ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: framework overview and in-the-small features

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: framework overview

What's in a name

Initially was "Oak" (James Gosling, 1991), then "Green"

- Ruled out by the trademark lawyers

Twelve people locked in a room together with a "naming consultant"

- "How does this thing make you feel?"
- "What else makes you feel that way?"

After listing and sorting, 12 names were sent to the lawyers

- #1 was "Silk"
- Gosling's favorite was "Lyric" (#3)
- "Java" was # 4

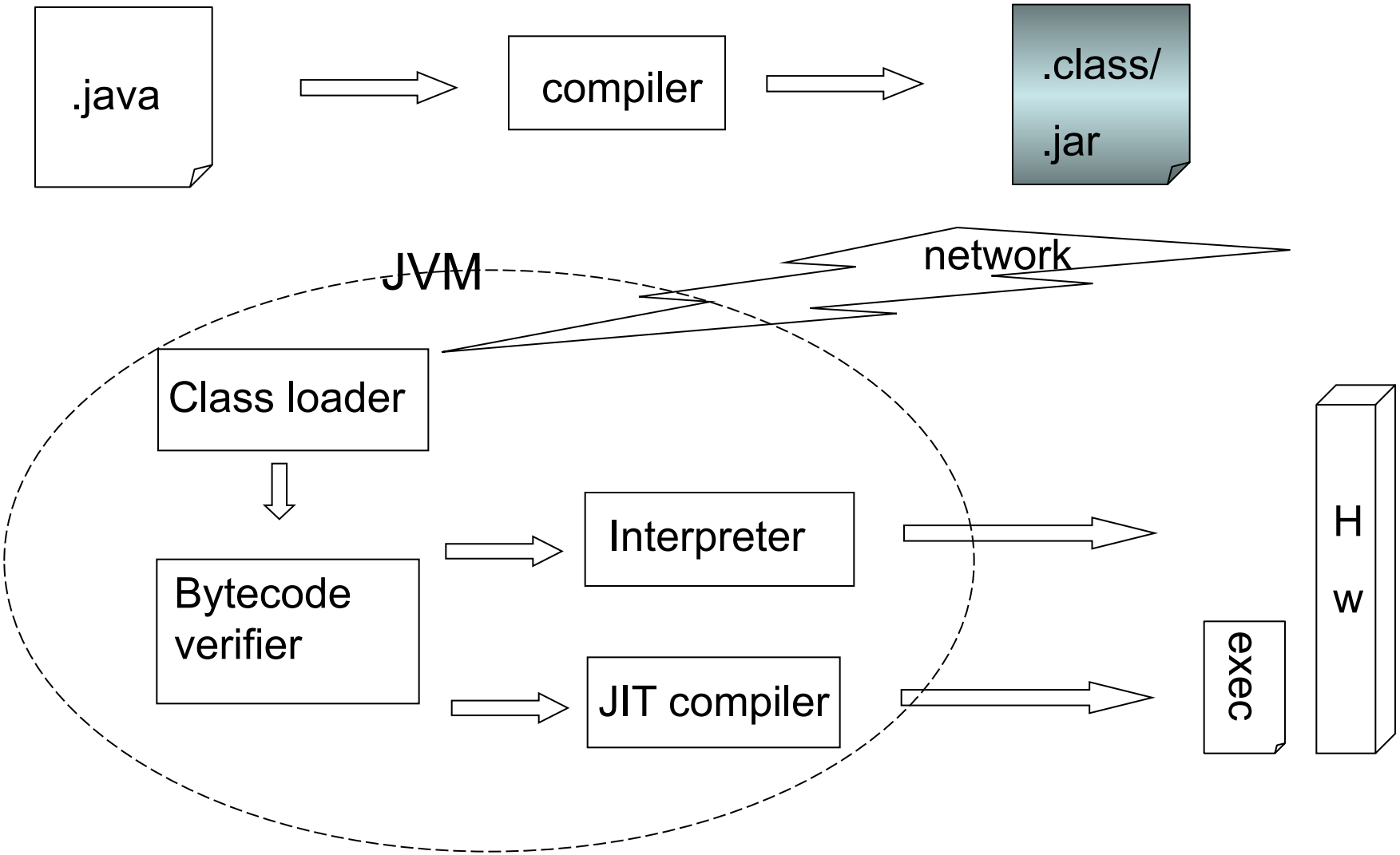
Version 1.0: 1995

Latest stable version: 7 Update 13 (1.2.13)

