301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-05*: The JVM instruction set*

Outline

- The JVM instruction set architecture
  - Execution model
  - Instruction format & Addressing modes
  - Types and non-orthogonality of instructions
  - Classes of instructions

  ➔Chapter 2 and 3 of the JVM Specification

The JVM interpreter loop

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
}   while (there is more to do);
```

Instruction set properties

- 32 bit stack machine
- Variable length instruction set
  - One-byte opcode followed by arguments
- Simple to very complex instructions
- Symbolic references
- Only relative branches
- Byte aligned (except for operands of `tableswitch` and `lookupswitch`)
- Compactness vs. performance

JVM Instruction Set

- Load and store (operand stack <-> local vars)
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack manipulation
- Control transfer
- Method invocation and return
- Monitor entry/exit

Instruction format

- Each instruction may have different "forms" supporting different kinds of operands.
- **Example**: different forms of "iload" (i.e. push)

Assembly code	Binary instruction code layout			
`iload_0`	26			Pushes local variable 0 on operand stack
`iload_1`	27			
`iload_2`	28			
`iload_3`	29			
`iload` *n*	21	*n*		
`wide iload` *n*	196	21	*n*	

Runtime memory

- Memory:
  - Local variable array (frame)
  - Operand stack (frame)
  - Object fields (heap)
  - Static fields (method area)
- JVM stack instructions
  - implicitly take arguments from the top of the operand stack of the current frame
  - put their result on the top of the operand stack
- The operand stack is used to
  - pass arguments to methods
  - return a result from a method
  - store intermediate results while evaluating expressions
  - store local variables

JVM Addressing Modes

- JVM supports three addressing modes
  - Immediate addressing mode
    - Constant is part of instruction
  - Indexed addressing mode
    - Accessing variables from **local variable array**
  - Stack addressing mode
    - Retrieving values from **operand stack** using pop

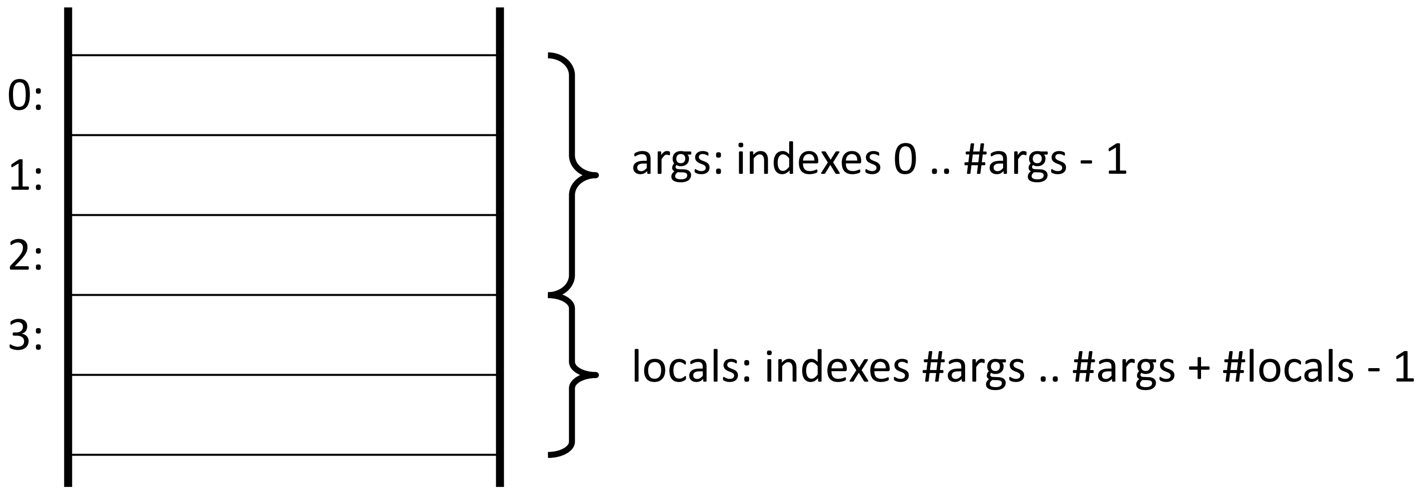
Instruction-set: typed instructions

- JVM instructions are explicitly typed: different opCodes for instructions for integers, floats, arrays, reference types, etc.
- This is reflected by a naming convention in the first letter of the opCode mnemonics
- **Example:** different types of "load" instructions

`i`	`int`
`l`	`long`
`s`	`short`
`b`	`byte`
`c`	`char`
`f`	`float`
`d`	`double`
`a`	`for reference`

`iload`	integer load
`lload`	long load
`fload`	float load
`dload`	double load
`aload`	reference-type load

Instruction-set: accessing arguments and locals in the Local Variable array



Instruction examples:

iload_1	istore_1
iload_3	astore_1
aload_5	fstore_3
aload_0	

- A ***load*** instruction takes something from the args/locals area and pushes it onto the top of the operand stack.
- A ***store*** instruction pops something from the top of the operand stack and places it in the args/locals area.

Opcode “pressure” and non-orthogonality

- Since op-codes are bytes, there are at most 256 distinct ones
- Impossible to have for each instruction one opcode *per type*
- Careful selection of which types to support for each instruction
- Non-supported types have to be converted
- Result: **non-orthogonality** of the Instruction Set Architecture

Type support in the JVM instruction set

- Design choice: almost no support for `byte`, `char` and `short` – using `int` as "computational type"

Table 2.11.1-A. Type support in the Java Virtual Machine instruction set

opcode	`byte`	`short`	`int`	`long`	`float`	`double`	`char`	`reference`
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aaload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			*iushr*	*lushr*				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					*fcmpl*	*dcmpl*		
Tcmpg					*fcmpg*	*dcmpg*		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Specification of an instruction: iadd

iadd ***iadd***

Operation	Add `int`
Format	*iadd*
Forms	*iadd* = 96 (0x60)
Operand Stack	..., *value1*, *value2* → ..., *result*
Description	Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack. The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values. Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

Computational Types

Table 2.11.1-B. Actual and Computational types in the Java Virtual Machine

Actual type	**Computational type**	**Category**
`boolean`	`int`	1
`byte`	`int`	1
`char`	`int`	1
`short`	`int`	1
`int`	`int`	1
`float`	`float`	1
`reference`	`reference`	1
`returnAddress`	`returnAddress`	1
`long`	`long`	2
`double`	`double`	2

Compiling Constants, Local Variables, and Control Constructs

- Sample Code
- Can compile to

