

COMPILER CONSTRUCTION

Intermediate Representations & Runtime Support

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2020



Chapter 10
Intermediate Representations

Chapter 12
Runtime Support



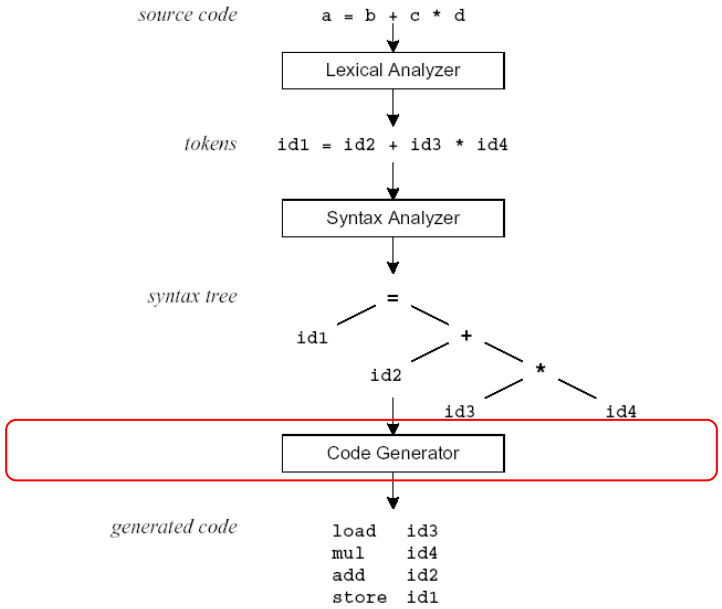
Outline

1. This file gives an overview of the **Intermediate Representation** (Ch. 10) for the compilers
2. This file also covers the *key concepts* of the **Runtime Support** (Ch. 12)
3. We talk about the intermediate language for Java
4. The above covers the basic knowledge required for the programming assignment #3



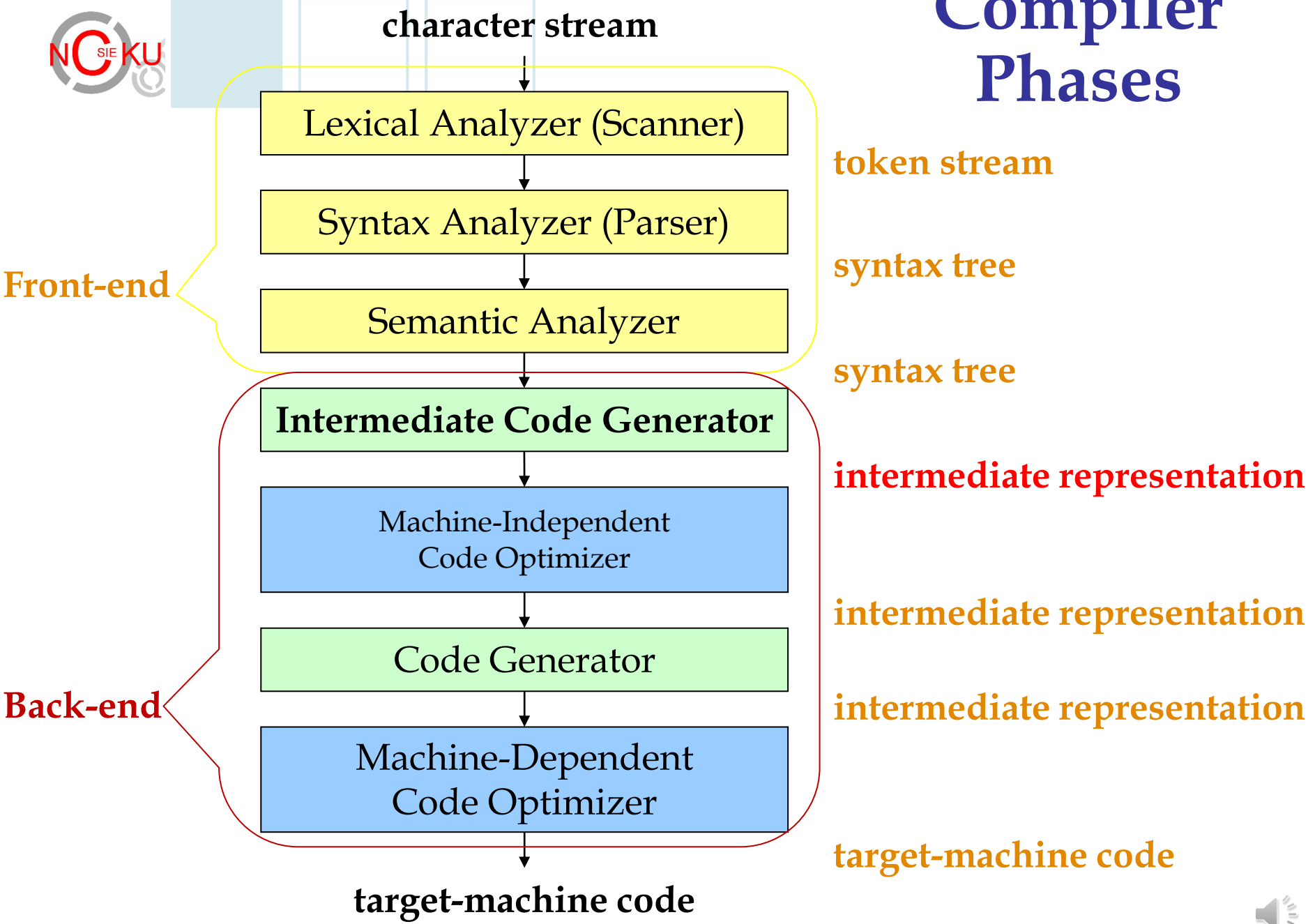
About the 3rd Assignment

- Lex and Yacc are able to do the following
- Now, our target is to generate the Java Assembly code
  - More precisely, the assembly code generation is at the Intermediate Code Generator phase of the compiler (see the next page)



A simplified view

Compiler Phases





Intermediate Representation

- An intermediate representation (IR)
  - is the **data structure** or **code** used internally by a compiler or virtual machine **to represent source code**
  - Compiler passes are performed on the IR
  - E.g., during the code generation process, an iteration process is performed to iterate over the IR node and generate the corresponding code for each IR node
- An IR is designed to be conducive for further processing, such as optimization and translation
- A "good" IR must be accurate
  - capable of representing the source code without loss of information and
  - independent of any particular source or target language
- An IR may take one of several forms
  1. an in-memory data structure, or
  2. a special tuple- or stack-based code readable by the program
  - In the 2nd case, it is also called **an intermediate language**

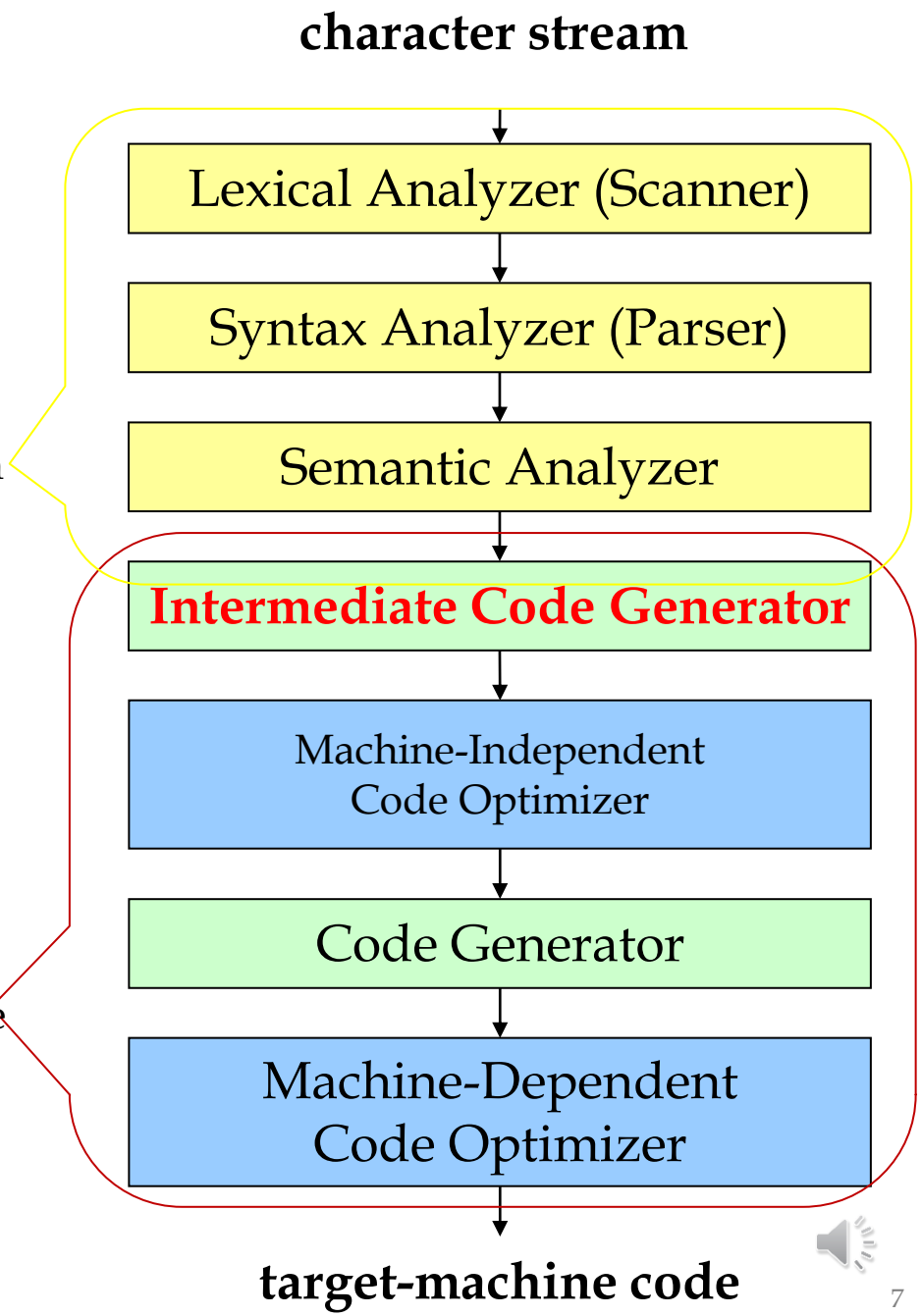
Courtesy of https://en.wikipedia.org/wiki/Intermediate_representation

Front- & Back-ends

- **Front-end** handles different source languages (e.g., C, C++)
- Conventionally, front-end is tightly coupled with back-end in a compiler tool
- E.g., GCC is **monolithic architecture**, making reusing part of their code almost impossible
- Huge obstacle if you want to try new ideas on compilers for real world impact

- **Back-end** is more close to target machine (e.g., ARM or x86 machines)
- Various back-ends are built for target-dependent optimizations

- Current trend is defining a clear Intermediate Language so as to **decouple the front-end and back-end**
- **The notable example is Clang and LLVM**





Classical Compiler Design (1/2)

- **Three phases structure**
  - The most popular design for a traditional compiler (like most C compilers)
  - whose major components are the front end, the optimizer and the backend, as shown in the figure below
  - **Communications are done through the intermediate representation(s), e.g., AST**
- Front end
  - parses source code, checking it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code
- AST
  - is optionally converted to a new representation for optimization, and the optimizer and backend are run on the code

Courtesy of http://www.aosabook.org/en/llvm.html



Classical Compiler Design (2/2)