Java platform goals

- Write Once, Run Anywhere
- Built-in security
- Automatic memory management
- API + documentation generation
- Object-Oriented
- Familiar C/C++ syntax

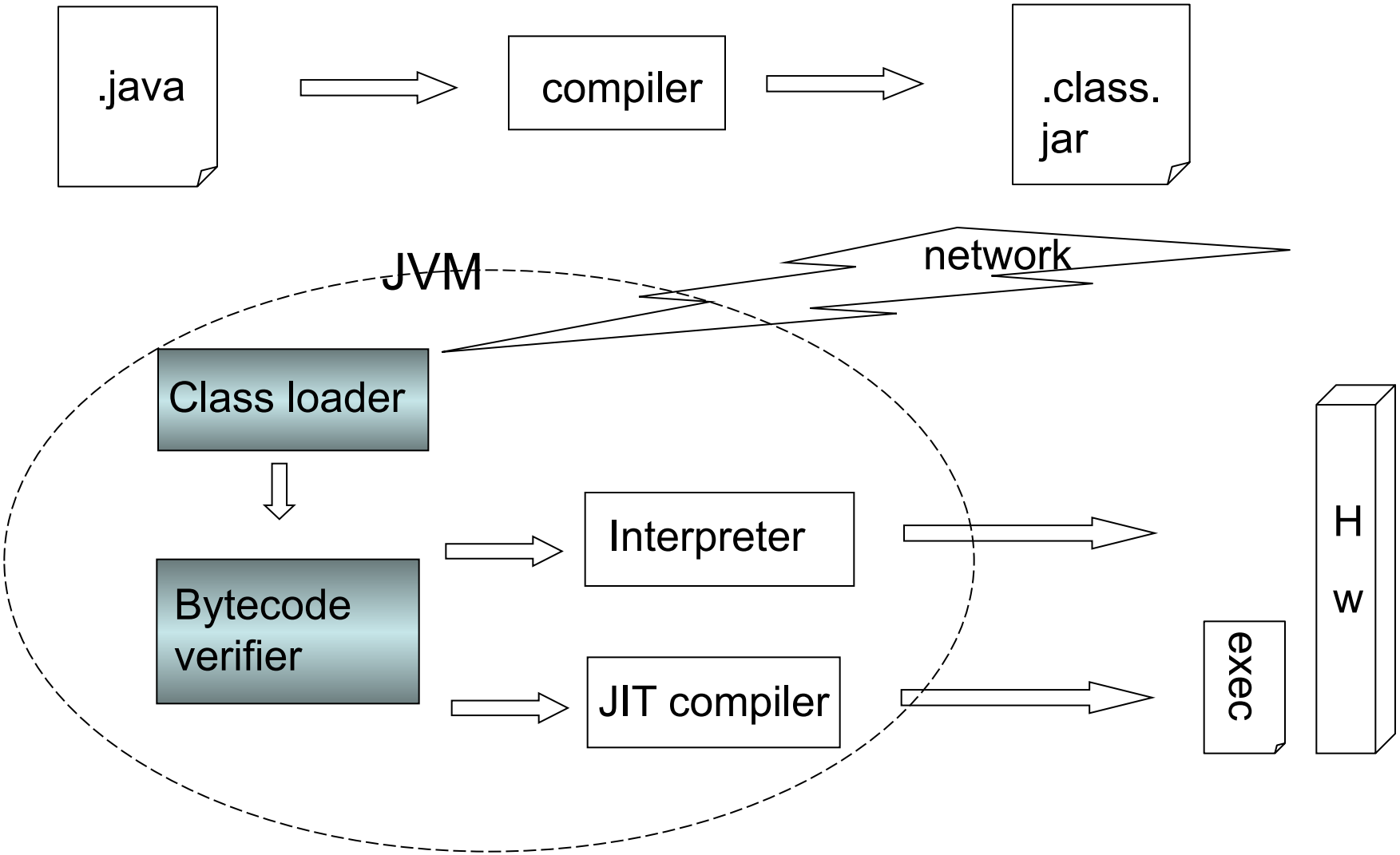
Write once, run anywhere



Bytecode

- Intermediate format resulting from Java compilation
- Instruction set of an architecture that
  - is stack-oriented
  - provides capability (object access rights)
- 1 bytecode instruction = 1 byte
- Executed by any platform-specific Virtual Machine (VM)

JVM overview



Security: language restrictions and support

- No pointers, no explicit memory de-allocation
- Checked type casts (at compile time and runtime)
- Enforced array bounds (at runtime)
- Security APIs
  - SecurityManager (standard security)
  - XML digital signature, Public Key Infrastructure, cryptographic services, authentication

