Università di Pisa

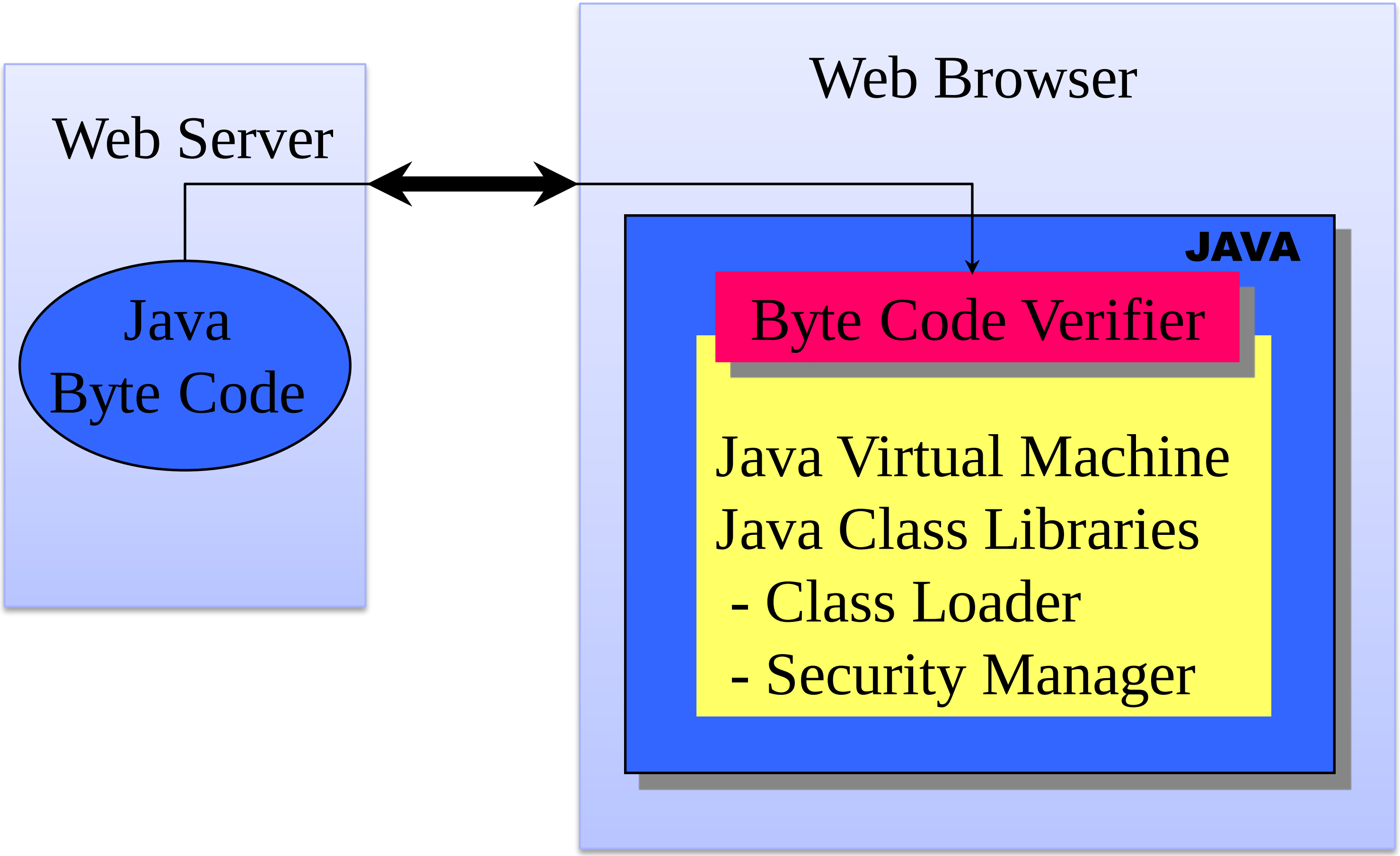
Java Virtual Machine

Giuseppe Attardi

Dipartimento di Informatica

Università di Pisa

Java Virtual Machine



Web Server

Java Byte Code

Web Browser

JAVA

Byte Code Verifier

Java Virtual Machine

Java Class Libraries

- Class Loader
- Security Manager

Java

- linguaggio ad oggetti (~ C++ senza puntatori)
- Abstract Window Toolkit
- Applets: codice migrabile
- incorporato in Netscape, Internet Explorer, etc.
- JBuilder, Café, J++: ambienti visuali

Java Language

- object-oriented
- portable
- interpreted byte-code
- high performance
- architecture neutral
- distributed
- multi-threaded

Java (continua)

- dynamic
  - memory allocation: garbage collection
  - dynamic linking
- exception handling
- robust
- secure
- no pointers
- strongly typed

Overview

- C++ like syntax
- Basic types
  - integer, floating, character, boolean, array
- Classes
  - single inheritance
  - interface inheritance
  - access: public, private, protected
  - initialization and finalization
- Interfaces
  - as types

Overview

- Methods
  - polymorphic
  - static, virtual, non-virtual, abstract
  - synchronized
- Threads
- Packages
- Exceptions

Overview

- Garbage collection
- Stream I/O
- Dynamic loading
- System resources

What is a Virtual Machine?

