CS 153: Concepts of Compiler Design

November 2 Class Meeting

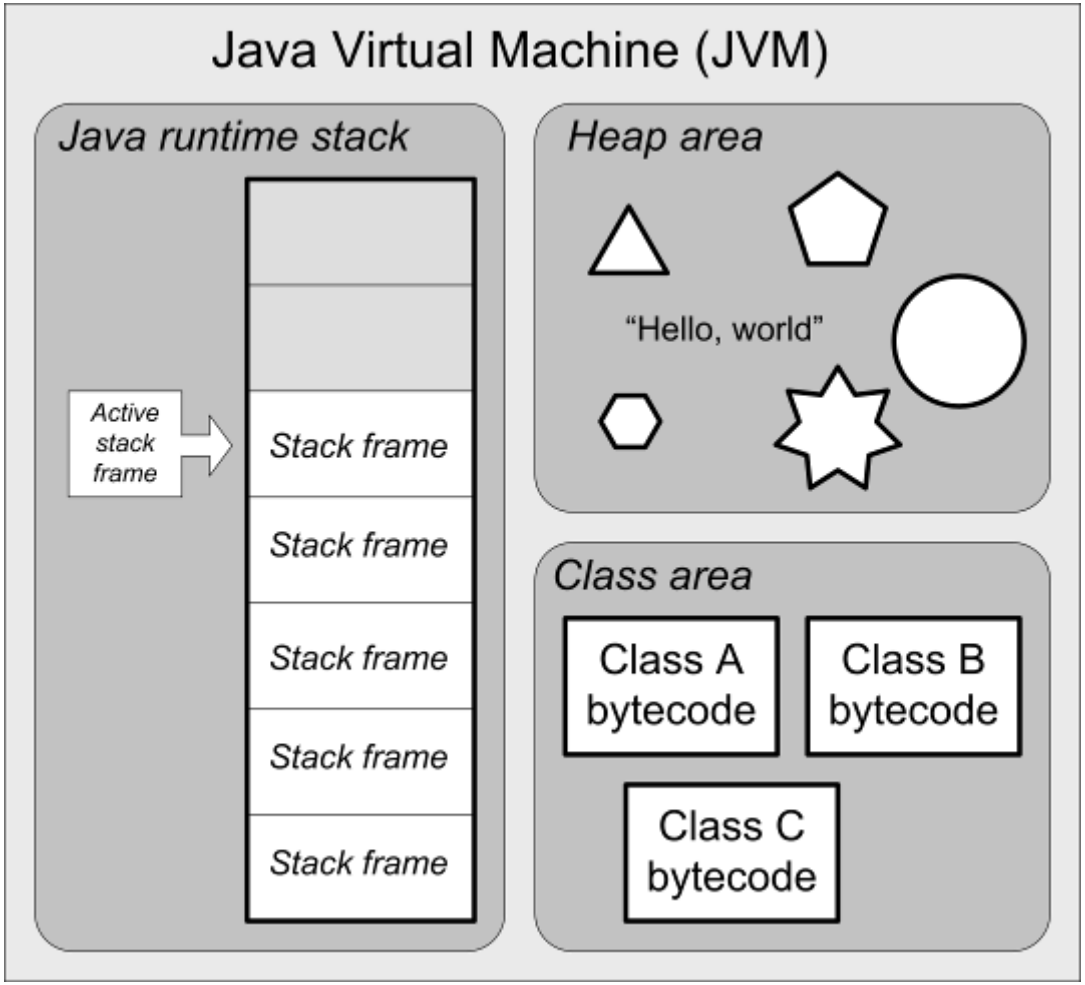
Department of Computer Science
San Jose State University