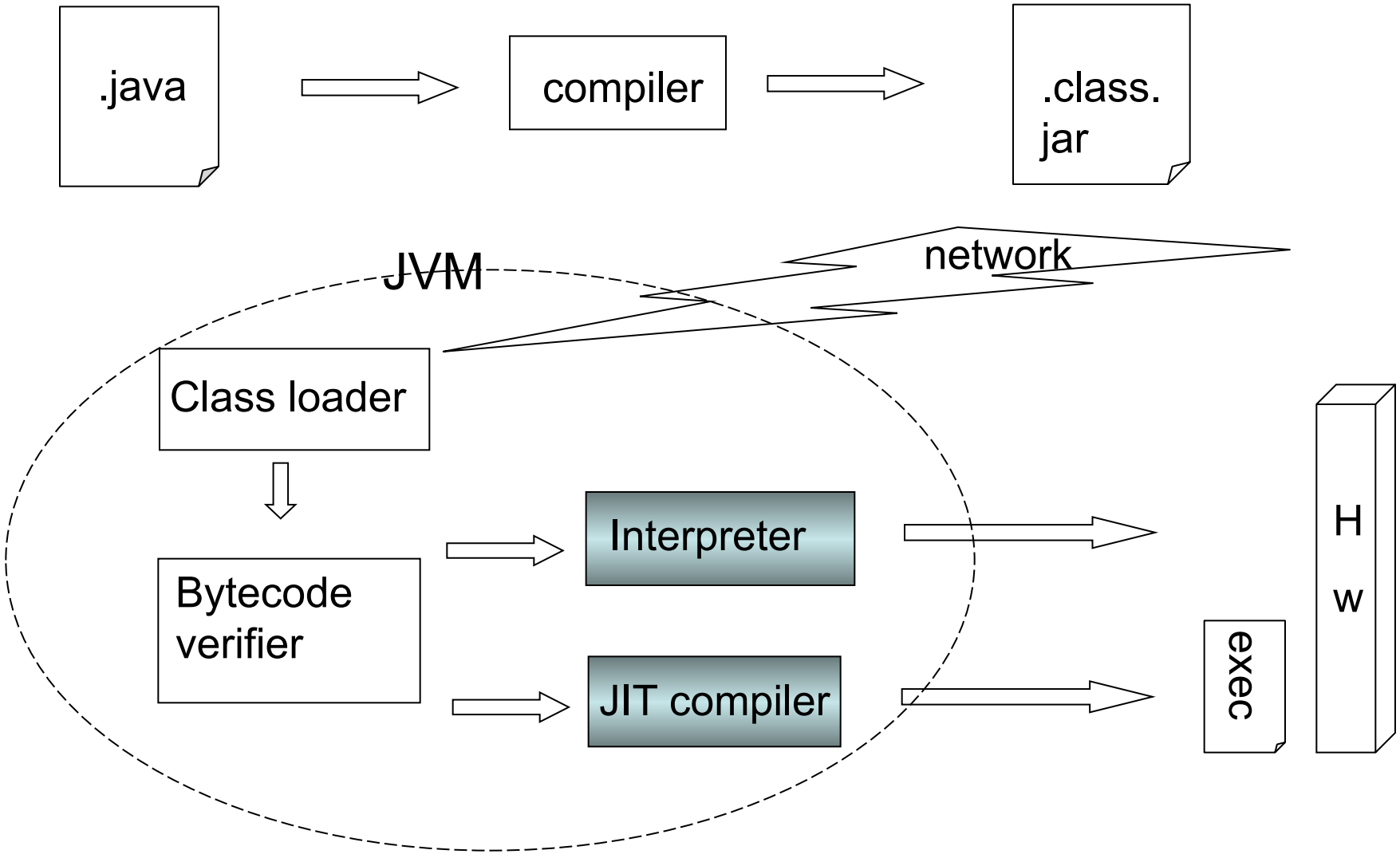
Security: class loaders

- Take care of files and file systems
- Locate libraries and dynamically load classes
- Partition classes into realms (e.g. local machine, local network, all the rest) and restrict what they can do

Security: Bytecode verifier

- Verifier checks bytecode using a “theorem prover”
  - Branches always to valid locations
  - Data always initialized
  - Types of parameters of bytecode instructions always correct
  - Data and methods access checked for visibility
  - Arbitrary bit patterns cannot get used as an address
  - No operand stack overflows and underflows

