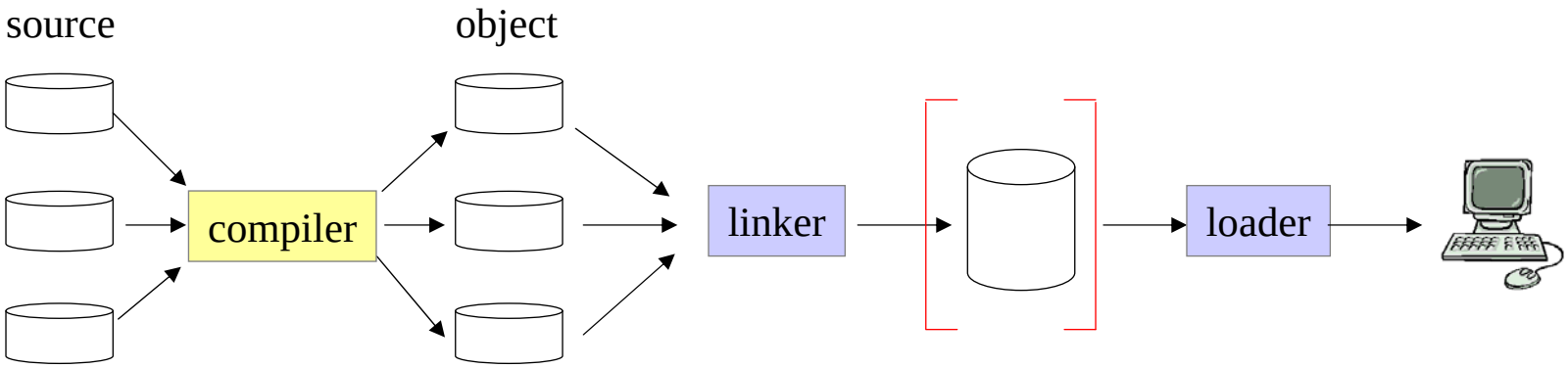
3. Linkers and Loaders

Overview



- combines object files
- resolves external references

- allocates memory for code and data
- loads the program into memory
- allocates stack and heap
- starts the program

Common practice today:

- *Linking loaders*: After linking, the program is not written to a file but immediately executed.
- *Dynamic linking and loading*: New program parts can be added to a running program.

Contents of an object file

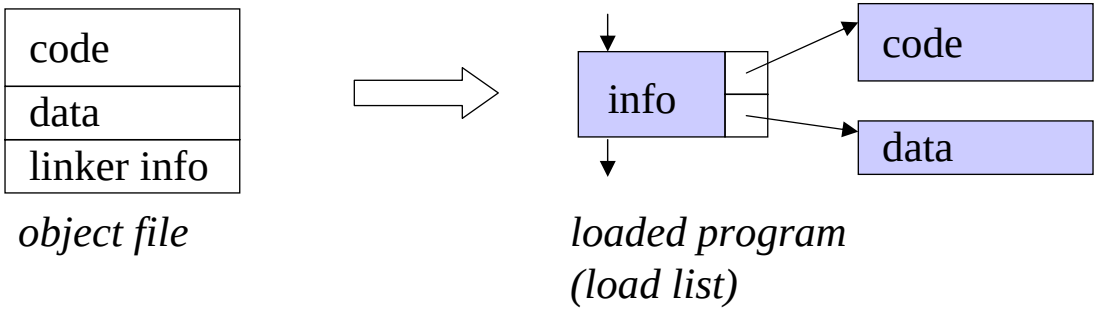
- Name
- Version
- Code
- Data (initialized global data, string constants)
- List of imported names (classes, modules)
- List of exported names (variables, methods)
- Fixup table

to resolve references between object files

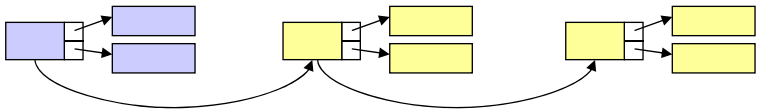
- Additional information
  - Pointer offsets for the garbage collector
  - Reference information for the debugger and for reflection

Tasks of a linking loader (1)

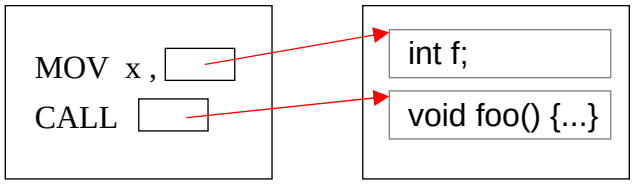
1. Reads the object file and allocates memory for code and data



2. Loads imported classes or modules recursively



3. Resolves external references ("fixups")



- Compiler cannot resolve these references
- Generates *fixup information* that is used to resolve the references at link time

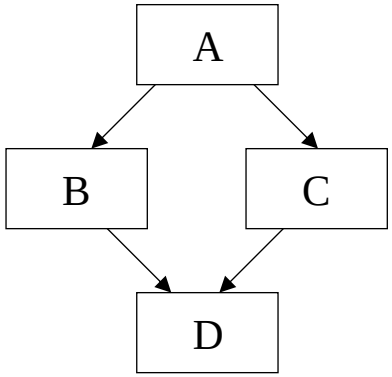
Tasks of a linking loader (2)

4. Initializes run-time data structures

module or class descriptor

- method table
- pointer offsets for garbage collector
- reference information for debugger and reflection
- infos for dynamically linking other modules or classes

5. Executes initialization code (static constructor or module body)



A uses *B*, *C*

```
load A
    load B
        load D
        init D
    fixup B
    init B
    load C
        -- D already loaded!
    fixup C
    init C
fixup A
init A
```

the loader is called recursively for imported classes or modules

Relocatable code

Goal

- Code should contain as few unresolved references as possible
- Code should be placeable at arbitrary addresses (relocatable code, position-independent code)

Solution: relocatable accesses

Access to local variables

```
MOV EAX, -8[EBP]
```

offsets relative to EBP (base pointer, frame pointer)

Access to fields

```
MOV EAX, 4[EBX]
```

offsets relative to the base address of the object (e.g. EBX)

Call of local static methods

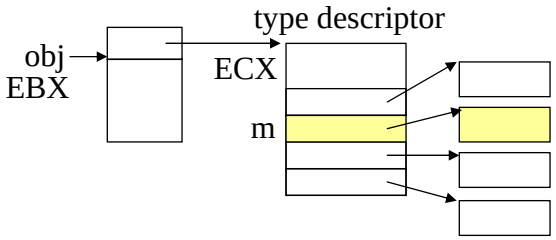
```
CALL -320
```

offsets relative to EIP (instruction pointer)

Call of dynamically bound methods (e.g. *obj.m(x);*)

```
MOV   EBX, obj
MOV   ECX, -4[EBX]
PUSH  EBX
PUSH  x
CALL  [m[ECX]]
```

offsets relative to a method table



offset *m* is known at compile time

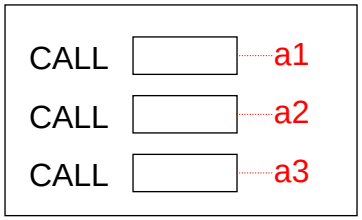
but method table needs to be set up by the linker

External references

For accessing

- external static variables
- external static methods

Direct references

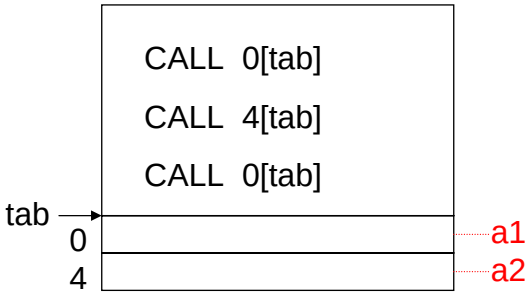


fixup table

a1	1	3
a2	2	5
a3	1	3

module entry

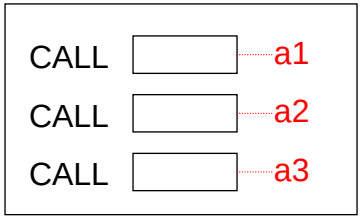
Indirect references



fixup table

a1	1	3
a2	2	5

Symbolic references

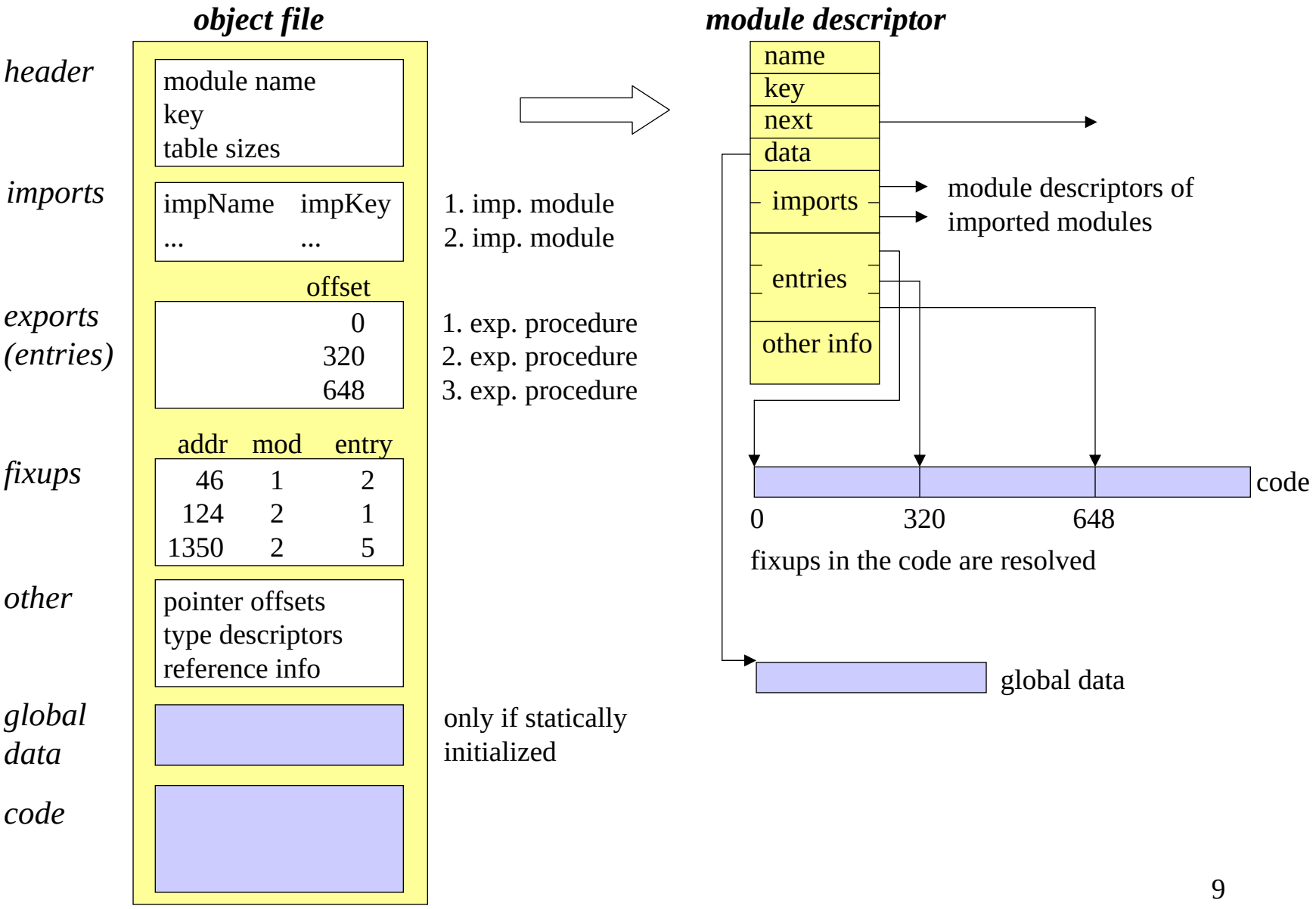


fixup table

a1	ModA	foo()
a2	ModB	bar()
a3	ModA	foo()

