Java Langage & JRE Internal Basics

Compile – (Link -) Runtime

This document:
https://github.com/Arnaud-Nauwynck/presentations/blob/main/java/Java-Langage-JRE-Internal-Basics.pdf

Outline

- Overview compile – runtime chain
- Compiler basics: grammar, parser to AST
  - Declaration-Statement-Expression
  - Bytecode, stack
- Langage Class Symbol resolution
  - Class.forName() / ClassLoader
  - First reference, Hot swap code
- Method Symbol resolution + call
  - Invokestatic, invokespecial
  - Invokevirtual
  - Invokeinterface
  - invokedynamic

Compile – Runtime Chain

UTF-8 files
src/main/java/*.java

Main.java

```java
package test;

public class Main {

    public static void main(String[] args) {
        System.out.println("Hello world");
    }

}
```

javac

Bytecode (binary) files
target/classes/*.class

jar

jar file = zip of *.class
target/*.jar

*.jar or .class
In CLASSPATH

+ main FQN

java

JRE

Symbols

C0
bytecode
interpreter

.. bytecode
link resolved
On first use

C1
assembly
langage

C2
assembly
... optimized

Compile (+Extension) – Runtime(+JVM Agent) Chain

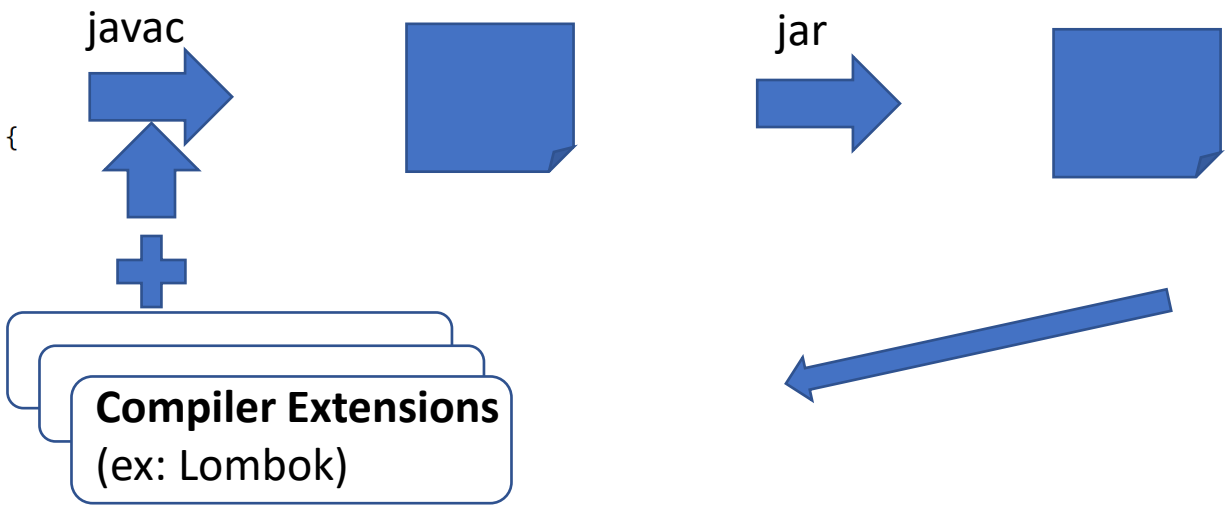
UTF-8 files
src/main/java/*.java

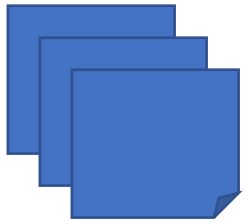
Main.java

```
1 package test;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.println("Hello world");
7     }
8
9 }
```

Bytecode (binary) files
target/classes/*.class

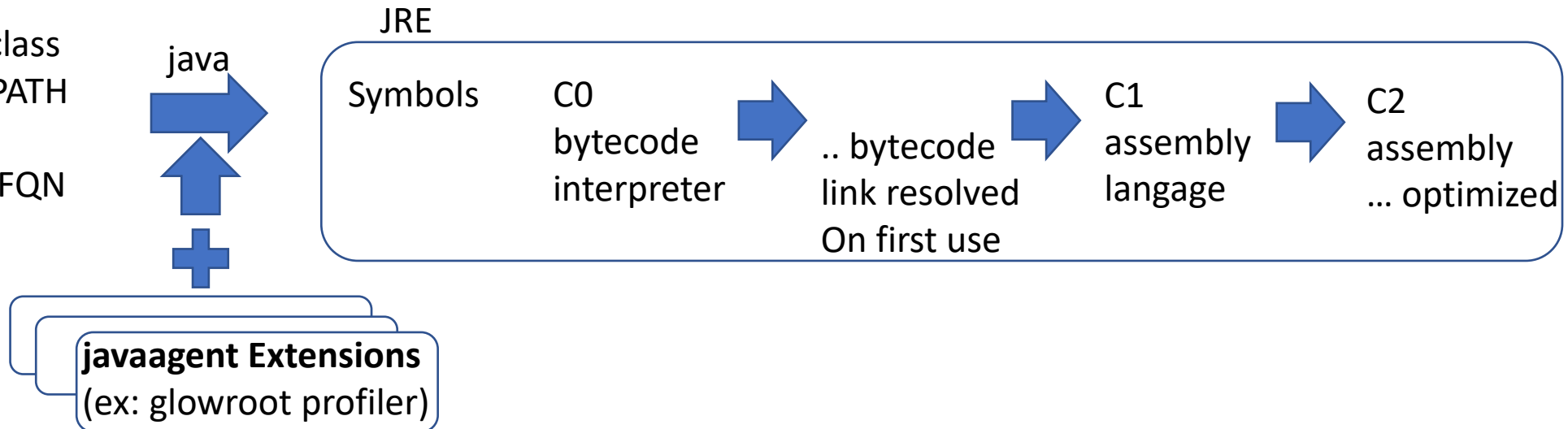
target/*.jar
(jar = zip file
of *.class)





*.jar or .class
In CLASSPATH

+ main FQN



Compile steps

UTF-8 files
src/main/java/*.java

Main.java

```
1 package test;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         System.out.println("Hello world");
7     }
8
9 }
```

javac

Bytecode (binary) files
target/classes/*.class

```
$ cat target/classes/test/Main.class
□□□□   ; ↔
☻ ♥ ♦♀ ♣ ♠☺ ►java/lang/Object☺ ♠<init>☺ ♥()V
♀ ♂ ♀☺ ►java/lang/System☺ ♥out☺ §Ljava/io/PrintStream ♫☺ ♂Hello world
► ◄ ↕♀ ‼ ¶☺ ‼java/io/PrintStream☺ println☺ §(Ljava/lang/String;)V ■☺
      test/Main☺ ♦Code☺ ☼LineNumberTable☺ ♦main☺ ■([Ljava/lang/String
;)V☺
SourceFile☺     Main.java ! § ☻     ☻ ☺ ♣ ♠ ☺ ↨   ↔ ☺ ☺   ♣*♦ ☺♦   ☺ ↑
♦ ☼♦   ☺ ↑       ↓ → ☺ ↨   % ☻ ☺        ♦ ↕
☻   ♠  ☺ ?2004h
```

jar

Zip file
target/*.jar

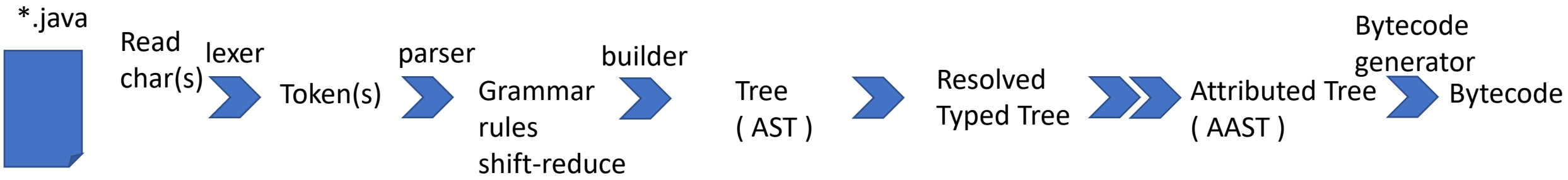
mvn package

```
$ javac -verbose -d target/classes src/main/java/test/Main.java
[parsing started SimpleFileObject[C:\Users\arnaud\eclipse-ws\ws1\test\src\main\java\test\Main.java]]
[parsing completed 30ms]
[loading /modules/jdk.security.jgss/module-info.class]
[loading /modules/java.smartcardio/module-info.class]
[loading /modules/jdk.crypto.ec/module-info.class]
[loading /modules/jdk.charsets/module-info.class]
```

```
[checking test.Main]
```

```
[wrote target\classes\test\Main.class]
[total 480ms]
```