- **A virtual machine (VM) is an *abstract* computer architecture**
- **Software on top of a real hardware**
- **Can run the same application on different machines where the VM is available**

The Java Virtual Machine

- **An abstract computing machine that executes bytecode programs**
  - **An instruction set and the meaning of those instructions – the *bytecodes***
  - **A binary format – the *class file* format**
  - **An algorithm to *verify* the class file**

JVM cont.

- Runtime environment for Java
- Implementation NOT defined
- Runs Java .class files
- Has to conform to Sun‘s specification

Implementations of the JVM

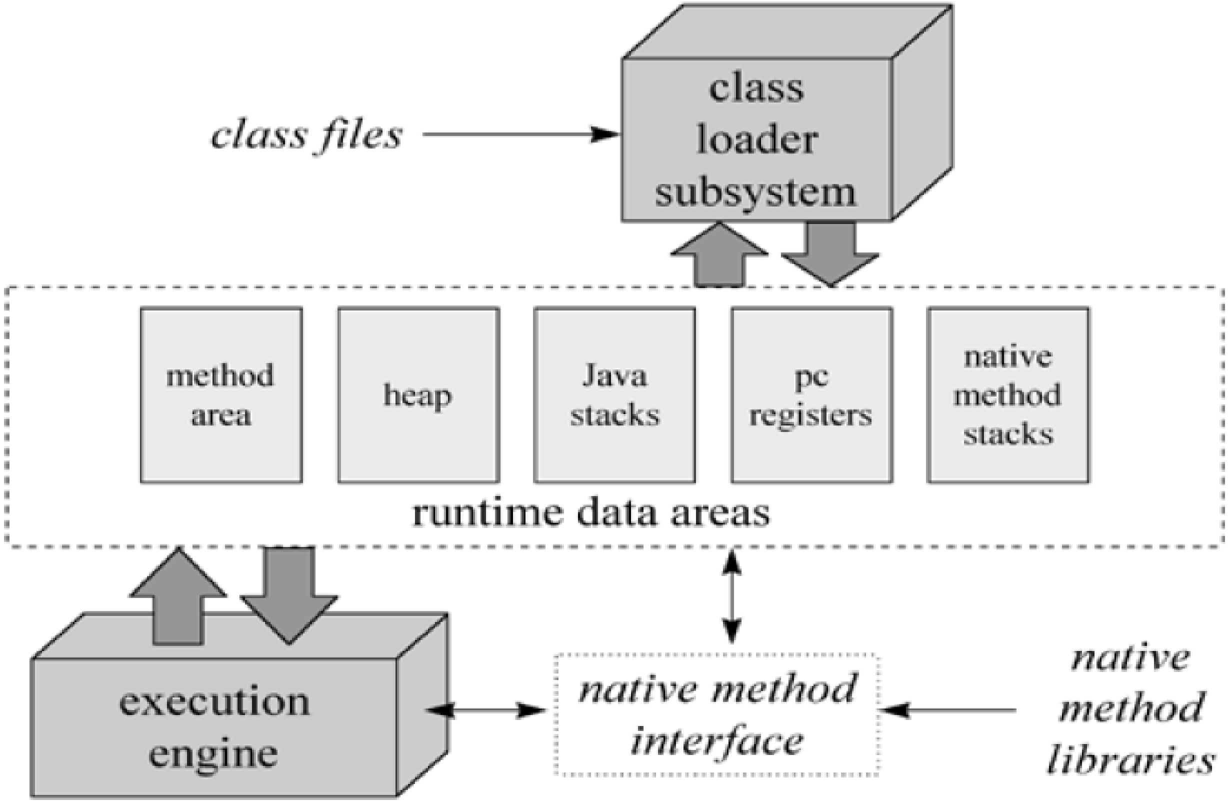
- Interpreter
  - Simple, compact
  - Slow
- Just-in-time compilation
  - State-of-the-art for desktop/server
  - Too resource consuming in embedded systems
- Batch compilation
- Hardware implementation

JVM Data Types

`reference`	Pointer to an object or array
`int`	32-bit integer (signed)
`long`	64-bit integer (signed)
`float`	32-bit floating-point (IEEE 754-1985)
`double`	64-bit floating-point (IEEE 754-1985)

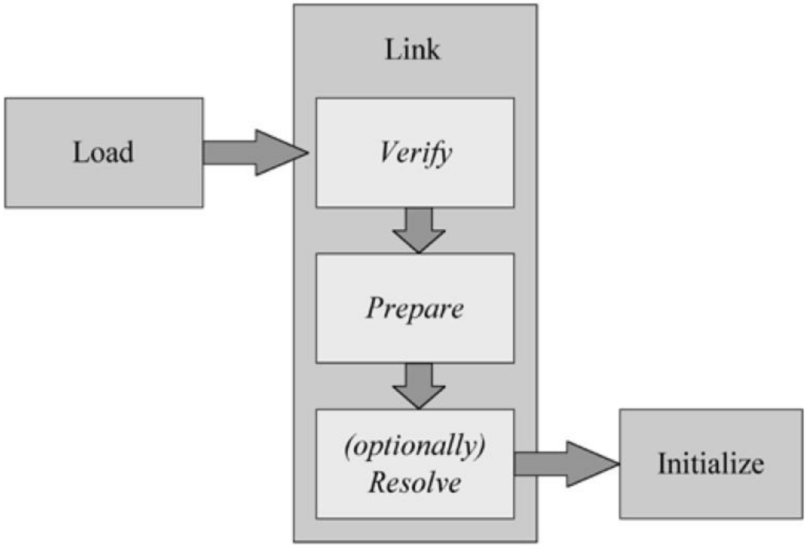
- **No `boolean`, `char`, `byte`, and `short` types**
  - **Stack contains only 32-bit and 64-bit data**
  - **Conversion instructions**