- Optimizer
  - is responsible for doing a broad variety of **transformations** to try to improve the code's **performance**
    - such as minimizing the execution time (or energy consumption) by eliminating redundant computations
  - is usually more or less **independent of language and target**
- Back-end
  - also known as the **code generator**
  - then maps the **intermediate code** onto the **target instruction set**
  - In addition to making *correct* code, it is responsible for generating *good* code that **takes advantage of special features of the supported architecture**
  - Common parts of a compiler back-end include instruction selection, register allocation, and instruction scheduling
- This model applies equally well to interpreters and JIT compilers
  - The Java Virtual Machine (JVM) is also an implementation of this model,
  - which uses **Java bytecode** as the interface between the front end and optimizer

Courtesy of http://www.aosabook.org/en/llvm.html



Implications of the Three Phases Design

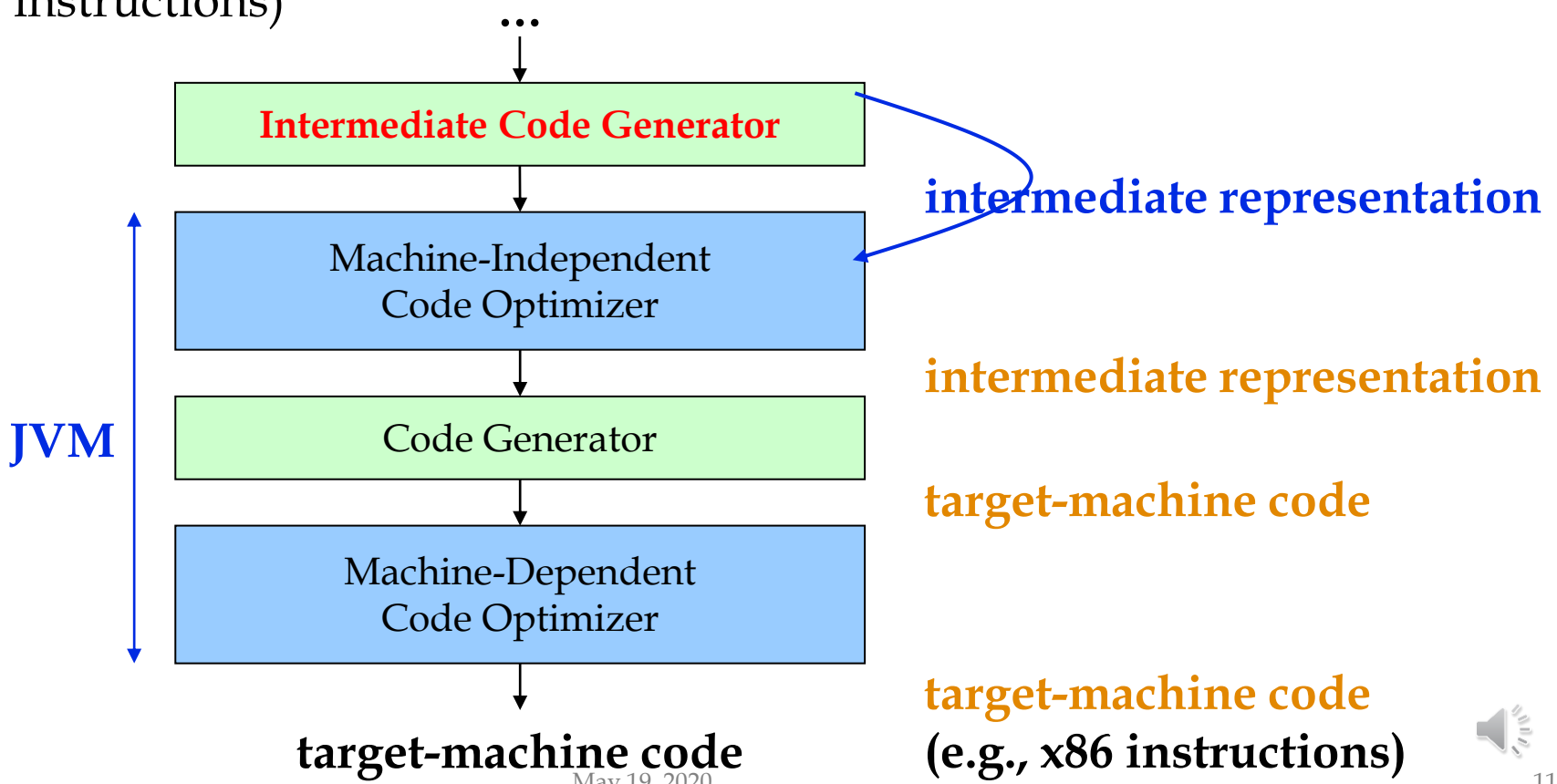
- The most important win of this classical design comes when a compiler decides to support **multiple source languages or target architectures**

- If the compiler uses a **common code representation** in its optimizer,
  - then a front-end can be written for any language that can compile to it, and
  - a back-end can be written for any target that can compile from it

Courtesy of http://www.aosabook.org/en/llvm.html

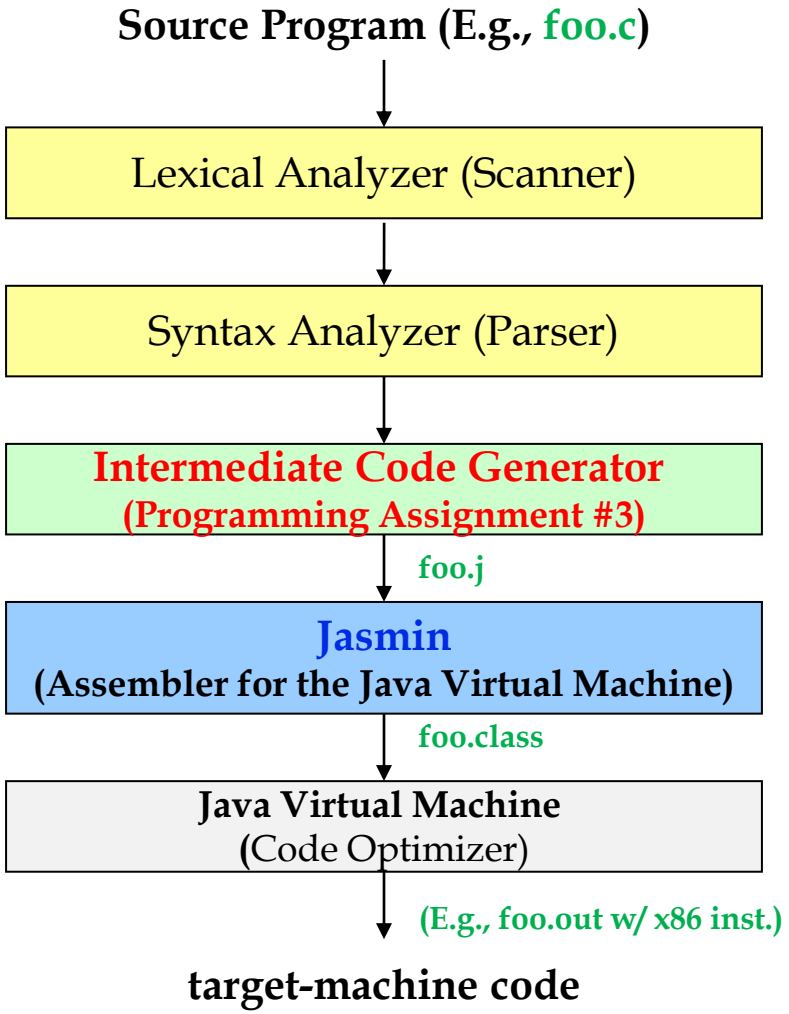
Java Virtual Machine Example

- For Java-like high-level languages, we can think that the generated intermediate representation (e.g., **Java bytecode**) is the input for Java Virtual Machine (JVM)
- Java Virtual Machine converts the **Java bytecode** (.class files) into the **machine code** accepted by the target machine (e.g., x86 instructions)





Workflow of Our 3rd Programming Assignment



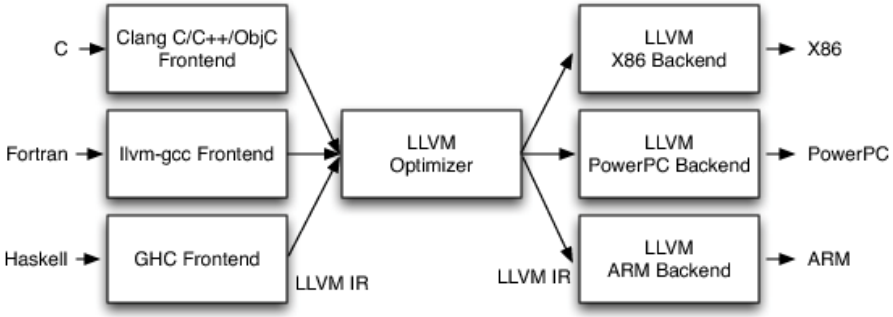


Another Three Phases Design Example: LLVM

In an LLVM-based compiler

- A front-end
  - is responsible for parsing, validating and diagnosing errors in the **input code**,
  - then translating the parsed code into **LLVM IR** (usually, but not always, by building an AST and then **converting the AST to LLVM IR**)
- LLVM IR
  - is optionally fed through a series of analysis and optimization passes which improve the code,
  - then is sent into a code generator to produce native machine code, as shown in the figure below
  - This is a very straightforward implementation of the three-phase design, but this simple description glosses over some of the power and flexibility that the LLVM architecture derives from LLVM IR

Multiple languages



Multiple targets



LLVM IR is a Complete Code Representation

- LLVM IR is both well specified and the ***only*** interface to the optimizer
  - This property means that all you need to know to write a front-end for LLVM is what LLVM IR is, and how it works
  - Since LLVM IR has a first-class textual form, it is both possible and reasonable to **build a front end that outputs LLVM IR** as ***text***,
  - then uses Unix pipes to **send it through the optimizer** sequence and code generator of your choice
  - Serialization: LLVM IR is (de)serialized to/from a binary format known as LLVM **bitcode**

- It might be surprising, but this is actually a pretty novel property to LLVM
  - Even the widely successful and relatively well-architected GCC compiler does not have this property: its **GIMPLE mid-level representation** is not a self-contained representation
  - As a simple example, when the GCC code generator goes to emit DWARF debug information, it reaches back and walks the source level "tree" form
  - GIMPLE itself uses a "tuple" representation for the operations in the code, but (at least as of GCC 4.5) still represents operands as **references back to the source level tree form**

- The implications of this are that **front-end authors need to know and produce GCC's tree data structures as well as GIMPLE to write a GCC front end**

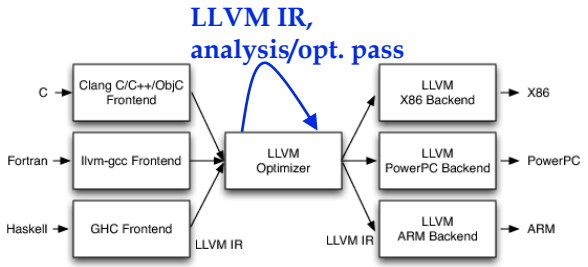
Courtesy of http://www.aosabook.org/en/llvm.html



LLVM is a Collection of Libraries

- LLVM is designed as a set of libraries,
  - rather than as a monolithic command line compiler like GCC or an opaque virtual machine like the JVM or .NET virtual machines
  - LLVM is an infrastructure, a collection of useful compiler technology that can be brought to bear on specific problems (like building a C compiler, or an optimizer in a special effects pipeline)

- The design of the optimizer
  - It reads LLVM IR in, chews on it a bit, then emits LLVM IR which hopefully will execute faster
  - It is organized as **a pipeline of distinct optimization passes** each of which is run on the input and has a chance to do something
  - Examples of passes are the inliner (which substitutes the body of a function into call sites), expression reassociation, loop invariant code motion
  - For example, at **-O3** it runs a series of 67 passes in its optimizer (as of LLVM 2.8)
  - **The sequence of applied passes does matter!!!**