Javac .. steps



CST = **Concrete** Syntaxic Tree

contain « ; » and parenthesis « () »

AST = **Abstract Syntaxic Tree**

AAST =
Attributed
Abstract Syntaxic Tree

Stack
Operator

Example.. Compiling « (1+x)*3.1 » from source to bytecode

(1 + x) * 3.1 → → ?? →

```
push 1
push x
fadd
push 3.1
dmult
```

Javac .. steps

*.java

Read char(s) → lexer → Token(s) → parser → Grammar rules shift-reduce → builder → Tree (AST) → Resolved Typed Tree → Attributed Tree (AAST) → Bytecode generator → Bytecode

CST = **Concrete** Syntaxic Tree

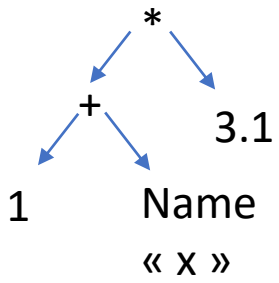
contain « ; » and parenthesis « () »

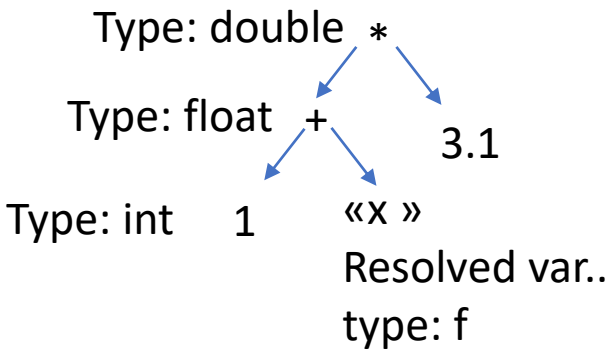
AST = **Abstract Syntaxic Tree**

AAST = **Attributed** Abstract Syntaxic Tree

Stack Operator

« (1 + x) * 3.1 »





push 1
push x
fadd
push 3.1
dmult

Lexer to Tokens

« (1 + x) * 3.1 »

(Literal number op Name) op Literal number

UTF-8 source file

API to read char file:

```
reader = ..
for(;;) {
  char c = reader.read();
  ..
}
```

API to read tokens:

```
tokenLexer = ..
for(;;) {
  token tk = lexer.readToken();
  ..
}
```

Grammar Rule

Expression ::= LitteralNumberExpr | VariableExpr | ParenthesisExpr |
UnaryOpExpr | BinaryOp | FunctionCallExpr

LitteralNumberExpr ::= <number>

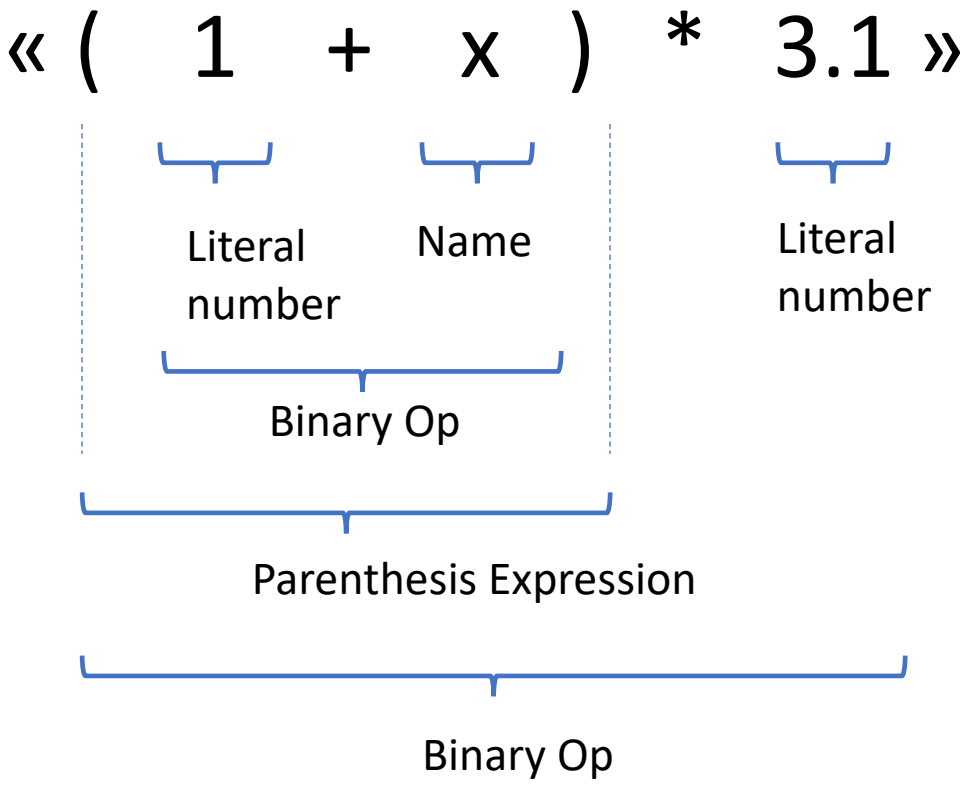
VariableExpr ::= <name>

ParenthesisExpr ::= « (» Expression «) »

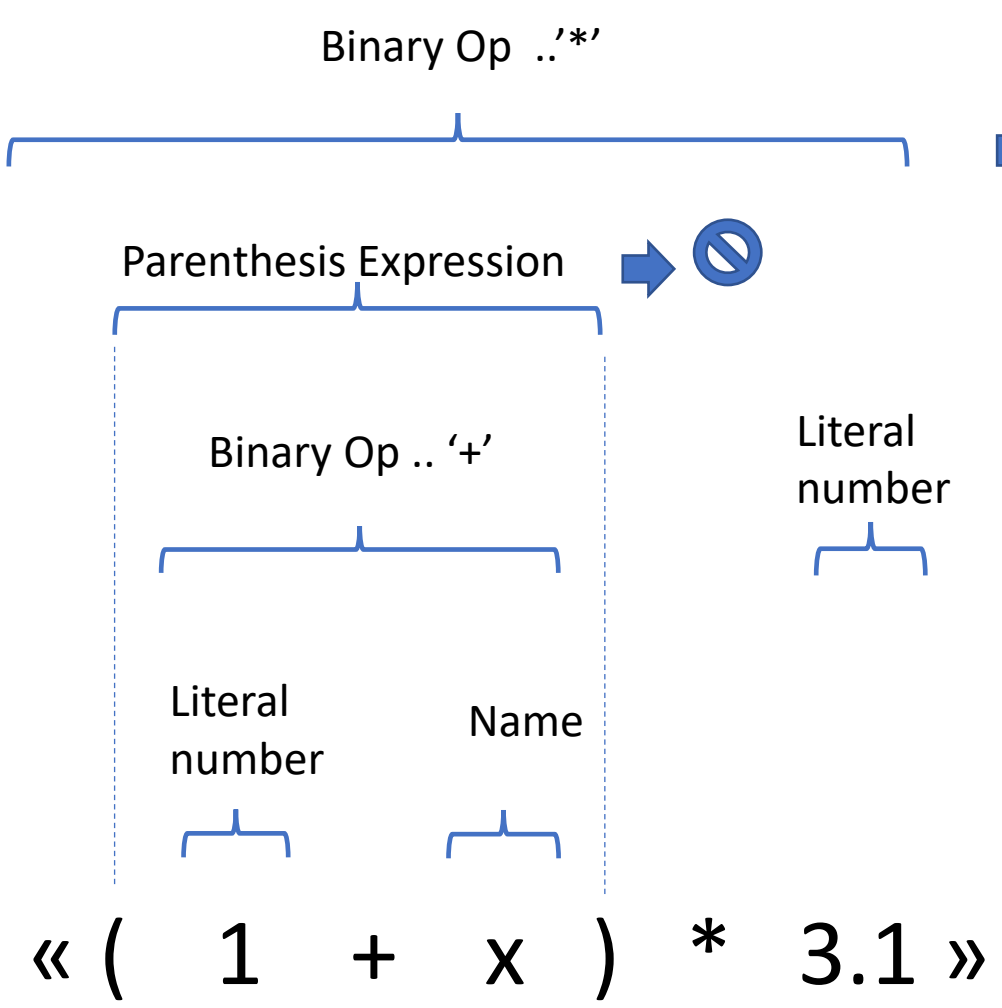
UnaryOpExpr ::= unoperator Expression
+, -, !