JVM Architecture

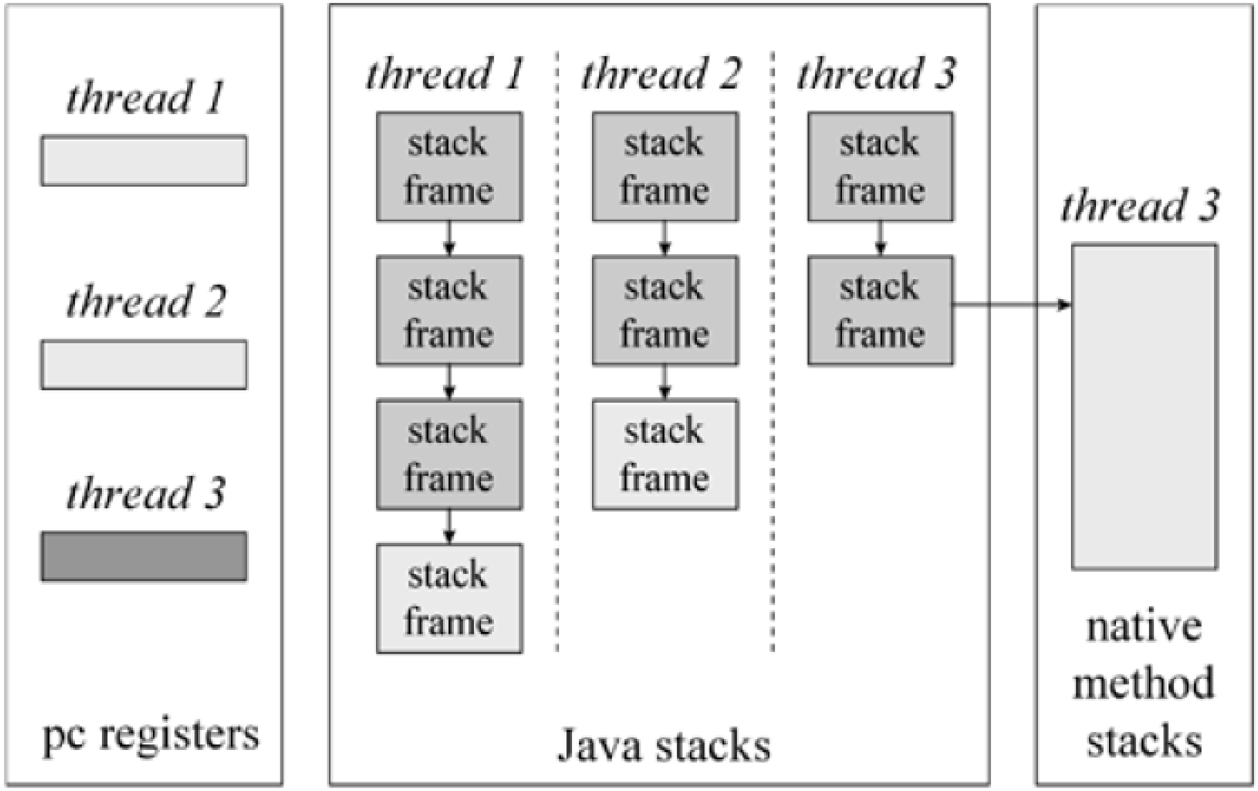


Class Loader

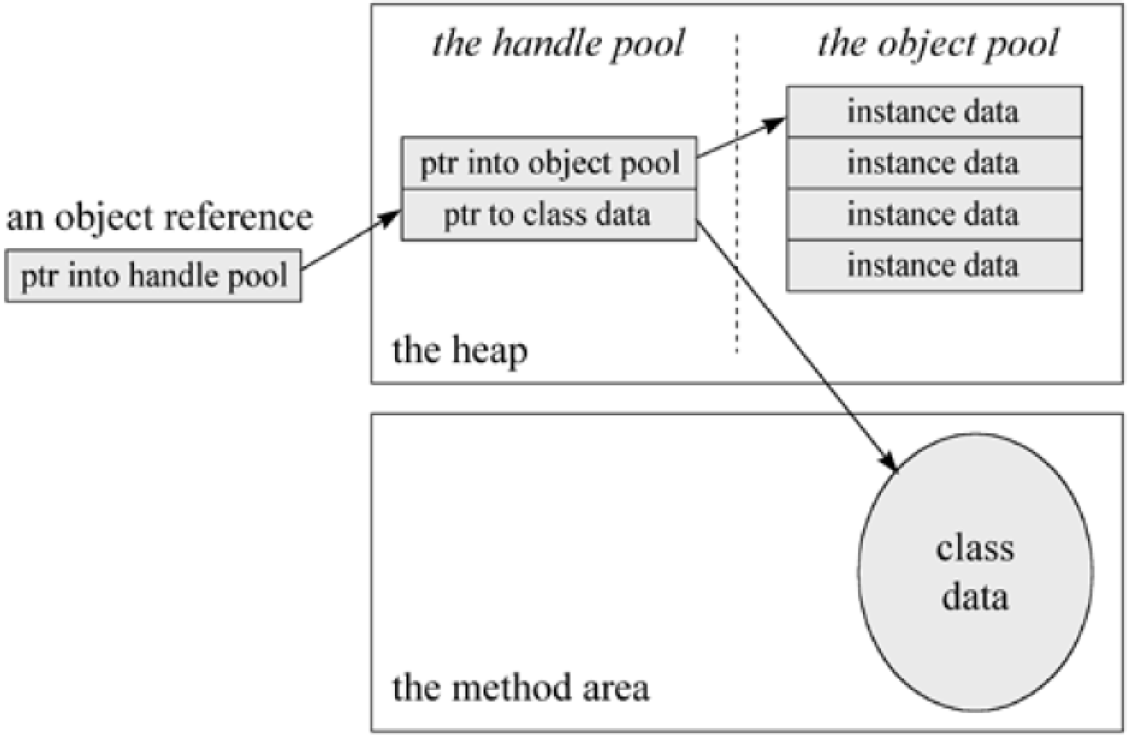
- **Loading: finding and importing the binary data for a type**
- **Linking:**
  - Verification
  - Preparation
  - Resolution
- **Initialization: invoking Java code that initializes class variables to their proper starting values**



Runtime Areas



Heap


the handle pool
the object pool
instance data
instance data
instance data
instance data
ptr into object pool
ptr to class data
an object reference
ptr into handle pool
the heap
class data
the method area

Method Area

- The fully qualified name of the type
- The fully qualified name of the type’s direct superclass
- Whether or not the type is a class or an interface
- The type’s modifiers (public,…)
- An ordered list of the fully qualified names of any direct superinterfaces
- The constant pool for the type
- Field information
- Method information
- All class (static) variables declared in the type
- A reference to class ClassLoader
- A reference to class Class

Activation Record

