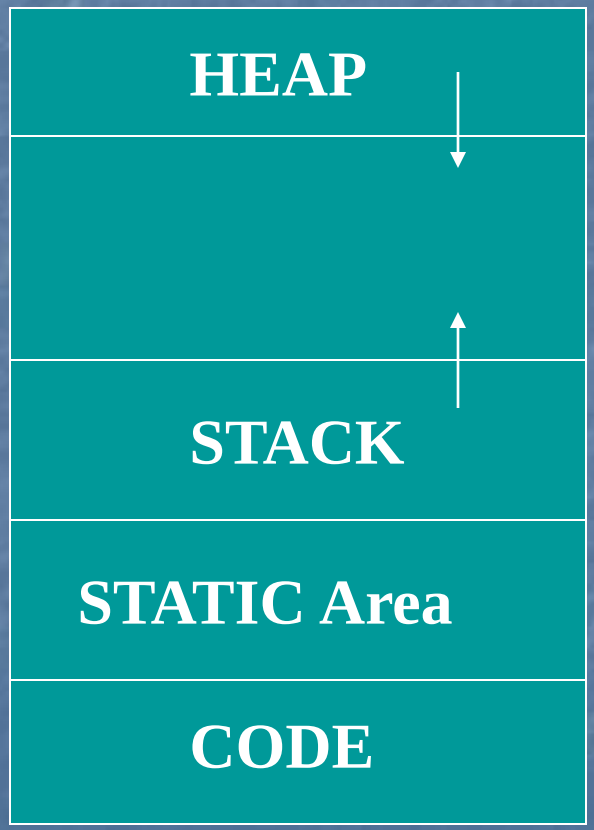
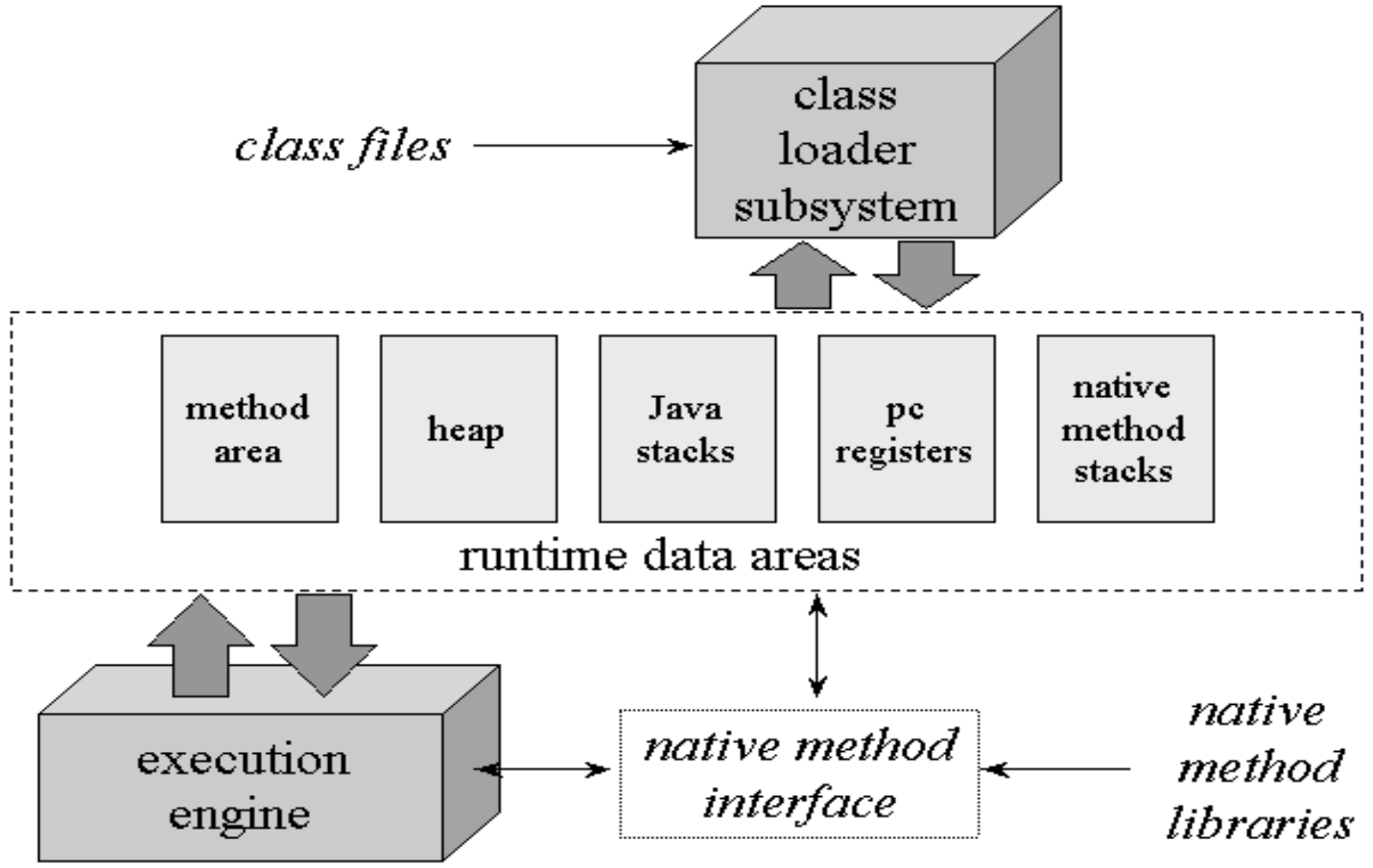
Memory Management Case study (JVM & CLR)

- Parameswari

C++ Memory Architecture Specification



JVM Architecture Specification



Method Area

Type information

constant pool

Field information

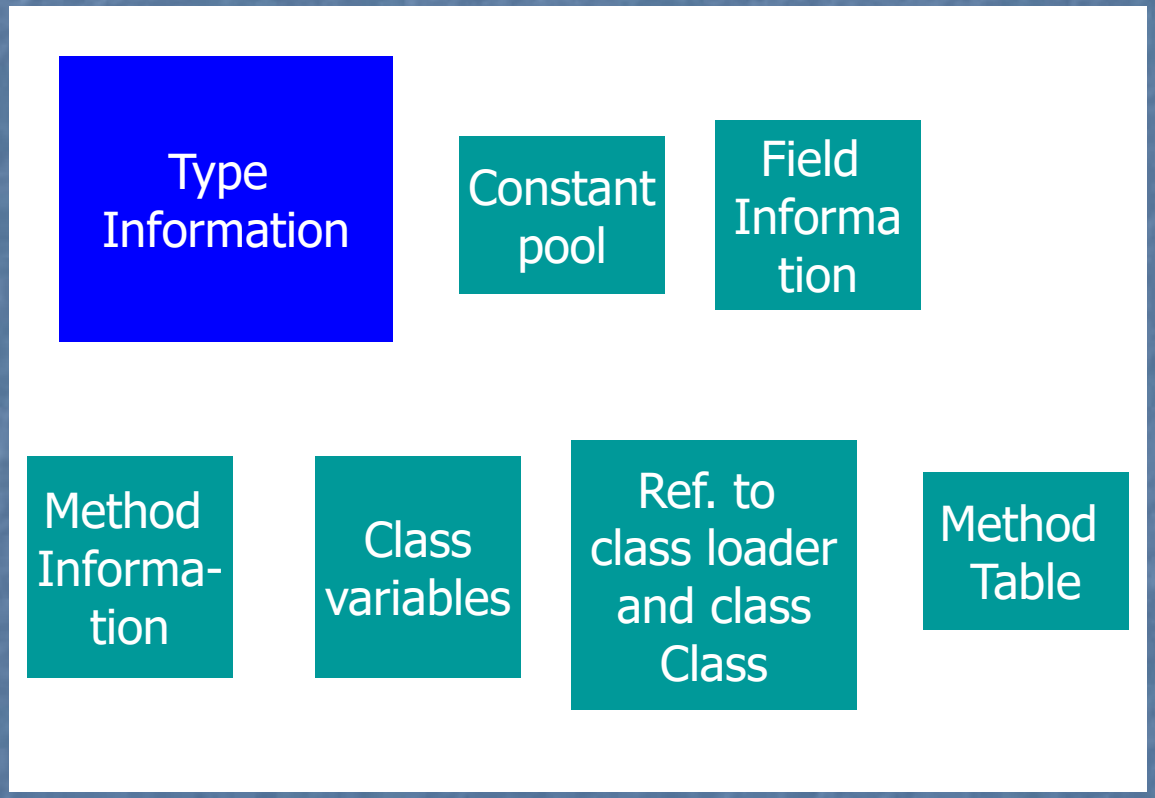
Method Table

Method information

Class variables

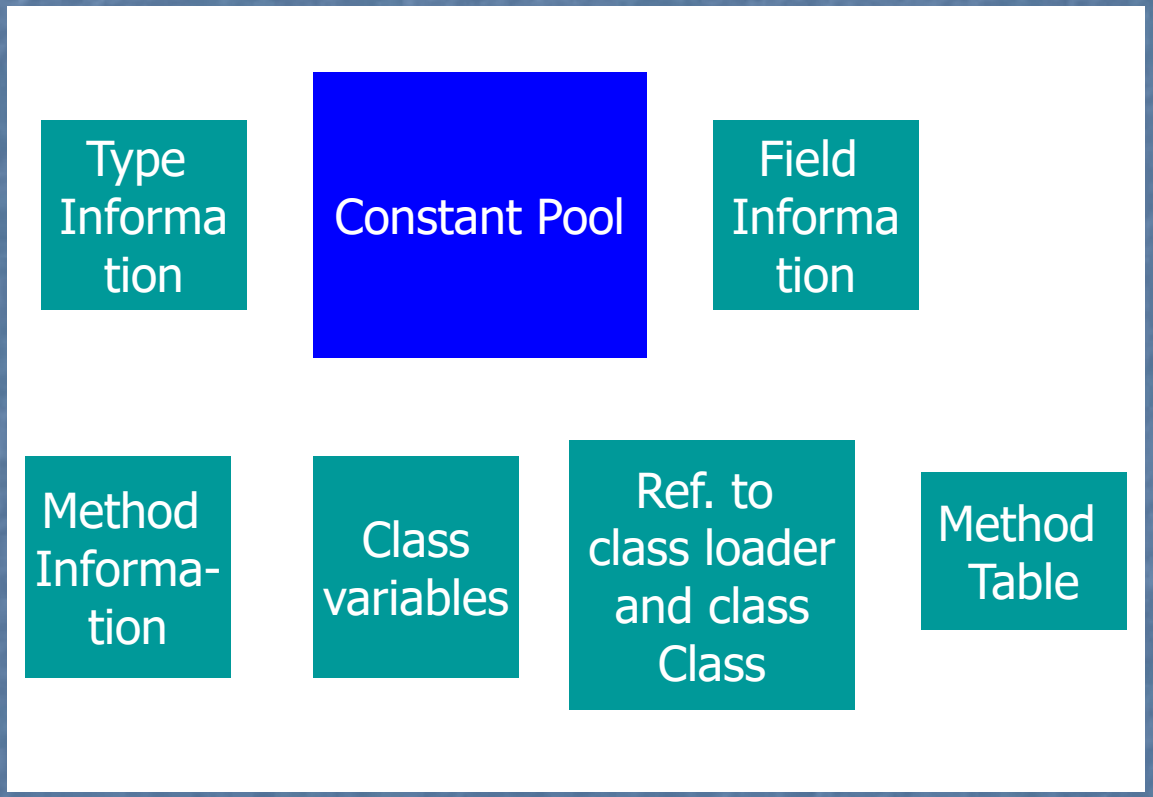
Reference to ***class loader*** and class ***Class***