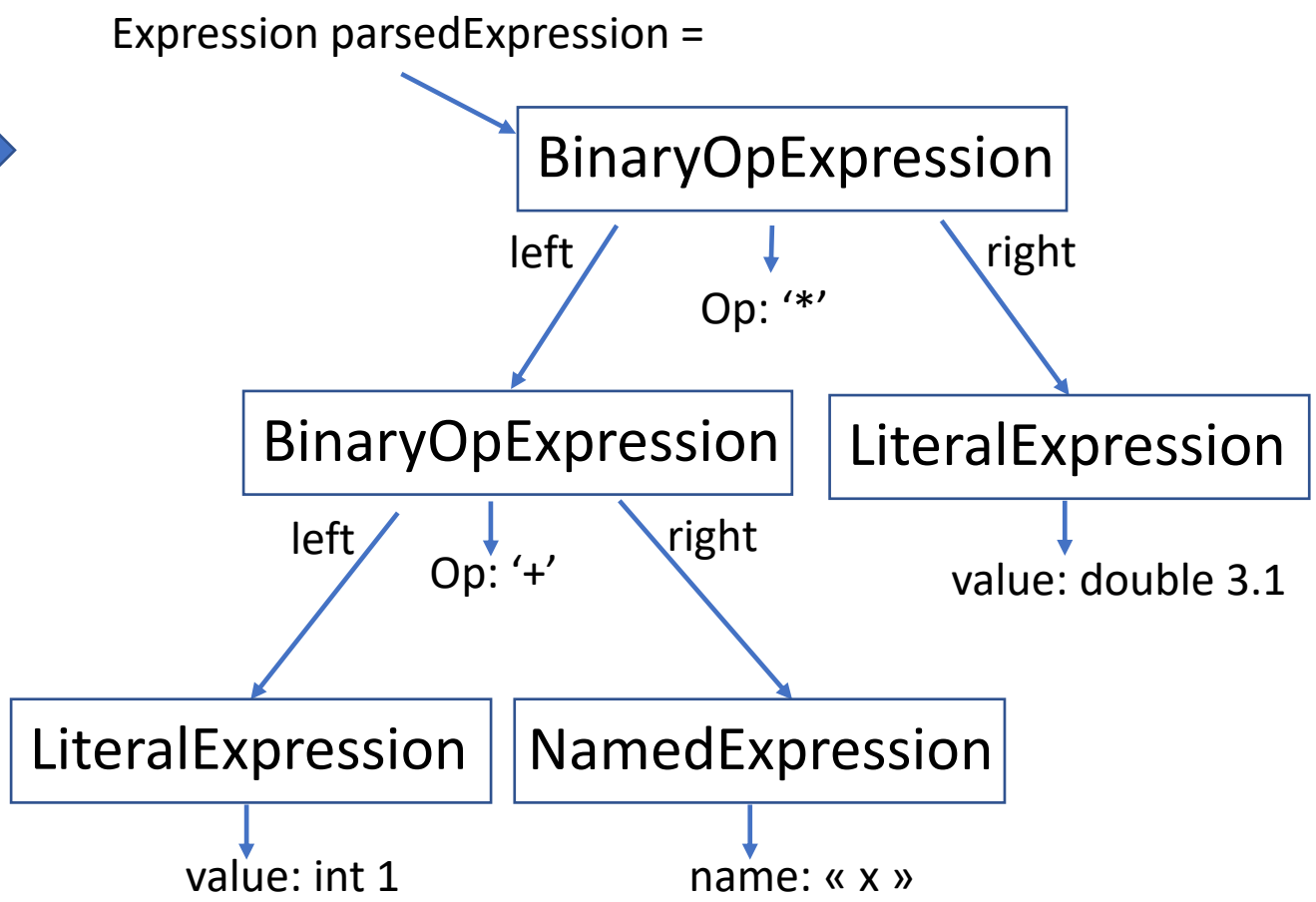
BinaryOpExpr ::= Expression binayOperator Expression
+, -, *, /, %, &, |, ^, && ..

Grammar Rule Parsed



Tree (in-memory) Representation



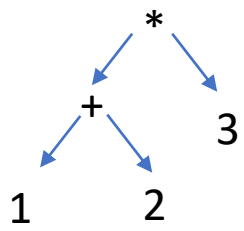


Concrete -> Abstract Syntaxic Tree
lost parenthesis, « ; », indentation, comment…

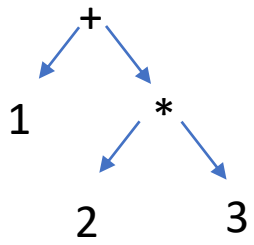
AST is NOT ambiguous, no need parenthesis

(parser need handling operator precendence)