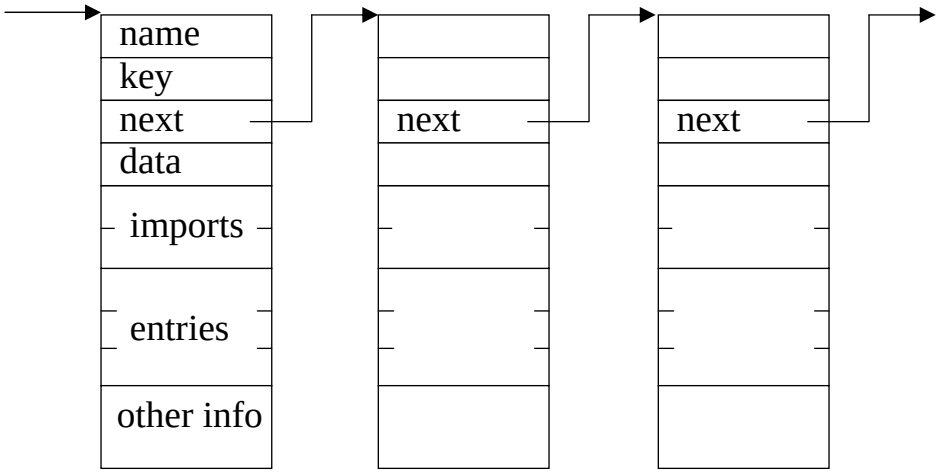
3. Linkers and Loaders

Object file and module descriptors



Load list

Module descriptors of all loaded modules




```
moduleDescr = load (moduleName);
```

- loads the specified module *M* from the file *moduleName.Obj*
- creates a module descriptor and adds it to the load list
- loads the modules that are imported by *M* (if not already loaded) and links them with *M*
- initializes all loaded modules

```
moduleDescr = find (moduleName);
```

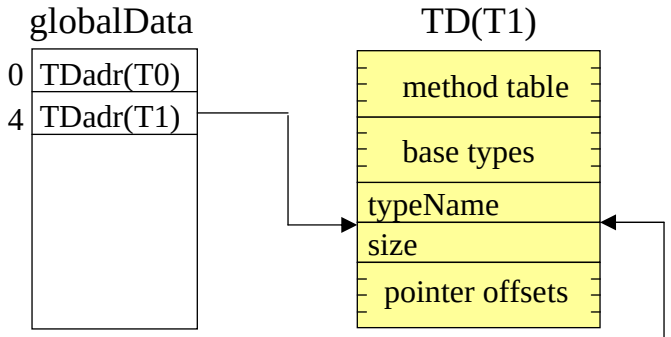
- searches the specified module in the load list and returns its descriptor or NIL

Type descriptors

Record types are described by type descriptors at run time

```
TYPE
   T0 = RECORD
      a: INTEGER;
      b: POINTER TO ...;
   END;

   T1 = RECORD (T0)
      c: POINTER TO ...;
      PROCEDURE (VAR this: T1) Foo();
      PROCEDURE (VAR this: T1) Bar();
   END;
```

- Compiler writes type info into the object file
- Loader loads type info and creates type descriptor



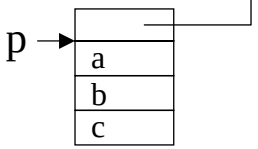
- Loader stores TD addr. at fixed location in *GlobalData*

What happens when a new object is created?

```
VAR p: POINTER TO T1;
NEW(p);
```

code generated by the compiler

```
p = alloc(size(T1)) + 4;
mem[p-4] = GlobalData[TDadr(T1)];
```



size(T1) ... 16
TDadr(T1) ... 4
} known at compile time

3. Linkers and Loaders

Resolving accesses to global variables

At compile time, global variables get offsets relative to the global data area

```
MODULE M0;
   VAR x, y: LONGINT;
   ...
END M0.
```

```
MODULE M1;
   VAR a*, b*: LONGINT;
   ...
END M1.
```

globalData(M0)

0	x
4	y
8	
12	

globalData(M1)

0	a
4	b
8	
12	

For accesses to global variables, the compiler generates fixup cells containing offsets

```
MODULE M0;
   ...
   x := 10;

   M1.b := 20;
```

code for M0

```
MOV  a1: 0 , 10
MOV  a2: 4 , 20
```

fixup table for M0

addr	mod	entry
a1	0	255
a2	1	255
...	...	...

special entry denoting globalData

0 ... this module
1 ... 1. imported module

Loader allocates globalData areas

e.g. globalData(M0) at address 1000
globalData(M1) at address 2000

Loader resolves fixups: adds globalData address to offsets in fixup cells

```
MOV  a1: 1000 , 10
MOV  a2: 2004 , 20
```

global variables are accessed via absolute addresses

Resolving calls to imported procedures

At compile time, every exported procedure gets an entry number

```
MODULE M1;
   PROCEDURE P0*; ...
   PROCEDURE P1*; ...
   PROCEDURE P2*; ...
END M1.
```

entry numbers

----- 0
----- 1
----- 2

entries in object file

0	0
1	500
2	1000

} entry offsets

For calls of imported procedures the compiler generates fixup cells

```
MODULE M0;
   M1.P1;
   M1.P2;
   M1.P1;
END M0.
```

code for M0

CALL	-1	a1
CALL	-1	a2
CALL	a1	a3

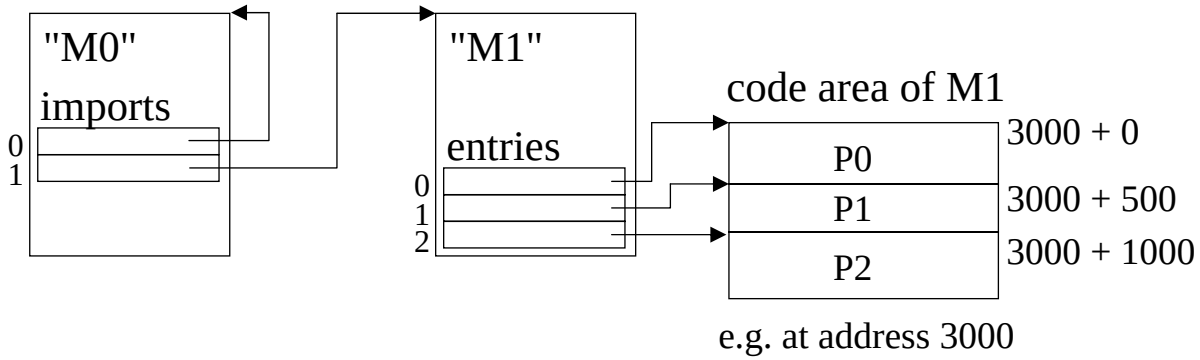
fixup table for M0

addr	mod	entry
a2	1	2
a3	1	1
...	...	...

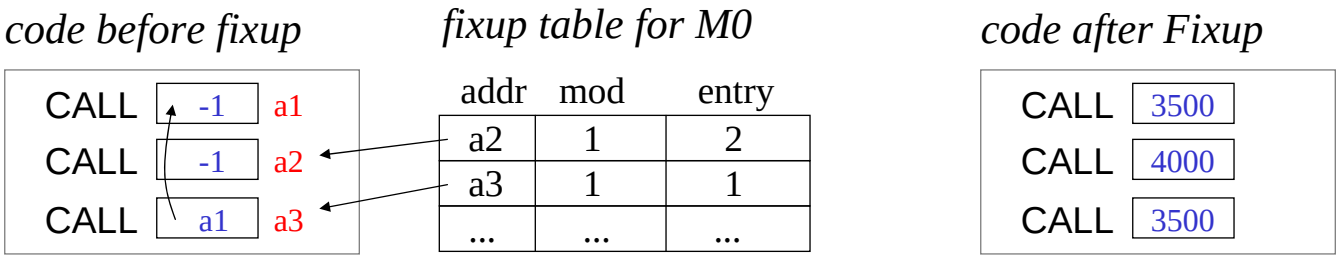
fixup cells of the same procedure are linked

Resolving calls to imported procedures

Loader creates module descriptors and allocates code area

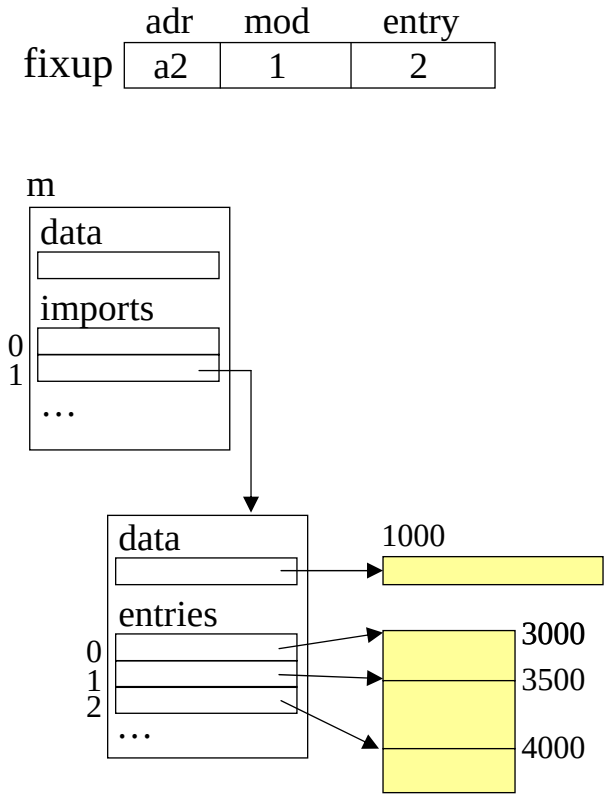


Loader resolves fixups



Fixup algorithm

```
void fixup (Module m) {
    for (int i = 0; i < m.nofFixups; i++) {
        Fixup fixup = m.fixups[i];
        if (fixup.entry = 255) {  // resolve variable access
            int adr = m.imports[fixup.mod].data;
            put(fixup.adr, adr + get(fixup.adr));
        } else {  // resolve procedure call
            int adr = m.imports[fixup.mod].entries[fixup.entry];
            int p = fixup.adr;
            while (p >= 0) {
                int q = get(p);
                put(p, adr);
                p = q;
            }
        }
    }
}
```