Method Area



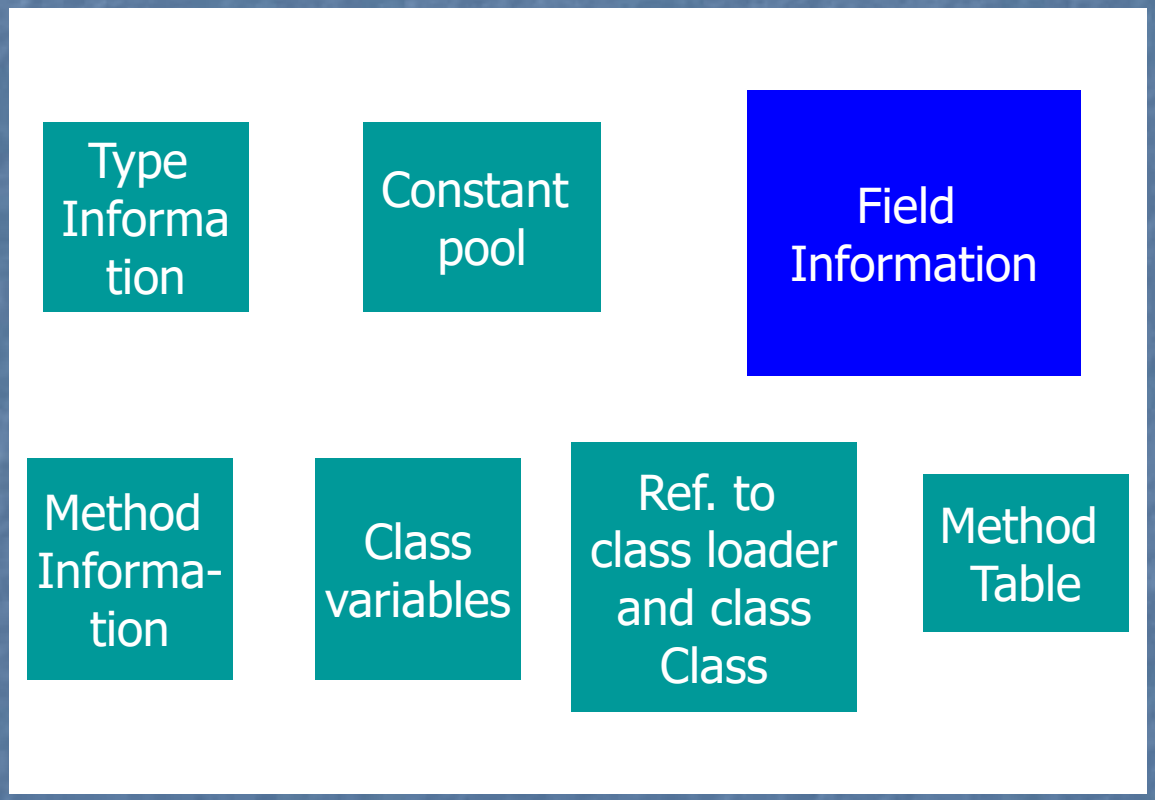
- Fully qualified type's name.
- Fully qualified direct super class name.
- Whether class or an interface
- type's modifiers
- list of the fully qualified names of any direct super interfaces

Method Area



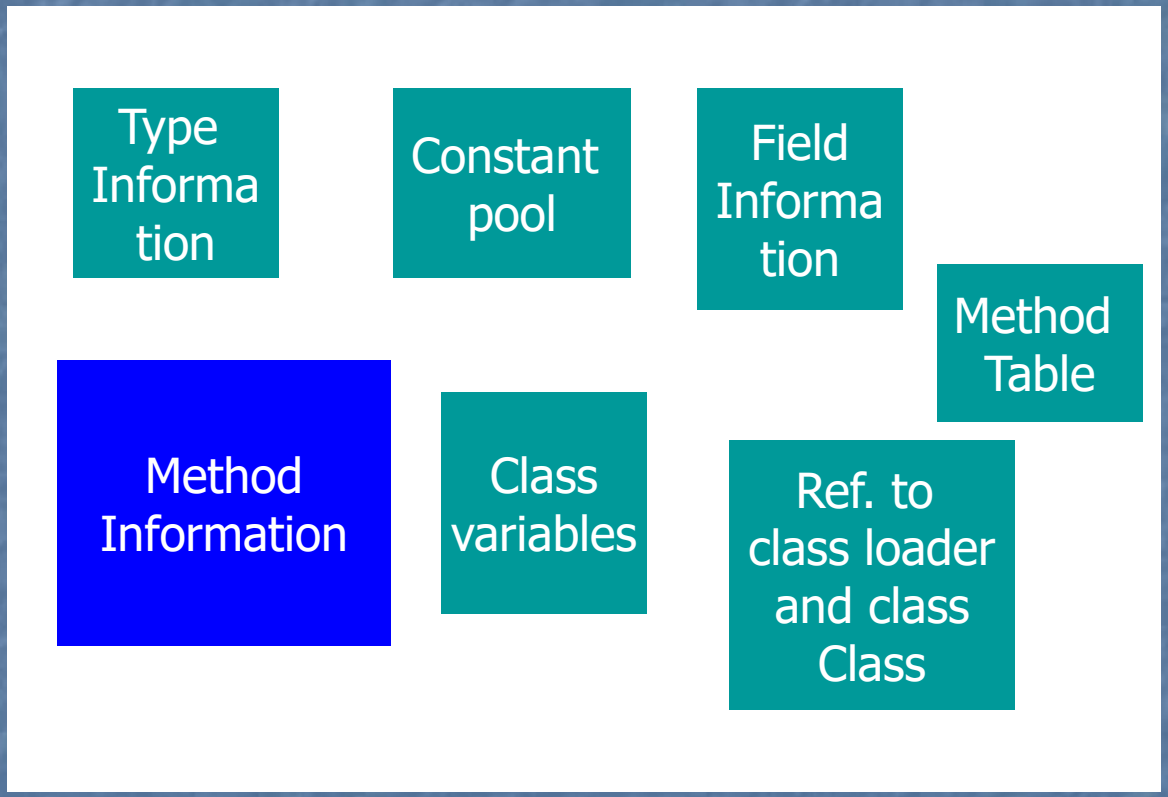
- Ordered set of constants
  - string
  - integer
  - floating point
  - final variables
- symbolic references to
  - types
  - fields
  - Methods

Method Area



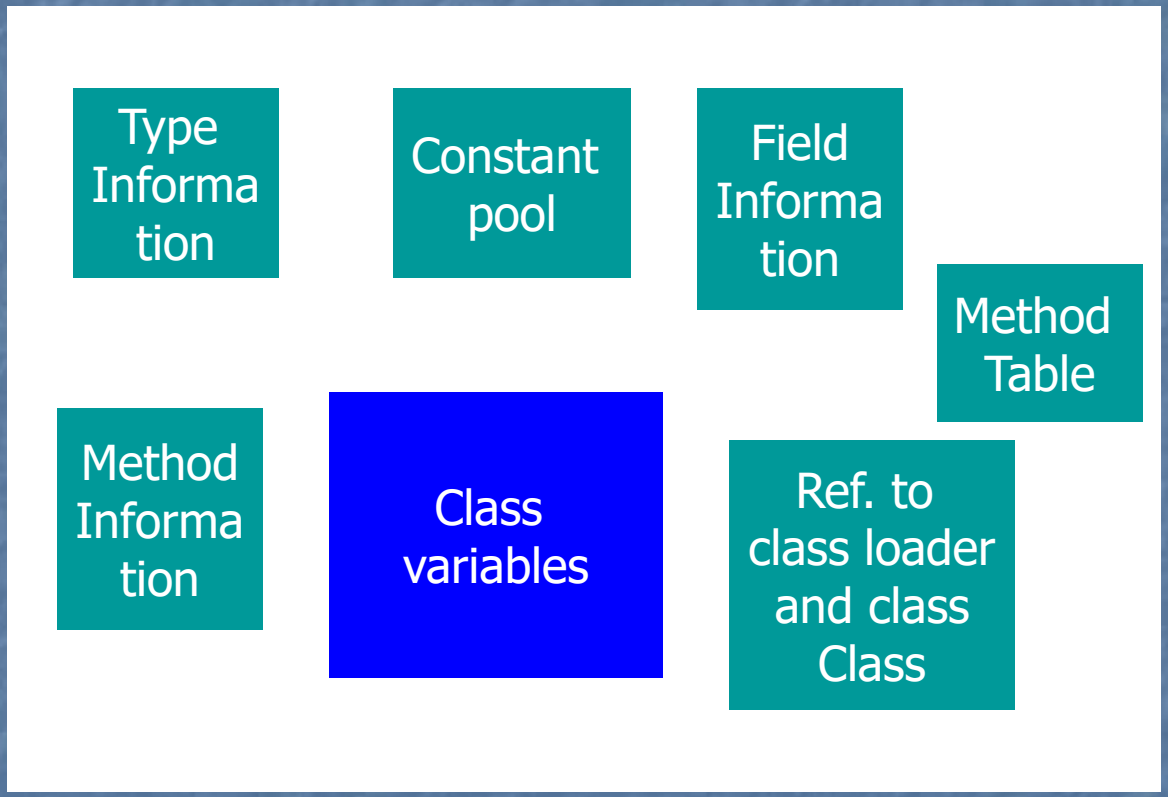
- field's name
- field's type
- field's modifiers (subset)
  - public
  - private
  - protected
  - static
  - final
  - volatile
  - transient