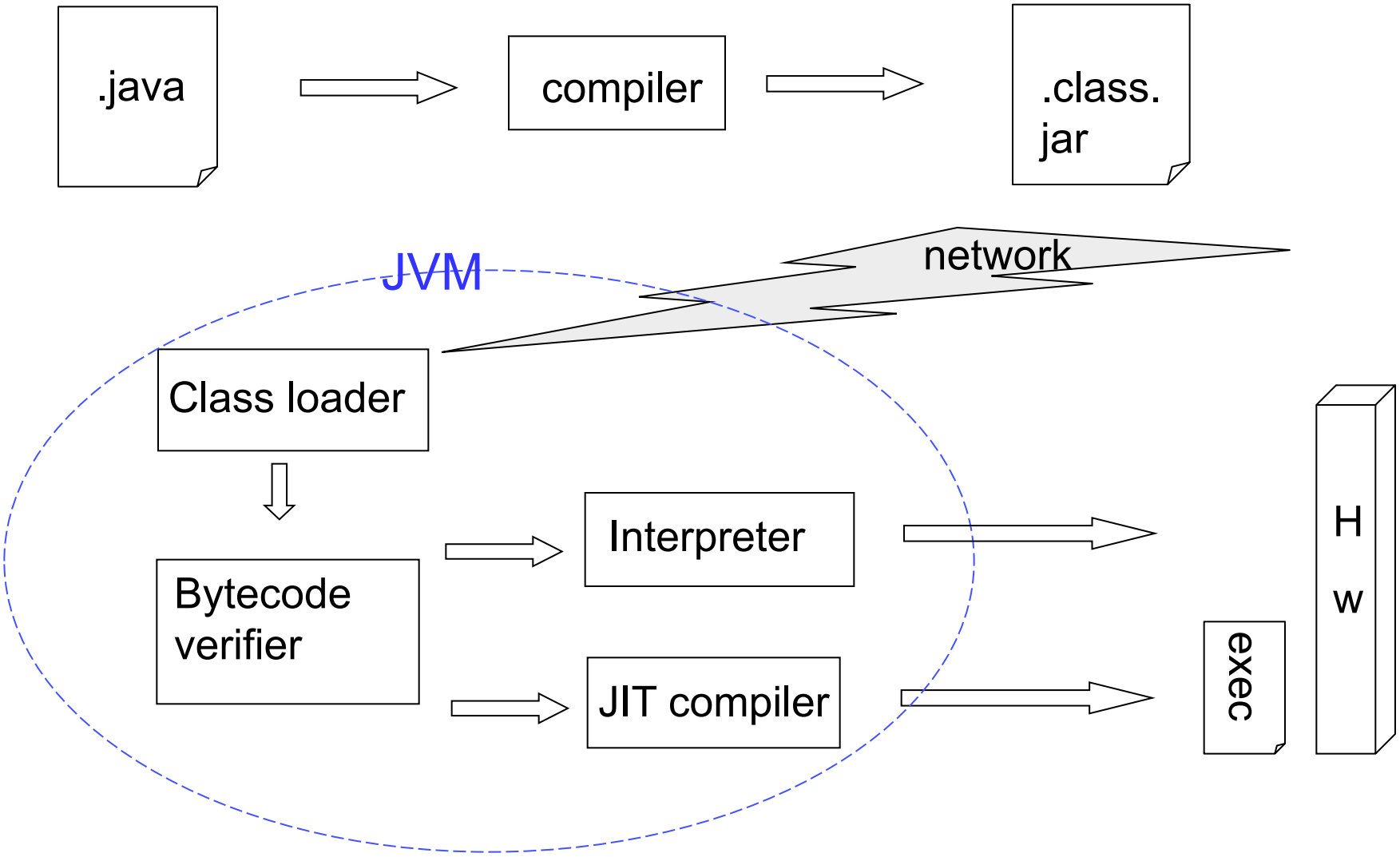
JVM: code generation



Code generation: HotSpot

- The interpreter is the software CPU of the JVM
  - Examines each bytecode and executes a unique native procedure
  - No native code is produced
- A JIT "compiler" converts the bytecode into native code just before running it
  - Keeps a log (cache) of the native code that it has to run to execute each bytecode
  - May optimize substituting a short set of instructions with a shorter/faster one
  - Like the back-end of a traditional compiler, the java compiler being the front-end
- HotSpot is the default SUN JVM since 2000

JVM Overview



JVM: more features

- Automated exception handling
  - Provides “root cause” debugging info for every exception
- Responsible for garbage collection
- Ships as JRE (VM + libraries)
- Can have other languages run on top of it, e.g.
  - JRuby (Ruby)
  - Rhino (JavaScript)
  - Jython (Python)
  - Scala
- From 6.0 scripting languages can be mixed with Java code

Command-line Java

- Compile

```
javac MainClass.java
```

- Execute

```
java MainClass
```

- Generate documentation

```
javadoc MainClass.java
```

- Generate an archive from `.class` files in current dir

```
jar cf myarchive.jar *.class
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Java: in-the-small language features

Encoding and formatting

- Uses unicode as encoding system: www.unicode.org
- Free format
  - Blanks, tabs, new lines, form feeds are only used to keep tokens separate
- Comments
  - Single line: `//Single line comment`
  - Multiple lines:
```
/* non-nested, multi-line
                    comment*/
```
  - Javadoc comment: `/** processed by javadoc */`

Identifiers