3. Linkers and Loaders

Load algorithm

```
Module load (String name) {
   Module m = find(name);  // search in module list
   if (m == null) {
      m = new Module();
      readObjectFile(name + ".Obj", m);
      modules.add(m);
      for (int i = 1; i < m.nofImports; i++) {
         Module m1 = load(m.importName[i]);
         if (m1 != null && m1.key == m.importKey[i])
            m.imports[i] = m1;
      }
      fixup(m);
      for (int i = 1; i < m.nofImports; i++) {
         m.imports[i].refCount++;
      }
      callModuleBody(m);
   }
   return m;
}
```

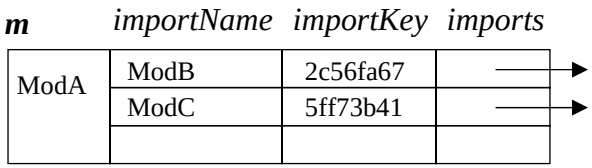
read object file and create module descriptor

load imported modules and check for version consistency

resolve external references

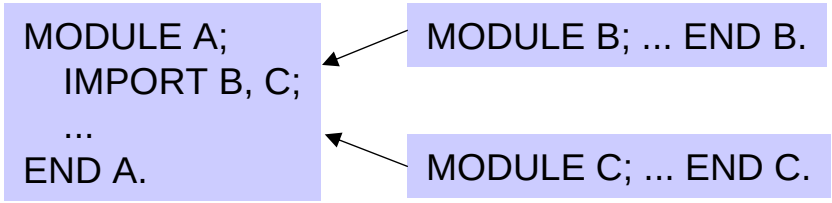
increment reference counter in imported modules

initialize loaded module



Reference counters are needed for module unloading:
only modules that are not referenced any more can be unloaded

Version numbers

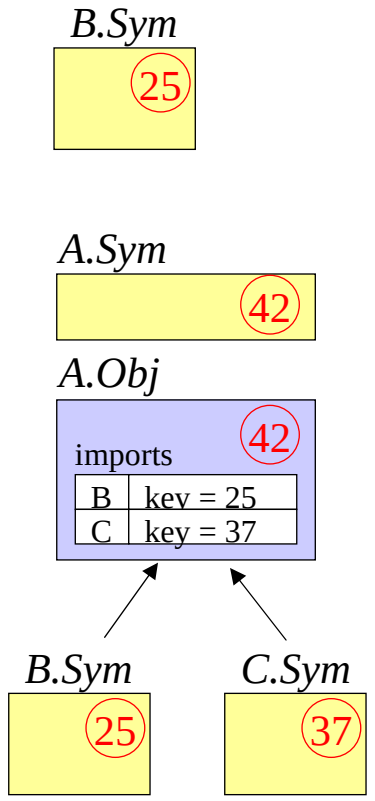


Symbol file (e.g. *B.Sym*)

- contains interface description of *B* (in binary form)
- contains version number of *B* (time stamp or check sum)

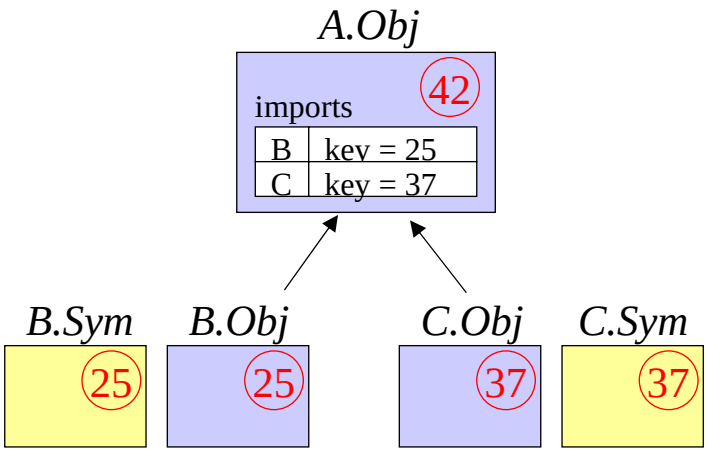
Object file (e.g. *A.Obj*)

- contains object code of *A*
- contains version number of *A*
- contains version numbers of all imported modules
  (i.e. compiler has type-checked *A* against these versions)

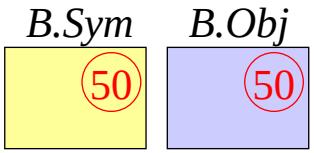


Version check during loading

Loader checks if the object files have the expected version numbers



If the interface of *B* is changed, *B.Sym* and *B.Obj* get a new version number



If we now try to load *A* without recompiling it the loader reports an error, because it expects *B* in version 25 but finds version 50.

3. Linkers and Loaders

Concepts of class loading in Java

1. **Dynamic loading**
   - Classes are loaded at run time "on demand" (lazy loading)
   - Class loading is delayed as long as possible, in order to get fast startup times
2. **Type-safe symbolic linking**
   - Class files and bytecodes are verified during loading
   - External references are denoted by symbolic signatures that are resolved during linking
     e.g. getfield <Field myUtil.Buffer.f int> => getfield 2
   - During resolution, type checking is partially redone
3. **Users can implement custom loaders**
   - There are multiple loaders, each responsible for a different kind of classes
   - E.g., users can implement a loader, which loads classes over the network
4. **Namespaces**
   - All classes loaded by the same loader form a *namespace*
   - Classes in different namespaces don't see each other

Basic classes for loading


```
class Class {
    static Class forName (String name);
    ...
}
```

Standard way to load a class

```
abstract class ClassLoader {
    Class loadClass (String name);
    ...
}
```

Ask a specific class loader to load a class

Explicit loading

```
Class c = Class.forName("myUtil.Buffer");
```

assume: *MyLoader* is a subclass of the abstract class *ClassLoader*

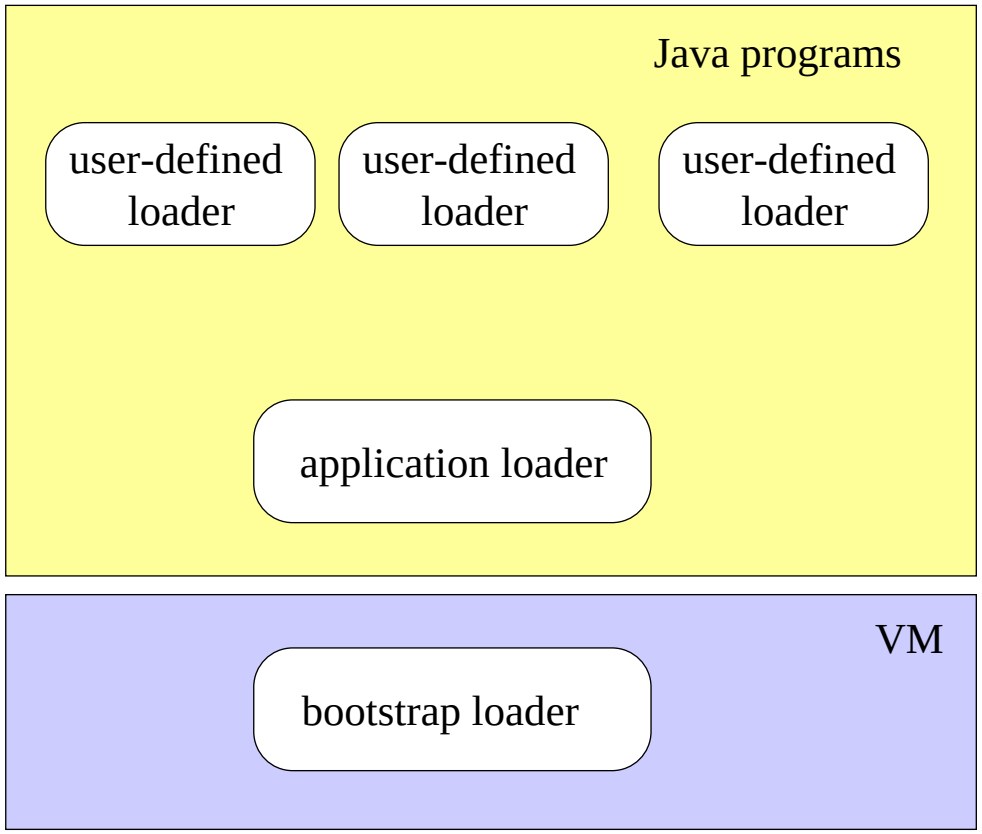
```
MyLoader loader = new MyLoader(...);
loader.loadClass("myUtil.Buffer");
```

Implicit loading

```
class A {
    B b;
    void foo(C c);
}
```

If *A* is loaded, *B* and *C* are automatically loaded as well (by the same loader)

Kinds of class loaders



User-defined loaders

- written in Java (by the user)
- load classes in their own way, e.g.:
  - over the network (e.g. applets)
  - from a special directory
  - ...

Application loader

- written in Java (part of the library)
- loads all classes in CLASSPATH

Bootstrap loader

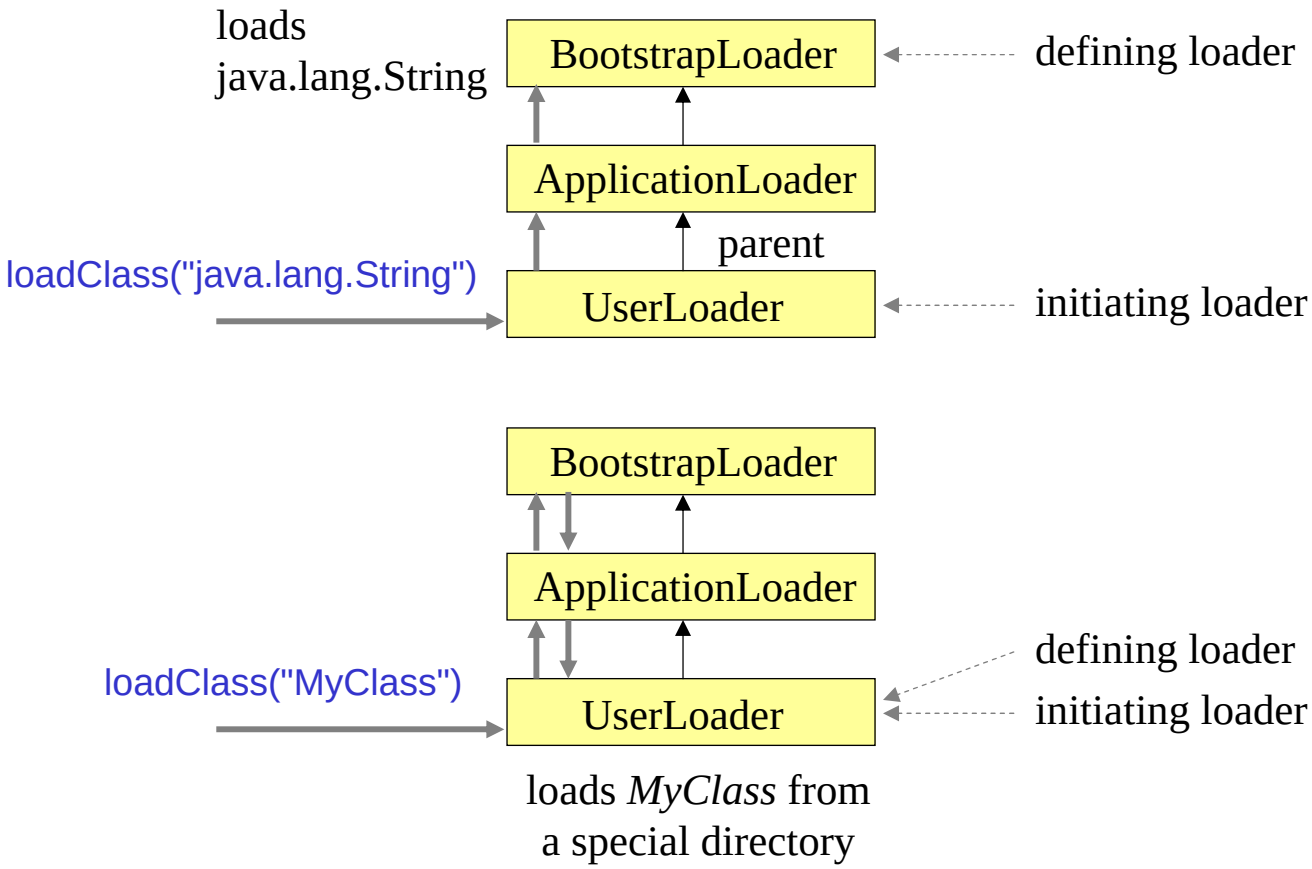
- written in C++ (part of the VM)
- loads all system classes (java.lang.*)