Fall 2017

Instructor: Ron Mak

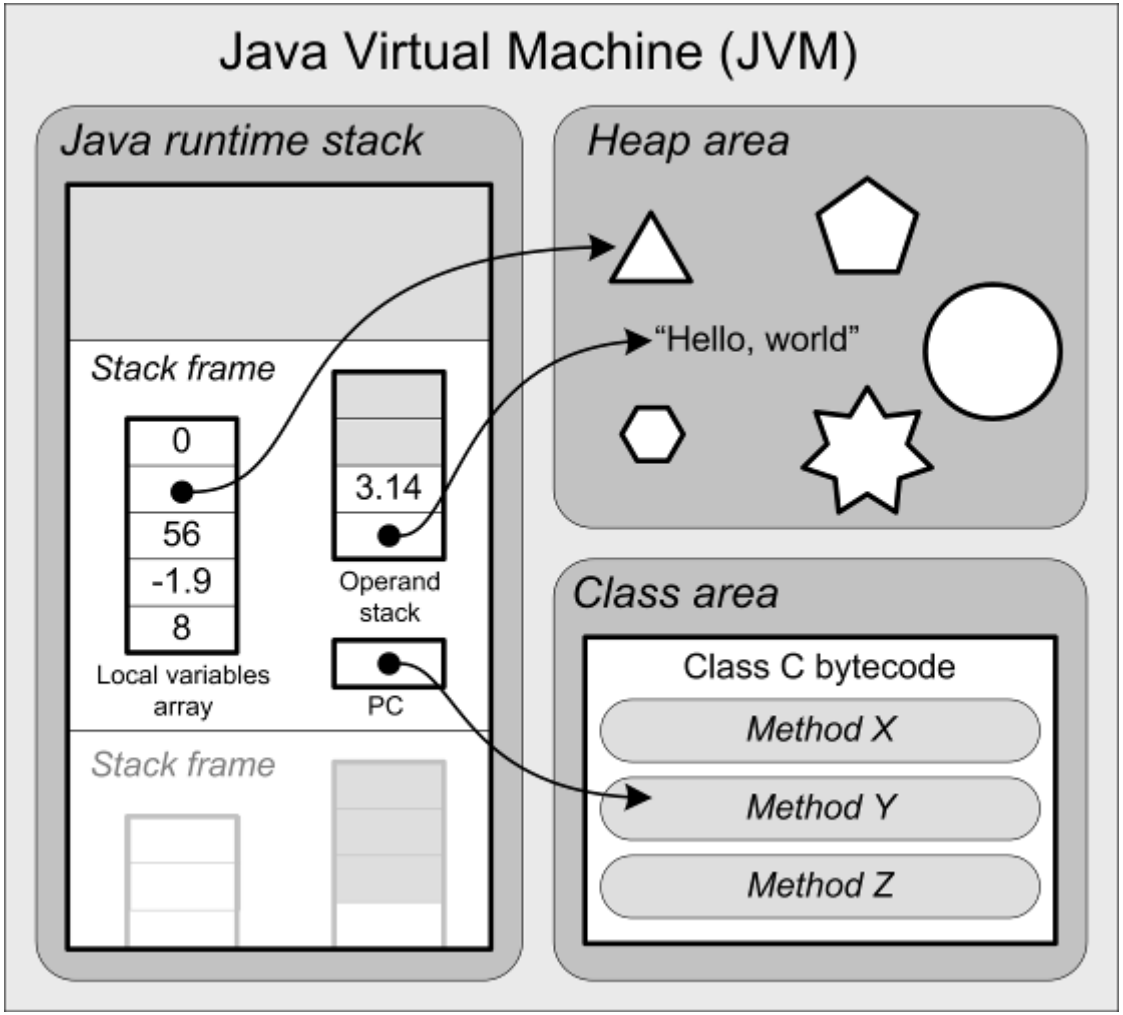
www.cs.sjsu.edu/~mak

Java Virtual Machine (JVM) Architecture



- Java stack
  - runtime stack
- Heap area
  - dynamically allocated objects
  - automatic garbage collection
- Class area
  - code for methods
  - constants pool
- Native method stacks
  - support native methods, e.g., written in C
  - (not shown)