- The activation record has three parts:
  - Local variables
  - Operand stack
  - Frame data
- The sizes of the local variables and operand stack are determined at compile time

Java Bytecode

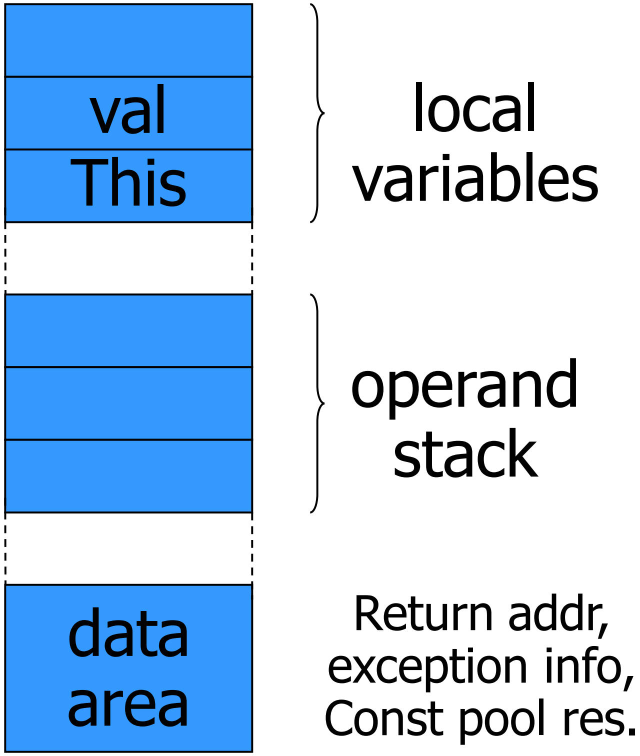
- **Java**

```
Class A extends Object {
    int i;
    void f(int val) {
        i = val + 1;
    }
}
```

- **Bytecode**

```
Method void f(int)
    aload  0   ;object ref this
    iload  1   ; int val
    iconst 1
    iadd       ; add val +1
    putfield #4 <Field int i>
    return
```

JVM Activation Record



Field and Method Access

- **Instruction includes index into constant pool**
  - **Constant pool stores symbolic names**
- **First execution**
  - **Use symbolic name to find field or method**
- **Second execution**
  - **Use modified “quick” instruction to simplify search**
  - **Putfield_quick 6**

invokevirtual <method-spec>

- Search for method
  - find the method entry in the constant pool
  - pop arguments
  - find method with the given name and signature using the reference
- Java

```
Object x;
x.equals(”test”);
```