What happens if a Java application is started? (java MyApp)

```
AppLoader appLoader = new AppLoader();
Class app = appLoader.loadClass(appName + ".class");
... // invoke main() method of app
```

Parent delegation model

- Every loader (except the bootstrap loader) has a parent loader
- Every loader delegates a load request to the parent loader first
- If the parent loader cannot load the class, the first loader loads it



- Purpose: user-defined loaders should not be allowed to load (critical) system classes

Setting the parent class loader


```
abstract class ClassLoader {

    protected ClassLoader () {...}

    protected ClassLoader (ClassLoader parent) {...}
    ...
}
```

parent = application loader

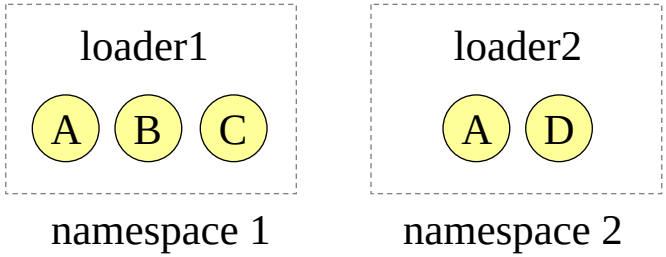
sets parent explicitly

```
class MyLoader extends ClassLoader {

    public MyLoader (String path) {
        super();     // sets application loader as the parent of MyLoader
        ...
    }

    ...
}
```

e.g., should load classes
that are in the specified path

Namespaces

All classes loaded by the same loader form a namespace



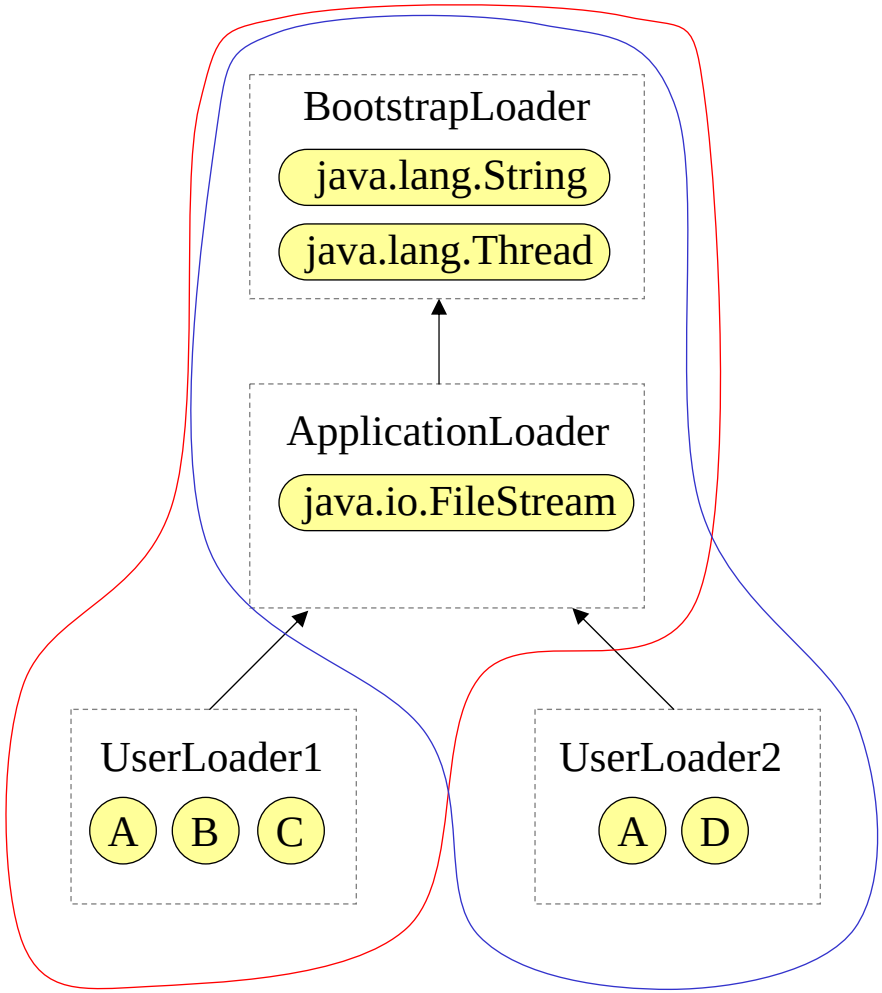
- Classes in namespace 1 don't see the classes in namespace 2 (protection e.g. for applets)
- A class can be loaded into multiple namespaces at the same time
- Unique class identification:
  class name + defining loader

Referenced classes are loaded by the same loader as the referencing class

```
class A {
    B b;
    void foo(C c) {...}
}
```

A references *B* and *C*
=> *B* and *C* are loaded by the same loader as *A*
=> *A*, *B* and *C* are in the same namespace

Sharing of namespaces

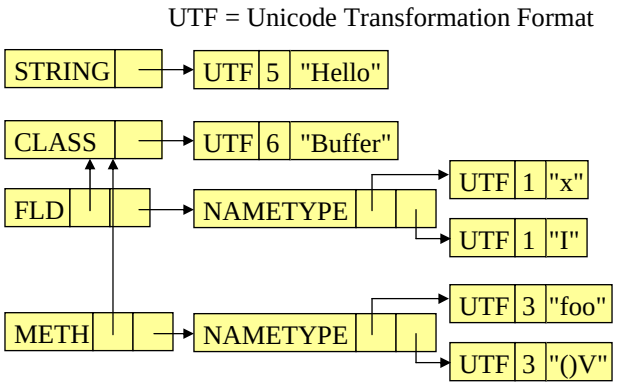


- Classes see also the namespaces of their parent loaders
- *B* and *D* see the same *String* class, but they don't see each other

3. Linkers and Loaders

Class file format (1)


```
ClassFile  =  magic4 version4                          // magic = 0xCAFEBABE
              nconst2 {Constant}                       // constant pool
              flags2 thisx superx
              nintf2 {interfacex}                      // interfaces
              nflds2 {Field}                           // fields
              nmeth2 {Method}                          // methods
              nattr2 {SourceFile | OtherAttr} .        // attributes

Constant   =  INT          val4                        // INT          = 3
           |  FLOAT        val4                        // FLOAT        = 4
           |  LONG         val8                        // LONG         = 5
           |  DOUBLE       val8                        // DOUBLE       = 6
           |  STRING       valx                        // STRING       = 8
           |  UTF          len2 bytes len              // UTF          = 1
           |  CLASS        namex                       // CLASS        = 7
           |  FLD          classx nameTypex            // FLD          = 9
           |  METH         classx nameTypex            // METH         = 10
           |  IMETH        classx nameTypex            // IMETH        = 11
           |  NAMETYPE     namex typex .               // NAMETYPE     = 12
```




```
Field      =  flags2 namex typex n2 {ConstantValue | OtherAttr}.

Method     =  flags2 namex typex n2 {Code | Exceptions | OtherAttr}.
```

Class file format (2)

SourceFile = $name_x$ len_4 $sourceFileName_x$. // name = "SourceFile"

ConstantValue = $name_x$ len_4 val_x. // name = "ConstantValue"

Code = $name_x$ len_4 $maxStack_2$ $maxLocals_2$ // name = "Code"
$codeLen_4$ {byte}
$excLen_2$ {$startPC_2$ $endPC_2$ $handlerPC_2$ $exception_x$}
n_2 {LineNumberTable | LocalVariableTable | OtherAttr}.

Exceptions = $name_x$ len_4 n_2 {$excClass_x$}. // name = "Exceptions"

LineNumberTable = $name_x$ len_4 n_2 {$startPC_2$ $lineNum_2$}. // name = "LineNumberTable"

LocalVariableTable = $name_x$ len_4 // name = "LocalVariableTable"
n_2 {$startPC_2$ len_2 $name_x$ $type_x$ adr_2}. // valid in [startPC .. startPC + len]

OtherAttr = $name_x$ len_4 $\{byte\}_{len}$.

Encoding of types and signatures

Primitive types

byte	B
char	C
double	D
float	F
int	I
long	J
short	S
boolean	Z
void	V

Arrays

int[]	[I
long[][]	[[J

Classes

String	Ljava/lang/String;
Hashtable[]	[Ljava/util/Hashtable;

Method signatures

int getSize()	()I
String toString()	()Ljava/lang/String;
void main(String[] args)	([Ljava/lang/String;)V
void wait(long timeout, int nanos)	(JI)V
int read(byte[] b, int off, int len)	([BII)I

Sample class


```
class Buffer {
   private int[] data;
   private int len;

   Buffer(int size) {
      data = new int[size];
      len = 0;
   }

   void insert(String s, int pos) {}

   void insert(char[] a) {}

   String substring(int from, int to) { return null; }
}
```

class file decoder

```
javap -l -s -c -verbose Buffer
```

-l	print line number and local variable table
-s	print internal type signatures
-c	disassemble the code
-verbose	print stack size, number of locals, method args
...	