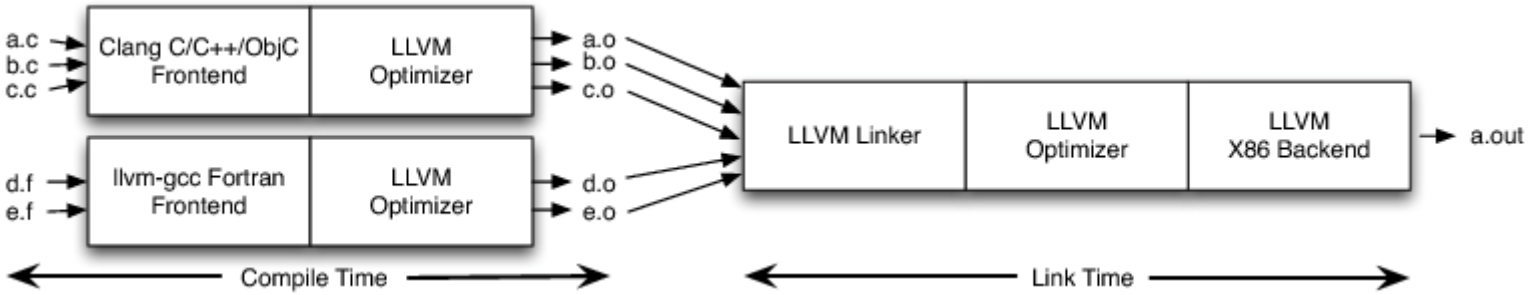
Courtesy of http://www.aosabook.org/en/llvm.html



A Modular Design of LLVM (LTO) (1/2)

- LLVM provides functionality, but lets the client decide most of the ***policies*** on how to use it
- Thanks to the serialization property
  - we can do part of compilation, save our progress to disk, then continue work at some point in the future
  - This feature provides a number of interesting capabilities: including support for **link-time** and **install-time optimization**, both of which delay code generation from "compile time"
- Link-Time Optimization (LTO)
  - The compiler traditionally only sees one translation unit (e.g., a *.c* file with all its headers) at a time and therefore cannot do **optimizations** (like inlining) **across file boundaries**
  - LLVM compilers (like Clang) support this with the **-flto** or **-O4** command line option
    - This option instructs the compiler to emit LLVM **bitcode** to the **.o** file instead of writing out a native object file, and delays code generation to link time, as shown below

Link-Time Optimization

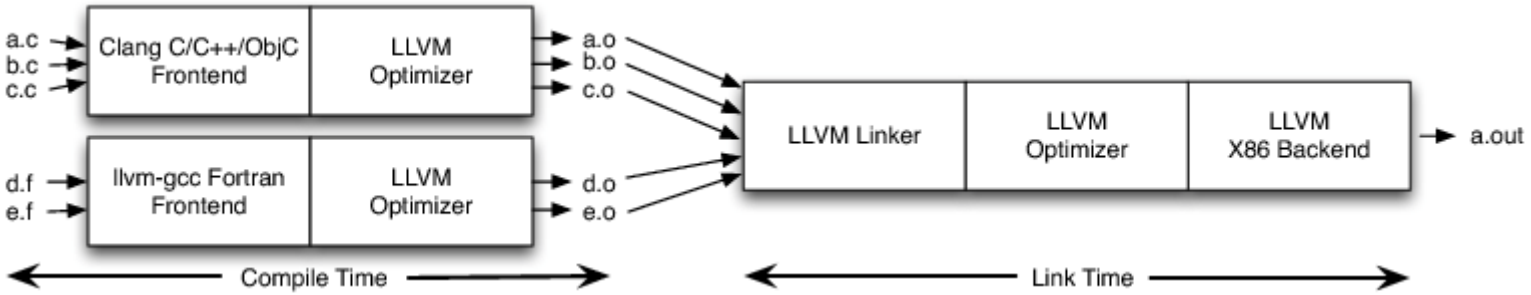


Courtesy of http://www.aosabook.org/en/llvm.html

A Modular Design of LLVM (LTO) (2/2)

- The linker detects LLVM **bitcode** in the **.o** files instead of native object files
  - Then, linker reads all the bitcode files into memory, links them together, and runs the LLVM optimizer over the aggregated program
  - Details differ depending on which operating system you're on
- Since the optimizer can now see across a much larger portion of the code,
  - it can **inline**, **propagate constants**, do more **aggressive dead code elimination**, and more across file boundaries
  - While many modern compilers support LTO, most of them (e.g., GCC, Open64, the Intel compiler, etc.) do so by having an expensive and slow serialization process
- In LLVM, LTO falls out naturally from the design of the system, and
  - **works across different source languages** (unlike many other compilers) because **the IR is truly source language neutral**

Link-Time Optimization



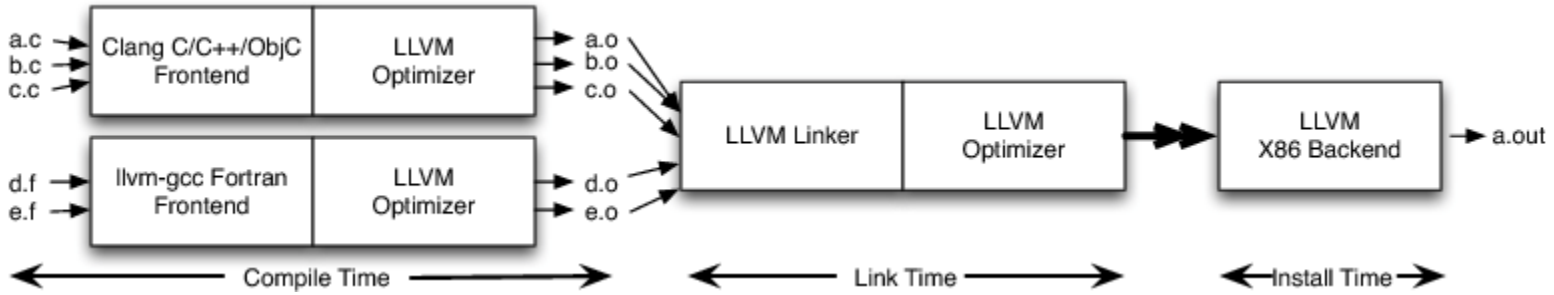
Courtesy of http://www.aosabook.org/en/llvm.html

- In fact, LLVM supports link-time, install-time, run-time, and offline optimization
- You can find out more by yourself

A Modular Design of LLVM (ITO)

- Install-time optimization (ITO)
  - refers to the idea of delaying code generation even later than link time, all the way to install time, as shown below
- Install time is a very interesting time
  - the cases when software is shipped in a box, downloaded, uploaded to a mobile device, etc.
  - This is when you find out **the specifics of the device you're targeting**
  - In the x86 family for example, there are broad variety of chips and characteristics
  - By delaying instruction choice, scheduling, and other aspects of code generation, **you can pick the best answers for the specific hardware an application ends up running on**

Install-Time Optimization



Courtesy of http://www.aosabook.org/en/llvm.html



Key Features of JVM

- The JVM is designed with the following principles in mind

1. Compactness

- Java class files are designed to be relatively compact
  - because JVMs are deployed in browsers and mobile devices
  - compared with the register machines
- In particular, the JVM's instructions are in nearly **zero-address form**
  - so that most instructions manipulate data at the top of Java's **runtime stack**
  - We refer to the topmost location as **top-of-stack** (TOS)



Key Features of JVM (Compactness)

- Example
  - Executing the **iadd** instruction, it
  1. pops two items off the stack and
  2. pushes their sum onto the TOS
  - Only **a single byte** is needed to represent such an operation (**iadd**) because the instruction's operands are **implicit**
- Generally, such compaction is achieved with **a loss of runtime performance**
  - stack manipulation is typically slower than processing operands in a register file
- **Mixing multiple instructions** to further achieve compactness
- Example
  - There are many ways to push 0 on TOS
  1. The shortest instruction, **iconst_0**, takes only a single byte
  2. The most general instruction, **ldc_w 0**, takes three bytes
     - for the instruction and consumes a constant-pool entry
  - While the **iconst_0** instruction is not strictly necessary, its inclusion allows greater code compaction, because pushing 0 on TOS is a frequent operation



Key Features of JVM (Cont'd)

2. Safety

- The JVM's instructions are designed to execute safely: an instruction can reference storage only if said storage is of the **type** allowed by the instruction and only if the storage is located in an area **appropriate for access**
- because the JVM may be deployed in an environment that cannot tolerate badly behaved programs

- Moreover, the instructions are designed so that most **safety errors can be caught prior to executing code**



Key Features of JVM (Cont'd)

- JVM adopts a more secured address form
- In a **pure zero-address form** (which the JVM is not), a **register load** is accomplished by
  a) computing a register number that is **pushed on TOS**, then
  b) pops the register number from the stack,
  c) accesses the register's contents, and
  d) pushes the contents on TOS
- However, the **purely zero-address form** is problematic,
  - from a security point of view,
  - because **the registers that could be accessed by a load instruction may not be known until runtime**



Key Features of JVM (Cont'd)

- The **purely zero-address form** is problematic

- **Runtime checks could be deployed to check the validity of a load instruction,**
  - **but degrades performance**

- Or, a more reliable and efficient approach would check such instructions prior to running the code



Key Features of JVM (Cont'd)

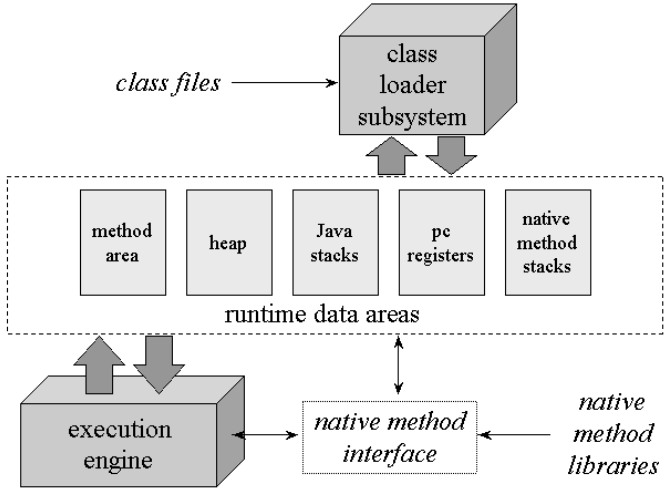
- A more reliable approach in JVM
  - The load instruction in the JVM is not zero-address, but instead
  - **specifies the register number as an immediate operand of the instruction**
- Example
  - the instruction **iload 5** causes
  - the contents of register 5 to be pushed on TOS

- When a class file is loaded, **bytecode verifier** checks the instructions
  - to ensure that register 5 falls within the range of registers its method can access
  - The instruction will always access register 5, because the immediate operands of an instruction cannot be changed at runtime
  - Thus, by discovering that the register reference is valid prior to running the code, no further checks of this kind are necessary at runtime
- The JVM instruction set and class file format are designed to facilitate such checks



Architecture of Java Virtual Machine

- The block diagram of the Java virtual machine that includes:
  - **a class loader subsystem**: a mechanism for loading types (classes and interfaces) given fully qualified names
  - **an execution engine**: a mechanism responsible for executing the instructions contained in the methods of loaded classes

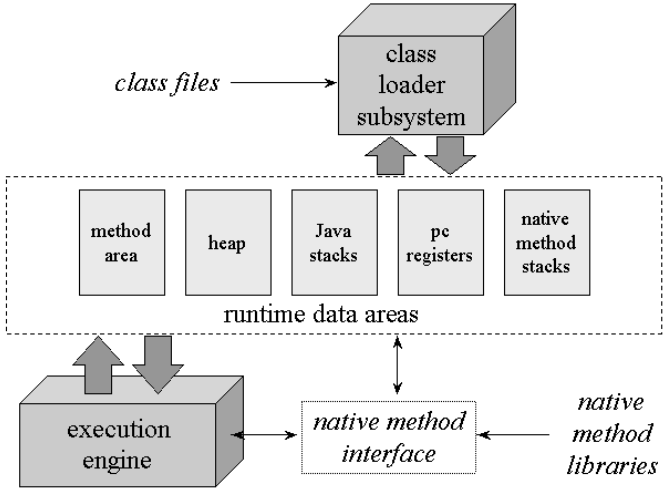


Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Architecture of Java Virtual Machine (Cont'd)