- ByteCode

```
aload_1
ldc ”test”
invokevirtual java/lang/Object/equals
```

Polymorphism

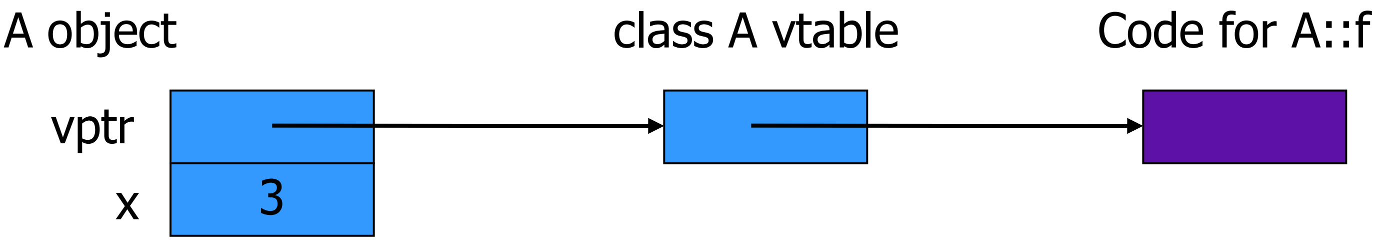
- **Ad-hoc polymorphism**
  - Supported by both C++ and Java
    - C++: allow overloading both operators and functions
    - Java: disallow overloading of operators
  - Overloading both resolved at compile-time
- **Subtype polymorphism**
  - Implicit cast from subtype to basetype, explicit cast from
  - basetype to subtype
  - C++: allow static casting from basetype to subtype
  - Java: runtime check required for basetype to subtype castings
- **Parametric polymorphism**
  - C++ templates: type-checked at link-time
  - Java generics: based on dynamic casting

Object Implementation

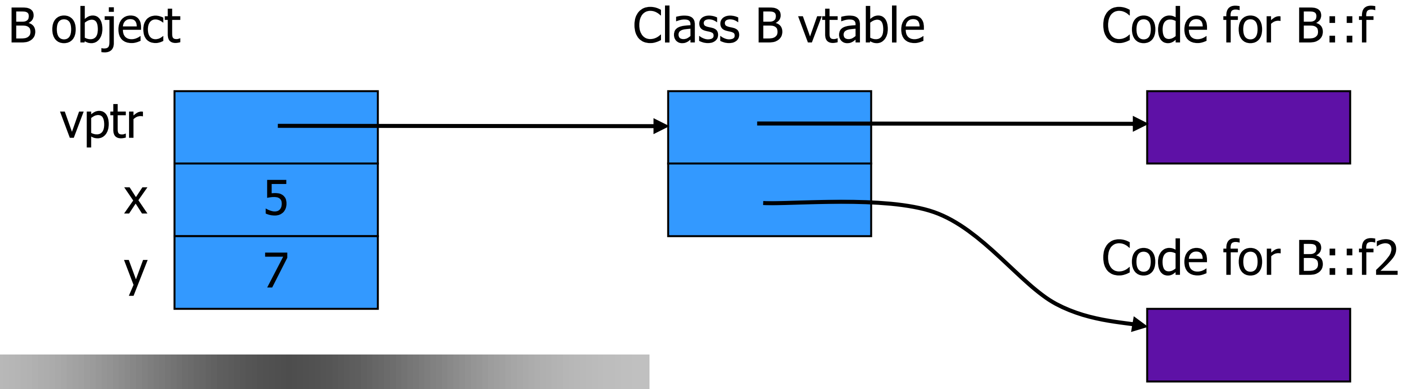
- **C++ classes can be seen as extending C structs with**
  - **Encapsulation (access control)**
    - **Extend type system to check member access**
  - **Dynamic binding (function pointers as members)**
    - **Store the function pointer inside the class object**
    - **Collect all function pointers separately into a table (vtable in C++)**
  - **Subtyping and implementating inheritance**
    - **A derived class object should look just like a base class object so that it can be used as a base class object**
    - **Need to be able to dynamically extend the collections of both data and function members**

C++ Object Layout & Method Lookup

```
class A { int x; public: virtual int f() { return x; }};
```