Contents of the class file (1)

	CA FE BA BE	magic
	00 00 00 32	version
	00 1B	number of constant pool entries
1:	0A 00 05 00 15	METH 5 21
2:	09 00 04 00 16	FLD 4 22
3:	09 00 04 00 17	FLD 4 23
4:	07 00 18	CLASS 24
5:	07 00 19	CLASS 25
6:	01 00 04 64 61 74 61	UTF 4 data
7:	01 00 02 5B 49	UTF 2 [I
8:	01 00 03 6C 65 6E	UTF 3 len
9:	01 00 01 49	UTF 1 I
10:	01 00 06 3C 69 6E 69 75 3E	UTF 6 <init>
11:	01 00 04 28 49 29 56	UTF 4 (I)V
12:	01 00 04 43 6F 64 65	UTF 4 Code
13:	01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65	UTF 15 LineNumberTable
14:	01 00 06 69 6E 73 65 72 74	UTF 6 insert
15:	01 00 16 28 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B 49 29 56	UTF 22 (Ljava/lang/String;I)V
16:	01 00 05 28 5B 43 29 56	UTF 5 ([C)V
17:	01 00 09 73 75 62 73 74 72 69 6E 67	UTF 9 substring
18:	01 00 16 28 49 49 29 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B	UTF 22 (II)Ljava/lang/String;
19:	01 00 0A 53 6F 75 72 63 65 46 69 6C 65	UTF 10 SourceFile
20:	01 00 0B 42 75 66 66 65 71 2E 6A 61 76 61	UTF 11 Buffer.java
21:	0C 00 0A 00 1A	NAMETYPE 10 26
22:	0C 00 06 00 07	NAMETYPE 6 7
23:	0C 00 08 00 09	NAMETYPE 8 9
24:	01 00 06 42 75 66 66 65 72	UTF 6 Buffer
25:	01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74	UTF 16 java/lang/Object
26:	01 00 03 28 29 56	UTF 3 ()V

constant pool

Contents of the class file (2)

class

```
00 20 00 04 00 05        flags this super
00 00                    nintf
```

fields

```
00 02                    number of fields
1: 00 02 00 06 00 07 00 00   flags name type n
2: 00 02 00 08 00 09 00 00   flags name type n
```

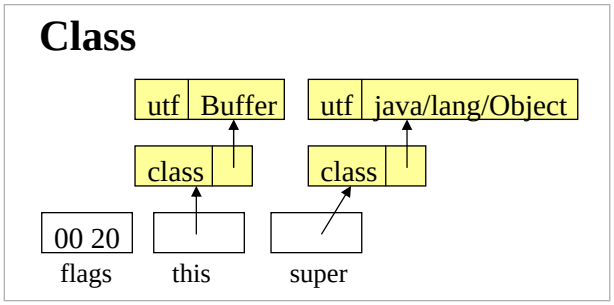
methods

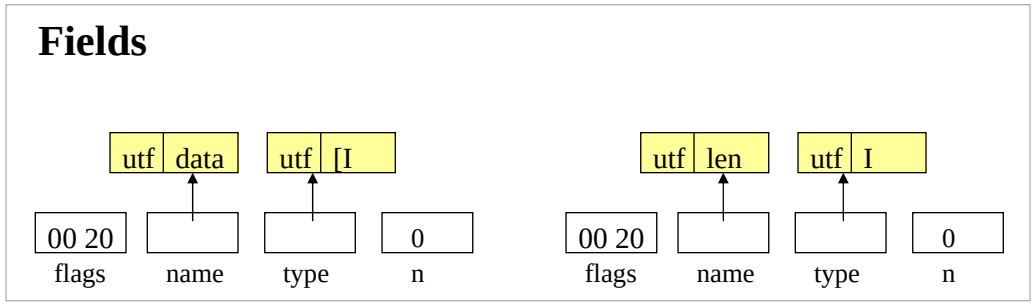
```
00 04                    number of methods

1: 00 00 00 0A 00 0B     flags name type
      00 01 00 0C 00 00 00 35 00 02 00 02   n name len maxstack maxlocals
      00 00 00 11 2A B7 00 01 2A 1B BC 0A B5 00 02 2A 03 B5 00 03 B1   codelen code
      00 00              exclen
      00 01 00 0D 00 00 00 12 00 04 00 00 00 05 00 04 00 06 00 0B 00 07 00 10 00 08   n name len nLineNumbers {pc line}

2: 00 00 00 0E 00 0F     flags name type
      00 01 00 0C 00 00 00 19 00 00 00 03   n name len maxstack maxlocals
      00 00 00 01 B1     codelen code
      00 00              exclen
      00 01 00 0D 00 00 00 06 00 01 00 00 00 0A   n name len nLineNumbers {pc line}

3: 00 00 00 0E 00 10     flags name type
      00 01 00 0C 00 00 00 19 00 00 00 02   n name len maxstack maxlocals
      00 00 00 01 B1     codelen code
      00 00              exclen
      00 01 00 0D 00 00 00 06 00 01 00 00 00 0B   n name len nLineNumbers {pc line}

4: 00 00 00 11 00 12     flags name type
      00 01 00 0C 00 00 00 1A 00 01 00 03   n name len maxstack maxlocals
      00 00 00 02  01 B0 codelen code
      00 00              exclen
      00 01 00 0D 00 00 00 06 00 01 00 00 0 0C   n name len nLineNumbers {pc line}
```

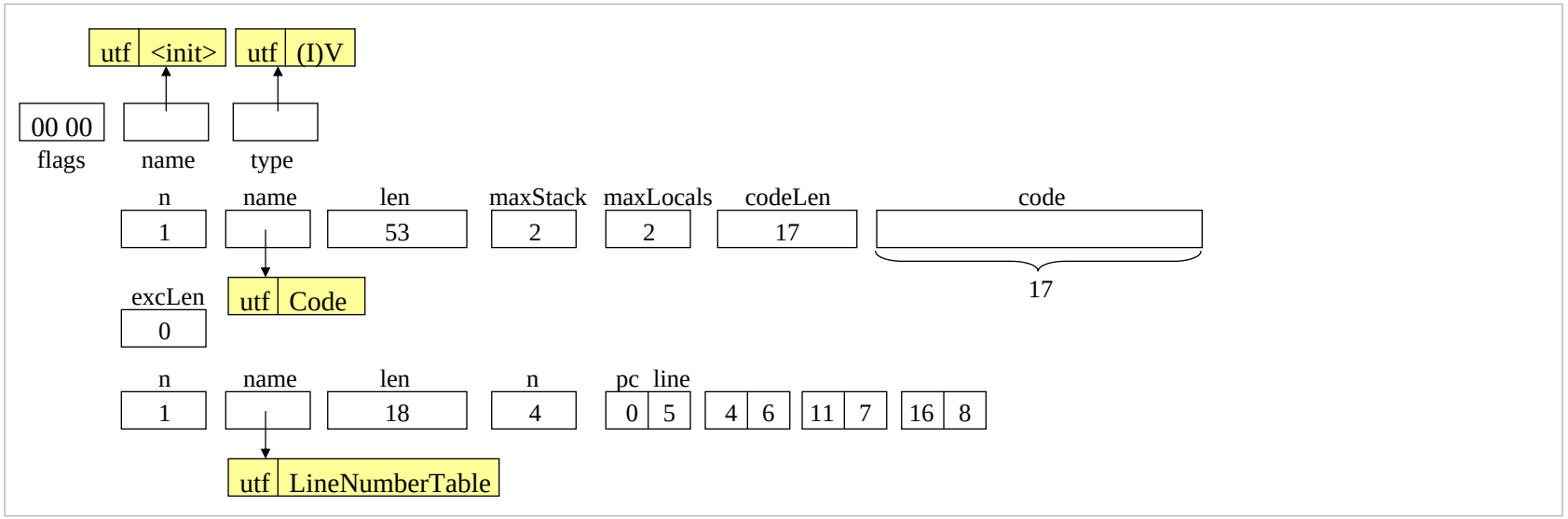
Contents of the class file (3)

00 01	number of attributes
00 13 00 00 00 02 00 14	name len sourceFileName

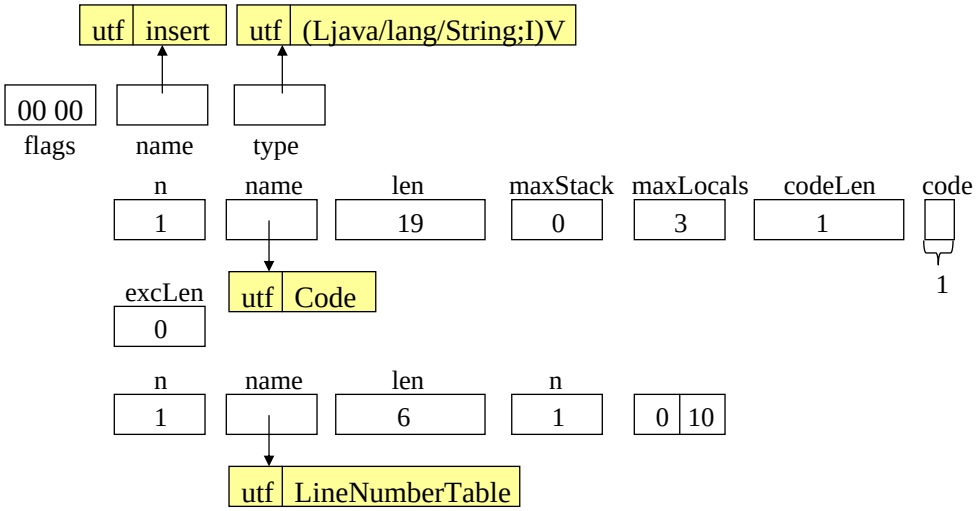
Visualization of the class file (1)



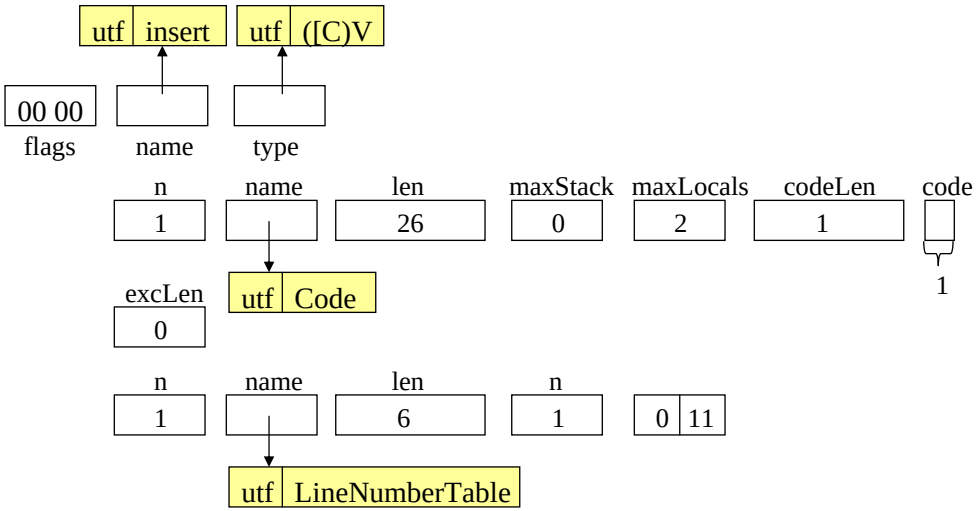




Visualization of the class file (2)