Java Virtual Machine Architecture, *cont'd*



- The runtime stack contains stack frames.
  - Stack frame = activation record.
- Each stack frame contains
  - local variables array
  - operand stack
  - program counter (PC)

Example Jasmin Program

hello.j

```
.class public HelloWorld
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack  2
.limit locals 1

  getstatic     java/lang/System/out Ljava/io/PrintStream;
  ldc           "Hello World."
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  return

.end method
```

- Assemble:
  - `java –jar jasmin.jar hello.j`
- Execute:
  - `java HelloWorld`

Jamin Assembly Instructions

- An Jasmin instruction consists of a mnemonic optionally followed by arguments.
  - Example:

```
aload 5     ; Push a reference to local variable #5
```

- Some instructions require operands on the operand stack.
  - Example:

```
iadd     ; Pop the two integer values on top of the
         ; stack, add them, and push the result
```

Jasmin Assembly Instructions, *cont'd*

- The JVM (and Jasmin) supports five basic data types:
  - int
  - long
  - float
  - double
  - reference
- Examples:

```
isub  ; integer subtraction
fmul  ; float multiplication
```

- Long and double values each requires two consecutive entries in the local variables array and two elements on the operand stack.

Letter	Type
a	reference
b	byte or boolean
c	char
d	double
f	float
i	int
l	long
s	short

Byte, boolean, char, and short are treated as ints on the operand stack and in the local variables array.

Loading Constants onto the Operand Stack

- Use the instructions **`ldc`** and **`ldc2_w`** (load constant and load double-word constant) to push constant values onto the **operand stack**.
  - Examples:

```
ldc      2
ldc      "Hello, world”
ldc      1.0
ldc2_w   1234567890L
ldc2_w   2.7182818284D
aconst_null  ; push null
```

Shortcuts for Loading Constants

- Special shortcuts for loading certain small constants x:

```
iconst_m1  ; Push int -1
iconst_x   ; Push int x, x = 0, 1, 2, 3, 4, or 5
lconst_x   ; Push long x, x = 0 or 1
fconst_x   ; Push float x, x = 0, 1, or 2
dconst_x   ; Push double x, x = 0 or 1

bipush x   ; Push byte x, -128 <= x <= 127
sipush x   ; Push short x, -32,768 <= x <= 32,767
```

- Shortcut instructions take up less memory and can execute faster.

Local Variables

- Local variables do not have names in Jasmin.
  - Fields of a class do have names, which we'll see later.
- Refer to a local variable by its slot number in the local variables array.
  - Example:

```
iload 5  ; Push the int value in local slot #5
```

Local Variables, *cont'd*

- Since each long and double value requires two consecutive slots, refer to it using the lower slot number.
  - Example:

```
lstore 3   ; Pop the long value
           ; from the top two stack elements
           ; and store it into local slots #3 and 4
```

Local Variables, *cont'd*

- Do not confuse constant values with slot numbers!
  - It depends on the instruction.
  - Examples:

```
bipush 14   ; push the constant value 14
iload  14   ; push the value in local slot #14
```

Local Variables, *cont'd*

- Local variables starting with slot #0 are automatically initialized to any method arguments.

```
public static double meth(int k, long m,
                          float x, String[][] s)
```

  - `k` ➔ local slot #0
    `m` ➔ local slot #1
    `x` ➔ local slot #3
    `s` ➔ local slot #4

What happened to slot #2?

  - Jasmin method signature:

```
.method public static meth(IJF[[Ljava/lang/String;)D
```

Load and Store Instructions

- In general:

```
iload n  ; push the int value in local slot #n
lload n  ; push the long value in local slot #n
fload n  ; push the float value in local slot #n
dload n  ; push the double value in local slot #n
aload n  ; push the reference in local slot #n
```

- Shortcut examples
  (for certain small values of n):

```
iload_0  ; push the int value in local slot #0
lload_2  ; push the long value in local slot #2
fload_1  ; push the float value in local slot #1
dload_3  ; push the double value in local slot #3
aload_2  ; push the reference in local slot #2
```

- Store instructions are similar.