Method Area



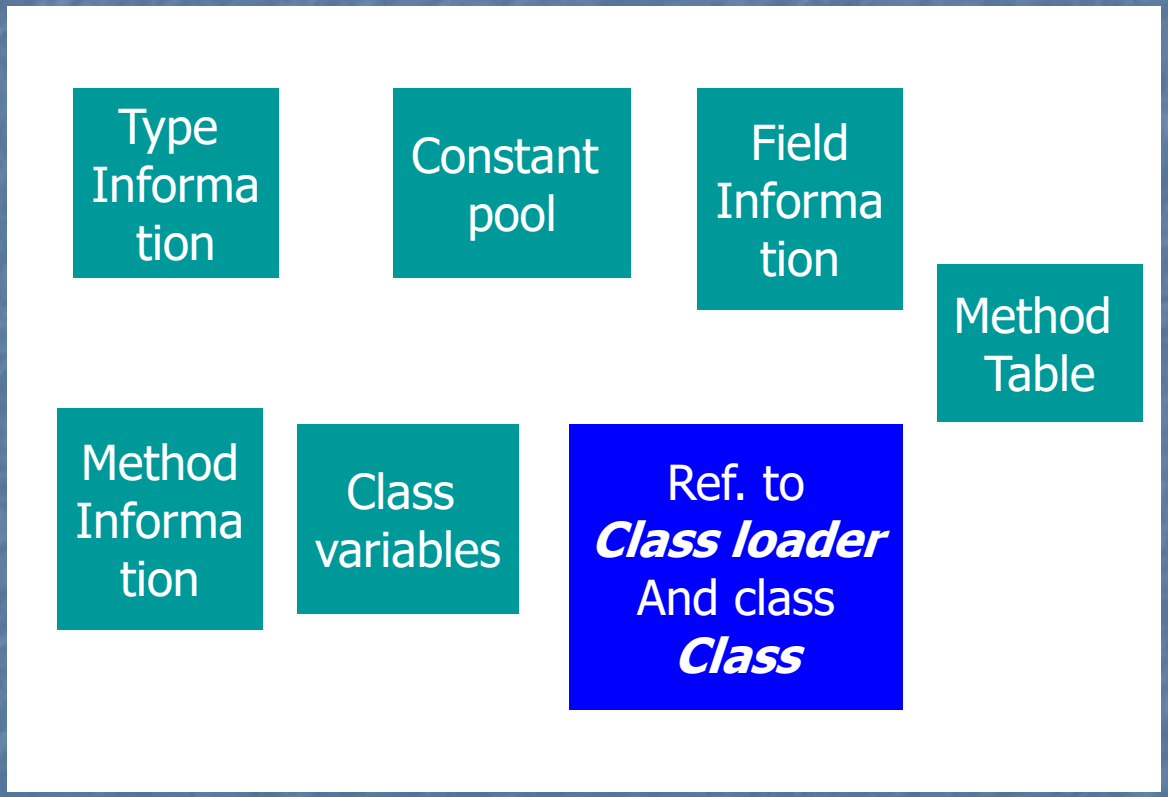
- Method's name
- Method's return type
- Number and type of parameters
- Modifiers (subset)
  - public
  - private
  - protected
  - static
  - final
  - synchronized
  - native
  - abstract

Method Area



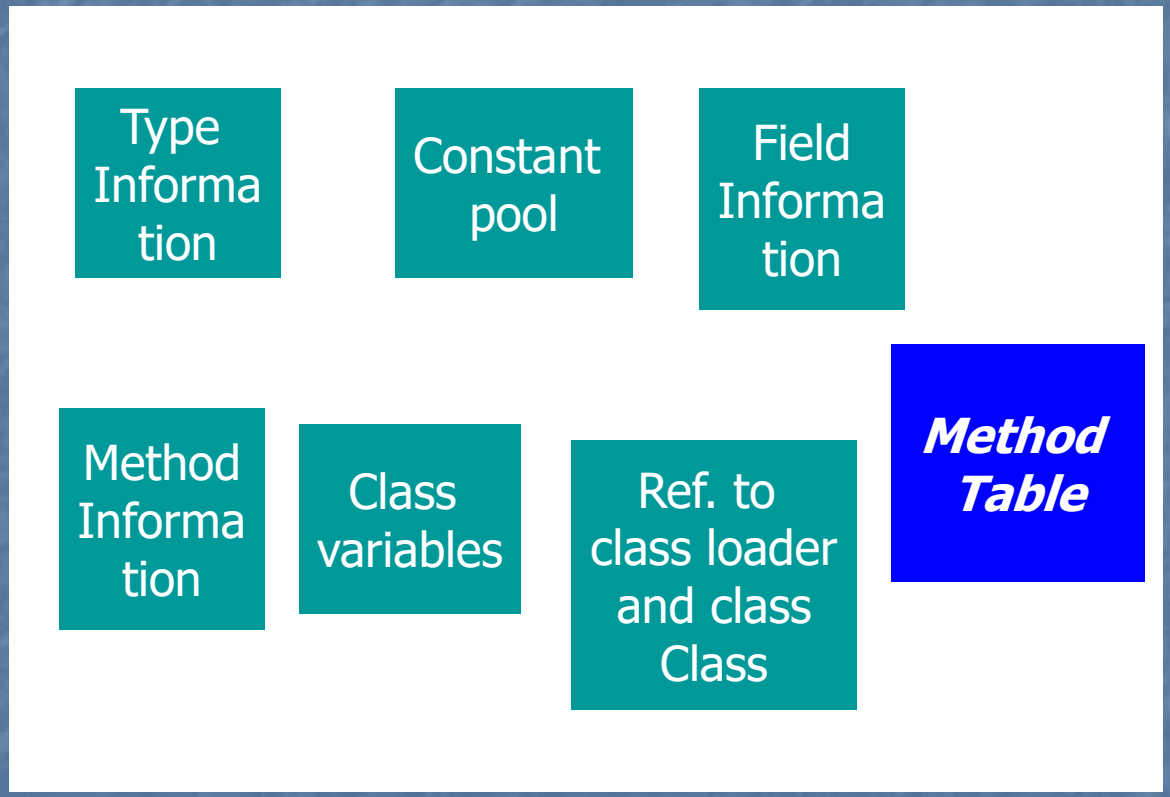
- ordered set of class variables
  - static variables

Method Area



- Reference to class loader is used for dynamic linking.
- instance java.lang.Class is created every type for the following info.
  - getName();
  - getSuperClass();
  - isInterface();
  - getInterfaces();
  - getClassLoader();