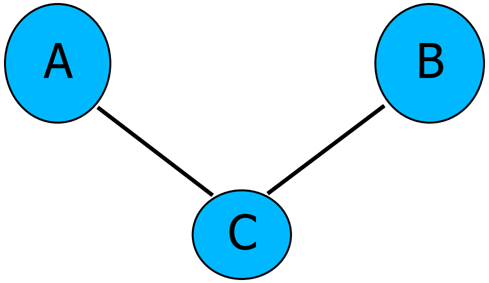


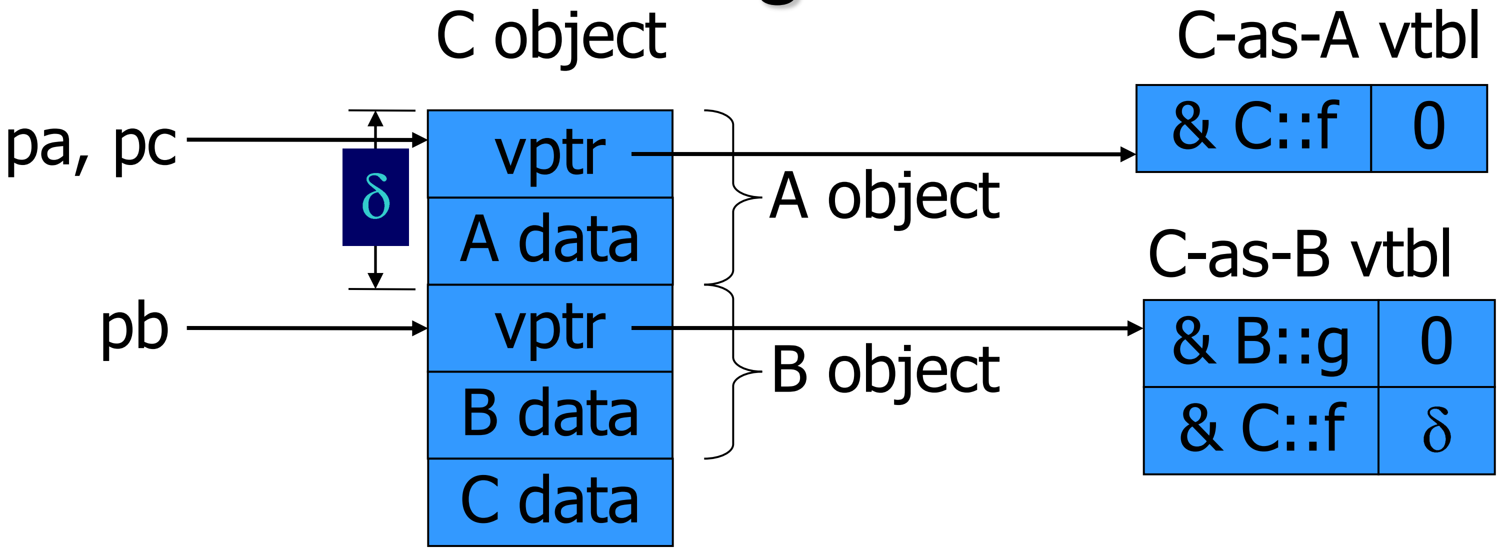
```
class B : public A { int y; public: virtual int f() { return y; }
        virtual void f2() { ... }
}; A a = new B(); a.f();
```