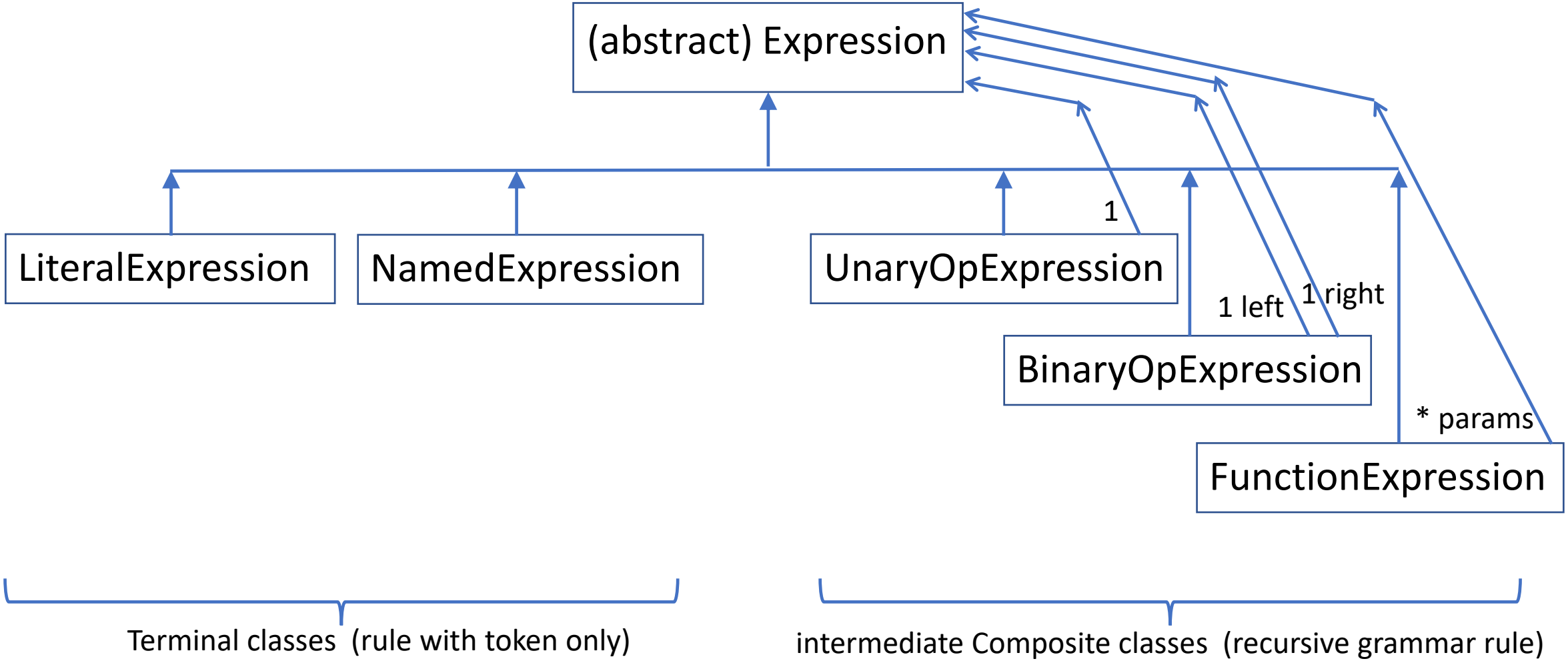
(1 + 2) * 3



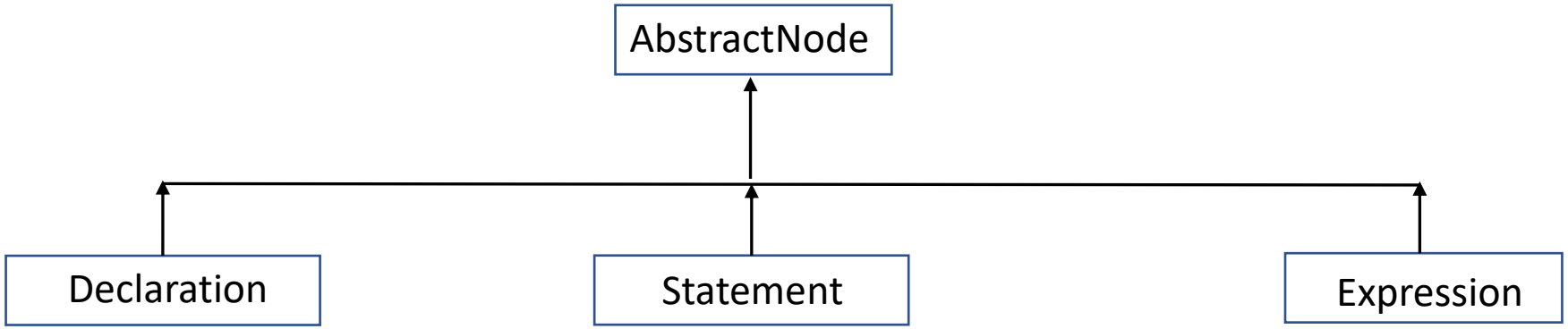
1 + (2 * 3) = 1 + 2 * 3



Expression classes
Abstract Syntaxic Tree (AST)



.. Not only Expressions
need Procedural (Statement) + Declarations

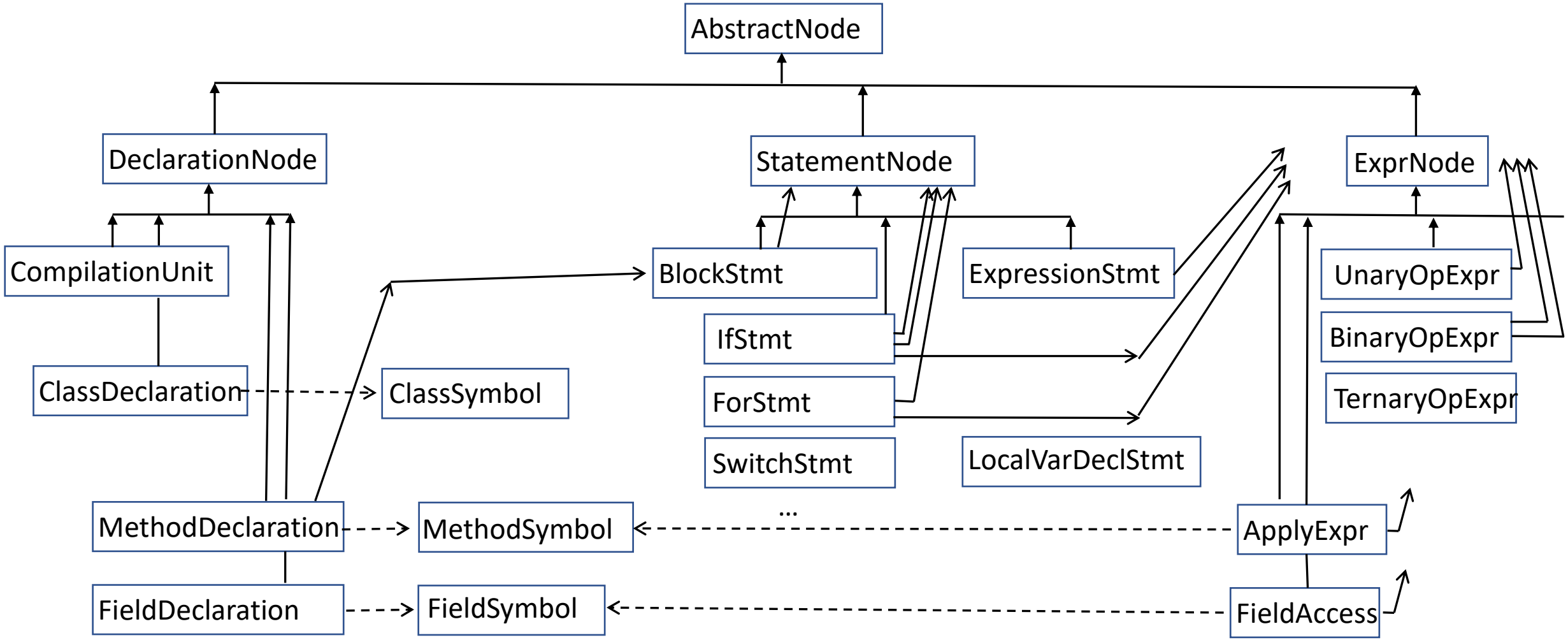


For program structure
package, class, method, ..

For procedural parts
« if », « for », « switch »..

For values,
and operators

Abstract Syntaxic Tree



Correspondance
Grammar Rule <-> AST Classes

1 grammar Rule ~ 1 AST sub-class

Parse 1 grammar rule => build 1 AST sub tree

Example

Rule XYZ::= <expr1> token <expr2> .. token <exprN>

```
// => trigger AST builder code:
{ return new AstXYZ (expr1, expr2, .. exprN); }
```

Parser

Using Code Generated from Grammar Rule ?

Example tools:
« lex & yacc » (origin C)
« Antlr », « JavaCC », ..