Arithmetic Instructions

- Addition

  `iadd ladd fadd dadd`

- Subtraction

  `isub lsub fsub dsub`

- Multiplication

  `imul lmul fmul dmul`

- Division and remaindering

  `idiv ldiv fdiv ddiv`

- Negation

  `ineg lneg fneg dneg`

- Operands are on top of the operand stack.
- Pop off the operands and replace them with the result value.
- Negation has only one operand, the others each has two.
- Int and float operands each takes a single stack element.
- Long and double operands each takes two stack elements.

_

Other Instructions

- Bitwise operations
  - Left and right shifts
  - And, or, exclusive or
- Type conversions
  - int ➔ float
- Widening and narrowing
  - int ➔ long
  - double ➔ long
- Stack manipulations
  - Push and pop
  - Swap and duplicate
- Array operations
  - Allocate array
  - Index element
- Object operations
- Control instructions

Using Java

- Your compilers will generate .class files to run on the Java Virtual Machine (JVM),
- You can write Java classes whose methods invoke methods in your compiled code.
  - Create wrappers and test harnesses.

Using Java, *cont'd*

- Your compiled code can invoke methods in classes that you write in Java.
- Create a runtime library.
  - Example: You invent a new source language with statements that do regular expression searches on strings. You can write the RE algorithms in Java and call them from your compiled code.

Testing Jasmin Programs

- Jasmin multiply engine:

Method **engines.multiply** takes two integer parameters and returns an integer value.

Locals
#0: first parameter value
#1: second parameter value

```
.class public engines/MultiplyEngine
.super java/lang/Object

.method public static multiply(II)I
.limit stack  2
.limit locals 2

    iload_0  ; push the local variable in slot #0 (1st parm)
    iload_1  ; push the local variable in slot #1 (2nd parm)
    imul     ; multiply
    ireturn  ; return the product on the stack

.end method
```

Testing Jasmin Programs, *cont'd*

- Java test harness:

```
package test;

public class MultiplyTester
{
    public static void main(String args[])
    {
        int op0 = Integer.parseInt(args[0]);
        int op1 = Integer.parseInt(args[1]);

        int prod = MultiplyEngine.multiply(op0, op1);

        System.out.println(op0 + " times " + op1 +
                           " equals " + prod);
    }
}
```

Building Hybrid Java + Jasmin in Eclipse

- Put your `.j` files inside the `src` subdirectory with your `.java` files.
- Create a `jbin` subdirectory in your project directory that will contain the `.class` files generated from your `.j` files.
- Right-click the project name in Eclipse.
  - Select Build Path ➔ Configure Build Path ...
  - Select the Libraries tab.
  - Click the Add External Class Folder ... button.
  - Navigate to your `jbin` directory and click the OK button.
  - Click the OK button.
  - Your `jbin` directory should now appear under Referenced Libraries in the project tree.
- Create a `jasmin.bat` or `jasmin.sh` script:
  - `java -jar G:\jasmin-2.3\jasmin.jar %1 %2 %3 %4 %5`
  - `java -jar /jasmin-2.3/jasmin.jar $1 $2 $3 $4 $5`
- Select Run ➔ External Tools ➔ External Tools Configuration ...
  - Name: jasmin
  - Location: path to your `jasmin.bat` or `jasmin.sh` script
  - Working directory: ${project_loc}\jbin
  - Arguments: ${selected_resource_loc}
- Select a `.j` file in the project tree.
  - Select Run ➔ External Tools ➔ Jasmin to assemble the `.j` file into a `.class` file under the `jbin` subdirectory.