- When a JVM runs a program, it needs **memory** to store many things
  - including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations
- The JVM organizes the memory it needs to execute a program into several ***runtime data areas***
  - Many decisions about the structural details of the runtime data areas are implementation dependent

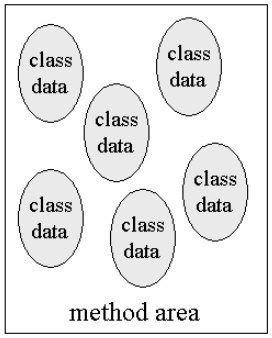


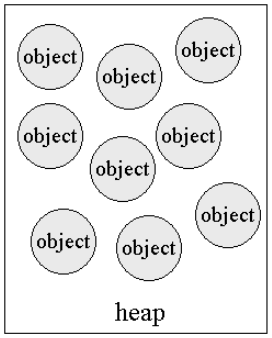
Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Runtime Data Areas (Shared)

- Runtime data areas are either
  - **shared among all of an application's threads or**
  - private to individual threads





- An instance of the JVM is usually created for an application
  - has ***one method area*** and ***one heap***
  - these areas are shared by all threads running inside the virtual machine
- When the virtual machine loads a **class file**,
  - it parses information about a type from the binary data contained in the class file
  - it places this type information into the method area
  - Class (static) variables are kept in **method area**
- As the program runs,
  - the virtual machine places all objects the program instantiates onto the **heap** as shown in the figure
  - The heap memory is managed by JVM (through Garbage Collection)

Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Runtime Data Areas (Private)

- Runtime data areas are either
  - shared among all of an application's threads or
  - **private to individual threads**

- A new Java thread gets its own ***pc register*** (program counter) and ***Java stack***
- If the thread is **executing a Java method** (not a native method),
  - the value of the **pc register** indicates the next instruction to execute
  - A thread's **Java stack** stores the state of Java (not native) method invocations for the thread

- The state of native method invocations is stored in an implementation-dependent way in ***native method stacks***
  - as well as possibly in registers or other implementation-dependent memory areas

Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Java Stack

- The Java stack is composed of ***stack frames*** (or ***frames***)
- A stack frame contains **the state of one Java method invocation**
  - The state includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations
- When a thread invokes a method,
  - the JVM pushes a new frame onto that thread's Java stack
- When the method completes,
  - the JVM pops and discards the frame for that method

Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Java Stack (Cont'd)

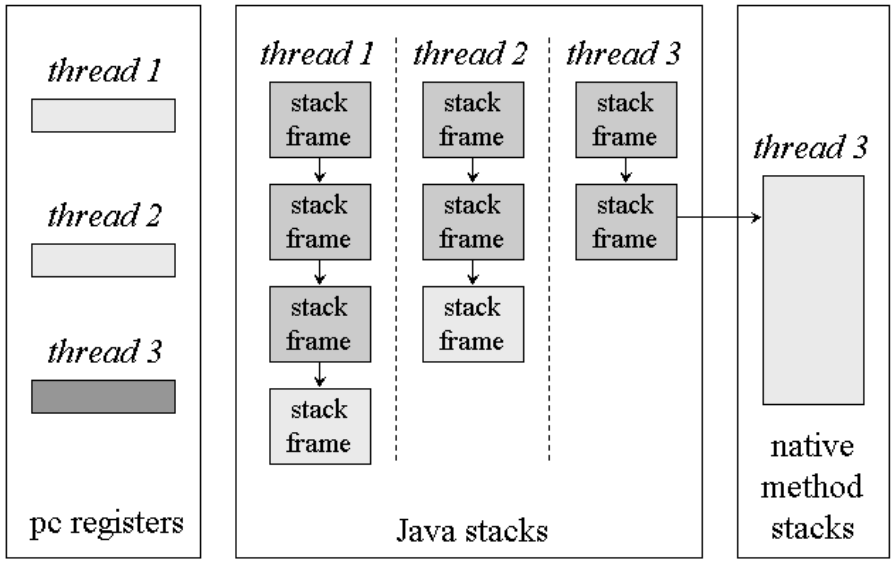
- The **JVM has no registers** to hold intermediate data
  - The instruction set uses the Java stack for storage of intermediate data values
- This approach was taken by Java's designers to
  - keep the Java virtual machine's instruction set compact and
  - facilitate implementation on architectures with few or irregular general purpose registers
- The **stack-based architecture** of the JVM's instruction set
  - facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run-time in some virtual machine implementations

Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Memory Areas for Java Threads

- The figure shows memory areas that JVM creates for each thread
- These areas are private to the owning thread
- No thread can access the pc register or Java stack of another thread

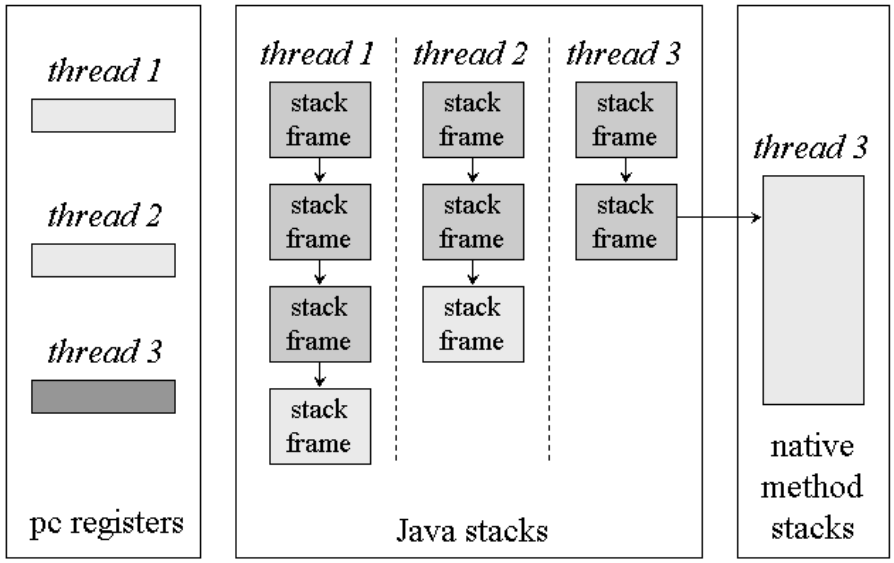


Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Memory Areas for Java Threads

- Figure shows a snapshot of a JVM instance
  - in which three threads are executing
- At the instant of the snapshot, threads one and two are executing Java methods
- Thread three is executing a native method
- The stacks are shown growing downwards
  - The ***top of each stack*** is shown at the bottom of the figure



Courtesy of http://www.artima.com/insidejvm/ed2/jvm2.html



Intermediate Language for Java Virtual Machine

- Java Virtual Machine interprets **class** files
  - which are binary encodings of the data and instructions needed to execute a Java program
- The Java bytecode is considered as the intermediate language for Java, whereas
  - in the textbook, JVM is used to represent the same concept
  - Both terms are used in this file
  - Hence, JVM could refer to **the machine** or **the language** in this file
- To simplify exposition, the contents of a class file are typically discussed in printable form
  - For example, the instruction that specifies **integer addition** has the numerical value of 96, but we typically refer to it as **iadd**



A Class File (Formal Definition)

- A Java class file is consist of 10 basic sections:
  1. Magic Number: 0xCAFEBABE
  2. Version of Class File Format: the minor and major versions of the class file
  3. Constant Pool: Pool of constants for the class
  4. Access Flags: for example whether the class is abstract, static, etc.
  5. This Class: The name of the current class
  6. Super Class: The name of the super class
  7. Interfaces: Any interfaces in the class
  8. Fields: Any fields in the class
  9. Methods: Any methods in the class
  10. Attributes: Any attributes of the class (for example the name of the source file, etc.)

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

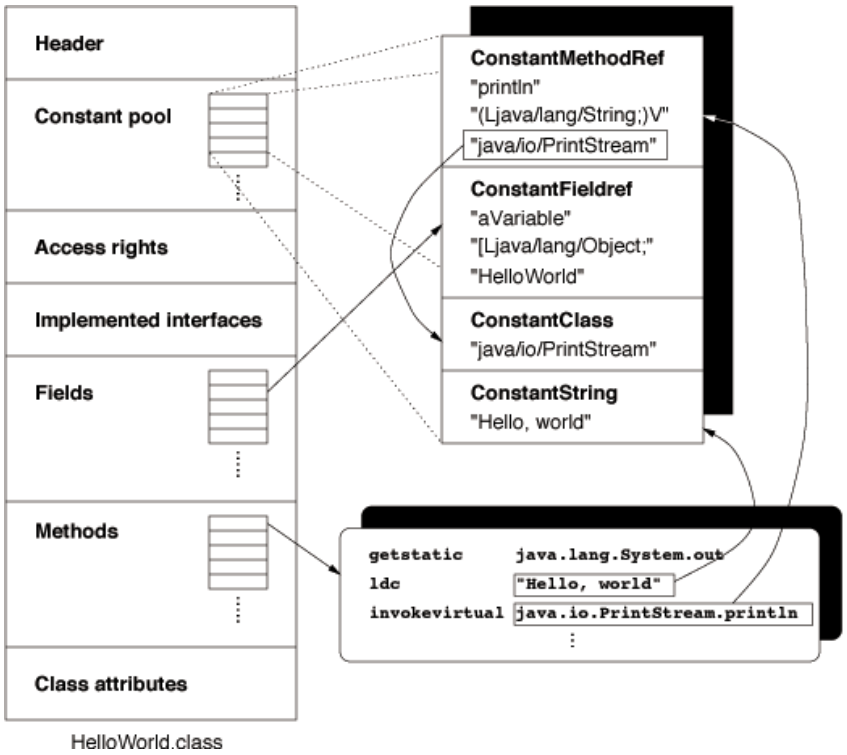
Courtesy of https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html; http://www.programering.com/a/MzN4QzNwATU.html; http://viralpatel.net/blogs/tutorial-java-class-file-format-revealed/



A Class File (Cont'd)

- The diagram depicts that a Java Class file is divided into different components
  - The length of the Java class is not known before it gets loaded
  - Variable length sections such as constant pool, methods, attributes, etc.
- A class file consists of a stream of **8-bit bytes** (one-byte-aligned)
  - All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively
  - Multibyte data items are always stored in **big-endian order**, where the high bytes come first
- The **order of different sections** in a Java Class file
  - is strictly defined so that the JVM knows what to expect in a Class file and in which order it is loading different components

Schematic Class File Structure



Courtesy of https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html; http://www.programering.com/a/MzN4QzNwATU.html; http://viralpatel.net/blogs/tutorial-java-class-file-format-revealed/



