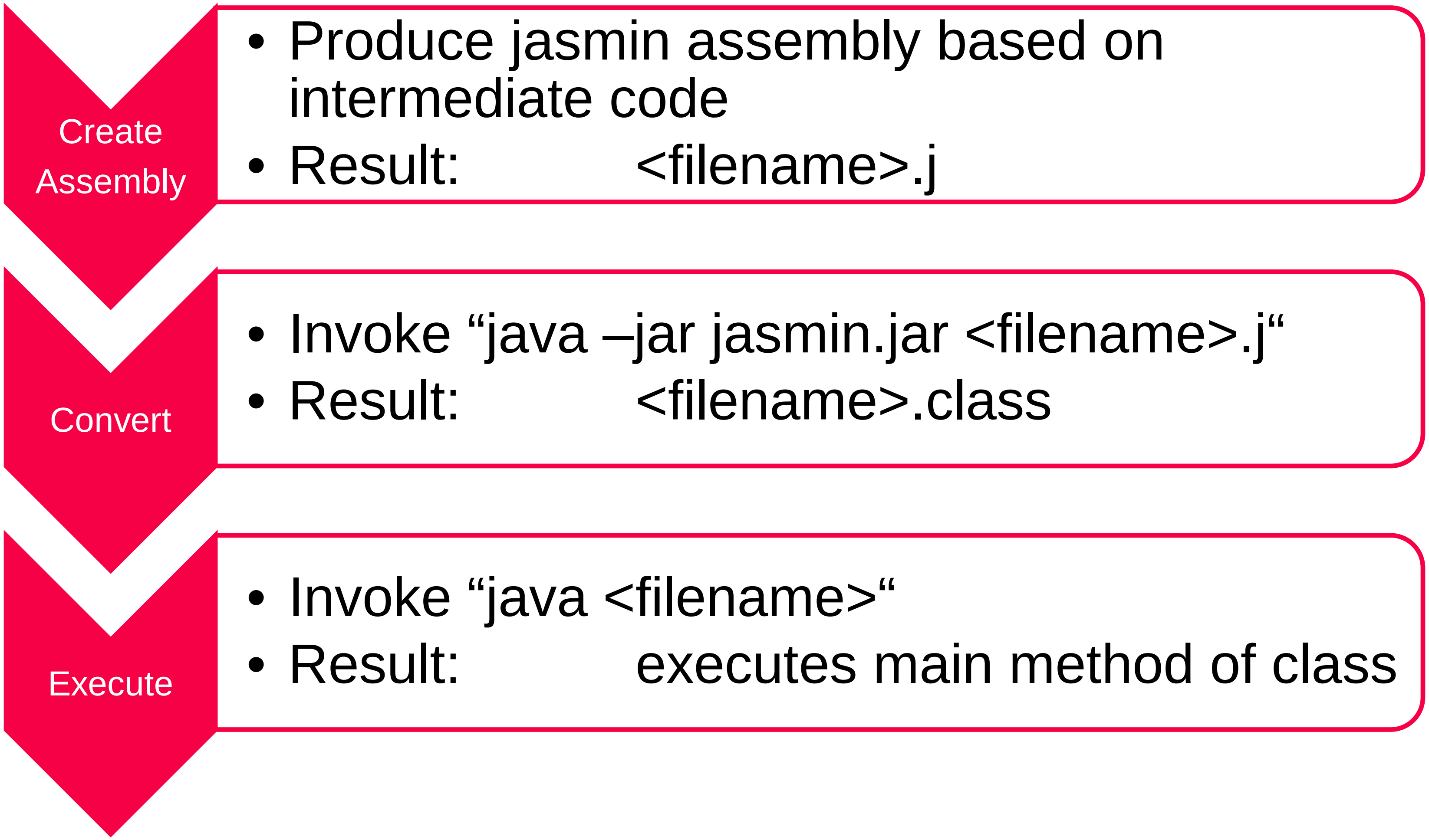
JASMIN

An assembler for the Java Virtual Machine

What is Jasmin?

- Assembler for Java bytecode
- **Input:** <filename>.j file
  contains assembly of intermediate code,
  written in asmin assembler language
- **Output:** executable Java .class file

Pipeline



Create Assembly

- Produce jasmin assembly based on intermediate code
- Result: <filename>.j

Convert

- Invoke “java –jar jasmin.jar <filename>.j“
- Result: <filename>.class

Execute

- Invoke “java <filename>“
- Result: executes main method of class

Jasmin syntax

- One statement per line
- Inline comments, initiated by ‘;‘
- Assembly setup:
  1. Required options
  2. Method 1
  3. Method 2
  4. …
  5. Method n

Required Options

- .**source:** Source of assembly
  - e.g.: `.source MyClass.calc`
- .**class:** Resulting java class description
  - e.g.: `.class public static MyClass`
- .**super**: Superclass of resulting java class
  - always: `.super java/lang/Object`

Methods

1. `.method <method signature>`

    - e.g. .method public static main([Ljava/lang/String;)V

2. `.limit stack n`

    n: choose realistic number

1. `.limit locals n`

    n = #parameters + #local_vars + #temp_vars

2. Instructions

3. `return`

    requires matching type on top-of-stack for non-void returns (e.g. `ireturn`)

1. `.end method`

Example: DoNothing

File: DoNothing.j

```
.source noSource
.class public static DoNothing
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack 0
.limit locals 1
;nothing to do here
return
.end method
```

Data Management

- Two data structures per function:
  - Stack
  - Locals Array
- Operations to manipulate both
- Built-in data types:
  - Primitives
  - Arrays
  - Objects

Data Management: Stack

- Each method has its own operand stack
- Size is definable per function
  (just assume a realistic number)
- Stack operations:
  - Push values onto stack
  - Pop/fetch values from stack
  - Instructions which require one or multiple values on stack (order does matter!)