Code Templates

- Syntax diagrams
  - Specify the source language grammar
  - Help us write the parsers
- Code templates
  - Specify what object code to generate
  - Help us write the code emitters

Code Template for a Pascal Program

```
.class public program-name                    Program header
.super java/lang/Object
```

Code for fields

```
.method public <init>()V                      Class constructor

      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return

.limit locals 1
.limit stack 1
.end method
```

Code for methods

Code for the main method

- Translate a Pascal program into a public class.
- Program variables become class fields.
- Must have a default constructor.
- Each procedure or function becomes a private **static** method.
- The main program code becomes the public **static** main method.

Compilation Strategy

- We'll compile a Pascal program as if it were a public Java class.
  - The Pascal program name becomes the Java class name.
- The main program becomes the main method of the Java class.
- We'll compile each program variable as if it were a field of the class.
  - Fields do have names in a Jasmin program.
  - Recall that local variables and parameters are referred to only by their slot numbers.

Compilation Strategy, *cont'd*

- We'll compile each Pascal procedure or function as if it were a private static method of the Java class.
- Local variables and formal parameters of the method do not have names in a Jasmin program.
- Jasmin instructions refer to local variables and parameters by their slot numbers of the local variables array.

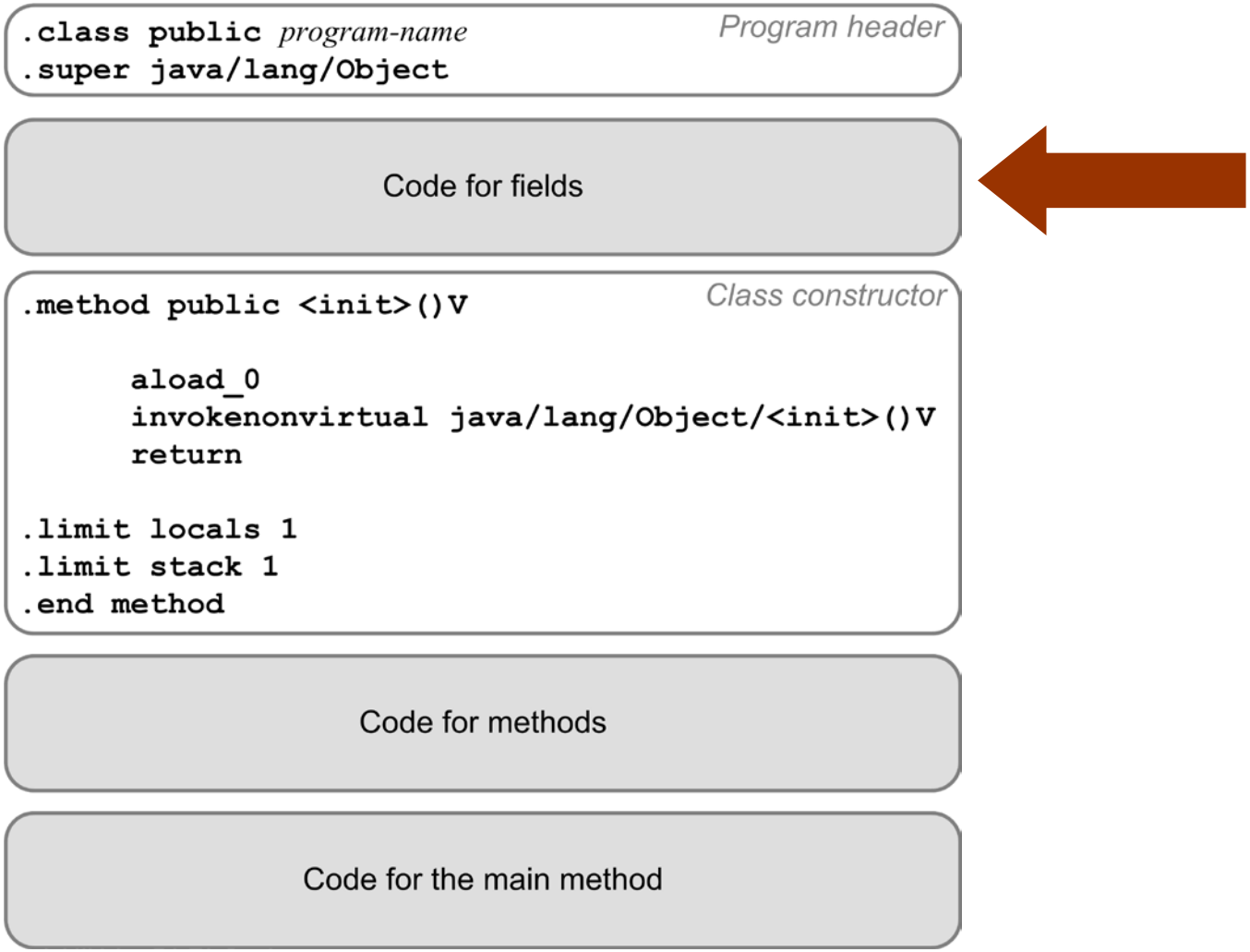
Jasmin Type Descriptors

Java Scalar type	Jasmin Type Descriptor
`int`	`I`
`float`	`F`
`boolean`	`Z`
`char`	`C`

Java Class	Jasmin Type Descriptor
`java.lang.String`	`Ljava/lang/String;`
`java.util.HashMap`	`Ljava/util/HashMap;`
`Newton`	`LNewton;`

Java Array type	Jasmin Type Descriptor
`java.lang.String[]`	`[Ljava/lang/String;`
`Newton[][]`	`[[LNewton;`
`int[][][]`	`[[[I;`

Program Fields



Program Fields, *cont'd*

- For example:

```
PROGRAM test;