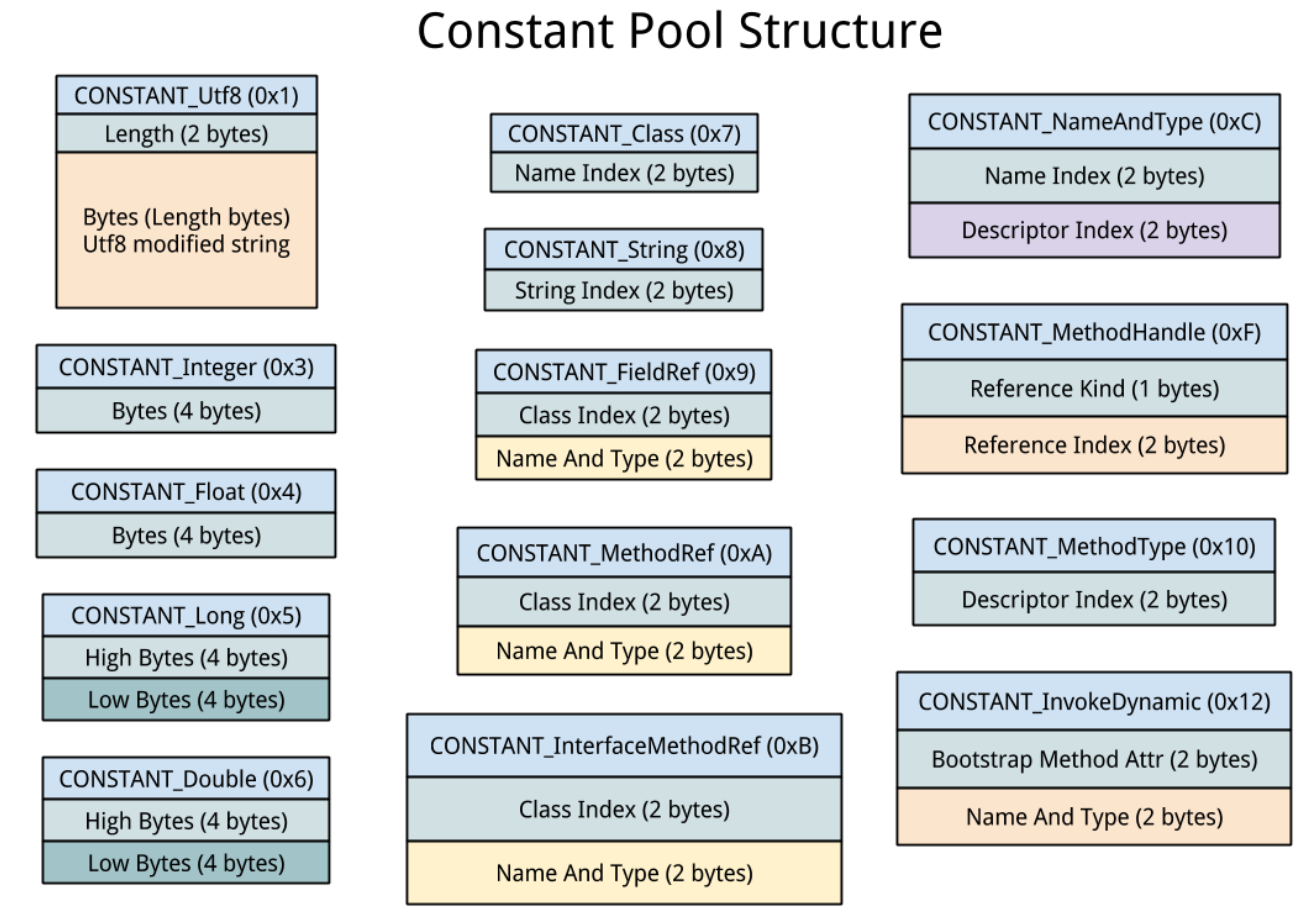
Constant Pool

- Keep all the **constants** related to the Class or an Interface
  - Include class names, variable names, interface names, method names and signature, final variable values, string literals, etc.
  - Contain variable-length arrays
  - Are placed by its array size; hence JVM knows how many constants it will expect while loading the class file
  - **constant_pool_count** is equal to the number of entries in the **constant_pool** array plus one
- Within each array elements, first byte represents a tag
  - specifying **the type of constant** at that position in the array
  - JVM identifies the type of the constant by reading **one-byte tag**
  - If one-byte tag represents a **UTF-8 String literal (Asciz),**
  - JVM knows that next 2 bytes represents length of the String literal and rest of the entry is string literal itself
- You can analyse the Constant Pool of any class file using **javap** command
- Explore more by yourself

```
C:\>javap -verbose Main
Compiled from "Main.java"
public class Main extends java.lang.Object
SourceFile: "Main.java"
minor version: 0
major version: 50
Constant pool:
const #1 = Method #4.#13; // java/lang/Object."<init>":()V
const #2 = int 16707053;
const #3 = class #14; // Main
const #4 = class #15; // java/lang/Object
const #5 = Asciz <init>;
const #6 = Asciz ()V;
const #7 = Asciz Code;
const #8 = Asciz LineNumberTable;
const #9 = Asciz main;
const #10 = Asciz ([Ljava/lang/String;)V;
const #11 = Asciz SourceFile;
const #12 = Asciz Main.java;
const #13 = NameAndType #5:#6;// "<init>":()V
const #14 = Asciz Main;
const #15 = Asciz java/lang/Object;
```

An entry Type Value

More about the Types of Constant Pool Elements

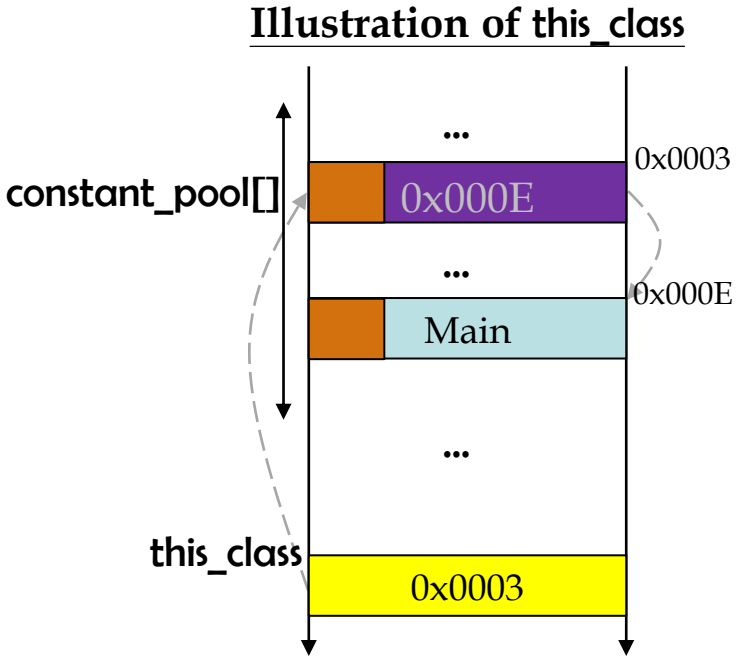


Courtesy of https://blog.lse.epita.fr/articles/69-0xcafebabe-java-class-file-format-an-overview.html
https://docs.oracle.com/javase/specs/jvms/se6/html/ClassFile.doc.html#29297



this Class

- **This Class** (**this_class**) is a two byte entry that points to **an index in Constant Pool**
- In the figure, **this_class** has a value 0x0003
  - which is an index in Constant Pool
  - At the address 0x0003, it has two parts:
  1. first part is the **one-byte tag** that represents the type of entry in constant pool, in this case it is **Class** or **Interface**
  2. second entry is **two-byte** having index again in Constant pool
  - The two-byte contains value 0x000E
  - Thus it points to **constant_pool**[0x000E] which is the String literal having name of the interface or class (e.g., Main)
  - **super_class** is similar



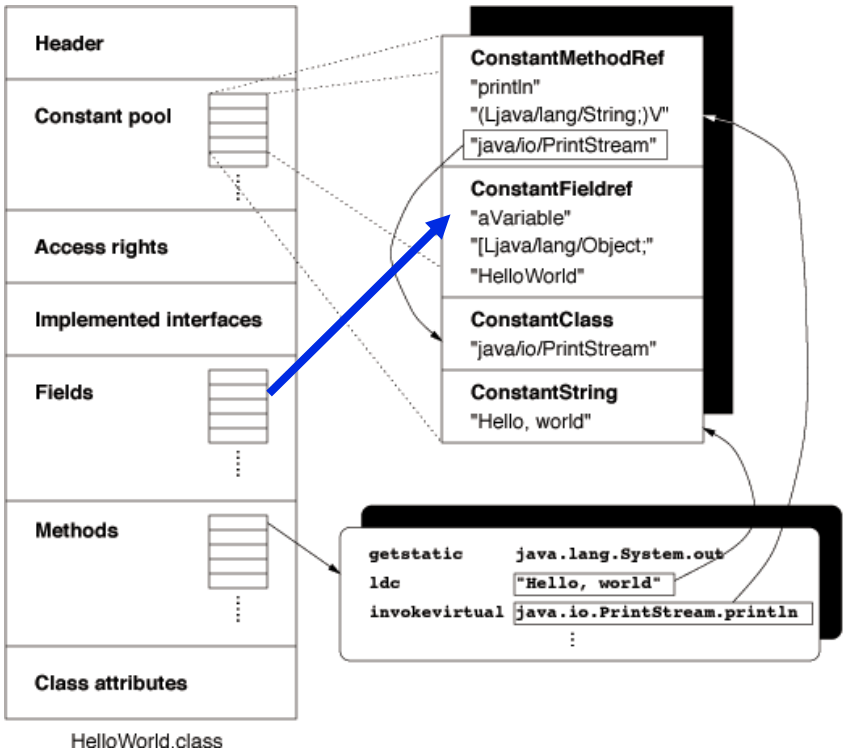
Courtesy of http://viralpatel.net/blogs/tutorial-java-class-file-format-revealed/
(Class instance creation) http://docs.oracle.com/javase/specs/jls/se8/html/jls-12.html#jls-12.5;



Fields

- A field
  - is an instance or a class level **variable** (property) of the class or interface
- Fields section contains only those fields
  - that **are defined by the class or an interface** of the file and
  - not those fields which are inherited from the super class or super interface
- First two bytes in Fields section represents count (**fields_count**)
  - that is the total number of fields in Fields Section
- Following the count is an array of variable length structure one for each field
  - Each element in this array represent one field
  - Some information is stored in this structure whereas some information, like **name of the fields**, are stored in Constant pool

Schematic Class File Structure



HelloWorld.class

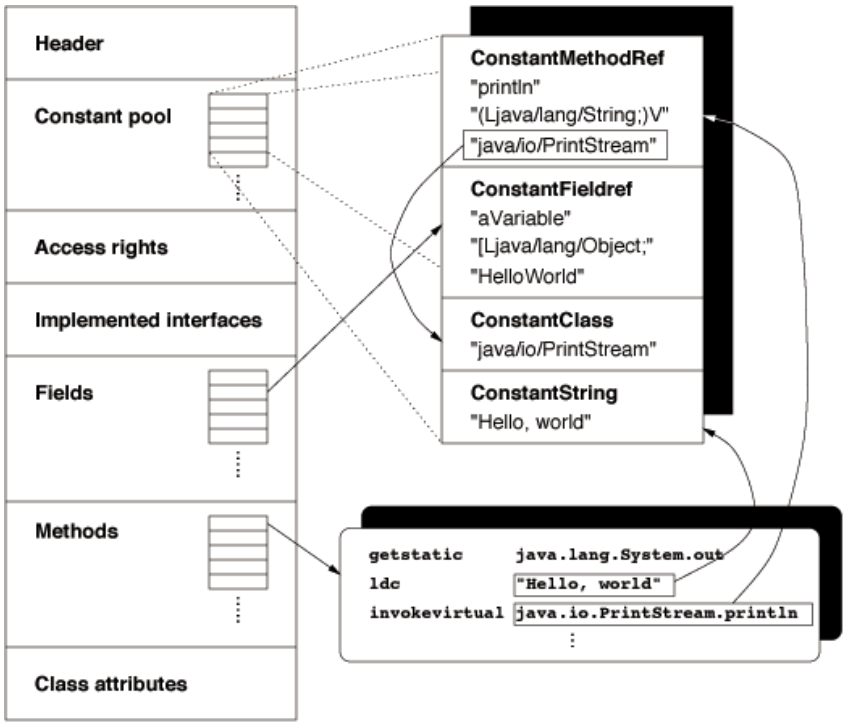
```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
} // You can check what are the attributes for field.
```



Methods

- The Methods component host the methods
  - that are explicitly **defined by this class**,
  - not any other methods that may be inherited from super class
- First two byte (**methods_count**) is
  - the count of the **number of methods** in the class or interface
- The rest is again a variable length array which holds each method structure
  - Method structure contains several pieces of information about the method
  - E.g., method argument list, its return type, the number of stack words required for the method's local variables, stack words required for method's operand stack, a table for exceptions, etc.