```
void spin() {
        int i;
        for (i = 0; i < 100; i++) {
            ; // Loop body is empty
        }
}
```

```
0  iconst_0       // Push int constant 0
1  istore_1       // Store into local variable 1 (i=0)
2  goto 8         // First time through don't increment
5  iinc 1 1       // Increment local variable 1 by 1 (i++)
8  iload_1        // Push local variable 1 (i)
9  bipush 100     // Push int constant 100
11  if_icmplt 5   // Compare and loop if less than (i < 100)
14  return        // Return void when done
```

- Pushing constants on the operand stacks
- Incrementing local variable, comparing

int vs. double: lack of opcodes for double requires longer bytecode

- Sample Code

```
void dspin() {
        double i;
        for (i = 0.0; i < 100.0; i++) {
            ;   // Loop body is empty
        }
}
```

- Can compile to

0 dconst_0	// Push double constant 0.0
1 dstore_1	// Store into local variables 1 and 2
2 goto 9	// First time through don't increment
5 dload_1	// Push local variables 1 and 2
6 dconst_1	// Push double constant 1.0
7 dadd	// Add; there is no dinc instruction
8 dstore_1	// Store result in local variables 1 and 2
9 dload_1	// Push local variables 1 and 2
10 ldc2_w #4	// Push double constant 100.0
13 dcmpg	// There is no if_dcmplt instruction
14 iflt 5	// Compare and loop if less than(i < 100.0)
17 return	// Return void when done

Accessing literals in the Constant Pool

- Sample Code

```
void useManyNumeric() {
       int i = 100;
       int j = 1000000;
       long l1 = 1;
       long l2 = 0xffffffff;
       double d = 2.2;
       ...do some calculations...  }
```

- Can compile to

0 bipush **100**	// Push small int constant with bipush
2 istore_1	
3 **ldc #1**	// Push large int (**1000000**) with ldc
5 istore_2	
6 lconst_**1**	// A tiny long value uses fast lconst_**1**
7 lstore_3	
8 **ldc2_w #6**	// Push long **0xffffffff** (that is, int -1)
11 lstore 5	// Any long can be pushed with ldc2_w
13 ldc2_w #8	// Push double constant **2.200000**
16 dstore 7	
...	...do those calculations…

Parameter passing: Receiving Arguments

- Sample Code