Using « hand-coded »
Recursive parser
… example « javac »

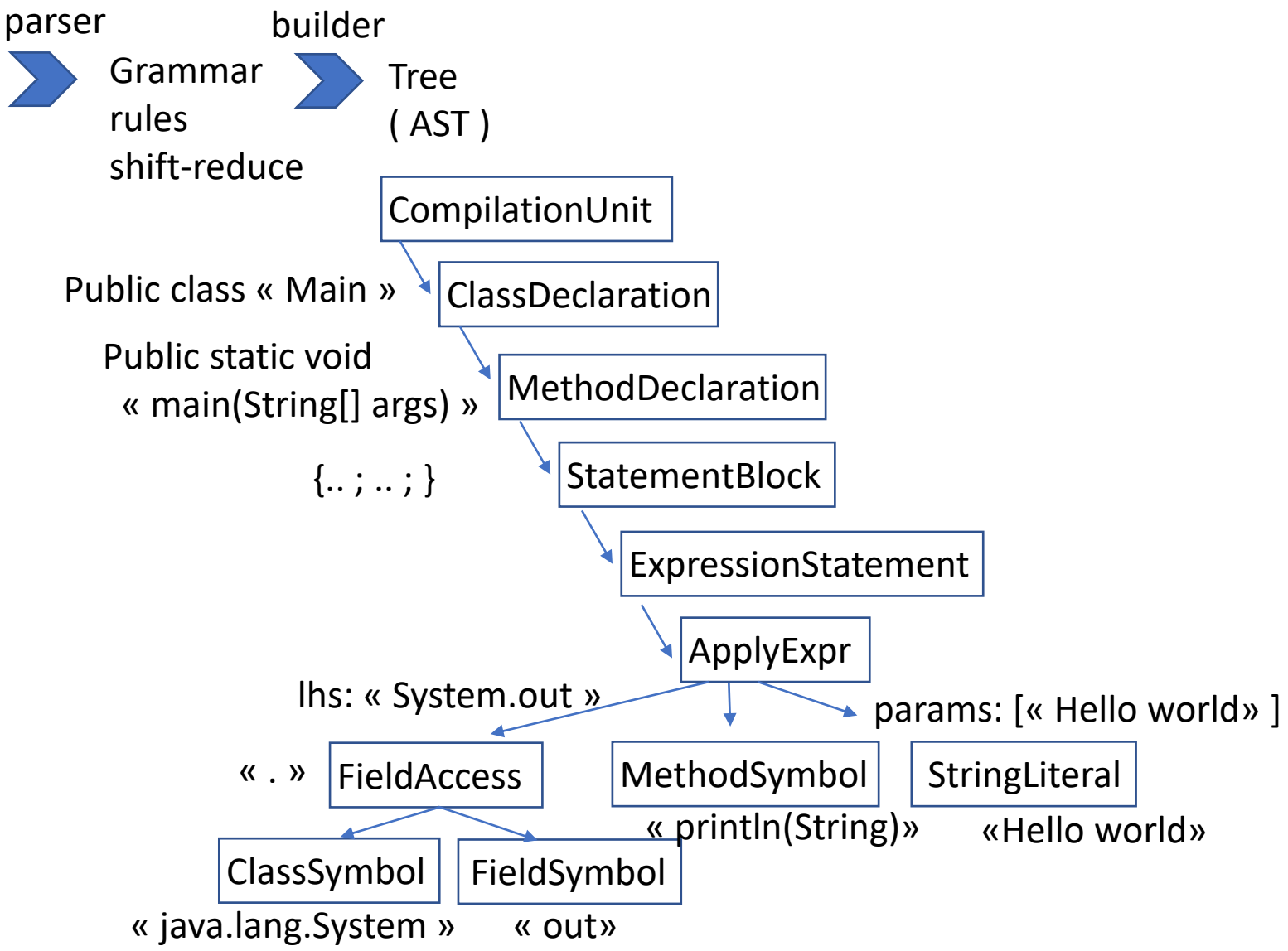
Detailed JDK extract.. JavacParser.java

```
case FOR: {
    nextToken();
    accept(LPAREN);
    List<JCStatement> inits = token.kind == SEMI ? List.nil() : forInit();
    if (inits.length() == 1 &&
        inits.head.hasTag(VARDEF) &&
        ((JCVariableDecl) inits.head).init == null &&
        token.kind == COLON) {
        JCVariableDecl var = (JCVariableDecl)inits.head;
        accept(COLON);
        JCExpression expr = parseExpression();
        accept(RPAREN);
        JCStatement body = parseStatementAsBlock();
        return F.at(pos).ForeachLoop(var, expr, body);
    } else {
        accept(SEMI);
        JCExpression cond = token.kind == SEMI ? null : parseExpression();
        accept(SEMI);
        List<JCExpressionStatement> steps = token.kind == RPAREN ? List.nil() : forUpdate();
        accept(RPAREN);
        JCStatement body = parseStatementAsBlock();
        return F.at(pos).ForLoop(inits, cond, steps, body);
    }
}
```

// <=Rule for
for(Type var : collection) stmt

// <=Rule for
for(init ; cond ; update) stmt

Full AST « Hello World »



AST to Text … Pretty.java
read easy code to understand AST

Pretty.java

```
781
782     public void visitForLoop(JCForLoop tree) {
783         try {
784             print("for (");
785             if (tree.init.nonEmpty()) {
786                 if (tree.init.head.hasTag(VARDEF)) {
787                     printExpr(tree.init.head);
788                     for (List<JCStatement> l = tree.init.tail; l.nonEmpty(); l = l.tail) {
789                         JCVariableDecl vdef = (JCVariableDecl)l.head;
790                         print(", " + vdef.name);
791                         if (vdef.init != null) {
792                             print(" = ");
793                             printExpr(vdef.init);
794                         }
795                     }
796                 } else {
797                     printExprs(tree.init);
798                 }
799             }
800             print("; ");
801             if (tree.cond != null) printExpr(tree.cond);
802             print("; ");
803             printExprs(tree.step);
804             print(") ");
805             printStat(tree.body);
806         } catch (IOException e) {
807             throw new UncheckedIOException(e);
808         }
809     }
```

Declaration

Something that can be prefixed with access modifier
« public|protected|private » « static » « final »

A declaration produces a « symbol », that can be imported / used

Statement

Something that can be suffixed by « ; »
or wrapped in « { ; ; } »

A statement has no type

No push/pop values on stack … statement need expressions

Expression

Something that can be wrapped by « (..) »

A statement has a type amoung
- Primitive type
- Pointer to class/enum/interface/..
- Array

Evaluating an expression => pop arguments
+ eval
+ push result value on stack

Declaration examples

```
public class Foo {

    private int field1;

    public Foo() {
    }

    public int getField1() {
        return field1;
    }

    public static class NestedBar {
    }

}
```

Class Declaration

- Field Declaration
- Constructor Declaration
- Method Declaration
- Nested class Declaration

Statement / Expression Examples

```
int x = 0 ;
```

LocalVariableDeclaration … statement, initializer: expression

```
x = 1 ;

int y = x + 1 ;

y = x = 2 ;

f() ;
```

ExpressionStatement … statement containing Expression

Symbol

Java (Source Code File)

.jar / .class library
Contains Symbol Tables
By [index] internally in bytecode
By [name] externally

Declaration

produce

Symbol

Compile/link

load

At Compile-Time
Import
+ resolve (ambiguous names)
+ type-check
... use in AST

At Runtime
Resolve by unique symbol name
+ redo type-check
... use in bytecode

Explicit Reflection Loading / Implicit Link for bytecode

Explicit from reflect code:
Class.forName()
/ ClassLoader.load() / .define()
/ Class.getMethod() / .getField()

May throw
ReflectiveOperation**Exception**

- ReflectiveOperationException
  - ClassNotFoundException
  - IllegalAccessException
  - InstantiationException
  - InvocationTargetException
  - NoSuchFieldException
  - NoSuchMethodException

Symbol

load

At Runtime
Resolve by unique
symbol name
+ redo type-check

Implicit Link For bytecode
Resolved at first usage

May throw Linkage**Error**

- Throwable
  - Error
    - LinkageError
      - BootstrapMethodError
      - ClassCircularityError
      - ClassFormatError
      - ExceptionInInitializerError
      - IncompatibleClassChangeError
        - AbstractMethodError
        - IllegalAccessError
        - InstantiationError
        - NoSuchFieldError
        - NoSuchMethodError
      - NoClassDefFoundError
      - UnsatisfiedLinkError
      - VerifyError

Type Descriptor & Symbol Name Mangling

Primitive Type => V, Z(boolean), B(byte), I(int),J(long), F(float),D(double),..