Schematic Class File Structure

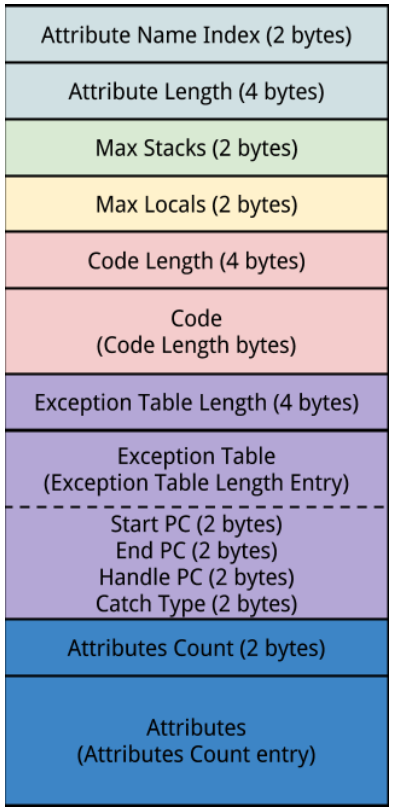




Attributes

- Attribute section contains several attributes about the fields, methods, and class
  - E.g., one of the attribute is the source code attribute which reveals the name of the source file from which this class file was compiled (Main.java in the previous example)
- There are several attribute types
  - each one of them can be applied to one or several fields, methods, classes and codes
- Example attribute types:
  - **Code** (as illustrate in the figure)
  - Local variables, constant value, information about the stack and exceptions
  - Inner Classes, Bootstrap Methods, Enclosing methods
  - Annotations
  - Information for debug/decompilation
  - Complementary information (Deprecated, Signature...)

Schematic Code Attribute

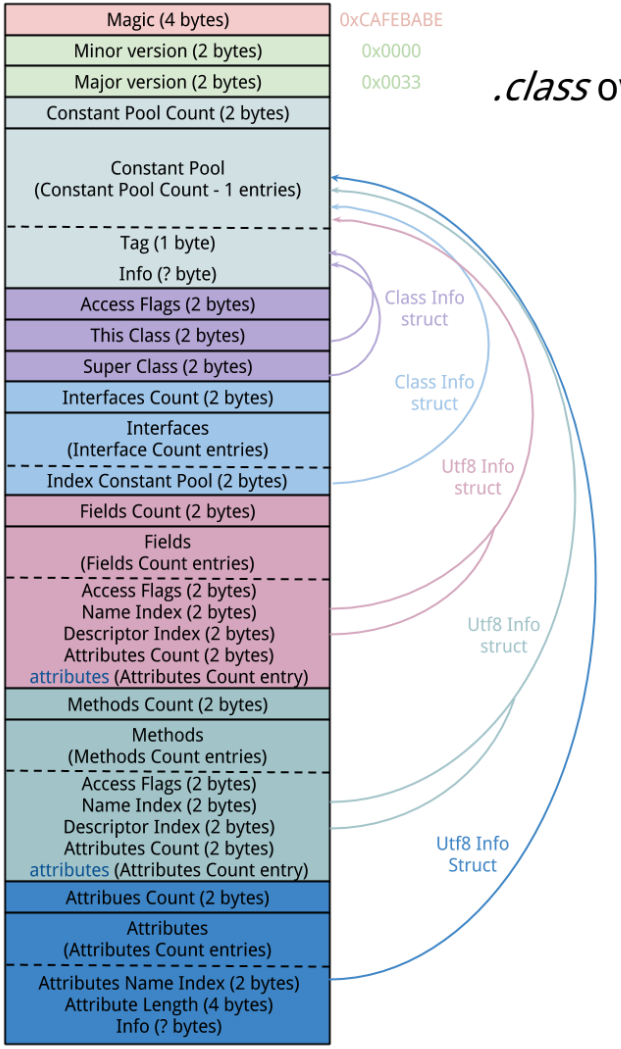


Courtesy of https://blog.lse.epita.fr/articles/69-0xcafebabe-java-class-file-format-an-overview.html; http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.2;



More about the class File

- For those who are interested in the details

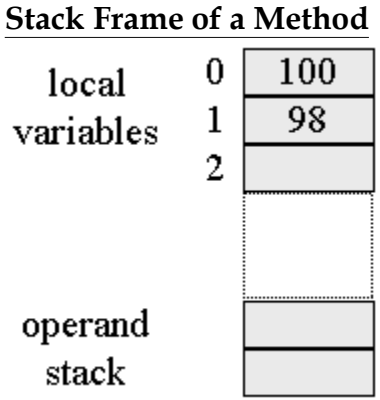


Courtesy of https://blog.lse.epita.fr/articles/69-0xcafebabe-java-class-file-format-an-overview.html



Java Stack Frame

- Java virtual machine creates a new Java Stack Frame for the thread
- The **stack frame** has three parts for each method
  - **local variables**, **operand stack**, and **frame data**
  - The method that is currently being executed by a thread is the thread's *current method*
  - The stack frame for the current method is the *current frame*
  - More about the **stack frame** is described in Sec. 12.2
  - Frame data is used to record constant pool resolution, normal method return, exception table, etc.
- An unlimited number of **virtual registers** in JVM
  - as is frequently the case with IRs, which is not tied to physical machines
  - Each method declares how many registers it can reference
- NOTE
  - **JVM does not have registers**
  - Registers used here refer to the **local variables**
  - Local variables and registers are used interchangeably here



Courtesy of http://www.artima.com/insidejvm/ed2/jvm8.html



Registers/Local Variables

- JVM registers typically host a method's **local variables**
  - which are similar to a machine's architected registers (Sec. 13.3)
  - Actually, we could see it this way: JVM local variables are mapped to the method's local variables
- JVM registers starting from 0 are set aside for a method's actual parameters
  - For **static methods**, register 0 holds the method's first declared parameter
  - For **instance-based methods**, register 0 holds **this** (the object's self-reference in the heap) and register 1 holds the method's first declared parameter

Registers/Local Variables (Cont'd)



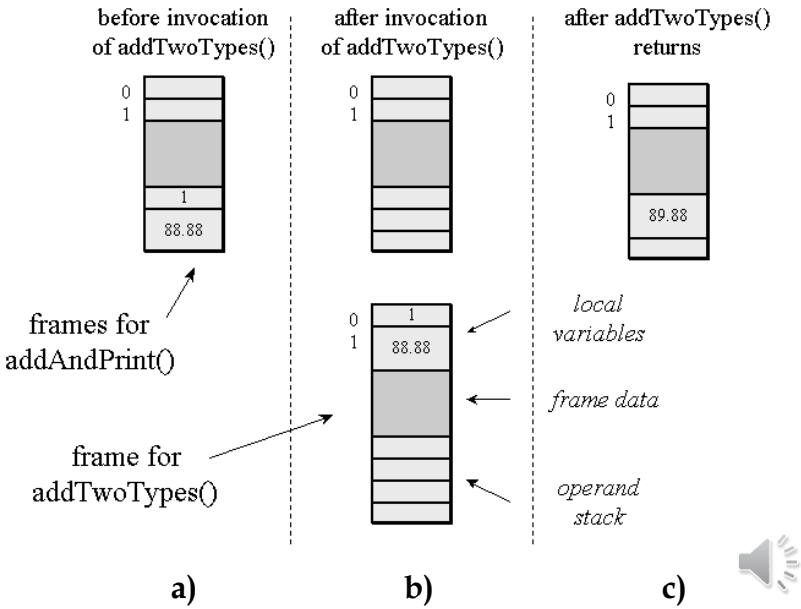
- When a method is invoked,
  - parameter values are automatically **popped from the caller's stack frame** and
  - **deposited into the callee's stack frame described as below**
- The figure depicts the stack status of a Java thread
  a) At the **addAndPrint()** method,
  - before invoking the **addTwoTypes()** method, the **addAndPrint()** method first pushes an int one and double 88.88 onto its operand stack
  - The **addAndPrint()** method uses the constant pool to identify the **addTwoTypes()** method
  - The resolved constant pool entry points to information in the method area about the **addTwoTypes()** method
  - The virtual machine allocates enough memory for the **addTwoTypes()** frame from a heap
  - It then pops the double and int parameters (88.88 and one) from **addAndPrint()**'s operand stack and places them into **addTwoType()**'s local variable slots one and zero

```
class Example3c {

   public static void addAndPrint() {
      double result = addTwoTypes(1, 88.88);
      System.out.println(result);
   }

   public static double addTwoTypes(int i, double d) {
      return i + d;
   }
}
```

Illustration of Stack Status before and after method invocation



Registers/Local Variables (Cont'd)



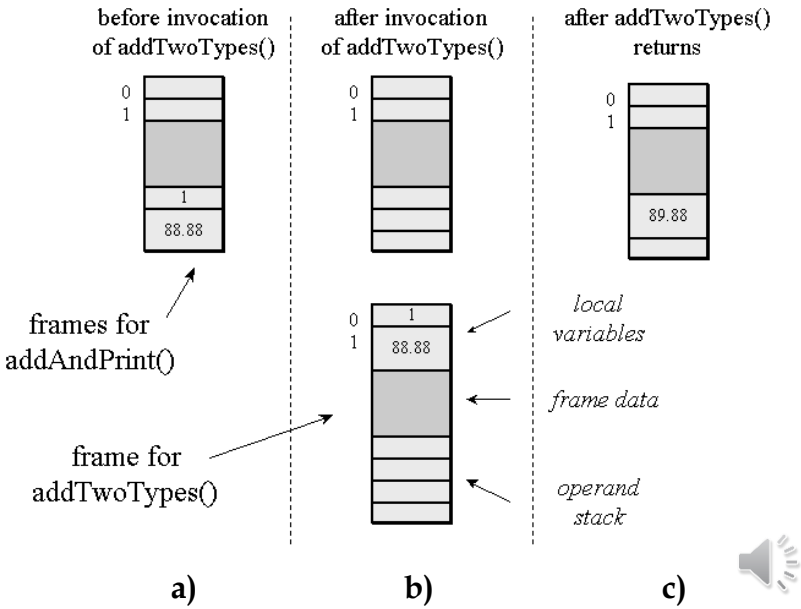
- The figure shows the stack status of a Java thread
  - b) It then invokes the **addTwoTypes()** method,
  - which calculates the sum for the local variables and it pushes the result (double return value, 89.88) onto its operand stack
  - Upon the method returns,
  - the virtual machine uses the information in the frame data to locate the stack frame of the invoking method, **addAndPrint()**
  - It pushes the double return value onto **addAndPrint()**'s operand stack and frees the memory occupied by **addTwoType()**'s frame
  - c) After **addTwoTypes()** returns,
  - it makes **addAndPrint()**'s frame current and continues executing the **addAndPrint()** method at the first instruction past the **addTwoType()** method invocation
  - NOTE: Some of the details described are implementation dependent

```
class Example3c {

   public static void addAndPrint() {
      double result = addTwoTypes(1, 88.88);
      System.out.println(result);
   }

   public static double addTwoTypes(int i, double d) {
      return i + d;
   }
}
```

Illustration of Stack Status before and after method invocation





Data Types for Intermediate Language

- Primitive types are designated by **a single character**, as shown in Fig. 10.4
- A reference type ***t*** is designated as **L*t*;** with ***t*** specified as follows
- Each dot in the type is replaced by a `/'
  - which results in a Unix–like file path to the class file for the type
  - The JVM uses such paths to locate a class file at runtime
- Example
  - The String type in Java is actually found in the java.lang package
  - Its full type name is therefore **java.lang.String** and it has the type designation **Ljava/lang/String;** in the JVM
  - Because of its need to represent String constants efficiently, the JVM is aware of the String reference type, almost as if it were a primitive type

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	`L`*t*`;`
Array of type *a*	`[`*a*

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

Be aware of the size of instructions and the size of data kept in Stack.