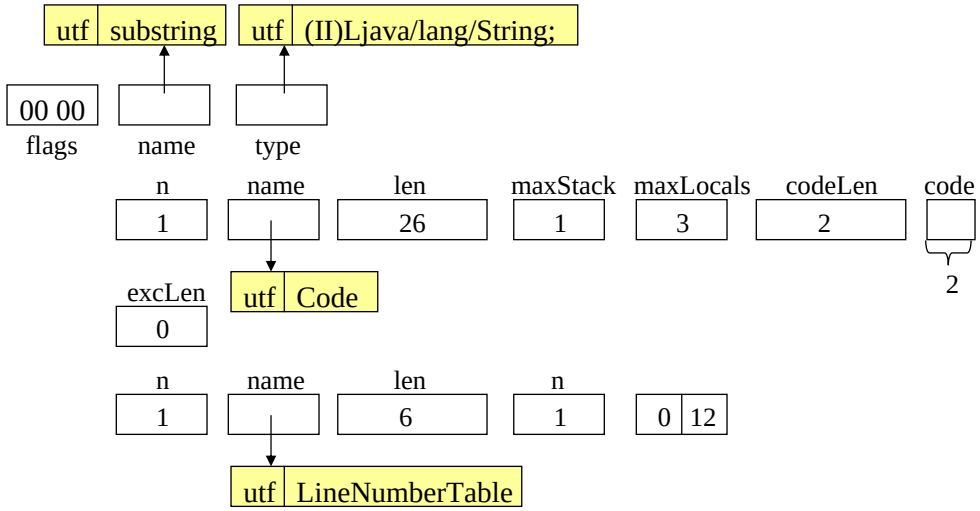


void insert(String s, int pos) {}



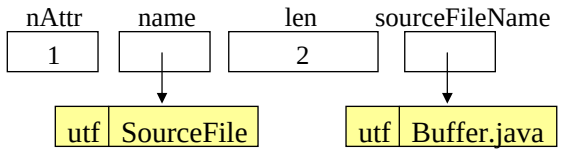
void insert(char[] a) {}

Visualization of the class file (3)




```
String substring(int from, int to) {
    return null;
}
```

File name

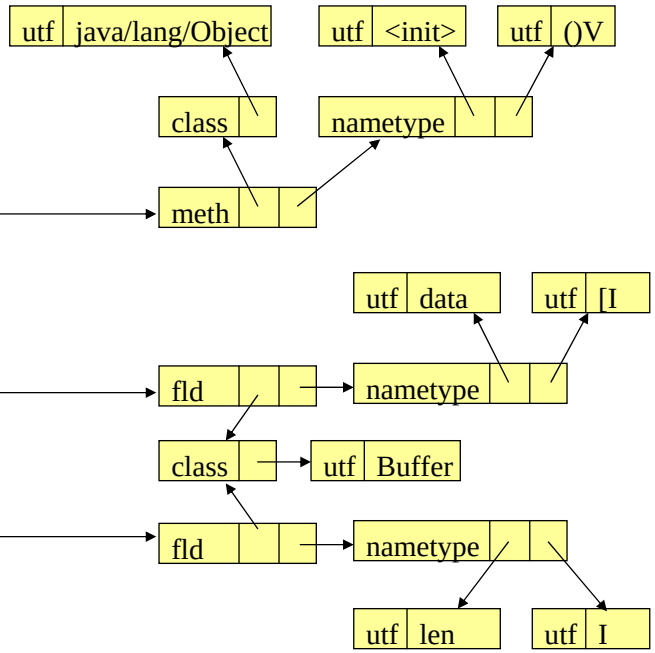


Visualization of the class file (4)

Code of the constructor

```
Buffer(int size) {
    data = new int[size];
    len = 0;
}
```

```
 0: aload_0
 1: invokespecial #1;
 4: aload_0
 5: iload_1
 6: newarray int
 8: putfield #2;
11: aload_0
12: iconst_0
13: putfield #3;
16: return
```



3. Linkers and Loaders

Loading phases

Not strictly sequential
Executed *as late as possible*!

Loading	load the class file and build the class data structure	done immediately when *loadClass* is invoked
Verification	check the consistency of the bytecodes	done when an instance of this type is referenced in the bytecodes for the first time
Initialization	initialize static fields and call the static constructor	done at the "first active use" of this type (e.g. `new T();`)
Resolution	replace symbolic references by direct references	done when the bytecode containing the symbolic reference is executed for the first time

3. Linkers and Loaders

Load phase

Tasks

1. Read the binary data stream from a class file

```
byte[] data = loadClassData(fullyQualifiedTypeName);
```

Ways to implement this:
- read a class file from the disk
- extract class from a JAR archive
- receive a class description over the network (e.g. for applets)
- generate class description "on the fly"
- ...

2. Parse the data stream and build internal data structures

```
Class c = defineClass(name, data, 0, data.length);
```

Creates a class object on the heap and connect it to the internal data structures

Implementation of the Load phase (1)

class ClassLoader


```
synchronized Class loadClass (String name) throws ClassNotFoundExeption {
    Class c = findLoadedClass(name);                    // already loaded?
    if (c == null) {
        try {
            c = getParent().loadClass(name);            // ask the parent loader
        } catch (ClassNotFoundException e) {
            c = findClass(name);                        // load the type yourself
        }
    }
    return c;
}
```

- Already loaded classes are not loaded again
- Before a loader loads a class, it gives the parent loader a chance to load it

```
Class findClass (String name) throws ClassNotFoundException {
    byte[] data = loadClassData(name);  // can throw ClassNotFoundException
    return defineClass(name, data, 0, data.length);
}
```

- Normally, a user-defined loader just overrides *findClass*
- *loadClassData* must be implemented by the programmer

Implementation of the Load phase (2)

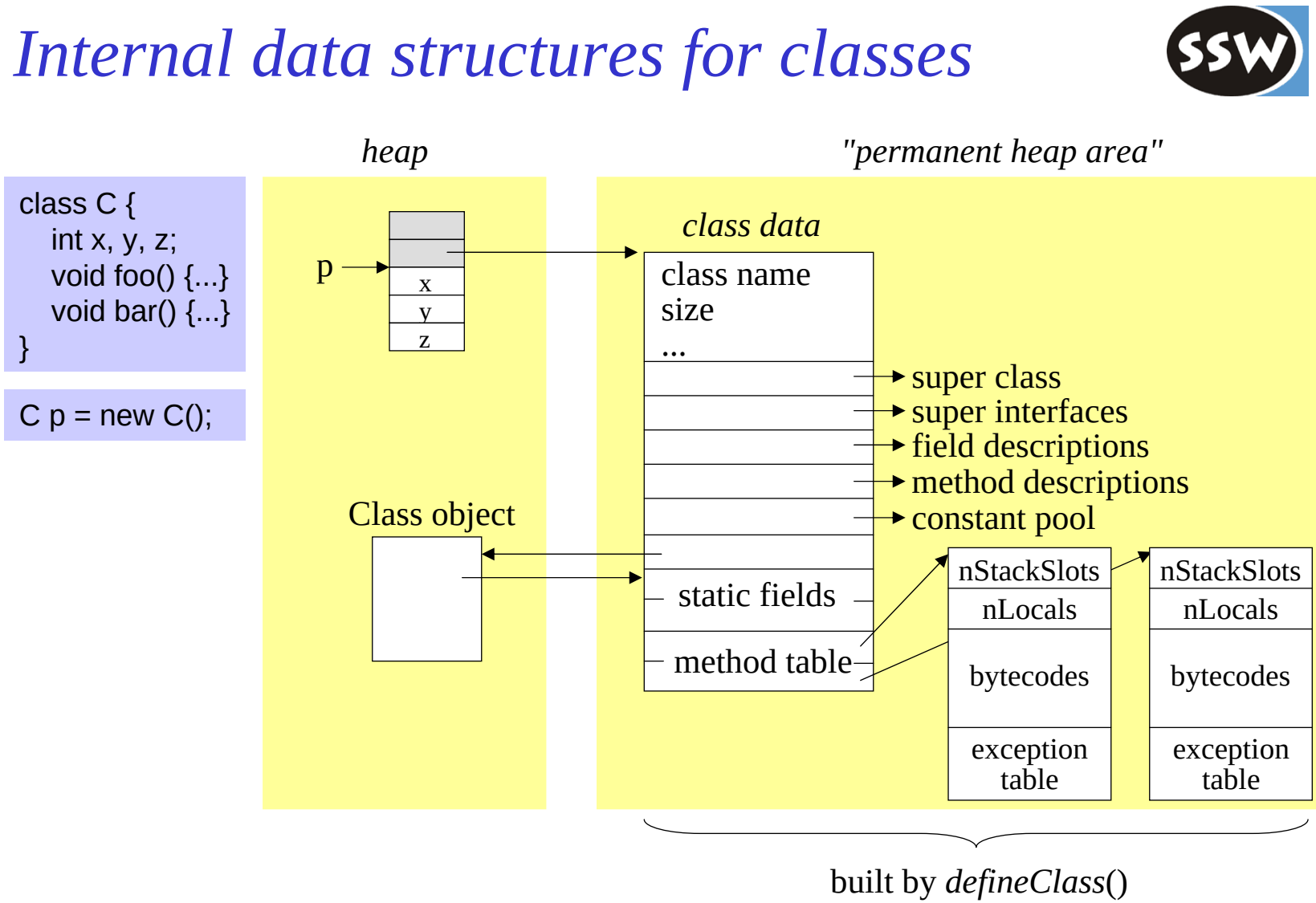
class ClassLoader


```
Class defineClass (String name, byte[] data, int pos, int len) {
   //--- parse data, build internal data structures, store them in variable cd (class data)
   ...
   if (invalid class format) throw new ClassFormatError();
   //--- load super types
   if (!name.equals("java.lang.Object"))
      cd.superClass = loadClass(cd.superClassName)    // just loaded but not
   for (all implemented interfaces i)                 // verified, initialized and resolved
      cd.interface[i] = loadClass(cd.interfaceName[i]);
   //--- create and return Class object
   Class c = new Class();
   ... // link c with loaded class data cd
   c.loader = this;
   return c;
}
```

If a type is loaded, its super types are loaded as well
(but without verification, initialization and resolution)

Super types are loaded by the same loader

Internal data structures for classes



static fields	only those that are declared in this class
method table	also the methods from the super class

3. Linkers and Loaders

Verification

Checks whether the bytecodes are correct and conform to the Java type rules

But hasn't that already been checked by the compiler?

- Not every class file was generated by the Java compiler
- Somebody could have manipulated the class file

Verification happens in different phases

Load phase

- Is the format of the class file correct?
- Does every class (except *Object*) have a super class?

Verify phase

- Are *final* classes not extended and *final* methods not overridden?
- Is method overloading unambiguous?
- Are the bytecodes of the methods correct (see later)?

Resolve phase

e.g. for: `invokevirtual #3 // Method MyClass.foo (I)V`

- Is the entry #3 in the constant pool a method entry?
- Does the class *MyClass* exist?
- Is *foo* a method of *MyClass* and does it have the specified signature?
- Does the referencing type have the necessary access rights for *foo*?

Bytecode verification

Structural checks

- Does every jump lead to the beginning of an instruction?
- Are exception handler ranges valid starts of instructions?
- Does every method end with a return instruction?