Method Area

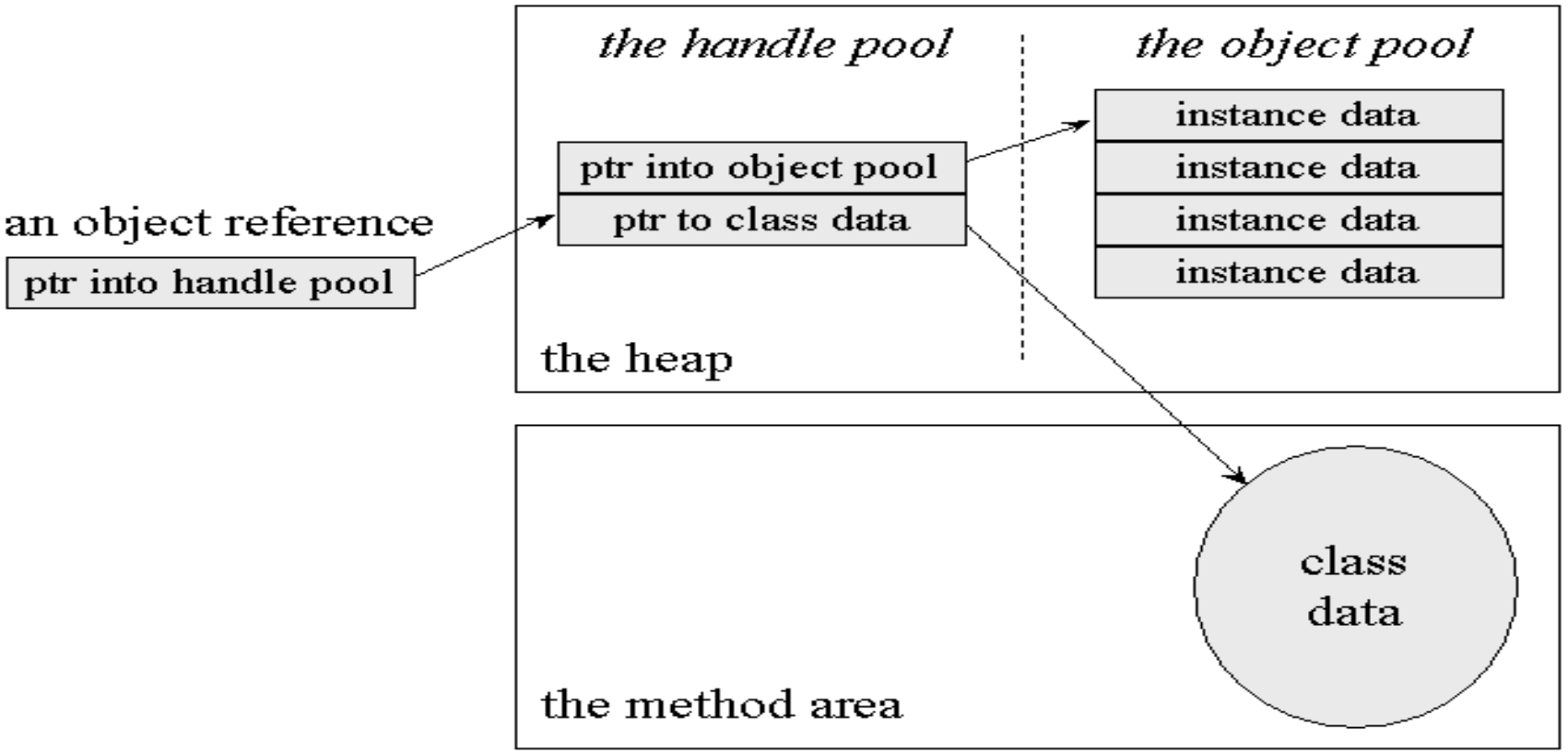


- Used for quick ref. to method.
- Contains name and index in symbol ref. array

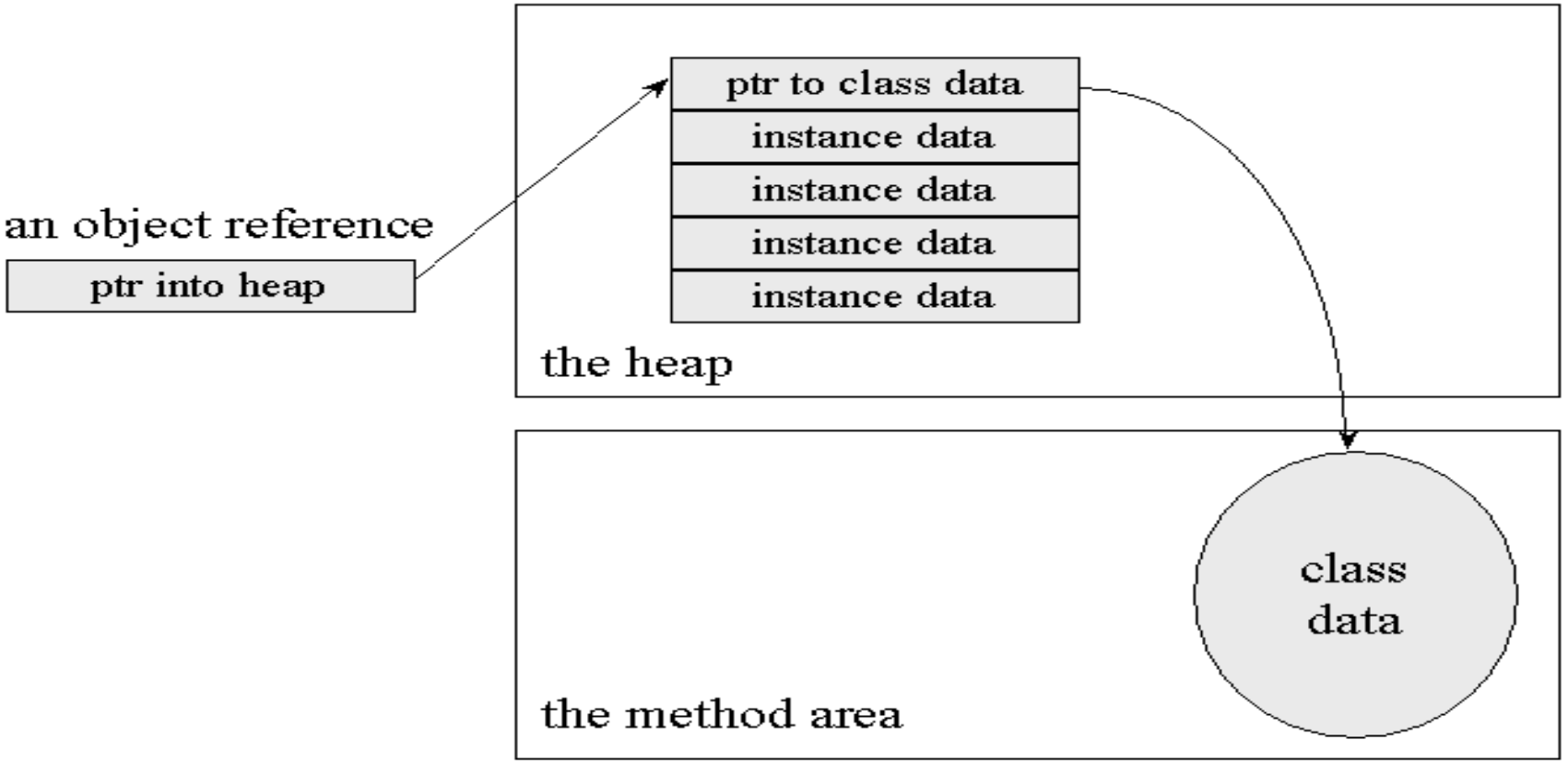
Heap Memory

- Objects and arrays are allocated in this area.
- Two different threads of the same application, however, could trample on each other's heap data.

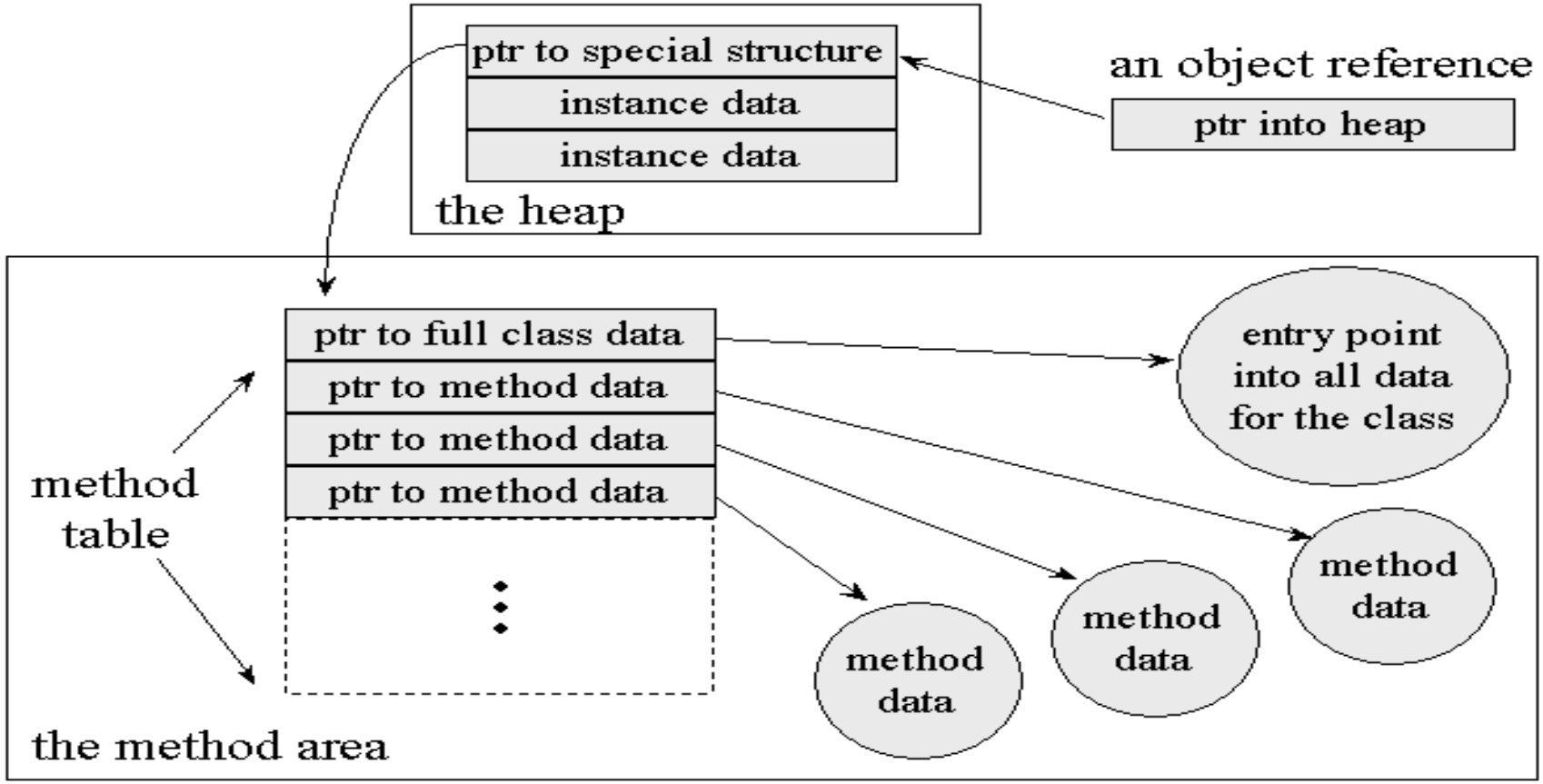
One possible Design of Object Allocation on Heap



Another Design of Object Allocation



Another Design of Object Allocation



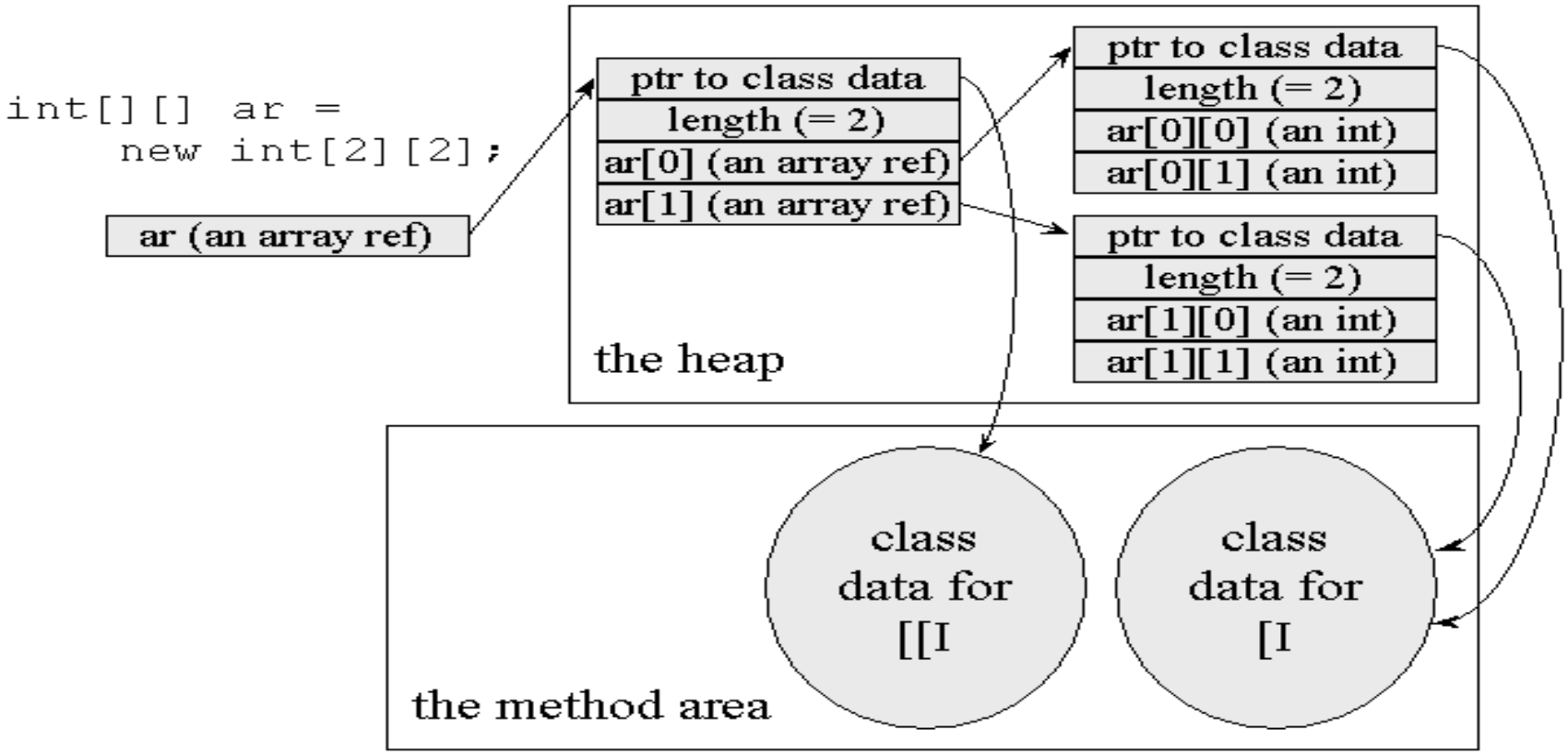
Lock on Objects

- object is associated with a lock (or mutex) to coordinate multi-threaded access to the object.
- Only one thread at a time can "own" an object's lock.
- Once a thread owns a lock, it can request the same lock again multiple times, but then has to release the lock the same number of times before it is made available to other threads.