Java Assembly Instructions

- Arithmetic
- Register Transfer (Local Variable)
- Accessing Static Fields
- Accessing Instance Fields
- Branching
- Static Method Calls
- Instance-Specific Method Calls
- Other Method Calls
- Stack Operations



Arithmetic Instructions

- Instructions that compute results based on simple mathematical functions
  - operate by popping their required number of operands (typically 2) from the operand stack,
  - computing the required result, and
  - finally pushing the result on TOS



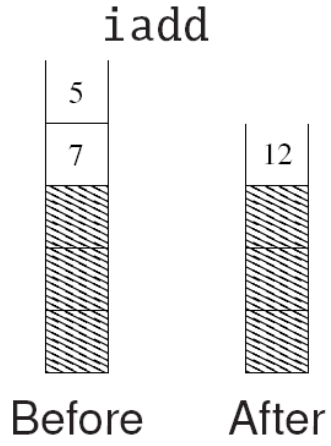
Arithmetic Instructions (Cont'd)

- Instructions that perform addition on primitive types:
  - **iadd** (int), **fadd** (float), **ladd** (long), and **dadd** (double)
- There are no instructions for integer types that are shorter than int, such as byte and short
  - Such arithmetic is performed on int types
- There are also instructions available for performing the usual arithmetic operations,
  - such as subtraction, multiplication, division, and remainder



Arithmetic Instructions: Example

- The **iadd** instruction
  - pops the top two elements off of the operand stack &
  - pushes their sum onto the stack



- The instruction expects the operands to be primitive **int** types, and the result is also **int** type
  - All operations involving **int** types are performed using 32-bit, **two's complement** arithmetic

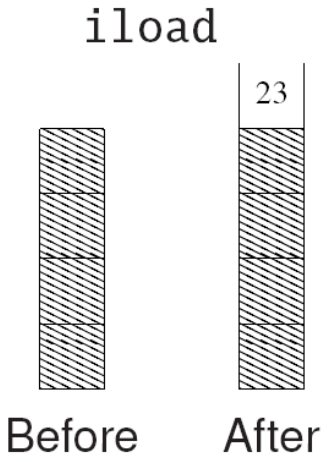
- Such arithmetic never throws any exceptions
  - I.e., overflow or underflow occur silently



Register Transfer Instructions

- The JVM's registers are **untyped**, so they can hold any kind of value
  - For values that require two registers (long and double types), an even-odd pair of registers must be used

- The **iload** instruction
  - pushes the contents of a JVM register on TOS
  - The instruction must contain an immediate operand designating the register whose contents should be loaded
  - If register 2 contains the value 23, then the example shows the results of executing **iload 2**



- The register is unaffected by the instruction



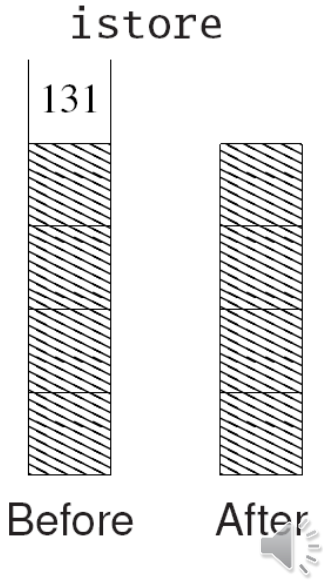
Register Transfer Instructions (Cont'd)

- To facilitate instruction compression,
  - the JVM has some single-byte instructions that **load low-numbered registers**
- The **iload 2** instruction takes two bytes:
  - one for the instruction and one for the operand
- The operation can be abbreviated as **iload_2**,
  - which takes only a single byte, as the opcode implies that register 2 is loaded



Register Transfer Instructions (Cont'd)

- Instructions are also available for moving data from the stack to a register
- The **istore** instruction
  - pops a value from the stack and
  - stores it in the register specified as an immediate operand of the instruction
- Example
  - An **istore 10** would pop the value 131 from the stack and store it in register 10
  - There is no abbreviated form of this instruction because register 10 is beyond the registers provisioned for the single-byte **istore** instructions





Register Transfer Instructions (Cont'd)

- There are variations of **iload** and **istore** for each supported data type
  - E.g., **fload** ***n*** reads a float value from register ***n*** and pushes it on TOS
- There are no register instructions specific to the boolean type
  - The JVM uses an int representation for boolean, with 0 representing false and 1 representing true
  - Types char, byte, and short are similarly treated as int at the register level, because 32 bits can accommodate their values as well

- All reference types are loaded and stored using **aload** and **astore** instructions, respectively
- An object reference takes the same space as an int, 32 bits
- The value of 0 is reserved to represent **null**



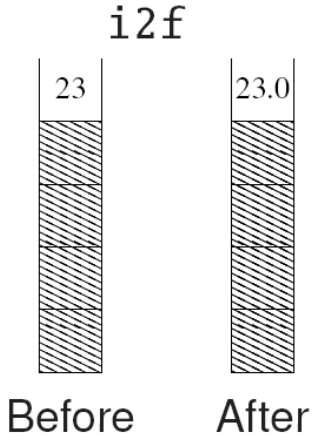
Registers and Types

- Static analysis is performed to check type consistency
  - Although registers are untyped,
  - **bytecode verification** is performed on the code to ensure that values flow in and out of registers without compromising Java's type system
  - Such analysis prevents a JVM program from loading a reference to an object (using **aload**) and
  - subsequently **performing math on the reference** to trick the JVM into accessing storage inappropriately
- In this regard, the JVM appears to be stricter than the Java language
  - an int value can be treated as a float without casting in Java
- However, it is important to understand that the transition from int to float could be done implicitly by JVM
  - Semantic analysis as described in Ch. 2 and 8 can insert the int-to-float **type conversion**, which is then realized by generating an **i2f** instruction



Registers and Types (Cont'd)

- Type conversion instructions operate by
  - popping a value off the stack and
  - pushing its converted value
- The stack must contain a value of the appropriate type at its top
  - In the example, the int 23 is at the TOS
  - When the instruction **i2f** has finished, the value is effectively replaced by its float representation
- **The bit patterns for 23 and 23.0 are different,**
  - with the former in two's complement and the latter in IEEE floating point format





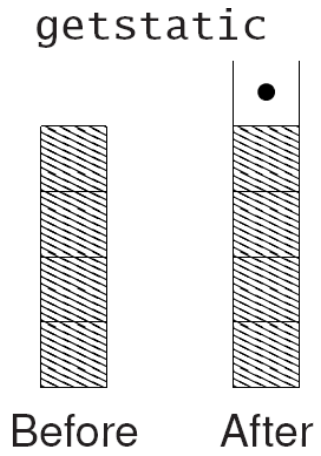
Accessing Static Fields

- A class's **static fields** are visible to every instance of the class
  - Space for such fields is provisioned when the class is loaded and initialized
  - Thereafter, a static field can be loaded or stored by the **getstatic** and **putstatic** instructions, respectively
- The action of **getstatic** is similar to
  - the various load instructions (**iload**, **aload**, etc.) in that the result of the fetch is pushed on TOS
- The get static field instruction as coded in Jasmin is follows:
  - **getstatic** ***name type***
  - where ***name*** is the name of the static field, prefaced by its fully qualified class name, and ***type*** is the expected type of the result



Accessing Static Fields (Cont'd)

- Many Java programs reference System.out for console output
- Such accesses are actually references to the static field **java.lang.System.out**, whose type is **java.io.PrintStream**
- Example
  - After the instruction executes: **getstatic java/lang/System/out Ljava/io/PrintStream**
  - the stack has a new element at TOS
  - The • on the stack represents the reference to System.out



- The length of **getstatic** instruction is 3 bytes
  - one specifies the getstatic opcode and the other two form a 16-bit integer specifying a **constant-pool** entry
  - Recall that constant-pool entries are denoted ordinally (0, 1, 2, etc.) and not by their byte-offset in the constant pool
  - In this case, the designated constant-pool entry contains the name and type operand values for the **getfield** instruction

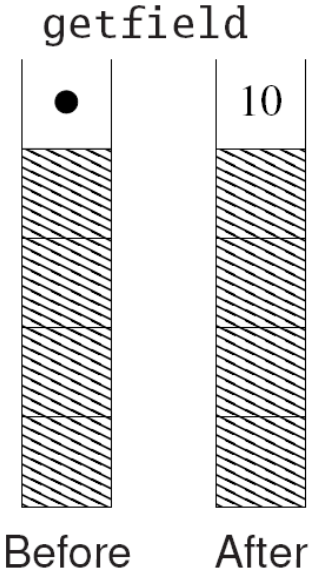


Accessing Instance Fields

- A class can declare **instance fields**
  - for which instance-specific storage is allocated
  - Every instance of type **t** is provided with storage for **t**'s instance fields
  - To access those fields, a particular instance of **t** must be provided
- The **getfield** instruction pushes the value of a particular instance field on TOS
- The syntax for a **getfield** instruction is exactly the syntax of the **getstatic** instruction
  - **a name** and **type** are specified as immediate operands
  - However, because the instruction also requires an instance of the accessed field, the semantics of the instruction specify that the **TOS must contain a reference to the instance** whose field is to be accessed

Accessing Instance Fields (Cont'd)

- Consider a **Point** class such that each instance has int fields: **x** and **y**
  - Consider a particular instance that has 10 and 20 values for its **x** and **y**, respectively
  - The example shows a **Point** reference (•) on TOS
  - The instruction **getfield Point/x I**
  - retrieves the value of • 's **x** field, making sure it is an int, and pushes the value (10) on TOS

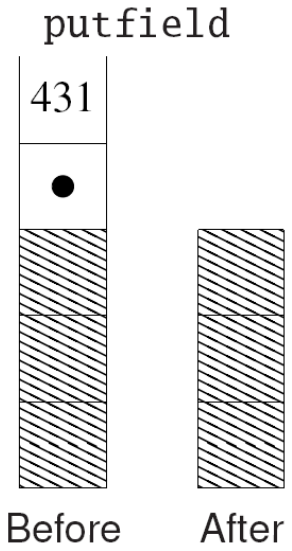


Accessing Instance Fields (Cont'd)

- The **putfield** instruction is the instance-specific version of the **putstatic** instruction
- Example
  - The object reference (•) refers to an instance of a **Point** object
  - The instruction **putfield Point/x I**
  - pops two values from TOS

1. The first is the value (431) that should be stored at the field (**x**) specified in the **putfield** instruction
2. The second is a reference to a **Point** object, shown as •

  - When the instruction completes, • 's **x** field will have the value 431, and the stack will have two fewer items





Branching Instructions

- The JVM provides instructions to
  - **alter the control flow of the executing program**
  - Control can be transferred unconditionally to the instruction at location **q** by a **goto** instruction
- The instruction occupies 3 bytes:
  - one byte specifies the **goto** **opcode** and
  - the other two bytes are concatenated to form **a signed 16-bit offset** denoted here as Δ
  - If **p** is the location of the current **goto** instruction, control is transferred to the opcode at **q = p+Δ**
  - In Jasmin, offsets are computed automatically, and targets are specified symbolically using labels



Branching Instructions (Cont'd)