Data flow analysis by abstract interpretation of the bytecodes

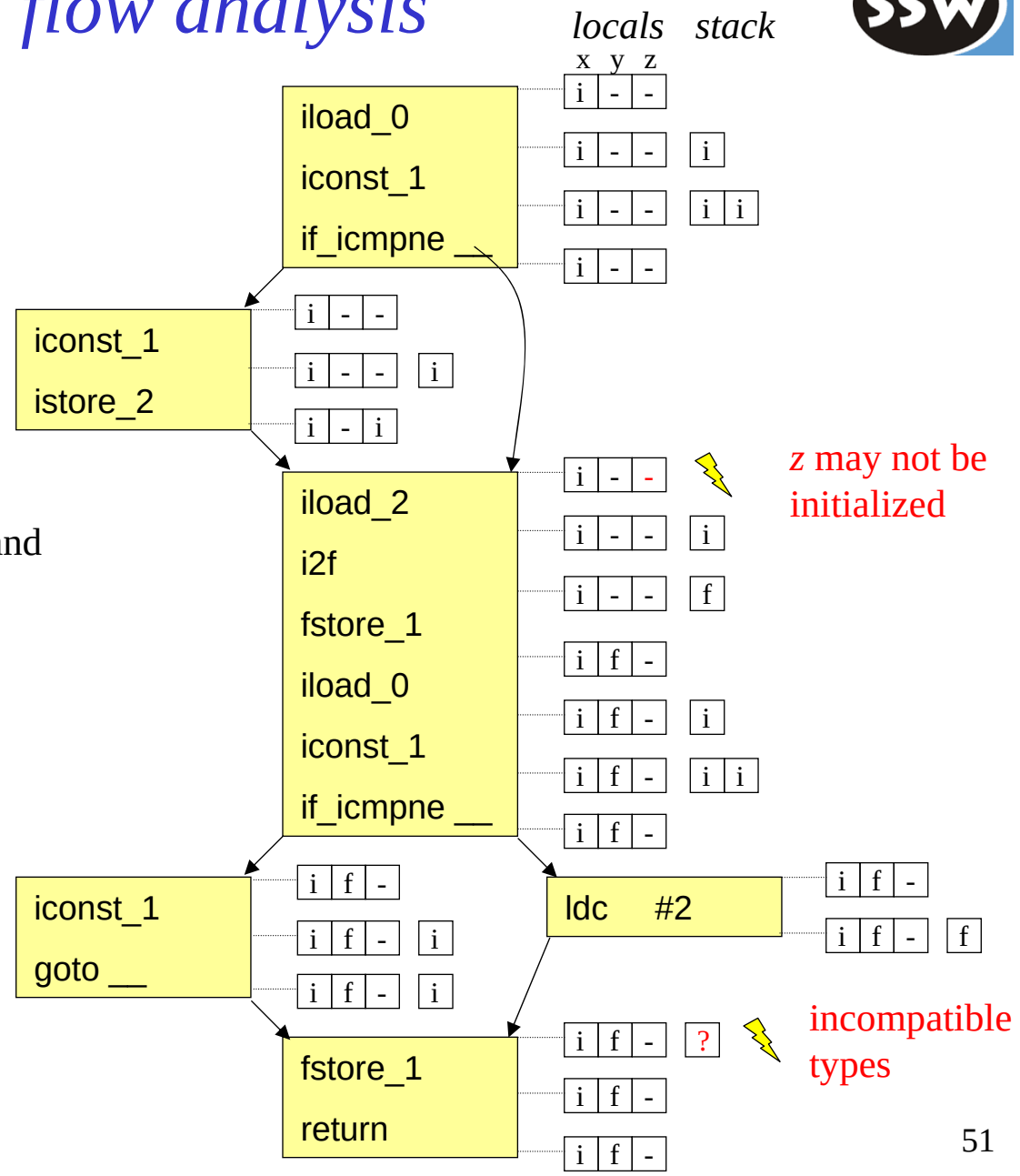
- Are all variable addresses in the range of the current stack frame?
- Is the size of the expression stack within the limits calculated by the compiler?
- Are local variables initialized before they are used?
- Do instructions access variables and stack slots with the correct type?
  (e.g.: istore_0 => stack top and variable at slot 0 must both be of type *int*)
- Are fields assigned values of the correct type?
  (e.g.: putfield #4 => stack top and field #4 must have compatible types)
- Are methods called with parameters of the correct type?
- Are local variables and the expression stack consistent, when program paths flow together?
  - do corresponding stack slots have the same type (or a common super type)?
  - do the expression stacks have the same lengths?

Example: data flow analysis

```
static void foo(int x) {
    float y;
    int z;
    if (x == 1) z = 1;
    y = z;
    y = (x == 1) ? 1 : 3.14f;
}
```

Types of the local variables and stack slots are traced

i ...	int, byte, short, char
f ...	float
- ...	undefined



3. Linkers and Loaders

When is a type initialized?

Types are initialized at their first active use (by the interpreter)

For classes

- When an object of this class is created
- When a static method of this class is called
- When a static field of this class is accessed (except for static final)
- When the class is accessed via reflection
- When one of its subclasses is initialized

For interfaces

- When a constant field of this interface is accessed

```
interface I {
    String s = new String("abc");  // static final field
}
```

- When the interface is accessed via reflection

What happens?

The static constructor of this type is called (<clinit>) if it exists

Initialization of static fields

Initialization code is copied to the static constructor

```
class A {
   static final  int x = 1;
   static        int y = 2;
   static        int z;
   static        String s = "Hello";

   static {
      z = 3;
   }
}
```

```
<clinit>()
   iconst_2
   putstatic #2   // field y
   ldc #3         // string "Hello"
   putstatic #4   // field s
   iconst_3
   putstatic #5   // field z
   return
```

} inserted

static final constants are not initialized by code, but they are inlined at the place where they are used.

Invocation of the initialization code

At the first active use of the type

```
void initialize (Class c) {
    if (! c.initialized) {
        initialize(c.superClass());
        c.<clinit>();
        c.initialized = true;
    }
}
```

- Every class is initialized exactly once
- Before a class is initialized, its superclasses are initialized
- Superinterfaces are not automatically initialized, but only at their first active use (e.g. when a static final field of this interface is accessed)

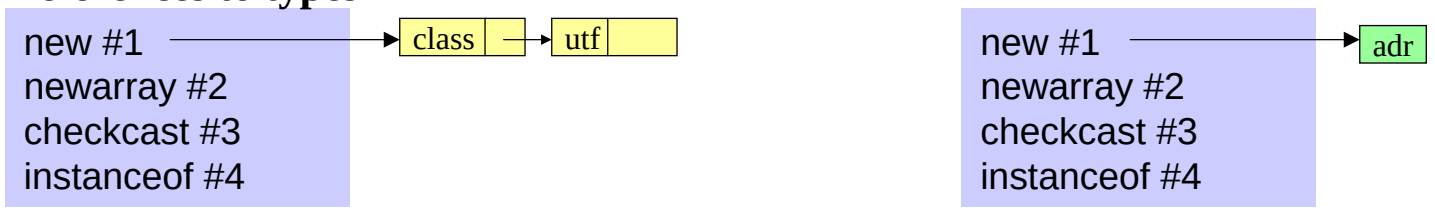
3. Linkers and Loaders

Resolving symbolic references

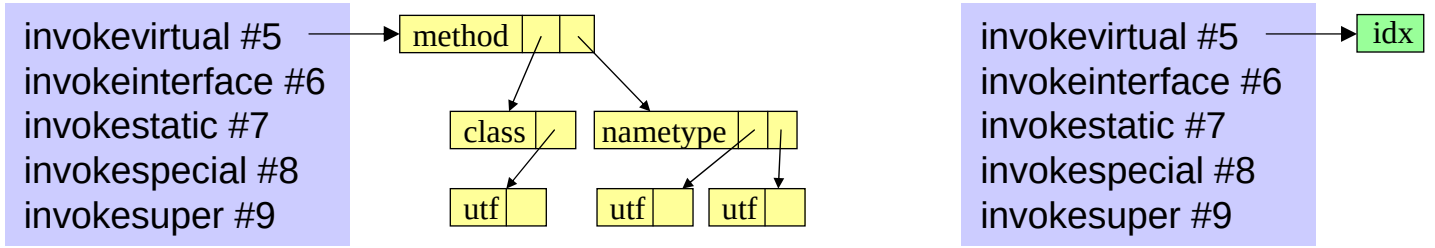
Symbolic references

Resolved references

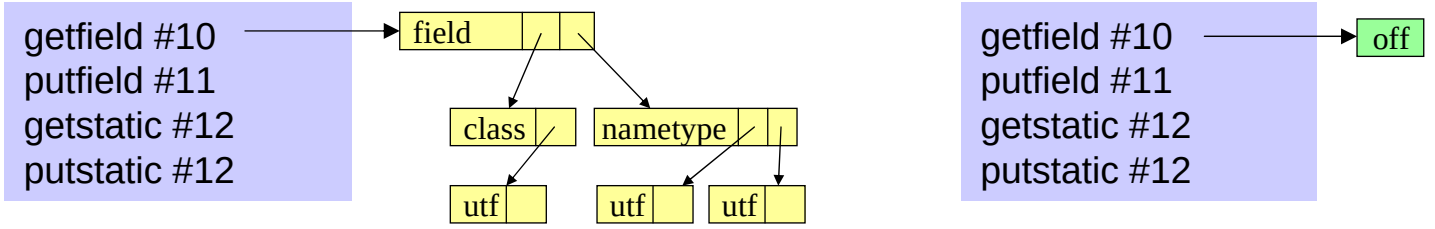
References to types



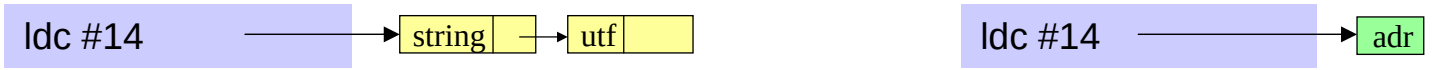
References to methods



References to fields

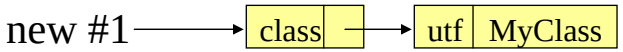


References to constants



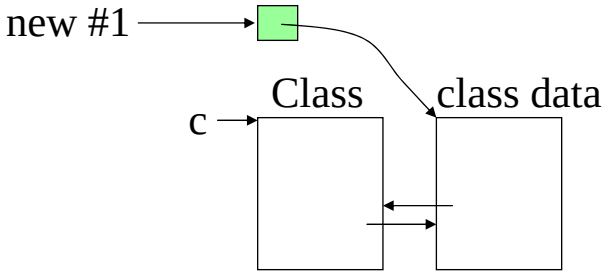
Resolving references to classes (1)

```
void resolveClassRef (int index, Class referrer) {
    if (!resolved) {
        String name = className(index);
        if (name starts with "[")
            resolveArrayClass(name);
        else
            resolveNonArrayClass(name, referrer);
    }
}
```