Array Allocation on Heap

- The name of an array's class has one open square bracket for each dimension plus a letter or string representing the array's type.
- The class name for an array of ints is "[I".
- The class name for three-dimensional array of bytes is "[[[B".
- The class name for a two-dimensional array of Objects is "[[Ljava.lang.Object".

Design of allocation of array on Heap



Java Stack

- Java stack stores a thread's state in discrete frames.
- Each frame contains
  - local variables Area.
  - operand stack
  - frame data

Java Stack

- Stack Area generates when a thread creates.
- It can be of either fixed or dynamic size.
- The stack memory is allocated per thread.
- It is used to store data and partial results.
- It contains references to heap objects.

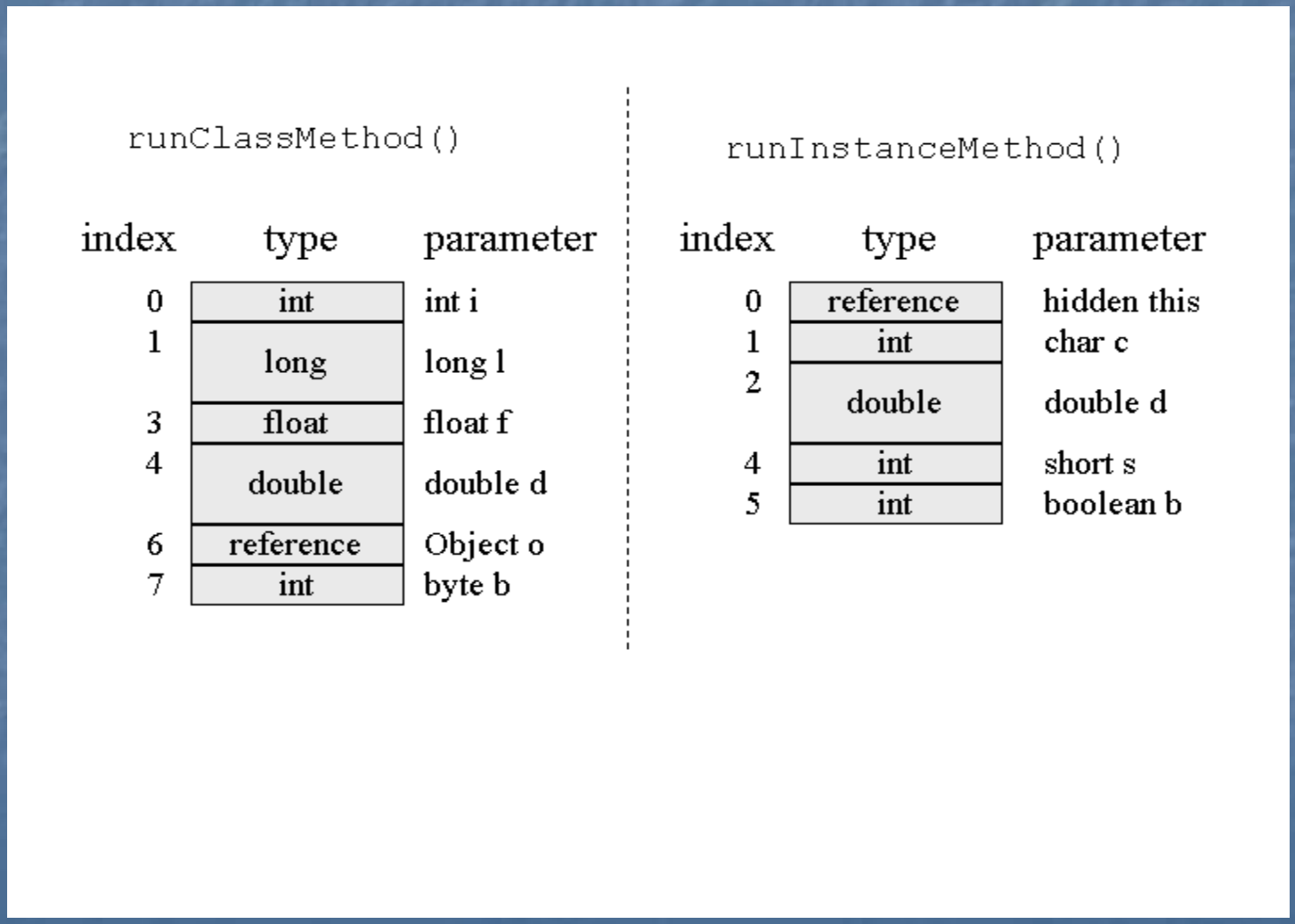
Java Stack

- It also holds the value itself rather than a reference to an object from the heap.
- The variables which are stored in the stack have certain visibility, called scope.

Local variable Area

- organized as a zero-based array of cells.
- Variables are accessed through their indices.
- Values of type int, float, reference, and return Address occupy one cell.
- Values of type byte, short, and char also occupy one cell.
- Values of type long and double occupy two consecutive cells in the array.

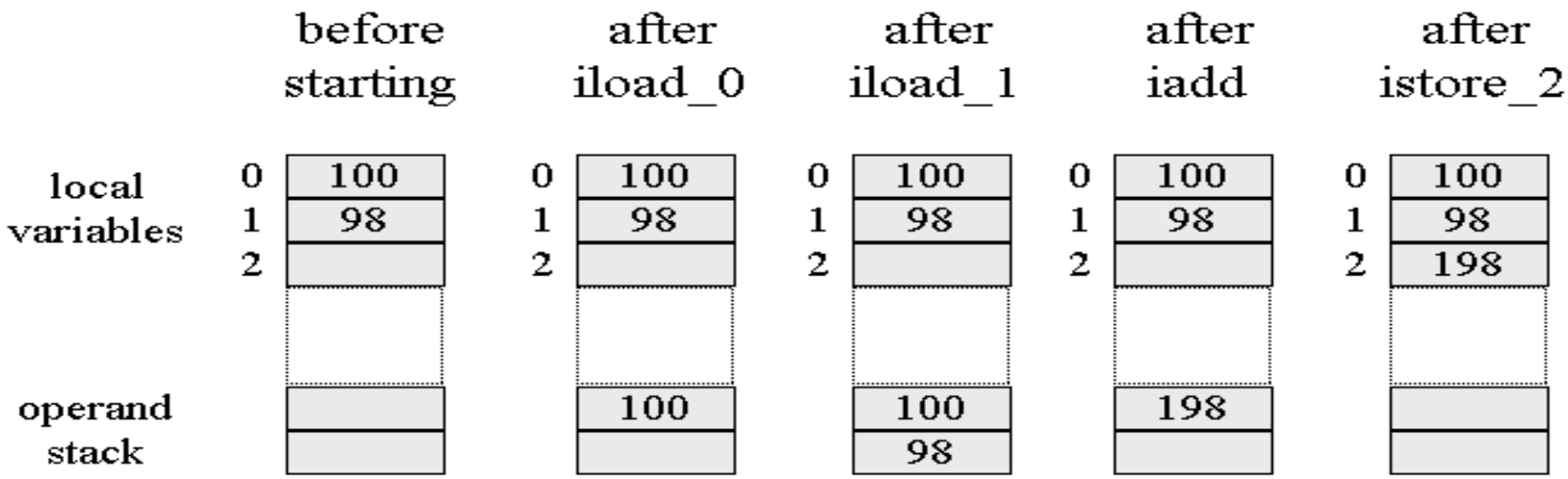
```
class Example3a {
   public static int runClassMethod(int i, long l, float f, double d, Object o, byte b) {
        return 0; }
  public int runInstanceMethod(char c, double d, short s, boolean b) {
        return 0; }
 }
```



Operand Stack

- operand stack is also organized as an array of cells.
- local variables are accessed via array indices, the operand stack is accessed by pushing and popping values.
- instructions take their operands from
  - operand stack
  - immediately following the opcode
  - constant pool

- iload_0 // push the int in local variable 0
- iload_1 // push the int in local variable 1
- iadd // pop two ints, add them, push result
- istore_2 // pop int, store into local variable 2



Stack Frame

- Stack Frame: Stack frame is a data structure that contains the thread?s data. Thread data represents the state of the thread in the current method.
- It is used to store partial results and data. It also performs dynamic linking, values return by methods and dispatch exceptions.

Stack Frame

- When a method invokes, a new frame creates. It destroys the frame when the invocation of the method completes.
- Each frame contains own Local Variable Array (LVA), Operand Stack (OS), and Frame Data (FD).

Stack Frame

- The sizes of LVA, OS, and FD determined at compile time.
- Only one frame (the frame for executing method) is active at any point in a given thread of control. This frame is called the current frame, and its method is known as the current method. The class of method is called the current class.