Array Type => « [» typeName

Class => type descriptor: « L » « pack »/ « subpack » /« ClassName » « ; »
symbol name : « pack » . « subpack » . « ClassName »
FQN (Fully Qualified Name)

Field => FQN « $ » « fieldName »

Method => type descriptor: « (type1 type2 ..typeN) returnType »
symbol name : «methodName(type1 type2 ..typeN) »

Example Type Descriptor & Symbol Names

```java
void fVoid();
byte fByte(byte p);
char fChar(char p);
int fInt(int p);
long fLong(long p);
float fFloat(float p);
double fDouble(double p);
boolean fBool(boolean p);

Boolean fBoolean(Boolean p);

boolean[] fArrayBool(boolean[] p);
Boolean[] fArrayBoolean(Boolean[] p);
```

```
$ javap -c -s target/classes/test/TestSignatures.class
Compiled from "TestSignatures.java"
interface test.TestSignatures {
  public abstract void fVoid();
    descriptor: ()V

  public abstract byte fByte(byte);
    descriptor: (B)B

  public abstract char fChar(char);
    descriptor: (C)C

  public abstract int fInt(int);
    descriptor: (I)I

  public abstract long fLong(long);
    descriptor: (J)J

  public abstract float fFloat(float);
    descriptor: (F)F

  public abstract double fDouble(double);
    descriptor: (D)D

  public abstract boolean fBool(boolean);
    descriptor: (Z)Z

  public abstract java.lang.Boolean fBoolean(java.lang.Boolean);
    descriptor: (Ljava/lang/Boolean;)Ljava/lang/Boolean;

  public abstract boolean[] fArrayBool(boolean[]);
    descriptor: ([Z)[Z

  public abstract java.lang.Boolean[] fArrayBoolean(java.lang.Boolean[]);
    descriptor: ([Ljava/lang/Boolean;)[Ljava/lang/Boolean;
}
```

Notice: Generic Types => same as <Object>
exact Type Erased in Descriptor
« Type Erasure »

```
<T> void fList1(List<? extends T> p1);

<T> void fList2(List<T> p1);

void fList3(List<Foo> p1);

void fList4(@SuppressWarnings("rawtypes") List p1);
```

```
public abstract <T> void fList1(java.util.List<? extends T>);
  descriptor: (Ljava/util/List;)V

public abstract <T> void fList2(java.util.List<T>);
  descriptor: (Ljava/util/List;)V

public abstract void fList3(java.util.List<test.Foo>);
  descriptor: (Ljava/util/List;)V

public abstract void fList4(java.util.List);
  descriptor: (Ljava/util/List;)V
```

Overload method signatures

Can not have 2 methods overload differing only by template type

```
38    public void compileErrorOverload(List<Foo> p);
39    public void compileErrorOverload(List<Bar> p);
```

Errors (2 items)

- Erasure of method compileErrorOverload(List<Bar>) is the same as another method in type TestSignatures
- Erasure of method compileErrorOverload(List<Foo>) is the same as another method in type TestSignatures

(not an error in other langages like C++)

Notice 2 : return type not in symbol name

Can not have 2 methods overload
differing only by return type or template type

```
38    public int    compileErrorOverload();
39    public double compileErrorOverload();
```

Errors (2 items)

- Duplicate method compileErrorOverload() in type TestSignatures
- Duplicate method compileErrorOverload() in type TestSignatures

Compiled OK … BUT change in CLASSPATH => LinkError

```java
public class Bar {

    public int field1;
```

```java
// previously compiled with..
/*
public int field1;
*/
// changed at runtime with..
public long field1;
```

Debug

- LinkTestApp [Java Application]
  - test.LinkTestApp at localhost:49773
    - Thread [main] (Suspended (uncaught exception NoSuchFieldError))
      - LinkTestApp.testBarField() line: 20
      - LinkTestApp.testFields() line: 12
      - LinkTestApp.main(String[]) line: 6
    - C:\apps\jdk\jdk-8\bin\javaw.exe (1 août 2022, 14:02:57) [pid: 4852]

LinkTestApp.java

```java
11          testFooField();
12          testBarField(); // <= will throw Link error on first call
13      }
14      private static void testFooField() {
15          Foo obj = new Foo();
16          System.out.println(obj.field1);
17      }
18      private static void testBarField() { // method contains Link error: referenced field changed at runtime
19          Bar obj = new Bar();
20          System.out.println(obj.field1); // <= cause Link error... ".field1" changed at runtime
21      }
```

Program start OK

Error method partially execute .. OK !

but fail on first bytecode Link error

```
0
Exception in thread "main" java.lang.NoSuchFieldError: field1
        at test.LinkTestApp.testBarField(LinkTestApp.java:20)
        at test.LinkTestApp.testFields(LinkTestApp.java:12)
        at test.LinkTestApp.main(LinkTestApp.java:6)
```

Internally … « getfield » => « _fast_*getfield »

```
public static void testBarField();
  Code:
     0: new           #51                 // class test/Bar
     3: dup
     4: invokespecial #53                 // Method test/Bar."<init>":()V
     7: astore_0
     8: getstatic     #33                 // Field java/lang/System.out:Ljava/io/PrintStream;
    11: aload_0
    12: getfield      #54                 // Field test/Bar.field1:I
    15: invokevirtual #43                 // Method java/io/PrintStream.println:(I)V
    18: return
```

On First use….

Resolve Bytecode « getfield #fieldIdx»

Load + type-check

Class c = .. Class.forName(« test.Bar »)
f = c.getField(« field1 »);
assert f.getType().equals(int.class);

Bytecode instruction
« getfield #idx»
HOT REPLACED BY internal
« **_fast_getfield offset**»

Code replaced (?) or re-executed
with « throw new …Error() »

«fast_getfield» on next uses

Cf OpenJdk .. src/hotspot/share/interpreter/bytecodes.hpp

bytecodes

Internal Reserved fast_* bytecodes

bytecodes.hpp

```
219     _return                 = 177, // 0xb1
220     _getstatic              = 178, // 0xb2
221     _putstatic              = 179, // 0xb3
222     _getfield               = 180, // 0xb4
223     _putfield               = 181, // 0xb5
224     _invokevirtual          = 182, // 0xb6
225     _invokespecial          = 183, // 0xb7
226     _invokestatic           = 184, // 0xb8
227     _invokeinterface        = 185, // 0xb9
228     _invokedynamic          = 186, // 0xba
229     _new                    = 187, // 0xbb
```

```
// JVM bytecodes
_fast_agetfield        =
_fast_bgetfield        ,
_fast_cgetfield        ,
_fast_dgetfield        ,
_fast_fgetfield        ,
_fast_igetfield        ,
_fast_lgetfield        ,
_fast_sgetfield        ,

_fast_aputfield        ,
_fast_bputfield        ,
_fast_zputfield        ,
_fast_cputfield        ,
_fast_dputfield        ,
_fast_fputfield        ,
_fast_iputfield        ,
_fast_lputfield        ,
_fast_sputfield        ,

_fast_aload_0          ,
_fast_iaccess_0        ,
_fast_aaccess_0        ,
_fast_faccess_0        ,

_fast_iload            ,
_fast_iload2           ,
_fast_icaload          ,

_fast_invokevfinal     ,
_fast_linearswitch     ,
_fast_binaryswitch     ,
```

Idem « invoke* » => « fast_invoke* »

```
public static void testFooMethod();
  Code:
     0: new           #30                 // class test/Foo
     3: dup
     4: invokespecial #32                 // Method test/Foo."<init>":()V
     7: astore_0
     8: aload_0
     9: invokevirtual #65                 // Method test/Foo.f:()V
    12: return
```

On First use….