References to non-array classes

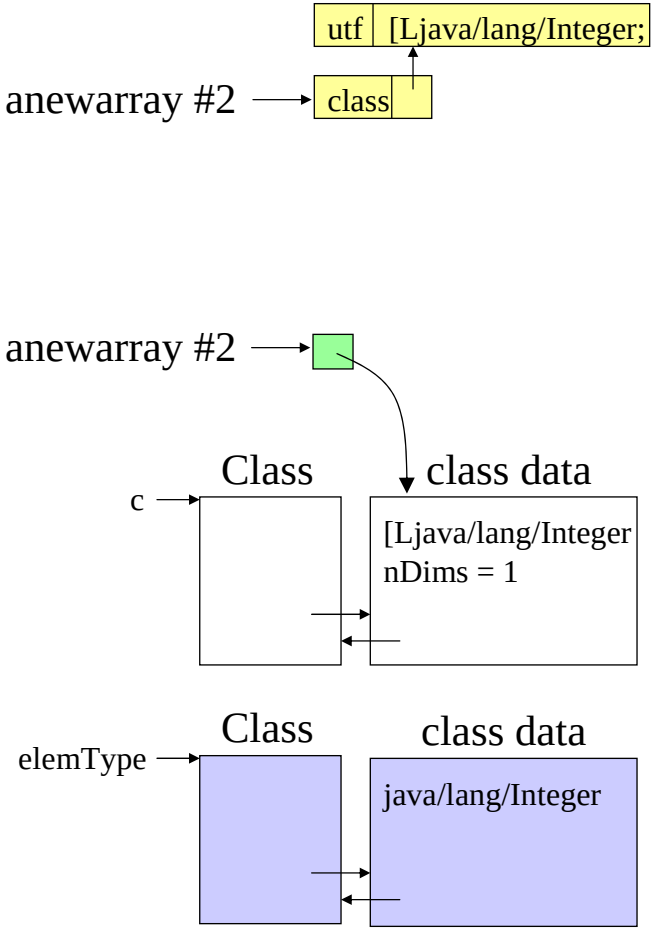
```
void resolveNonArrayClass (String name, Class referrer) {
    Class c = findLoadedClass(name);
    if (c == null) {
        c = loadClass(name); // lazy loading!!
        checkAccessPermissions(referrer, c);
        verify(c);
    }
    mark entry as resolved;
}
```



Resolving references to classes (2)

References to arrays

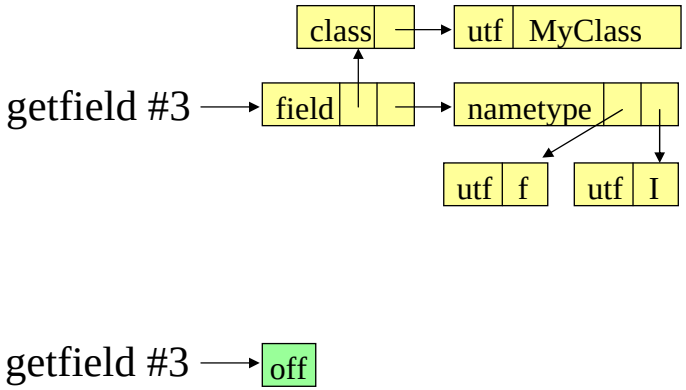
```
void resolveArrayClass(String name) {
    String elemTypeName = name of element type;
    int nDims = number of dimensions; // [[...
    Class c = findLoadedArrayType(elemTypeName, nDims);
    if (c == null) {
        if (elemTypeName is a reference type) {
            Class elemType = loadClass(elemTypeName);
            c = makeNewArrayClass(elemType, nDims);
            c.loader = this;
        } else { // primitive type
            c = makeNewArrayClass(elemTypeName, nDims);
            c.loader = bootstrapLoader;
        }
    }
    mark entry as resolved and store class pointer;
}
```



Array classes are not loaded, but created "on demand"
(but only once per element type and number of dimensions)

Resolving references to fields

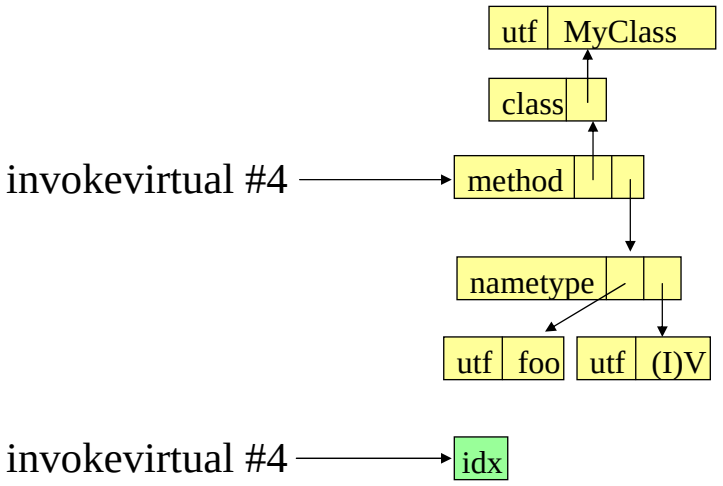
```
void resolveFieldRef (int index) {
    if (!resolved) {
        get className and fieldName from constant pool;
        Class c = loadClass(className);
        Field f = findField(c, fieldName);
        if (f == null) throw new NoSuchFieldError();
        mark entry as resolved and store field offset;
    }
}
```



- The type of the field is only resolved (i.e. loaded), when it is needed.

Resolving references to methods

```
void resolveMethodRef (int index) {
   if (!resolved) {
      get className and methodName from constant pool;
      Class c = loadClass(className);
      Method m = findMethod(c, methodName);
      if (m == null) throw new NoSuchMethodError();
      mark entry as resolved and store method index;
   }
}
```



Loading of parameter types

- If the actual and the formal parameter types are the same the types are not loaded during method resolution.

  void foo(Figure f) {...}

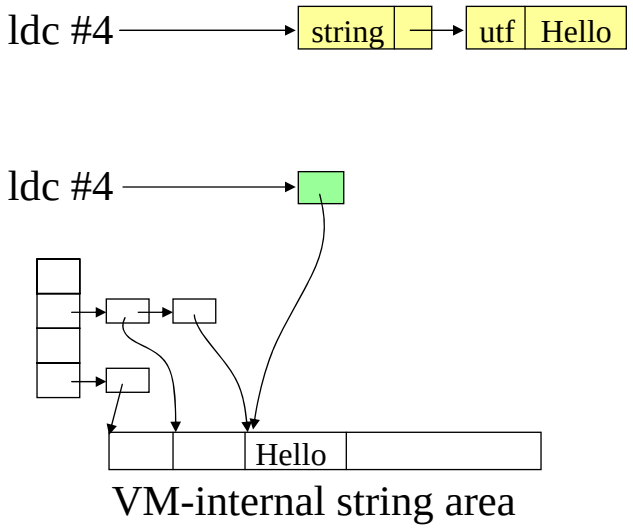
```
foo(this.figure);     aload_0          // this
                      getfield #2      // Field MyClass.figure: LFigure;
                      invokevirtual #4 // Method MyClass.foo: (LFigure;)V
```

- If the actual parameter type is a subclass of the formal parameter type both types are loaded and their compatibility is checked

```
foo(this.rect);       aload_0          // this
                      getfield #3      // Field MyClass.rect: LRectangle;
                      invokevirtual #4 // Method MyClass.foo: (LFigure;)V
```

Resolving references to strings

```
void resolveStringRef (int index) {
    if (!resolved) {
        get stringVal from constant pool;
        stringVal.intern();
        mark entry as resolved
            and store address of interned string;
    }
}
```



Strings are copied from the constant pool into an internal string area.

Example showing the effects of using an internal string area

```
void foo() {
    String s1 = In.readString(); // reads "Hello"
    String s2 = "Hello";
    if (s1 == s2) Out.println("true"); else Out.println("false");

    s1 = "Hello";
    if (s1 == s2) Out.println("true"); else Out.println("false");
}
```

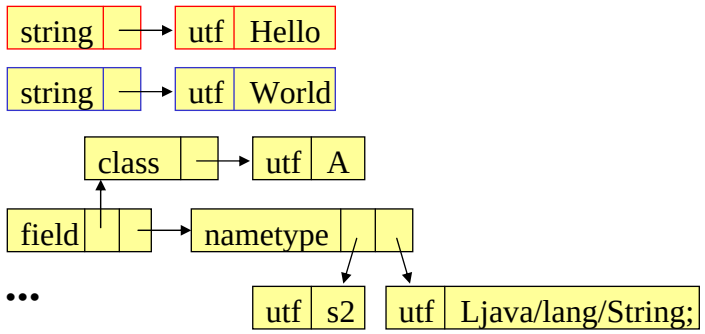
output: false
(*s1* not in the internal string area)

output: true
(*s1* and *s2* point to the same entry in the internal string area)

Imported constants

```
public class A {
    public static final   String s1 = "Hello";
    public static         String s2 = "World";
}
```

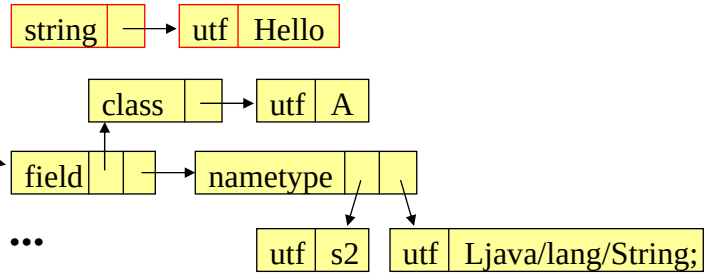
Constant pool of A




```
public class B {
    void foo() {
        String s;
        s = A.s1;
        s = A.s2;
    }
}
```

```
foo():
    ldc #2;
    astore_1
    getstatic #3
    astore_1
    return
```

Constant pool of B



- Constants (static final) are copied into the constant pool of the referencing class (in contrast to initial values of fields)
- If *A.s1* is changed and *B* is not recompiled, the constant pool of *B* contains the old value!

3. Linkers and Loaders

User-defined loader (1)

For example, a loader that loads a class from a specified path.

```
class PathLoader extends ClassLoader {
    private String path;
    private Hashtable loadedClasses = new Hashtable();  // classes loaded by this loader

    public PathLoader (String path) {
        super();  // makes the application loader the parent loader
        this.path = path;
    }

    public synchronized Class loadClass (String name) throws ClassNotFoundException {
        Class c = (Class) loadedClasses.get(name);
        if (c != null) return c;
        try {
            c = getParent().loadClass(name);
        } catch (ClassNotFoundException e) {
            byte[] data = loadClassData(name);
            c = defineClass(name, data, 0, data.length);
            loadedClasses.put(name, c);
        }
        return c;
    }
...
```

User-defined loader (2)

```
...
    private byte[] loadClassData (String name) throws ClassNotFoundException {
        try {
            FileInputStream s = new FileInputStream(path + "\\" + name + ".class");
            byte[] data = new byte[s.available()];
            s.read(data);
            return data;
        } catch (IOException e) {
            throw new ClassNotFoundException();
        }
    }
} // PathLoader
```

```
class Application {

    public static final void main (String[] arg) {
        PathLoader loader = new PathLoader("ssw\\projects");
        try {
            Class c = loader.loadClass("MyClass");
            ...
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```