Stack Frame

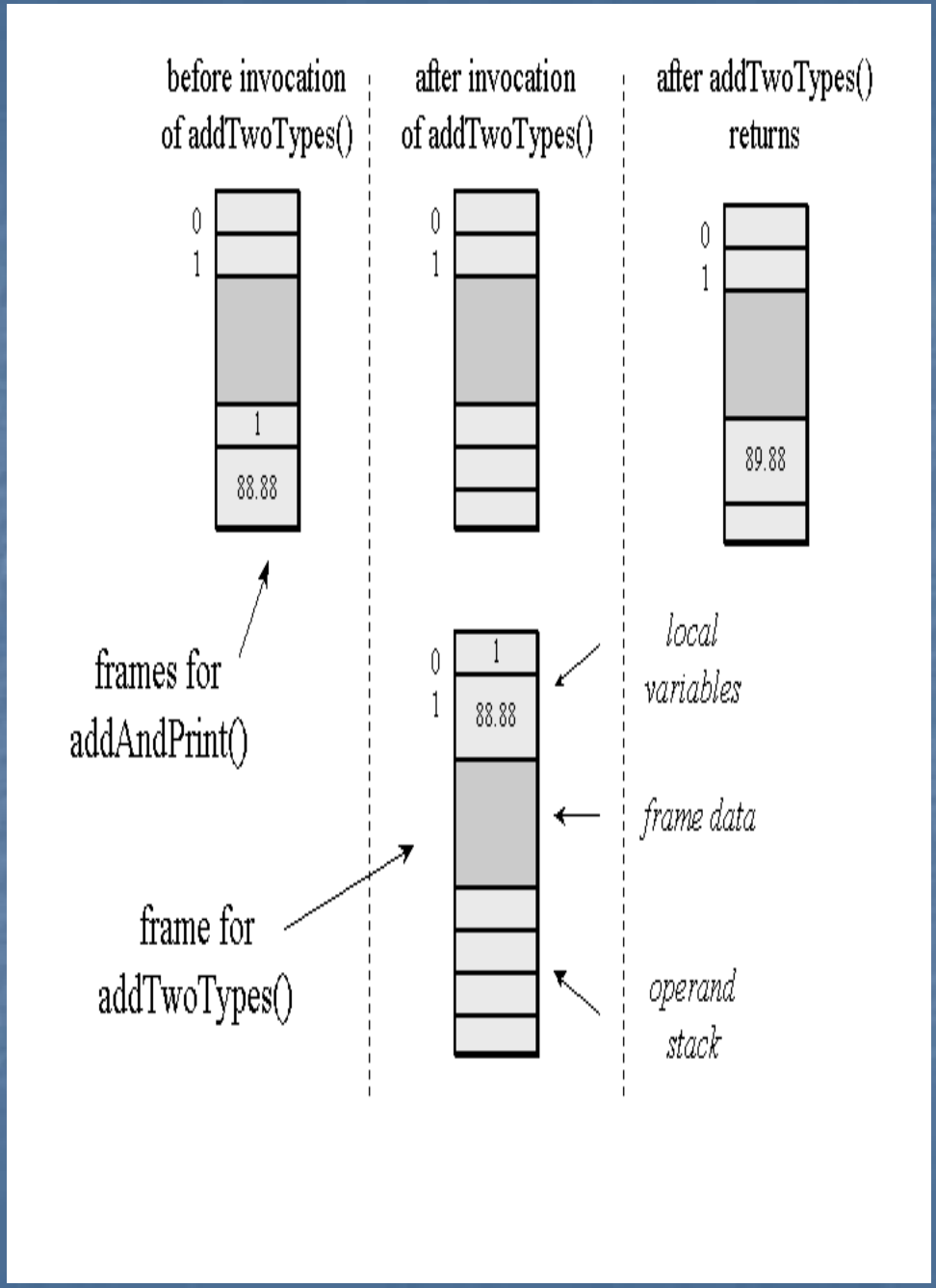
- The frame stops the current method, if its method invokes another method or if the method completes.
- The frame created by a thread is local to that thread and cannot be referenced by any other thread.

Frame data

- Frame data is needed to support
  - constant pool resolution
  - normal method return
  - exception dispatch
  - debugging.

Heap Space Memory	Stack Memory
All parts of the application invoke the heap memory.	The stack memory execution is limited to a single thread.
Anytime an object gets created, it is stored in the heap space.	The stack memory only comprises its reference and local primitive variables.
Objects here are accessible globally across the application.	Other threads cannot access stack memory objects.
Here, memory is defined according to young and old generations.	Memory management occurs on a Last-In-First-Out basis.
The memory remains as per the scope of the application.	Memory is temporary.
The methods like – XMX and XMS JVM are used to define the optimal size of the heap memory.	For stack memory, it gets determined by the -XSS method.
Here, the exception of *java.lang.OutOfMemoryError* occurs in the case of full memory.	Here, the error *java.lang.StackOverFlowError* happens in case the memory is full.
The size is more but takes time to process compared to the stack memory.	The size is lesser but faster in execution for its smooth LIFO operation.

```
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;
    }
}
```



Example

```
class abc  {
  public int a;
 String str;
 abc() {
   a=10;   atr="string1";
 }
 public void print{   System.out.print(a+"  "+str);   }
}
interface def  {
  void add();
}
class pqr  extends abc implements def {
 static int b;
 final int c=50;
 String s;
 pqr(int m) { super(); b=m;  s= new String("string2"); }
 void add() { a=a+c;  add1(); }
 static void add1() { c=b+50; }
}
```

```
class Main  {
 public static void main(String[] s) {
    pqr p=new pqr(20);
    p.add();
    p.print();
 }
}
```

```
class abc  {
  public int a;
 String str;
 abc() {
   a=10;
   str="string1";
 }
 public void print{
    System.out.print(a+" "+str);
  }
}
```

Type info

abc

java.lang.Object

Isclass=true

modifier=4

Constant pool

Symbol ref. array

a	str	<init>	print

10 "string1"