C++ Approach

- **C Extends A,B**
- **More memory usage**
- **Less dereferencing**





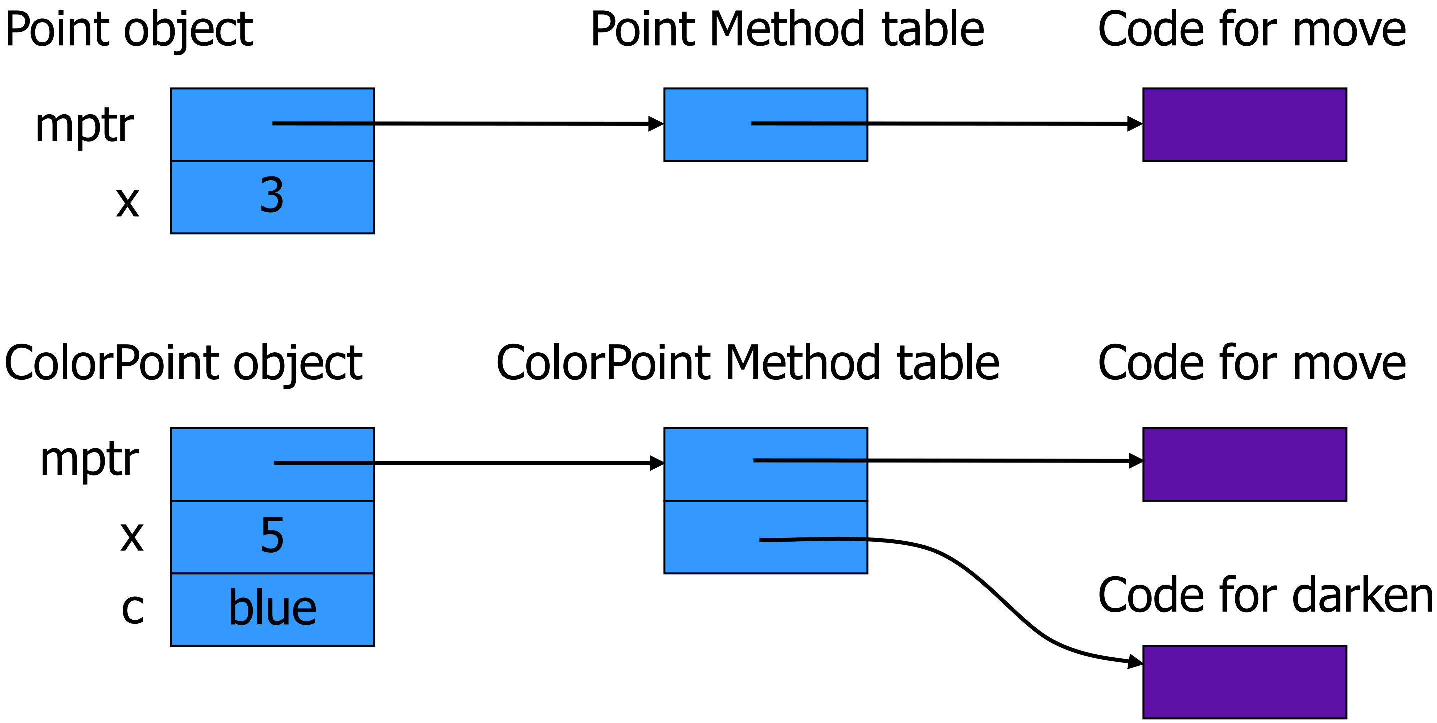
C-as-A vtbl

& C::f	0

C-as-B vtbl

& B::g	0
& C::f	δ

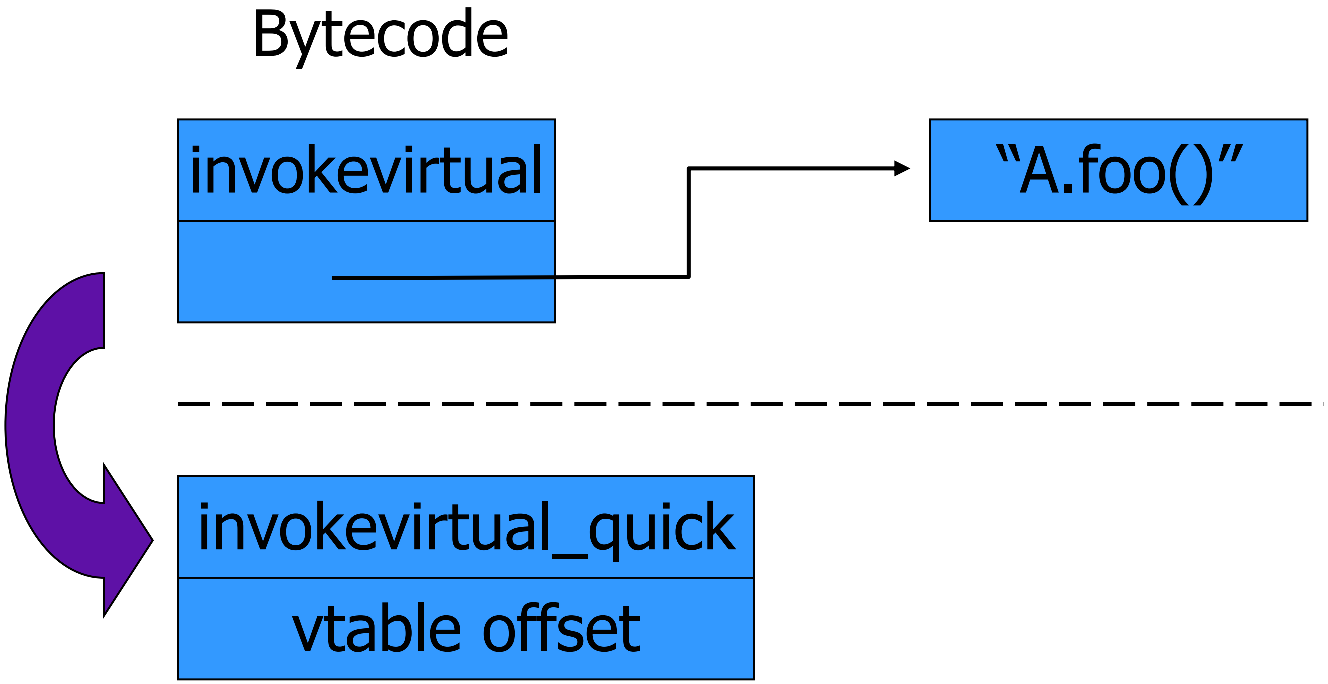
JVM Method Lookup



Point p = new ColorPoint(3, 2, “RED”);

p.move(2, 3);   // (p.mptr[0])(p,2, 3)

Bytecode Rewriting: invokevirtual



After search, rewrite bytecode to use fixed offset into the vtable.