- No restriction on length
- Case sensitive
- Cannot start with a digit
- Cannot include `/` or `-`
- Cannot be a keyword

Annotations

Meta-data about programs

- Compiler flags

  e.g: `@Deprecated, @Override, @SuppressWarnings`
- Information that can be used for compilation (or other forms of code analysis)

  e.g.: `@Inherited,` application-defined such as `@RevisionId`
- Some runtime processing
  e.g.: application-defined

Keywords

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient
class	(goto)	protected	try
(const)	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof		

- Literals `null`, `true`, `false` are also reserved

Operators

- Access, method call: `.`, `[]`, `()`
- Postfix: *`expr++`*, *`expr--`* (R to L)
- Other unary: `++`*`expr`*, `--`*`expr`*, `+`, `-`, `~`, `!`, `new`, `(`*`aType`*`)`
- Arithmetic: `*`, `/`, `%`
- Additive: `+`, `-`
- Shift: <<, >>, >>>
- Relational: `<`, `>`, `<=`, `>=`, `instanceof`
- Equality: `==`, `!=`
- Logical (L to R): `&`, `^`, `|`, `&&`, `||`
- Ternary: *`condition`* `?` `(`*`expr1`*`):(`*`expr2`*`)` (R to L)
- Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`
- Precedence: from top to bottom
- Tip: don't rely too much on precedence rules: use parentheses

Type system Basics

- Primitive types
  - `boolean, byte, short, int, long, char, float, double`
- Reference types
  - `class`, `interface`, `[]`
- `null`
- Automatic widening conversions (no precision loss)
  - `byte` to `short` to `int` to `long`
  - `char` to `int`, `int` to `double`, `float` to `double`
- Automatic widening conversions (precision loss)
  - `int` to `float`, `long` to `float`, `long` to `double`
- A cast is required for narrowing conversions