Field info

name	Type	Modifier	index
a	int	5	0
str	String	4	1

Method info

name	ret.type	npar	modifier	parlist	codeptr
<init>	void	0	1		
print	void	0	5		

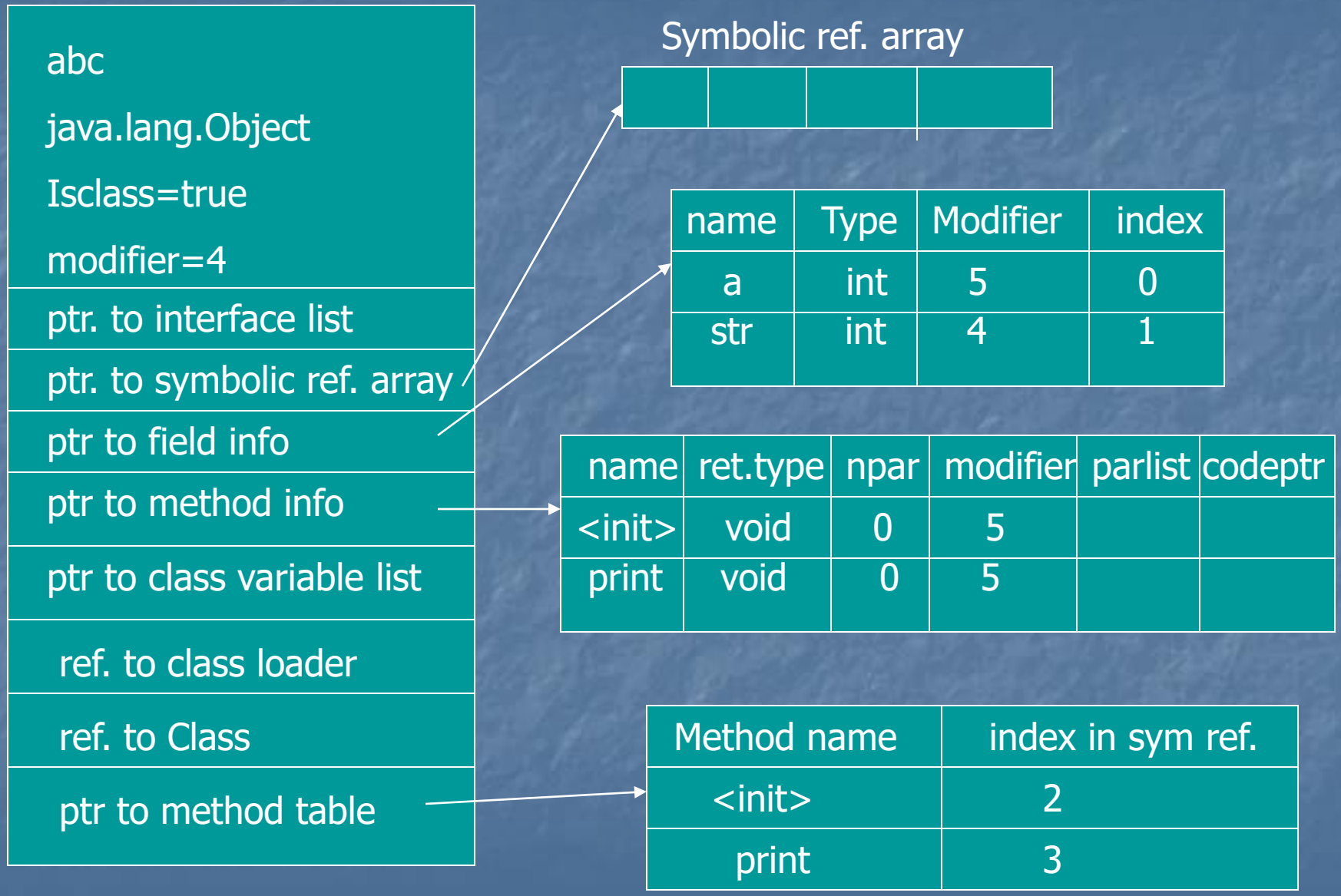
Method Table

name	index
<init>	2
print	3

Class variables

null

Class Area of abc in Method area



```
interface def
{
  void add();
}
```

Type info

def

java.lang.Object

Isclass=false

modifier=4

Constant pool

Symbol ref. array

add

Field info

null

Method info

name	ret.type	npar	modifier	parlist	codeptr
add	void	0	4		

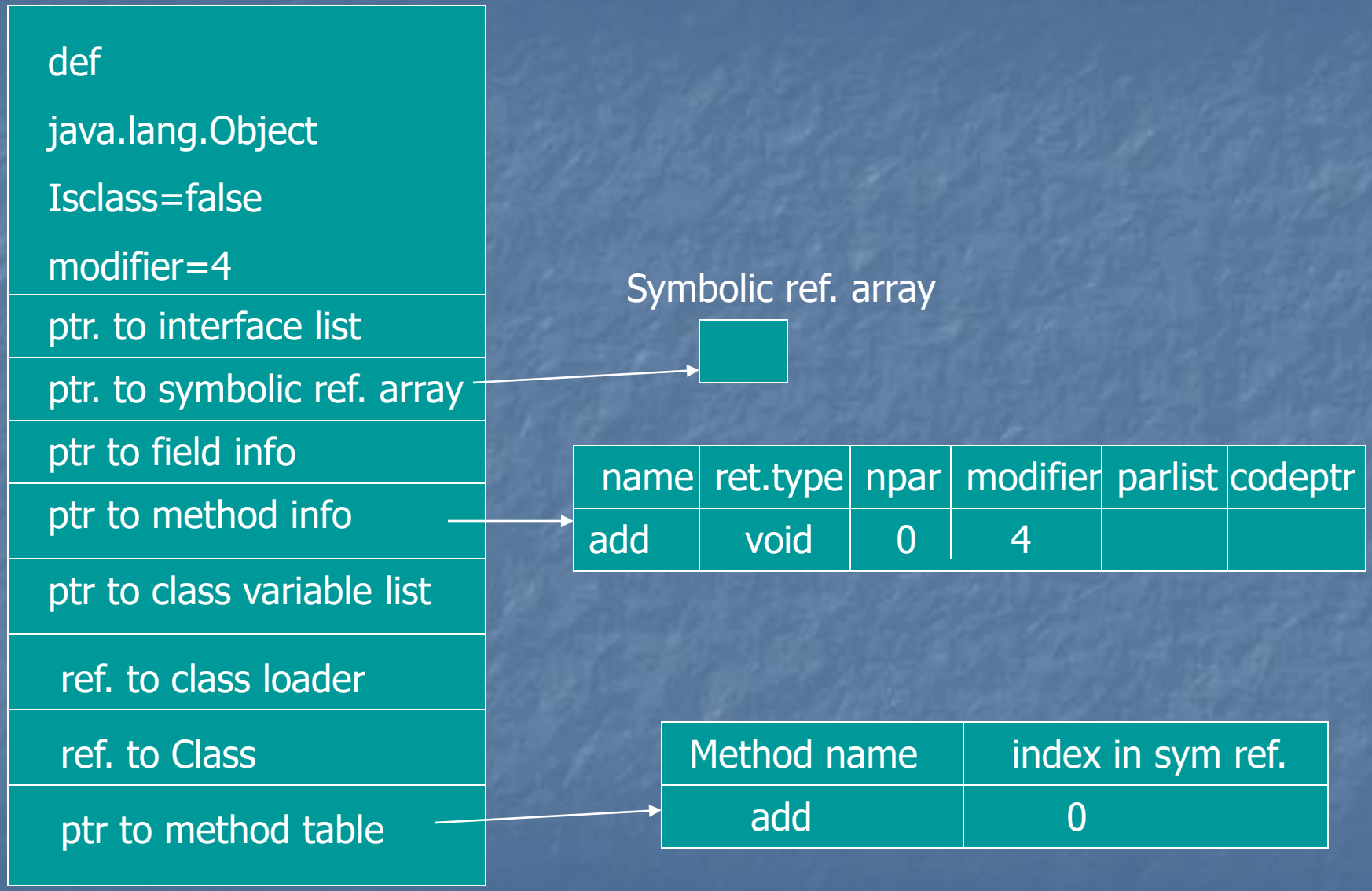
Class variables

null

Method Table

name	index
add	0

Class Area of def in Method area



```
class pqr  extends abc implements def {
 static int b;
 final int c=50;
 String s;
 pqr(int m) { super(); b=m;
                    s= new String("string2"); }
 void add() { a=a+c;  add1(); }
 static void add1() { c=b+50; }
}
```

Type info

pqr
abc
Isclass=true
modifier=4

Class variables

b

Constant pool

Symbolic ref. array

b	c	s	<init>	super	add	add1

50

Field info

name	Type	Modifier	index
b	int	4,6	0
c	int	4,7	1
S	String	4	2

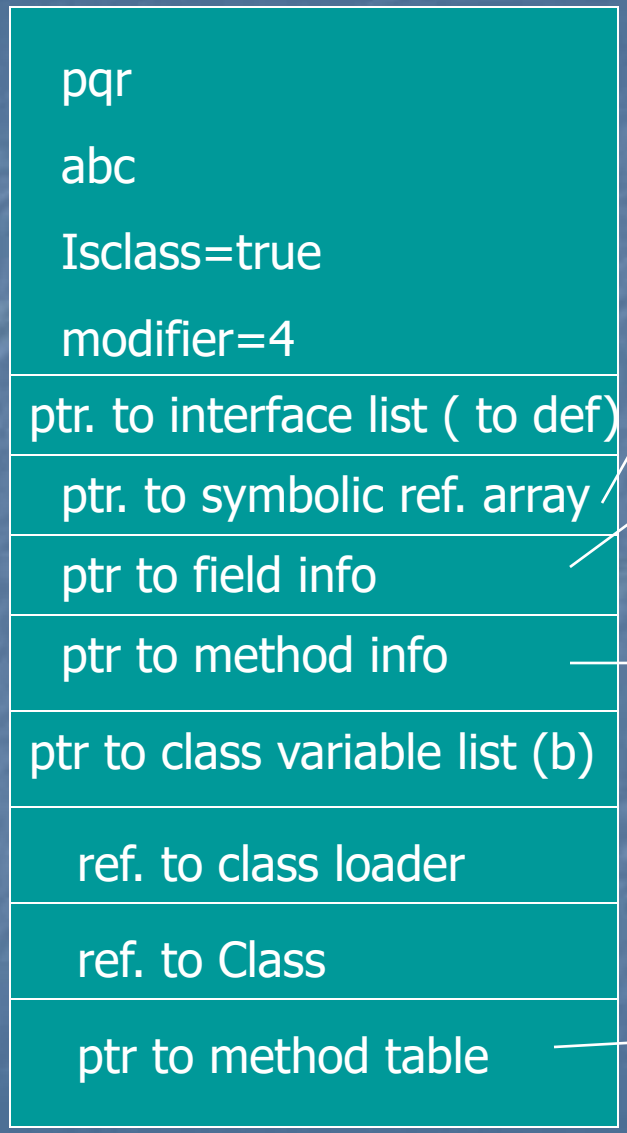
Method info

name	ret.type	npar	modifier	parlist	codeptr
<init>	void	1	4		
add	void	0	4		
add1	void	0	4		
super	void	0	4		

Method Table

name	index
<init>	3
add	5
add1	6
super	4

Class Area of pqr in Method area



Symbolic ref. array

name	Type	Modifier	index
b	int	4,6	0
c	int	4,7	1
S	String	4	2

name	ret.type	npar	modifier	parlist	codeptr
<init>	void	1	4		
add	void	0	4		
add1	void	0	4		
super	void	0	4		

Method name	index in sym ref.
<init>	3
add	4

```
class Main  {
 public static void main(String[] s) {
    pqr p=new pqr(20);
    p.add();
    p.print();
 }
}
```