```
int addTwo(int i, int j) {
       return i + j;
}
```

- Can compile to

0 iload_1	// Push value of local variable 1 (i)
1 iload_2	// Push value of local variable 2 (j)
2 iadd	// Add; leave int result on operand stack
3 **ireturn**	// Return int result

- Local variable 0 used for **this** in instance methods

- Sample Code

```
static int addTwo(int i, int j) {
       return i + j;
}
```

- Can compile to

```
0 iload_0
1 iload_1
2 iadd
3 ireturn
```

Invoking Methods

- Sample Code

```
int add12and13() {
        return addTwo(12, 13);
}
```

- Can compile to

0 aload_0	// Push local variable 0 (this)
1 bipush 12	// Push int constant 12
3 bipush 13	// Push int constant 13
5 **invokevirtual #4**	// Method Example.addtwo(II)I
8 ireturn	// Return int on top of operand stack; // it is the int result of addTwo()

- **invokevirtual** causes the allocation of a new frame, pops the arguments from the stack into the local variables of the callee (putting **this** in 0), and passes the control to it by changing the **pc**
- A resolution of the symbolic link is performed
- **ireturn** pushes the top of the current stack to the stack of the caller, and passes the control to it. Similarly for **dreturn**, …
- **return** just passes the control to the caller

Other kinds of method invocation

- **invokestatic –** for calling methods with "static" modifiers
  - **this** is not passed, arguments are copied to local vars from 0
- **invokespecial –** for calling ***constructors***, which are not dynamically dispatched, ***private methods*** or ***superclass methods***.
  - **this** is always passed
- **invokeinterface –** same as ***invokevirtual***, but used when the called method is declared in an interface (requires a different kind of method lookup)
- **invokedynamic –** introduced in Java SE 7 to support dynamic typing
  - We shall discuss it when presenting lambdas

Working with objects

- Sample Code
- Can compile to

```
Object create() {
        return new Object();
}
```

```
0 new #1              // Class java.lang.Object
3 dup
4 invokespecial #4    // Method java.lang.Object.<init>()V
7 areturn
```

- Objects are manipulated essentially like data of primitive types, but through **references** using the corresponding instructions (e.g. **areturn**)

Accessing fields (instance variables)

- Sample Code

```
void setIt(int value) {
    i = value;
}
int getIt() {
    return i;
}
```

- Can compile to

```
Method void setIt(int)
0 aload_0
1 iload_1
2 putfield #4 // Field Example.i I
5 return
Method int getIt()
0 aload_0
1 getfield #4 // Field Example.i I
4 ireturn
```

- Requires resolution of the symbolic reference in the constant pool
- Computes the offset of the field in the class, and uses it to access the field in **this**
- Similar for ***static variables***, using **putstatic** and **getstatic**

Using Arrays

- Sample Code
- Can compile to

```
void createBuffer() {
        int buffer[];
        int bufsz = 100;
        int value = 12;
        buffer = new int[bufsz];
        buffer[10] = value;
        value = buffer[11];
}
```

```
0 bipush 100        // Push int constant 100 (bufsz)
2 istore_2          // Store bufsz in local variable 2
3 bipush 12         // Push int constant 12 (value)
5 istore_3          // Store value in local variable 3
6 iload_2           // Push bufsz and...
7 newarray int      // ... create new int array of that length
9 astore_1          // Store new array in buffer
10 aload_1          // Push buffer
11 bipush 10        // Push int constant 10
13 iload_3          // Push value
14 iastore          // Store value at buffer[10]
15 aload_1          // Push buffer
16 bipush 11        // Push int constant 11
18 iaload           // Push value at buffer[11]...
19 istore_3         // ...and store it in value
20 return
```

Compiling switches (1)

- Sample Code

```
int chooseNear(int i) {
    switch (i) {
        case 0:  return  0;
        case 1:  return  1;
        case 2:  return  2;
        default: return -1;
} }
```

- Can compile to