VAR
    i, j, k : integer;
    x, y    : real;
    p, q    : boolean;
    ch      : char;
    index   : 1..10;
```

Pascal program variables

- Compiles to:

```
.field private static _runTimer LRunTimer;
.field private static _standardIn LPascalTextIn;
.field private static ch C
.field private static i I
.field private static index I
.field private static j I
.field private static k I
.field private static p Z
.field private static q Z
.field private static x F
.field private static y F
```

Classes **RunTimer** and **PascalTextIn** are defined in the **Pascal Runtime Library** **PascalRTL.jar** which contains runtime routines written in Java.

Code Template for the Main Method, *cont'd*

```
.class public program-name                    Program header
.super java/lang/Object
```

Code for fields

```
.method public <init>()V                      Class constructor

      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return

.limit locals 1
.limit stack 1
.end method
```

Code for methods

Code for the main method

Code Template for the Main Method, *cont'd*

Main method header

```
.method public static main([Ljava/lang/String;)V
```

Main method prologue

```
        new             RunTimer
        dup
        invokenonvirtual  RunTimer/<init>()V
        putstatic       program-name/_runTimer LRunTimer;
        new             PascalTextIn
        dup
        invokenonvirtual  PascalTextIn/<init>()V
        putstatic       program-name/_standardIn LPascalTextIn;
```

Code for structured data allocations

Code for compound statement

Main method epilogue

```
        getstatic       program-name/_runTimer LRunTimer;
        invokevirtual   RunTimer.printElapsedTime()V

        return

.limit locals n
.limit stack m
.end method
```

- The main method prologue initializes the runtime timer **`_runTimer`** and the standard input **`_standardIn`** fields.
- The main method epilogue prints the elapsed run time.
  - **`.limit locals`**
    `.limit stack`
    specify the size of the local variables array and the maximum size of the operand stack, respectively.