Resolve Bytecode « invoke* #methIdx»

Load + type-check

Class c = .. Class.forName(« test.Bar »)
m = c.getMethod(« field1(type1..typeN »);
assert m.getType().equals(…);

Bytecode instruction
« invoke* #idx»
HOT REPLACED BY internal
« **_fast_invoke* offset**»

Code replaced (?) or re-executed
with « throw new …Error() »

«fast_invoke*» on next uses

Different « invoke* » : {static|special|virtual|interface|dynamic}

static => … to call fixed (known) function, and update stack

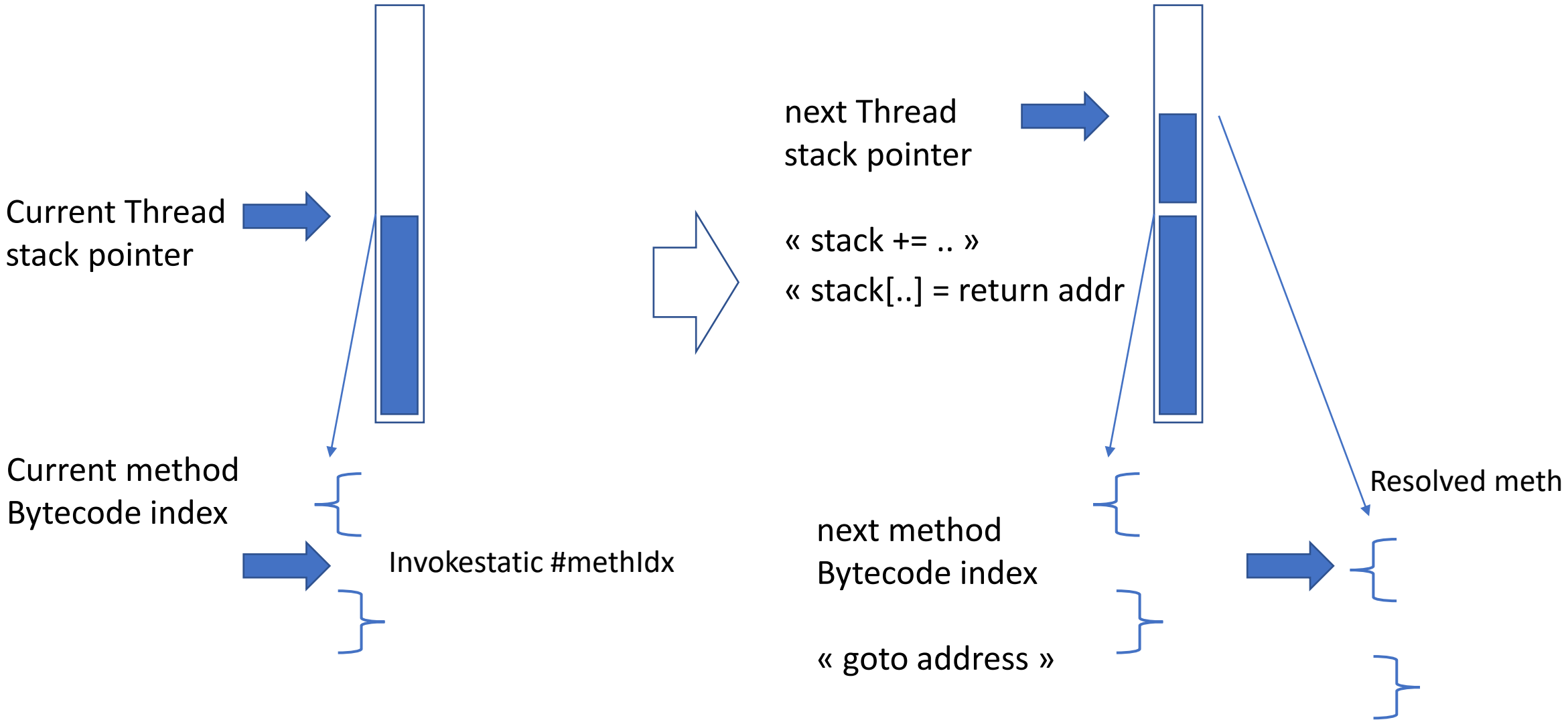
special => … idem … call fixed (known) function, and update stack
after « new »: « <init> » method, or « super() » method

virtual => … need array access to object class « virtual table », to determine exact method

interface => … need lookup interfaces table … then virtual table, to determine method

dynamic => … internal for JRE, allowing type evolution

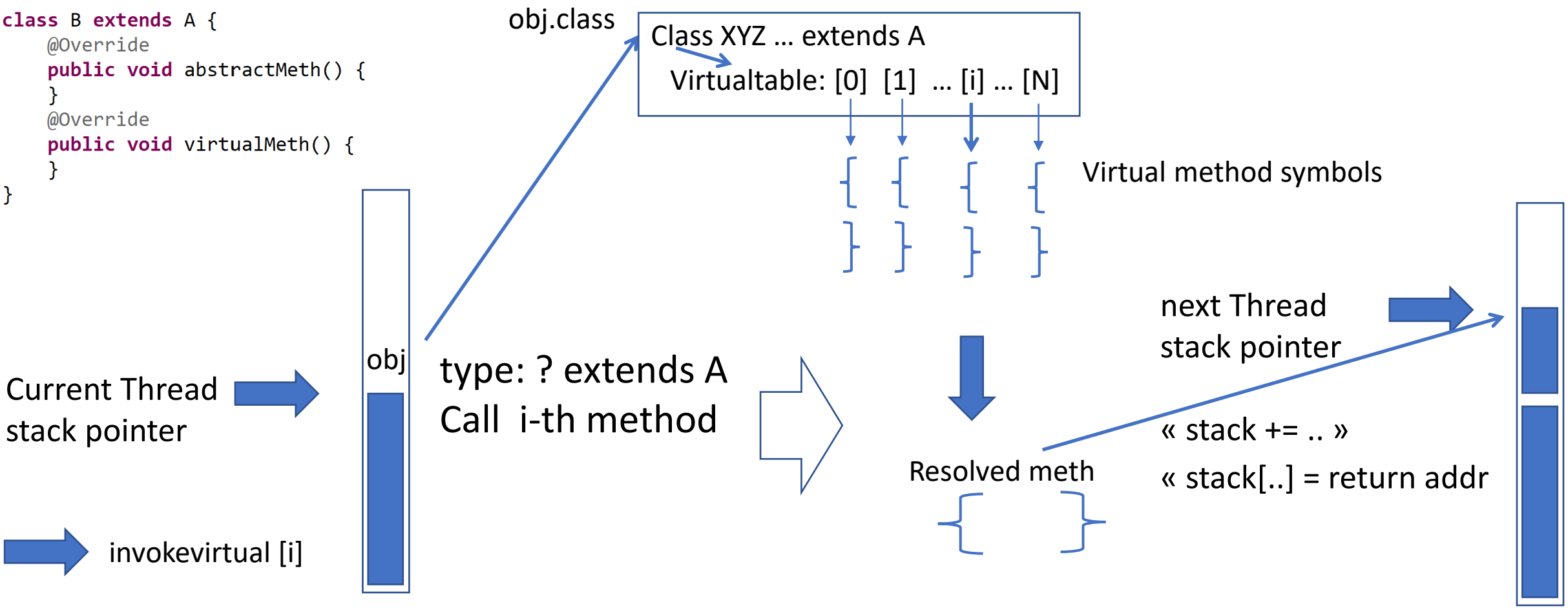
invokestatic



invokevirtual

```
public abstract class A {
    public abstract void abstractMeth();
    public void virtualMeth() {
        // maybe overriden
    }
}

class B extends A {
    @Override
    public void abstractMeth() {
    }
    @Override
    public void virtualMeth() {
    }
}
```



invokevirtual .. slower than invokestatic

Invokestatic …. O(1 call)

