1</td>
<td>VM1</td>
</tr>
<tr>
<td>Field 2</td>
<td>VM2</td>
</tr>
<tr>
<td>...</td>
<td>VM3</td>
</tr>
<tr>
<td>Object instances of same class</td>
<td>...</td>
</tr>
</tbody>
</table>
```
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

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[]) {
        ... 
    }
}
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[]) {
        ...
    }
}
```java
class Parent {
    static int field1 = 4;
    int field3 = 7;
    int field4;

    static void test1() {
    }
    static void test2() {
    }
    void test3() {
    }
    void test4() {
    }
}

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