Data management: Local arrays

- Each method has its own locals array
- Definable size for each function (0-based indices)
- Typing:
  - Can store arbitrary types
  - Items need to be initialized before read access
- Contains:
  - Function parameters (Stored in lowest indices)
  - Locally declared variables
  - Temporary variables

Data management: Types

- Primitives
  - Integer indicated by letter I
  - Void indicated by letter V
  - Float indicated by letter F
- Objects following the format „package/Classname;“
  - e.g. `Ljava/lang/String;` String object
- Arrays indicated by a leading [
  - E.g. `[Ljava/lang/String;` array of Strings

Instructions

- One instruction per line
- Can involve stack and locals array
- May require specific number/type of elements present
- Order for binary instructions like a – b:

1. `push a`   push value of a
2. `push b`   push value of b
3. `isub`   pop both and push value of a - b
4. result is now top of stack

Instructions: Variable Handling

- `Iload n` pushes integer, stored in index n of locals array, onto stack
- `Istore n` pops integer from stack and stores it into index n of locals array
- `aload n` pushes object, stored in index n of locals array, onto stack
- `astore n` pops object from stack and stores it into index n of locals array

Instructions: Constants

- `sipush n` / `bipush n`
  - pushes integer constant n onto stack
  - e.g. `sipush 10`
- `ldc      “<string>“`
  - Pushes string constant <string> onto stack
  - e.g. `ldc “Hello World"`
  - Note: Strings are Objects (variable access with astore/aload)

Instructions: Arithmetic Operators

- `ineg` toggles sign of int on top of stack
- `iadd` add two integers
- `imul` multiply two integers
- `idiv` divide two integers
- `irem` modulo division of two integers

Instructions: Logic Operators

- `iand` bitwise and of two integers
- `ior` bitwise or of two integers
- `inot` does not exist!
  - Needs to be assembled using custom labels and conditional jump operations

Example

File: BasicInstructions.j

```
.source noSource
.class public static BasicInstructions
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack 5
.limit locals 3
sipush 5            ;push integer 5 onto stack
istore 0            ;pop integer 5 and store in index 0
ldc "Hello World" ;push string Hello World onto stack
astore 1            ;store string in index 1

iload 0             ;load 5
dup                 ;duplicate stack entry 5
sipush 2            ;push integer 2 onto stack
isub                ;pop 5 and 2 and store result 3 onto stack
iadd                ;pop 5 and 3 and store result 8 onto stack
istore 0            ;store result 8 in index 0

return
.end method
```

Instructions: Relation Operators

- Do not exist!
- Need to be assembled using custom labels and conditional jump operations

Instructions: Labels and Jumps

- `<labelname>:`

  marks a label in the assembly
- `goto <labelname>`

  continues execution of current method at position of the label <labelname>
- `if_icmpXX <labelname>`

  pops two elements off the stack, relates them and jumps to <labelname> if comparison computes to true

Instructions : if_icmp variations

`if_icmplt`	relation using <
`if_icmple`	relation using <=
`if_icmpge`	relation using >=
`if_icmpgt`	relation using >
`if_icmpeq`	relation using ==
`if_icmpeq`	relation using !=

Example

File: LabelsAndJumps.j

```
.source noSource
.class public static LabelsAndJumps
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack 5
.limit locals 3
sipush 10
istore 0          ;store 10 in index 0
goto label_skip_redefinition
sipush 20
istore 0          ;store 20 to index 0 - skipped
label_skip_redefinition:
iload 0           ;push value of index 0: 10
sipush 15         ;push 15
if_icmplt label_is_lesser
ldc "greater"     ;push string to stack - skipped
goto label_end
label_is_lesser:
ldc "lesser"      ;push string to stack
label_end:
; result: string "lesser" on top of stack
return
.end method
```

Instructions: Static Method Calls

- Methods defined as static methods within class
- Call using

```
invokestatic <classname>.<method signature>
```

- Example

method: `public static int myMethod(int a);`

signature: `myMethod(I)I`

invokation: `invokestatic MyClass.myMethod(I)I`

Instructions: Non-static Method Calls

- Call method of object on stack
- Call using

```
invokevirtual <package>/<method signature>
```

- Requires parameters and object on stack
- Example:

method: `public int myMethod(int a);`

signature: `myMethod(I)I`

invokation: `invokevirtual foo/bar/myMethod(I)I`

Instructions: Print

- Based on virtual invokation
- How to:

1. Push PrintStream object onto stack:

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

2. Push value onto stack (iload, aload etc.)
3. Invoke matching PrintStream method

```
invokevirtual java/io/PrintStream/println(I)V
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

Example

File: MethodCalls.j

```
.source noSource
.class public static MethodCalls
.super java/lang/Object

.method public static print(ILjava/lang/String;)I
.limit stack 2
.limit locals 2
  ;print integer
iload 0
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(I)V
  ;print string
aload 1
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  ;return 0
sipush 0
ireturn
.end method

.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 2
sipush 42
ldc "Hello World"
invokestatic MethodCalls.print(ILjava/lang/String;)I
istore 1  ;store return value 0 in index 1
return
.end method
```