```
0 iload_1                   // Push local variable 1 (argument i)
1 tableswitch 0 to 2:       // Valid indices are 0 through 2
      0: 28                 // If i is 0, continue at 28
      1: 30                 // If i is 1, continue at 30
      2: 32                 // If i is 2, continue at 32
      default:34            // Otherwise, continue at 34
28 iconst_0                 // i was 0; push int constant 0...
29 ireturn                  // ...and return it
30 iconst_1                 // i was 1; push int constant 1...
31 ireturn                  // ...and return it
32 iconst_2                 // i was 2; push int constant 2...
33 ireturn                  // ...and return it
34 iconst_m1                // otherwise push int constant -1...
35 ireturn                  // ...and return it
```

tableswitch

Operation Access jump table by index and jump

Format

tableswitch
<0-3 byte pad>
defaultbyte1
defaultbyte2
defaultbyte3
defaultbyte4
lowbyte1
lowbyte2
lowbyte3
lowbyte4
highbyte1
highbyte2
highbyte3
highbyte4
jump offsets...

Forms *tableswitch* = 170 (0xaa)

Operand Stack ..., *index* →

...

tableswitch

A *tableswitch* is a variable-length instruction. Immediately after the *tableswitch* opcode, between zero and three bytes must act as padding, such that *defaultbyte1* begins at an address that is a multiple of four bytes from the start of the current method (the opcode of its first instruction). Immediately after the padding are bytes constituting three signed 32-bit values: *default*, *low*, and *high*. Immediately following are bytes constituting a series of *high* - *low* + 1 signed 32-bit offsets. The value *low* must be less than or equal to *high*. The *high* - *low* + 1 signed 32-bit offsets are treated as a 0-based jump table. Each of these signed 32-bit values is constructed as (*byte1* << 24) | (*byte2* << 16) | (*byte3* << 8) | *byte4*.

The *index* must be of type `int` and is popped from the operand stack. If *index* is less than *low* or *index* is greater than *high*, then a target address is calculated by adding *default* to the address of the opcode of this *tableswitch* instruction. Otherwise, the offset at position *index* - *low* of the jump table is extracted. The target address is calculated by adding that offset to the address of the opcode of this *tableswitch* instruction. Execution then continues at the target address.

The target address that can be calculated from each jump table offset, as well as the one that can be calculated from *default*, must be the address of an opcode of an instruction within the method that contains this *tableswitch* instruction.

Compiling switches (2)

- Sample Code

```
int chooseFar(int i) {
   switch (i) {
      case -100: return -1;
      case 0:    return  0;
      case 100:  return  1;
      default:   return -1;
} }
```

- Can compile to

```
0 iload_1
1 lookupswitch 3:
      -100: 36
         0: 38
       100: 40
   default: 42
36 iconst_m1
37 ireturn
38 iconst_0
39 ireturn
40 iconst_1
41 ireturn
42 iconst_m1
43 ireturn
```

- **lookupswitch** is used when the cases of the **switch** are sparse
- Each case is a pair <value: address>, instead of an offset in the table of addresses
- Cases are sorted, so binary search can be used

Note that only switches on **int** are supported: for other types conversions (**char**, **byte**, **short**) or non-trivial translations (**String**, using hashcode) are needed

Operand stack manipulation

- Sample Code
- Can compile to

```
public long nextIndex() {
        return index++;
}
   private long index = 0;
```

0 aload_0	// Push this
1 **dup**	// Make a copy of it

2 getfield #4	// One of the copies of this is consumed
	// pushing long field index,
	// above the original this

5 **dup2_x1**	// The long on top of the operand stack is
	// copied into the operand stack below the
	// original this

6 lconst_1	// Push long constant 1
7 ladd	// The index value is incremented...
8 putfield #4	// ...and the result stored in the field
11 lreturn	// The original value of index is on top of // the operand stack, ready to be returned

Operation Duplicate the top one or two operand stack values and insert two or three values down

Format

dup2_x1

Forms *dup2_x1* = 93 (0x5d)

Operand Stack

Form 1:

..., *value3*, *value2*, *value1* →

..., *value2*, *value1*, *value3*, *value2*, *value1*

where *value1*, *value2*, and *value3* are all values of a category 1 computational type (§2.11.1).

Form 2:

..., *value2*, *value1* →

..., *value1*, *value2*, *value1*

where *value1* is a value of a category 2 computational type and *value2* is a value of a category 1 computational type (§2.11.1).

Description Duplicate the top one or two values on the operand stack and insert the duplicated values, in the original order, one value beneath the original value or values in the operand stack.

Throwing Exceptions

- Sample Code
- Can compile to

```
void cantBeZero(int i) throws TestExc
{
        if (i == 0) {
            throw new TestExc();
        }}
```