Invokevirtual …. O(**1 array access** + **push 1 extra param** « this » + 1 call)

invokeinterface

```
public interface IA {
    public void meth();
    public default void defaultMeth() {
        // maybe overriden
    }
}

class B extends Foo implements IA1, IA2, IA3,   IA,   IA4, IA5 {
//                                              /\
//                                          j-th position
    @Override
    public void meth() {
    }
    @Override
    public void defaultMeth() {
    }
}
```

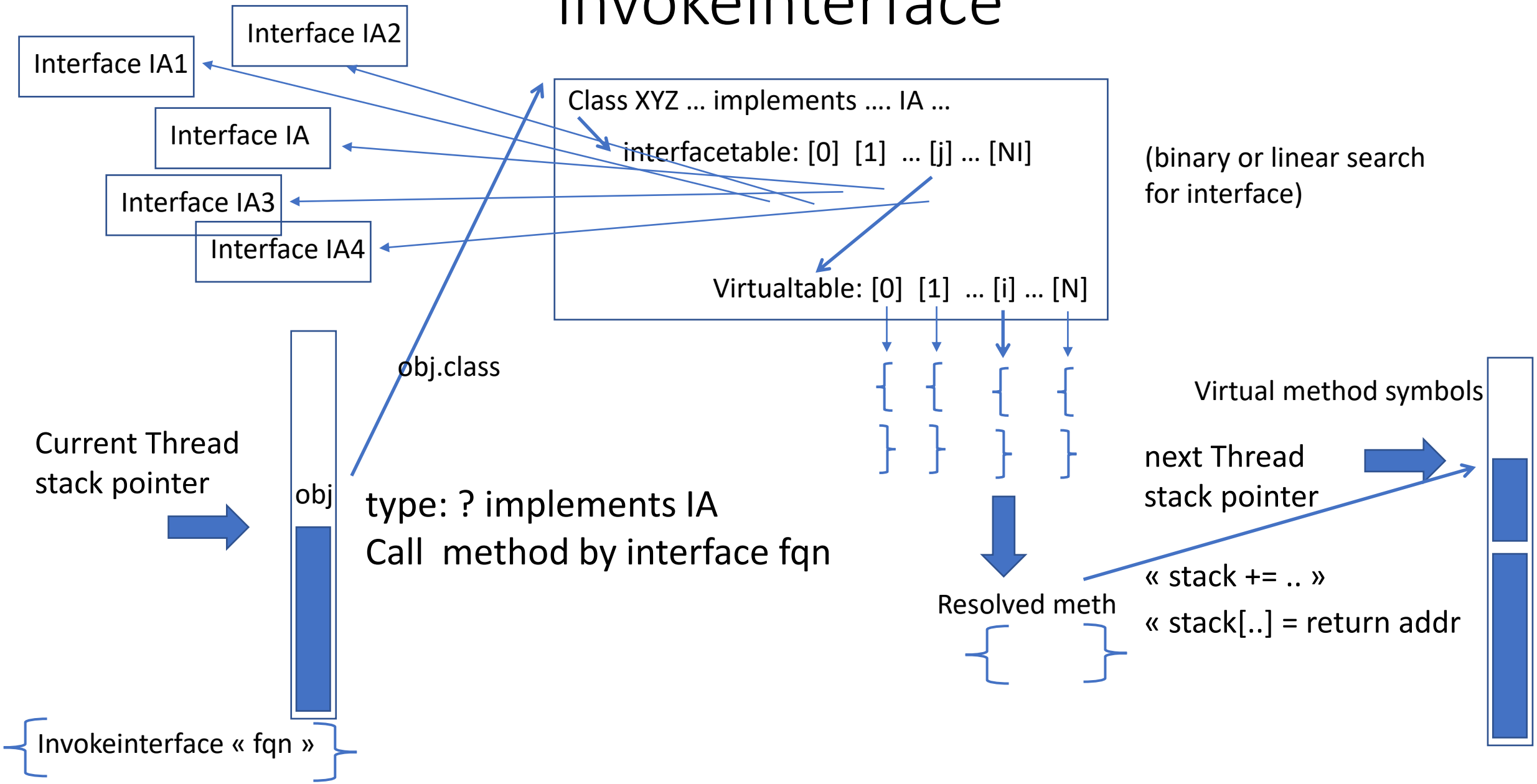
```
IA anyObj = new B();
anyObj.meth();  // <= invokeinterface
```

Performance Problem :

no relation between interface « IA » and extends class (here: « Foo »)

.. can not use virtualtable directly

invokeinterface



invokeinterface .. slower than invokevirtual

Invokestatic …. O(1 call)

Invokevirtual …. O(1 array access + push 1 extra param « this » + 1 call)

Invokeinterface …. O(1 pointer access
+ J linear OR log(J) binary search interface tables
+ 1 virtual table array access
+ push 1 extra param « this » + 1 call)

Remark: abstract class vs interface

For performance (with millions of calls…):

1/ Prefer declare virtual method in abstract class
Rather than method in interface

2/ do not use too many interfaces

3/ sort « implements » interfaces by most frequent usage first
(? .. If linear search)

Example: cf in JDK …

```
public abstract class InputStream {  public abstract int read(); }
public abstract class OutputStream {  public abstract void write(int data); }
```

ArrayList
extends AbstractList (..AbstractCollection)
implement List

```
// 1:
List<Integer> ls1 = new ArrayList<Integer>();
ls1.add(e);  // invokeinterface #41,  2          // InterfaceMethod java/util/List.add:(Ljava/lang/Object;)Z

// 2:
AbstractList<Integer> ls2 = new ArrayList<Integer>();
ls2.add(e);  // invokevirtual #51                // Method java/util/AbstractList.add:(Ljava/lang/Object;)Z

// 3:
ArrayList<Integer> ls3 = new ArrayList<Integer>();
ls3.add(e);  // invokevirtual #57                // Method java/util/ArrayList.add:(Ljava/lang/Object;)Z

// 4:
val ls4 = new ArrayList<Integer>();  // lombok val, or jdk>=15 var
ls4.add(e); // idem ArrayList .. invokevirtual
```

1: Slower than 2:,3:,4:

2: & 3: same perf (both invokevirtual, not same type)

4: idem 3:

HOTSPOT may optimize all as invokespecial !!

Declarations List, ArrayList, ..

```java
public interface Collection<E> extends Iterable<E> {
    boolean add(E e);
}

public abstract class AbstractCollection<E> implements Collection<E> {
    @Override
    public boolean add(E e) {
        throw new UnsupportedOperationException();
    }
}

public interface List<E> extends Collection<E> {
    boolean add(E e);
}

public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    @Override
    public boolean add(E e) {
        throw new UnsupportedOperationException();
    }
}

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    @Override
    public boolean add(E e) {
```

Miranda Methods
(implicit abstract methods from interface)

```java
public interface IFoo {
    void foo();
}

public abstract class AbstractFoo implements IFoo {
    //... implicitly miranda method
    // public @Override abstract void foo();
}

public final class Foo extends AbstractFoo {
    @Override
    public void foo() {
    }
}
```

```java
IFoo f1 = new Foo();
f1.foo(); // invoke interface

AbstractFoo f2 = new Foo();
f2.foo(); // invoke interface ?? => invoke virtual !!
          // invokevirtual #32                // Method test/TestMiranda$AbstractFoo.foo:()V

Foo f3 = new Foo();
f3.foo(); // invoke virtual
```

Questions ?