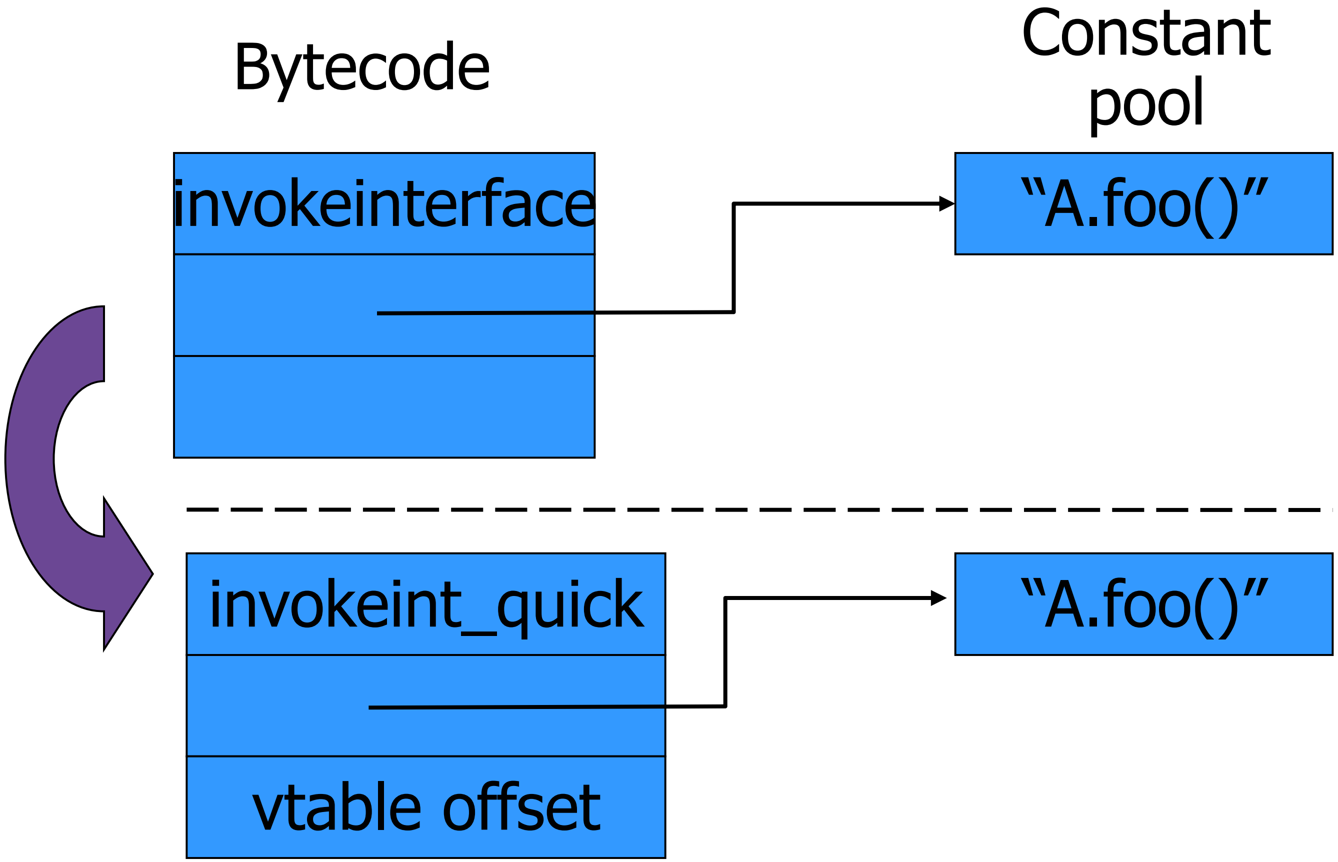
No search on second execution.

invokeinterface <method-spec>

- **Interfaces**
- **Problem with multiple interface**
- **Solutions:**
  - **Multiple method tables**
  - **Search method table**

```
interface Incrementable {
    public void inc();
}
class IntCounter implements Incrementable {
   public void add(int);
   public void inc();
}
class FloatCounter implements Incrementable {
   public void inc();
}
void add2(Incrementable x) { x.inc(); }
```

Bytecode Rewriting: invokeinterface



Cache address of method; check class on second use

Memory Areas for the JVM

- **Method area**
  - **Class description**
  - **Code**
  - **Constant pool**
- **Heap**
  - **Objects and Arrays**
  - **Shared by all threads**
  - **Garbage collected**

Memory Areas for the JVM

- **Stack**
  - **Thread private**
  - **Logical stack that contains:**
    - **Invocation frame**
    - **Local variable area**
    - **Operand stack**
  - **Not necessary a *single* stack**
  - **Local variables and operand stack are accessed frequently**

JVM Instruction Set

- 32 (64) bit stack machine
- Variable length instruction set
- Simple to very complex instructions
- Symbolic references
- Only relative branches

JVM Instruction Set

- Load and store
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack manipulation
- Control transfer
- Method invocation and return

Dissassembling Java

- Compile
  - `javac Hello.java`
- Run
  - `java Hello`
- Dissassemble
  - `javap -c Hello`

A Bytecode Example

```
public class X {

   public static void
   main(String[] args) {
        add(1, 2);
   }

   public static int
   add(int a, int b) {
        return a+b;
   }
}
```

```
public static void
   main(java.lang.String[]);
  Code:
   0:   iconst_1
   1:   iconst_2
   //Method add:(II)I
   2:   invokestatic    #2;
   5:   pop
   6:   return

public static int
   add(int,int);
  Code:
   0:   iload_0
   1:   iload_1
   2:   iadd
   3:   ireturn
```

Java Virtual Machine

- instruction set
  - set of registers
- pc, optop, frame, vars
  - stack
- local variables
- execution environment
- operand stack
  - garbage-collected heap
  - method area

Instruction Set

- **Operands on stack, result on stack**
- **Variants for byte, short, integer, float, double, address**
  - push
  - load
  - store
  - array
  - stack
- **pop, dup, dup_x, swap**
  - arithmetic
  - logic
  - conversion

Instruction Set (2)

- control
- ifeq, iflt, ifle, …
- jsr, ret, goto
  - function return
  - table jumping
- tableswitch, lookupswitch
  - object fields
  - method invocation
  - exception handling
  - new, instanceof, checkcast
  - monitor

Method Invocation

invokevirtual	index1	index2

Stack: …, object, arg1, arg2, … => …

index1 index2 used to retrieve signature from constant pool

1. retrieve object’s method table
2. lookup method => index
3. get method from method table of class
4. if method is synchronized, get lock
5. prepare operand stack and environment
6. jump

Other Method Invocations

- **invokenonvirtual, invokestatic, invokeinterface: similar**