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


Coming next (Java SE 8, 18.3.2014):
  - lambda expressions (closures)
  - embedded JavaScript
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

1. Java files (.java) are compiled by a compiler into class files (.class)
2. Class files can be packaged into a .jar file
3. The .jar file is loaded by the class loader
4. The class loader verifies the bytecode
5. Bytecode is then interpreted by the interpreter
6. Or, the bytecode can be compiled by a JIT (Just-In-Time) compiler
7. The compiled bytecode is then executed by the JVM
8. The JVM can connect to the network to load additional classes or resources

Java and C# in depth
Bytecode

- Intermediate format resulting from Java compilation
  - Instruction set of an architecture that
    - is stack-oriented (no registers)
    - provides capability (object access rights)
  - 1 bytecode instruction = 1 byte

- Executed by any platform-specific Virtual Machine (VM)
Bytecode format

- JVM loads class file → gets a stream of bytecodes
- One bytecode instruction: opcode + ≥0 operands
- Each opcode is associated with a mnemonic
  - 03 → icnst_0 // pushes int 0 on stack
  - 3b → istore_0 // pops int from stack to local in pos 0
  - 84 00 01 → iinc 0, 1 // increments local in pos 0 by 1
  - 1a → iload_0 // pushes int from local in pos 0 on stack
  - 05 → icnst_2 // pushes int 2 on stack
  - 68 → imul // pops 2 int values, multiplies them and puts the result on the stack
Example of bytecode translation

```java
class SimpleMath{
    byte inflexible_add(){
        byte x = 2;
        byte y = 2;
        byte z = (byte) (x + y);
        return z;
    }
}
```
# Bytecode example

<table>
<thead>
<tr>
<th>Opcode mnemonics</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>iconst_2</td>
<td>push an integer constant 2 into the stack</td>
</tr>
<tr>
<td>istore_1</td>
<td>pop into local in pos 1 (x)</td>
</tr>
<tr>
<td>iconst_2</td>
<td>push an integer constant 2 into the stack</td>
</tr>
<tr>
<td>istore_2</td>
<td>pop into local in pos 2 (y)</td>
</tr>
<tr>
<td>iload_1</td>
<td>push x into the stack</td>
</tr>
<tr>
<td>iload_2</td>
<td>push y into the stack</td>
</tr>
<tr>
<td>iadd</td>
<td>sum the two top values on the stack and push the result</td>
</tr>
<tr>
<td>int2byte</td>
<td>convert result into byte</td>
</tr>
<tr>
<td>istore_3</td>
<td>pop into local in pos 3 (z)</td>
</tr>
<tr>
<td>iload_3</td>
<td>push z into the stack</td>
</tr>
<tr>
<td>ireturn</td>
<td>return result (z)</td>
</tr>
</tbody>
</table>
Java and C# in depth

JVM overview

- .java
- compiler
- .class
- .jar

Class loader

Bytecode verifier

Interpreter

JIT compiler

JVM

network

H

W

exec
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

1. Java source code (.java)
2. Compiler
3. Class file (.class)
4. JAR file (.jar)
5. JVM
6. Class loader
7. Bytecode verifier
8. Interpreter
9. JIT compiler
10. Network
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 often occurring short sets of instructions (“hot spots”) with shorter/faster ones
  - Like the back-end of a traditional compiler, the java compiler being the front-end

- HotSpot is the default SUN JVM since 2000
HotSpot client and server

- **HotSpot client VM**
  - For platforms typically used for client applications (e.g. GUI)
  - Tuned for reducing start-up time and memory footprint
  - Invoked by using `–client` when launching an app

- **HotSpot server VM**
  - For all platforms
  - Tuned for max program execution speed
  - Invoked by using `–server` when launching an app

- Both use an interpreter to launch applications, and an adaptive compiler optimizing code hot spots

- They use different code inline policies and heap defaults
JVM Overview

- .java
- compiler
- .class
- .jar

Class loader

Bytecode verifier

Interpreter

JIT compiler

network

HW

eexec
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**
  
  \texttt{javac MainClass.java}

- **Execute**
  
  \texttt{java MainClass}

- **Generate documentation**
  
  \texttt{javadoc MainClass.java}

- **Generate an archive from .class files in current dir**
  
  \texttt{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](http://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

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Double</th>
<th>Int</th>
<th>Super</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Else</td>
<td>Interface</td>
<td>Switch</td>
</tr>
<tr>
<td>Break</td>
<td>Extends</td>
<td>Long</td>
<td>Synchronized</td>
</tr>
<tr>
<td>Byte</td>
<td>Final</td>
<td>Native</td>
<td>This</td>
</tr>
<tr>
<td>Case</td>
<td>Finally</td>
<td>New</td>
<td>Throw</td>
</tr>
<tr>
<td>Catch</td>
<td>Float</td>
<td>Package</td>
<td>Throws</td>
</tr>
<tr>
<td>Char</td>
<td>For</td>
<td>Private</td>
<td>Transient</td>
</tr>
<tr>
<td>Class</td>
<td>(goto)</td>
<td>Protected</td>
<td>Try</td>
</tr>
<tr>
<td>(const)</td>
<td>If</td>
<td>Public</td>
<td>Void</td>
</tr>
<tr>
<td>Continue</td>
<td>Implements</td>
<td>Return</td>
<td>While</td>
</tr>
<tr>
<td>Default</td>
<td>Import</td>
<td>Short</td>
<td></td>
</tr>
<tr>
<td>Do</td>
<td>Instanceof</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 (possible precision loss)**
  - `int to float, long to float, long to double`

- A cast is required for narrowing conversions
  ```java
  int i = 3; long j = 5; i = (int) j
  ```
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

```java
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++

```java
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

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

// equivalent to the following
```java
int i=0;
while (i < n)
{
    // executes loop body n times
    i++;
}
```
Control flow: enhanced for loop

Introduced in Java 5.0

```java
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: \texttt{switch} selector

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

Expr can be of type:
- \texttt{byte}, \texttt{short}, \texttt{int}, \texttt{char} (or wrapped counterparts)
- \texttt{enum} types
- \texttt{String} (compared with \texttt{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`