```
0 iload_1               // Push argument 1 (i)
1 ifne 12               // If i==0, allocate instance and throw
4 new #1                // Create instance of TestExc
7 dup                   // One reference goes to its constructor
8 invokespecial #7      // Method TestExc.<init>()V
11 athrow               // Second reference is thrown
12 return               // Never get here if we threw TestExc
```

- **athrow** looks in the method for a **catch** block for the thrown exception using the **exception table**
- If it exists, the operand stack is cleared and control passed to the first instruction
- Otherwise the current frame is discarded and the same exception is thrown on the caller
- If no method catches the exception, the thread is aborted

try-catch

- Sample Code

```
void catchOne() {
        try {
            tryItOut();
        } catch (TestExc e) {
            handleExc(e);
        }}
```

- Can compile to

```
0 aload_0               // Beginning of try block
1 invokevirtual #6      // Method Example.tryItOut()V
4 return                // End of try block; normal return
5 astore_1              // Store thrown value in local var 1
6 aload_0               // Push this
7 aload_1               // Push thrown value
8 invokevirtual #5      // Invoke handler method:
                        // Example.handleExc(LTestExc;)V
11 return               // Return after handling TestExc

Exception table:
From To Target  Type
0    4  5       Class TestExc
```

- Compilation of **finally** more tricky

- Compiles a catch clause like another method
- The table records boundaries of **try** and is used by **athrow** to dispatch the control

Other Instructions

- Handling synchronization: **monitorenter**, **monitorexit**
- verifying instances: **instanceof**
- checking a cast operation: **checkcast**
- No operation: **nop**

Limitations of the Java Virtual Machine

- Max number of entries in **constant pool**: **65535** (count in ClassFile structure)
- Max number of **fields**, of **methods**, of **direct superinterfaces**: **65535** (idem)
- Max number of **local variables** in the local variables array of a frame: **65535**, also by the 16-bit local variable indexing of the JVM instruction set.
- Max **operand stack** size: **65535**
- Max number of **parameters** of a method: **255**
- Max length of field and method names: **65535** characters by the 16-bit unsigned length item of the CONSTANT_Utf8_info structure
- Max number of **dimensions** in an **array**: **255**, by the size of the *dimensions* opcode of the *multianewarray* instruction and by the constraints imposed on the *multianewarray*, *anewarray*, and *newarray* instructions

JIT Compilation in the OpenJDK HotSpot JVM

JIT Compilation vs AOT Compilation and Interpretation

- **AOT (Ahead Of Time) Compilation** leads to better performance in general
  - Allocation of variables without variable lookup at run time
  - Aggressive code optimization to exploit hardware features
- **Interpretation** facilitates interactive debugging and testing
  - Interpretation leads to better diagnostics of a programming problem
  - Procedures can be invoked from command line by a user
  - Variable values can be inspected and modified by a user
- **Just-In-Time Compilation** tries to obtain the advantages of both

JIT Compilation: not only in the JVM

- “Dynamic compilation” first described in a paper by J. McCarthy on LISP in 1960
- Present for example in
  - Java: JVM (Java Virtual Machine)
  - C#: CLR (Common Language Runtime)
  - Android: DVM (Dalvik Virtual Machine) or ART (Android RunTime)
- JIT compiler has access to dynamic runtime information, enabling it to make better optimizations (such as inlining functions)

JIT vs AOT Compilation

- Primary difference: a just-in-time compiler runs in the same process as the application and competes with the application for resources.
- Therefore compilation time is more important for a JIT compiler than for an AOT compiler.
- But JIT compilation can exploit new possibilities for optimization, such as **deoptimization** and **speculation**.
- A Java-based JIT compiler takes **bytecode** as input and translate it into **machine code** that the CPU executes directly.
- A Java JIT compiler also differs from an AOT compiler because the JVM verifies class files at load time. When it's time to compile, there's little need for parsing or verification.

HotSpot's JIT execution model

Based on four observations:

1. Most code is only executed uncommonly, so getting it compiled would waste resources that the JIT compiler needs.
2. Only a subset of methods is run frequently.
3. The interpreter is ready right away to execute any code.
4. Compiled code is much faster, but it is only available after the compilation process is over, which takes resources and time.

The resulting execution model is:

1. Code starts executing interpreted with no delay.
2. Methods that are found commonly executed (hot) are JIT compiled.
3. Once compiled code is available, the execution switches to it.

Identifying and compiling hot code in HotSpot

- The interpreter instruments the code that it executes, keeping:
  - a per-method count of the number of times a method is entered;
  - a per-method count of the times a branch back to the start of a loop is taken in the method.
- On method entry, the two numbers are added: if the result crosses a threshold, the method is enqueued for compilation.
- A compiler deamon thread then processes the compilation request. While compilation is in progress, interpreted execution continues.
- Once the compiled code is available, the interpreter branches off to it.

Multi-tiered execution

There is a trade-off:

- “fast-to-start-but-slow-to-execute” interpreter vs “slow-to-start-but-fast-to-execute” compiled code.
- The compiler can be designed to optimize less (the code is available sooner but doesn't perform as well) or more (faster code at a later time).
- A practical design that leverages this observation is to have **a multi-tier system**.
- HotSpot has a **three-tiered system** consisting of **the interpreter**, **the quick compiler**, and **the optimizing compiler**. Each tier represents a different trade-off between the delay of execution and the speed of execution.

The three tiers of execution

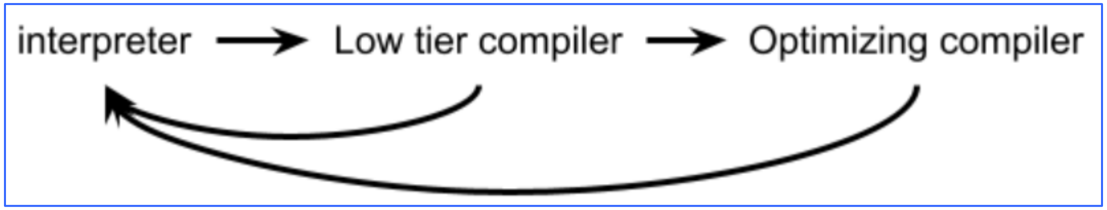
- Java code starts execution in the interpreter.
- When a method becomes warm (threshold: ≈1.500), it's enqueued for compilation by the quick compiler.
- Execution switches to that compiled code when it's ready.
- Method executing in the second tier is still instrumented: when it becomes hot (threshold: ≈10.000), then it's enqueued for compilation by the optimizing compiler.
- Execution continues in the second-tier compiled code until the faster code is available.

In HotSpot, for historical reasons, the second tier is known as **C1** or the **client compiler** and the optimizing tier is known as **C2**, or the **server compiler**.

Deoptimization and speculation

Usually method executions (can) pass in three phases:

But **deoptimization** can happen: the execution of the compiled method is stopped at some point, and the execution resumes in the interpreter at exactly the same point.



Two main possible causes:

- Corner cases in code
- **Speculation:** The compiler makes some assumption to generate better code: If an assumption is invalidated, then the thread that executes a method that makes the assumption deoptimizes in order to not execute code that's erroneous (being based on wrong assumptions).

Example of Speculation: Null checks in the C2 tier

- In Java, field or array access is usually guarded by a null check. Here is an example in pseudocode:

```
if (object == null) {
  throw new NullPointerException();
}
val = object.field;
```

- It's very uncommon for a NPE to not be caused by a programming error, so C2 speculates that NPEs never occur:

```
if (object == null) {
  deoptimize();
}
val = object.field;
```

- Clearly, if the object is null, execution must return to the interpreter which will throw the NPE.

Example of speculation: Class hierarchy analysis (CHA)

- Call in **`compiledMethod`** is virtual (subject to dynamic binding)
- If **`C`** is loaded but none of its sublcasses, the call can be "devirtualized" invoking **`C.virtualMethod`**

```
class C {
  void virtualMethod() {}
}

void compiledMethod(C c) {
  c.virtualMethod();
}
```

- JIT compilation of **`compileMethod`** can exploit this
- But if later a subclass of **`C`** is loaded, the compiled method could be incorrect: the method is marked for deoptimization

Deoptimization and safepoints

- Methods are compiled: deoptimization is only possible at locations known as **safepoints**.
- The JVM has to be able to reconstruct the state of execution so the interpreter can resume the thread where the compiled execution stopped.
- At a safepoint, a mapping exists between elements of the interpreter state (locals, locked monitors, and so on) and their location in compiled code, such as a register, stack, etc.
- Conflicting requirements: *common enough* safepoints, ensuring immediate deoptimization, vs *rare enough* safepoints, leaving the compiler the freedom to optimize between two of them.

Resources

- How the JIT compiler boosts Java performance in OpenJDK, by Roland Westrelin
  https://developers.redhat.com/articles/2021/06/23/how-jit-compiler-boosts-java-performance-openjdk
- "Just In Time" to understand, by Gabriele Pappalardo
  https://pages.di.unipi.it/corradini/Didattica/AP-21/SLIDES/GabrielePappalardo-Just_In_Time_to_understand.pdf