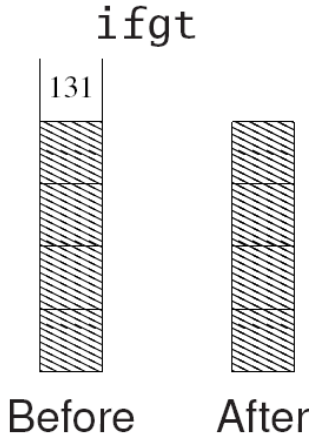
- Two classes of branching instructions

1. There are several kinds of instructions that accommodate conditional branches
   - Each such instruction contains an opcode that specifies the **conditional test** and a branch to be taken if the test is true
   - Six such instructions in JVM, one for **every possible comparison of an int value against 0**: **ifeq** (=), **ifne** (!=), **iflt** (<), **ifle** (≤), **ifgt** (>), and **ifge** (≥)
   - A separate opcode is provisioned for each instruction
2. Some programs call for comparisons of non-zero values
   - While the instructions described above are sufficient for such programs, a shorter sequence of instructions can be generated using the following relatively more complex instructions
   - Six instructions are in this family: **if_icmpeq** (a = b), **if_icmpne** (a != b), **if_icmplt** (a < b), **if_icmple** (a ≤ b), **if_icmpgt** (a > b), and **if_icmpge** (a ≥ b)

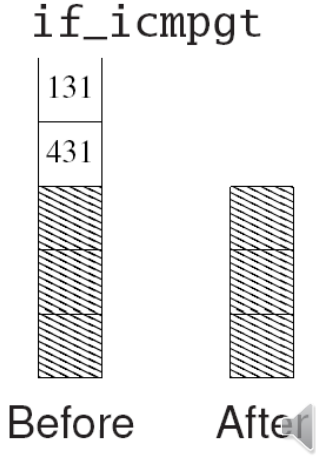


Branching Instructions (Cont'd)

- The **ifgt** instruction and its cognates expect that
  - the TOS is a signed int value
  - The branch is taken if the condition (in this case, greater-than-zero) is satisfied
- Example
  - 131 is popped and compared against 0
  - Because 131 > 0, the branch will be taken
  - Had the comparison failed, control would pass to the instruction following the **ifgt** instruction (i.e., **ifgt <label>**)



- Consider a source program that at some point has
  - a = 431 and b = 131 and must next evaluate whether a > b
  - After an **iload** instruction is issued for a and b, in that order, the stack appears as shown on the left
- Example
  - the **if_icmpgt** instruction pops the top two elements and performs the comparison 431 > 131
  - Because this test succeeds, the branch is taken



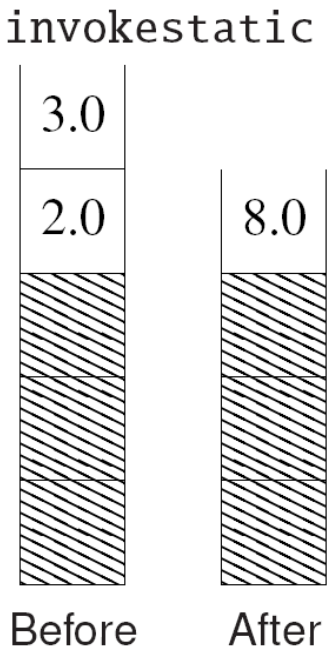


Static Method Calls Instructions

- There are several forms of method-calling instructions in the JVM
- In object-oriented languages, a method could be **common to all instances of some type *t*** or **instance specific**
  - In the former case, Java designates such methods as static
  - A static method of type ***t***, like a static field, is referenced by its type ***t*** and does not require an instance of ***t*** to be called
  - An example of a static method is **Math.pow** (**double a,double b**), which returns a^b
  - Such methods are called using the **invokestatic** instruction

Static Method Calls Instructions (Cont'd)

- The **Math.pow** static method is called using the Jasmin notation
  - **invokestatic java/lang/Math/pow(DD)D**
  - which specifies the full path to the method and also contains the **signature** of the method
- Example
  - Two values (2.0 and 3.0) are popped and supplied to the **Math.pow** static method as its first and second parameters, respectively
  - When **Math.pow** completes, its result (8.0) is pushed on TOS





Static Method Calls Instructions (Cont'd)

- From the above example, it should be clear that a method's parameters are pushed on the stack in **left-to-right order**
  - If a static method takes ***n*** parameters, then its ***n*** th parameter is on TOS just as the method is called
- For a **method signature**,
  - the symbols between the parentheses indicate the method's input parameter types, using the notation described in Fig. 10.4
  - I.e., in the above example, **two double parameters** (**(DD)**) are expected
  - A method's return type is specified just after the parentheses
  - In the above example, the return type is also **double**
- Although the Jasmin notation shows the method and its signature as part of the **invokestatic** instruction,
  - the method and signature descriptive information is actually held in the **constant pool**
  - The **invokestatic** instruction occupies 3 bytes,
    - with the second two bytes forming an ordinal index into the constant pool



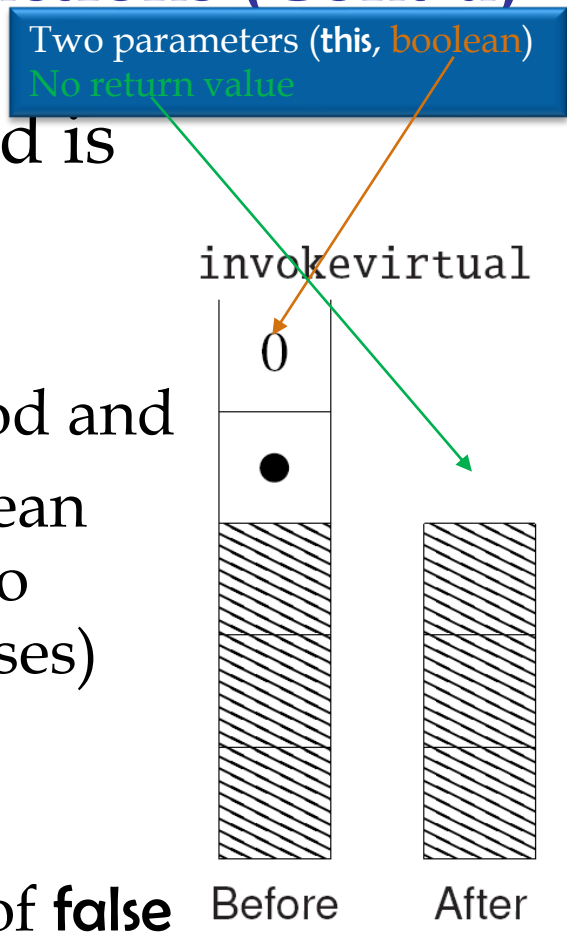
Instance-Specific Method Calls Instructions

- An instance-specific method
  - such as **PrintStream.print**()
  - is invoked in a manner similar to **invokestatic** with the following differences
  - Because the method is instance-specific, **an instance must be pushed on the stack** before the method's parameters
  - The instance becomes **this** inside the called method
  - The **invokevirtual** operator is used instead of **invokestatic**
- An instance-specific method
  - formally declaring that it takes **n** parameters ($p_1, p_2, \ldots p_n$)
  - actually takes **n + 1** parameters where p_0 is effectively the called method's **this**



Instance-Specific Method Calls Instructions (Cont'd)

- The **PrintStream.print(boolean)** method is called using the Jasmin notation
  - **invokevirtual java/io/PrintStream/print(Z)V**
  - which specifies the full path to the method and
  - indicates that the method takes one boolean parameter (indicated by **Z**) and returns no values (indicated by **V** after the parentheses)
  - In the example, the · is an instance of a **PrintStream** class (e.g., **System.out**)
  - The 0 on TOS is Java's integer encoding of **false** for the boolean data





Other Method Calls Instructions

- Almost all methods in Java are called by **invokevirtual** or **invokestatic**
  - However, important exceptions include **constructor calls** and **calls based on super**,
  - which are handled by the **invokespecial** instruction

- A **constructor call** is special in the following sense
  - An **uninitialized reference** to an object instance is pushed on TOS (usually by a new instruction)
  - The method name actually involved is **<init>** within the type of the object pushed on TOS
  - The constructor consumes the reference from TOS as its input parameter (**this**) along with any declared parameters
  - All constructors are **void** so nothing is returned from the constructor call
  - A code generator must be aware of this behavior and issue the appropriate instructions to be able to access the instantiated object
- Example
  - **invokespecial java/lang/StringBuffer/<init>()V**
  - calls the method "<init>" (the special name used for instance initialization methods) in the class "**java.lang.StringBuffer**", and
  - it has the descriptor "**()V**" (i.e. it takes no arguments and no results)



Other Method Calls Instructions (Cont'd)

- The **invokespecial** method
  - can also be used to invoke a **private** method, but this appears to be for efficiency reasons only
- A **private** method cannot be overridden,
  - so there is no reason to employ a virtual method dispatch



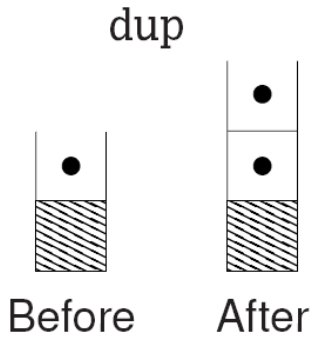
Stack Operations Instructions

- The JVM provides some instructions specifically for **manipulating items near the TOS**
- Such instructions may seem superfluous,
  - in that they **can be simulated using other instructions** and registers
  - They are included to facilitate shorter instruction sequences for common program fragments
  - E.g., **dup**, **dup_x1**, **pop** and **swap**



dup Instruction

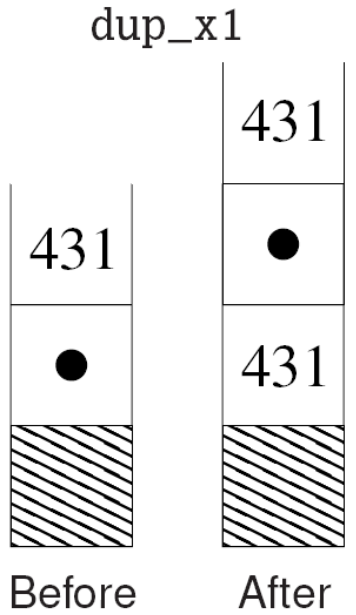
- Example
  - the duplication of the cell at TOS
  - The instruction works for any 32-bit type (i.e., all types except long and double)
- The instruction nicely accommodates multiple assignments (x=y=z=value)
  - There is a **dup2** instruction that duplicates the top two cells to accommodate long and double types
- It is also useful for constructing new objects
  - The **new *t*** instruction
  - leaves a reference on TOS to the newly allocated storage of type ***t***,
  - but that storage cannot be accessed until a constructor has been invoked
  - Constructors consume the reference to the allocated storage (along with their other parameters), but they **return nothing** (they are void)
  - Thus, **to remember the reference after the return of the Constructor**, the TOS is usually duped before the constructor invocation sequence is generated





dup_x1 Instructions

- This instruction **duplicates** the TOS element,
  - but it situates the duplicated cell as shown in the example, **two cells below the TOS**
- One application of this instruction is
  - the duplication of a value that participates in an embedded assignment
- Consider **foo(this.x ← y)**,
  - where **y** happens to have the value 431
  - The example starts with **y**'s value already loaded on the stack
  - The example ends with the stack prepared for the **putfield** instruction followed by the method call to **foo**
  - The **dup_x1** instruction duplicates the 431 and places it below the **this** reference (shown as •)
  - Recall that the field assigned by a **putfield** is an immediate operand of the instruction
  - Thus, when the **putfield** for **x** completes, the top two elements will be removed from the stack, leaving the duplicated 431 as the parameter value for **foo**
  - This instruction nicely demonstrates that the JVM instruction set was designed not to be simple but to allow for compact code sequences





QUESTIONS?