Loading a Program Variable's Value

- To load (push) a program variable's value onto the operand stack:

  `getstatic` *program-name/variable-name type-descriptor*

- Examples:

```
getstatic  Test/count I
getstatic  Test/radius F
```

Java Scalar type	Jasmin Type Descriptor
`int`	`I`
`float`	`F`
`boolean`	`Z`
`char`	`C`

Storing a Program Variable's Value

- To store (pop) a value from the operand stack into a program variable:

  `putstatic` *program-name/variable-name type-descriptor*

- Examples:

```
putstatic  Test/count I
putstatic  Test/radius F
```

Java Scalar type	Jasmin Type Descriptor
`int`	`I`
`float`	`F`
`boolean`	`Z`
`char`	`C`

Code for Procedures and Functions

```
.class public program-name                    Program header
.super java/lang/Object
```

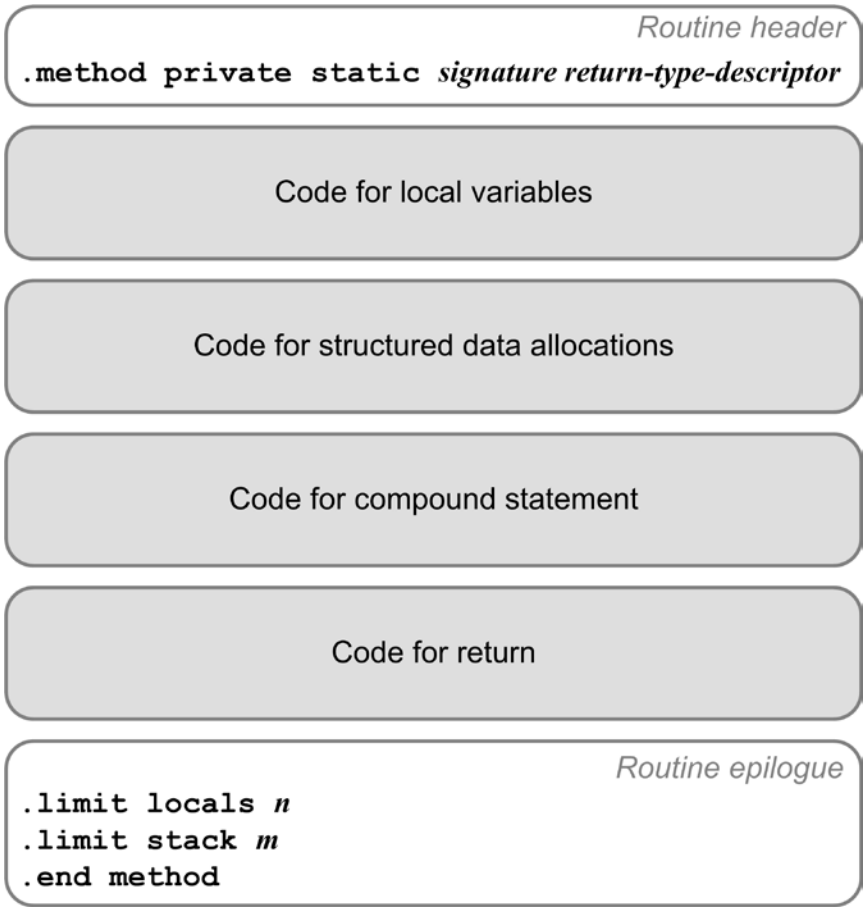
Code for fields

```
.method public <init>()V                      Class constructor

      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return

.limit locals 1
.limit stack 1
.end method
```

Code for methods

Code for the main method

Code for Procedures and Functions



- Each a private static method.
- Method signature:
  - Routine's name
  - Type descriptors of the formal parameters.
- Example:

```
TYPE
    arr = ARRAY [1..5] OF real;

FUNCTION func(i, j : integer;
              x, y : real;
              p : boolean;
              ch : char;
              vector : arr;
              length : integer)
          : real;
```

- Compiles to:

```
.method private static func(IIFFZC[FI)F
```