```
int i = 3; long j = 5; i = (int) j
```

Widening conversions with precision loss

```
float g(int x){
      return x;
}
...
int i = 1234567890;
float f = g(i);
System.out.println(i - (int)f)
// output: -46
...
```

Wrapper types and autoboxing

- For each primitive type there is a wrapper type
  - `Boolean, Byte, Short, Integer, Long, Character, Float, Double`
- Starting from 5.0, autoboxing provides automatic conversions between primitive and wrapper types
- Pro: reduces code complexity
- Cons: not efficient, sometimes unexpected behavior

Some surprises of autoboxing

```
new Integer(7).equals(7)//true

new Long(7).equals(7)//false. True if equals(7L)

new Integer(7).equals(new (Long(7))) //false

new Integer(7) == 7 //true

new Long(7) == 7 //true

new Integer(7) == new Long(7)//compiler error
```

Control flow: conditional branch

Same syntax as in C/C++

```
if   (booleanExpr)
{
     // do something
}
else       // else is optional
{
     // do something else
}
```

Control flow: loops

```
while (booleanExpr)
{
      // execute body
      // until booleanExpr becomes false
}


do
{
      // execute body (at least once)
      // until booleanExpr becomes false
}
while (booleanExpr);
```

Control flow: `for` loop

```
for (int i=0; i < n; i++)
{
      // execute loop body n times
}

// equivalent to the following
int i=0;
while (i < n)
{
      // executes loop body n times
      i++;
}
```

Control flow: enhanced `for` loop

Introduced in Java 5.0

```
for (variable : collection)
{
      // loop body
}
```

- **`collection`** is an array or an object of a class that implements **`interface Iterable`**
  - more on classes and interfaces later

- Executes the loop body for every element of the **`collection`**, assigned iteratively to **`variable`**

Control flow: `switch` selector

```
switch  (Expr)
{
      case Value1: instructions;
           break;
      case Value2: instructions;
           break;
      // ...
      default: instructions;
}
```

`Expr` can be of type:

- `byte`, `short`, `int`, `char` (or wrapped counterparts)
- `enum` types
- `String` (compared with `equals`) (new in Java 7)

Breaking the control flow: `break`

`label: [while | do | for]`

- Identifies a loop
- (Or a code block)

`break optionalLabel;`

- Within a loop or a **`switch`**
- No label: exit the loop or switch
- With label:
  - within loop: jump out of the loop to label **`optionalLabel`**
  - within **`switch`**: jump out of **`switch`** block to label **`optionalLabel`**

Breaking the control flow: `continue`

`label: [while | do | for]`

- Identifies a loop
- (Or a code block)

`continue optionalLabel;`

- Within a loop
- No label: skip the remainder of the current iteration and continue with the next iteration
- With label:
  - skip the remainder of the current iteration and continue with the next iteration of the loop with label **`optionalLabel`**