Type info

Main

java.lang.Object

Isclass=true

modifier=4

Constant pool

Symbol ref. array

main	20

Field info

null

Method info

name	ret.type	npar	modifier	parlist	codeptr
main	void	0	4		

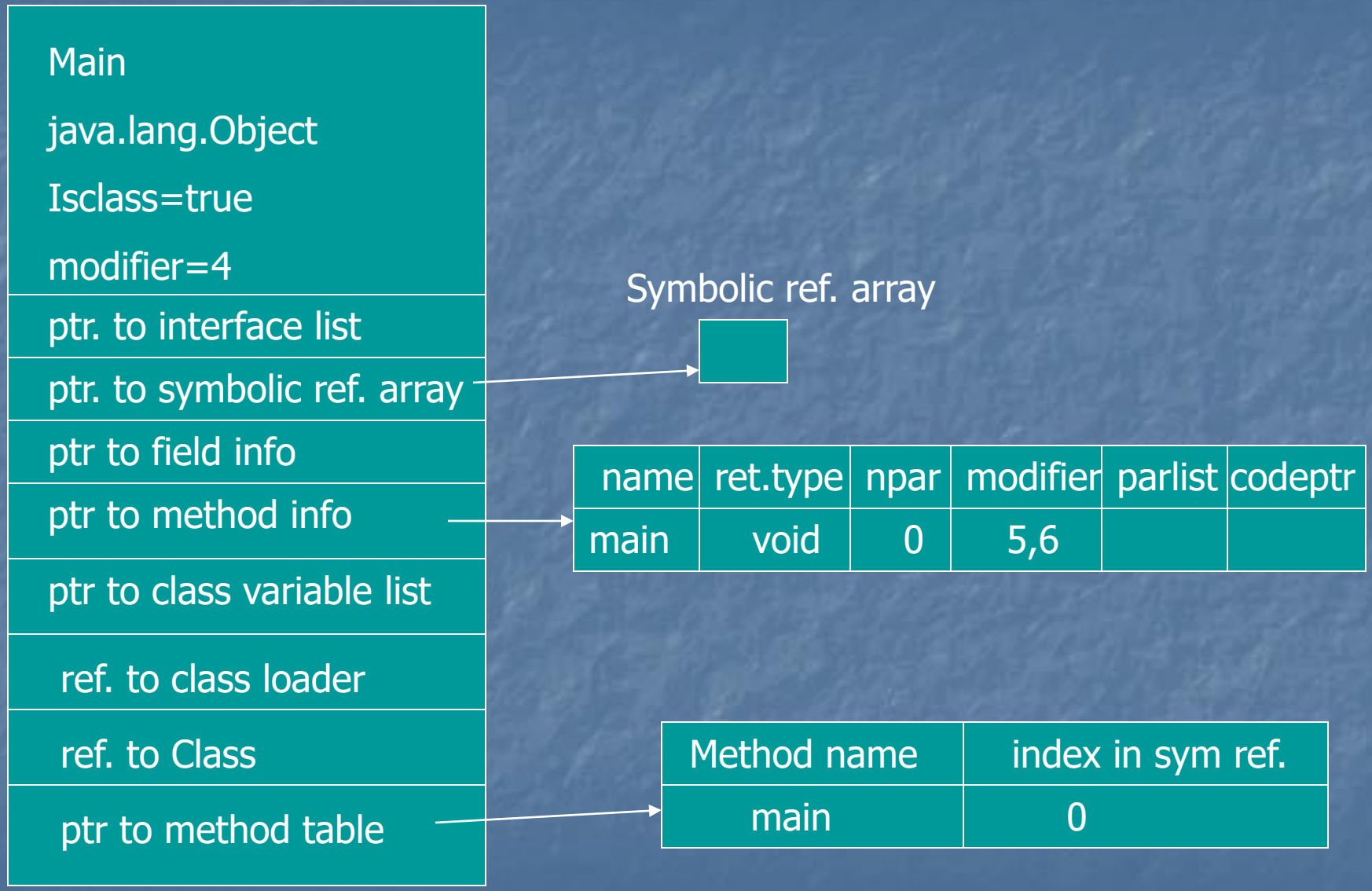
Class variables

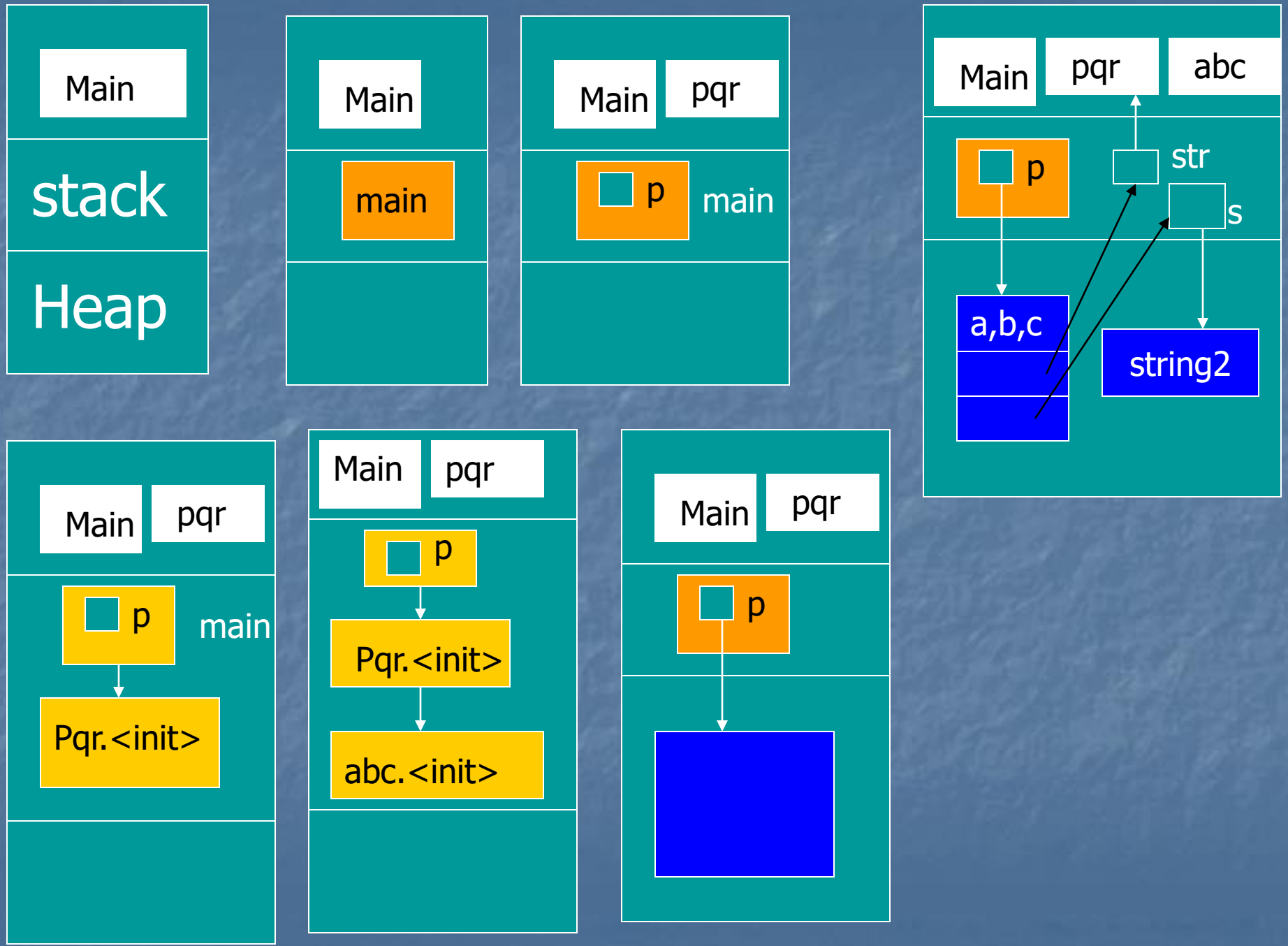
null

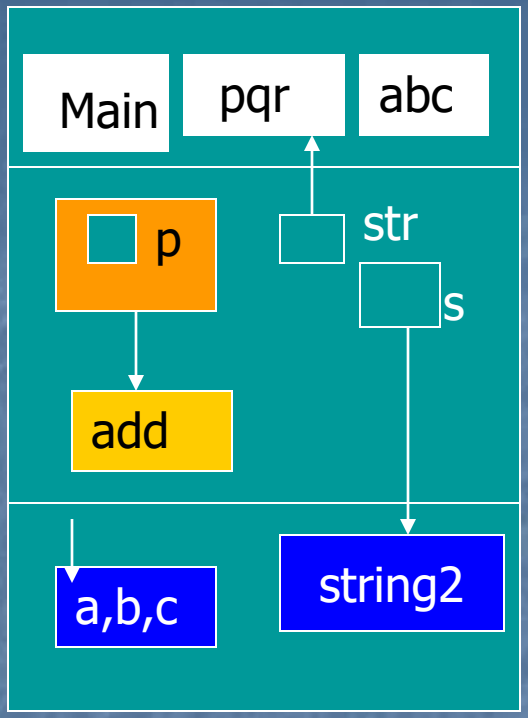
Method Table

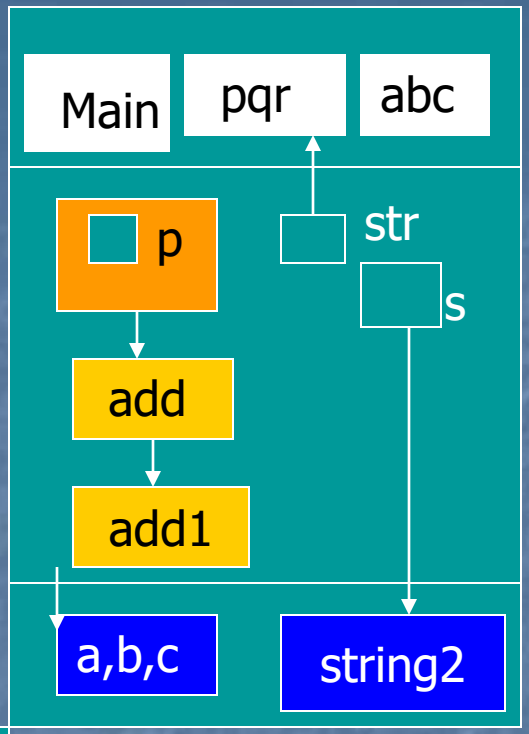
name	index
main	0

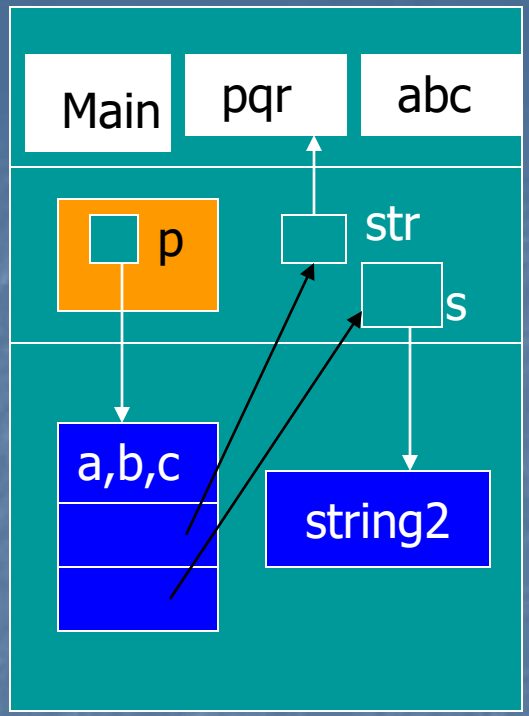
Class Area of Main in Method area

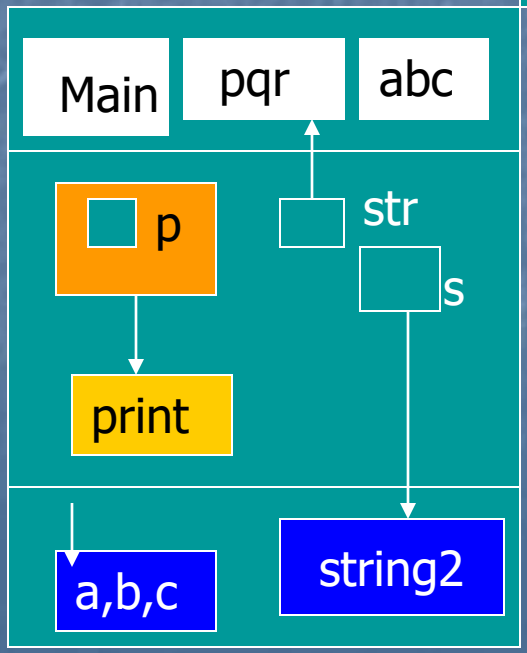


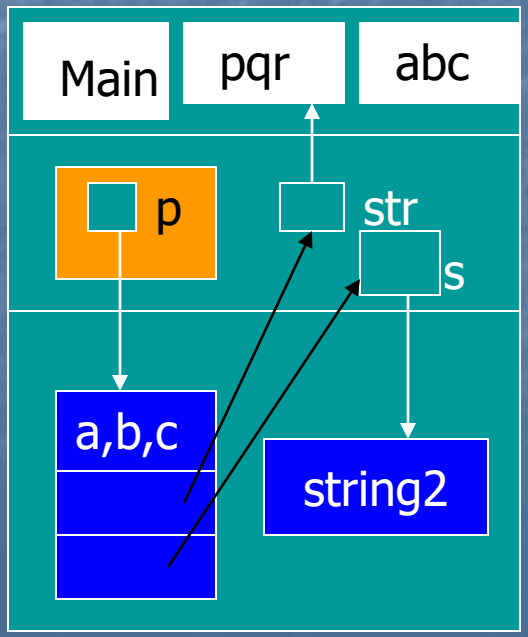

Main
stack
Heap
Main
main
Main
pqr
p
main
Main
pqr
abc
p
str
s
a,b,c
string2
Main
pqr
p
main
Pqr.<init>
Main
pqr
p
Pqr.<init>
abc.<init>
Main
pqr
p

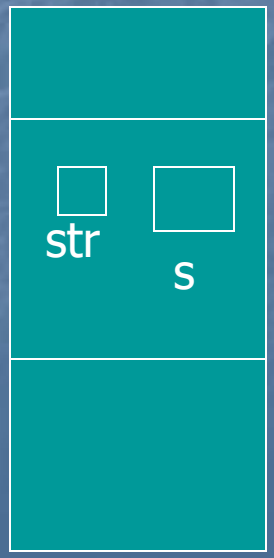

Main
pqr
abc
p
str
s
add
a,b,c
string2

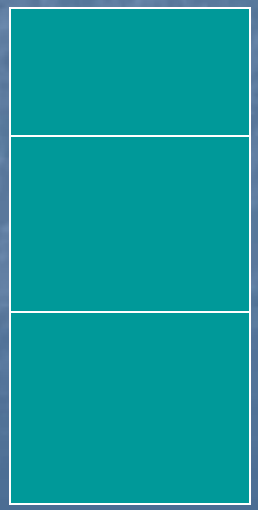

Main
pqr
abc
p
str
s
add
add1
a,b,c
string2


Main
pqr
abc
p
str
s
a,b,c
string2

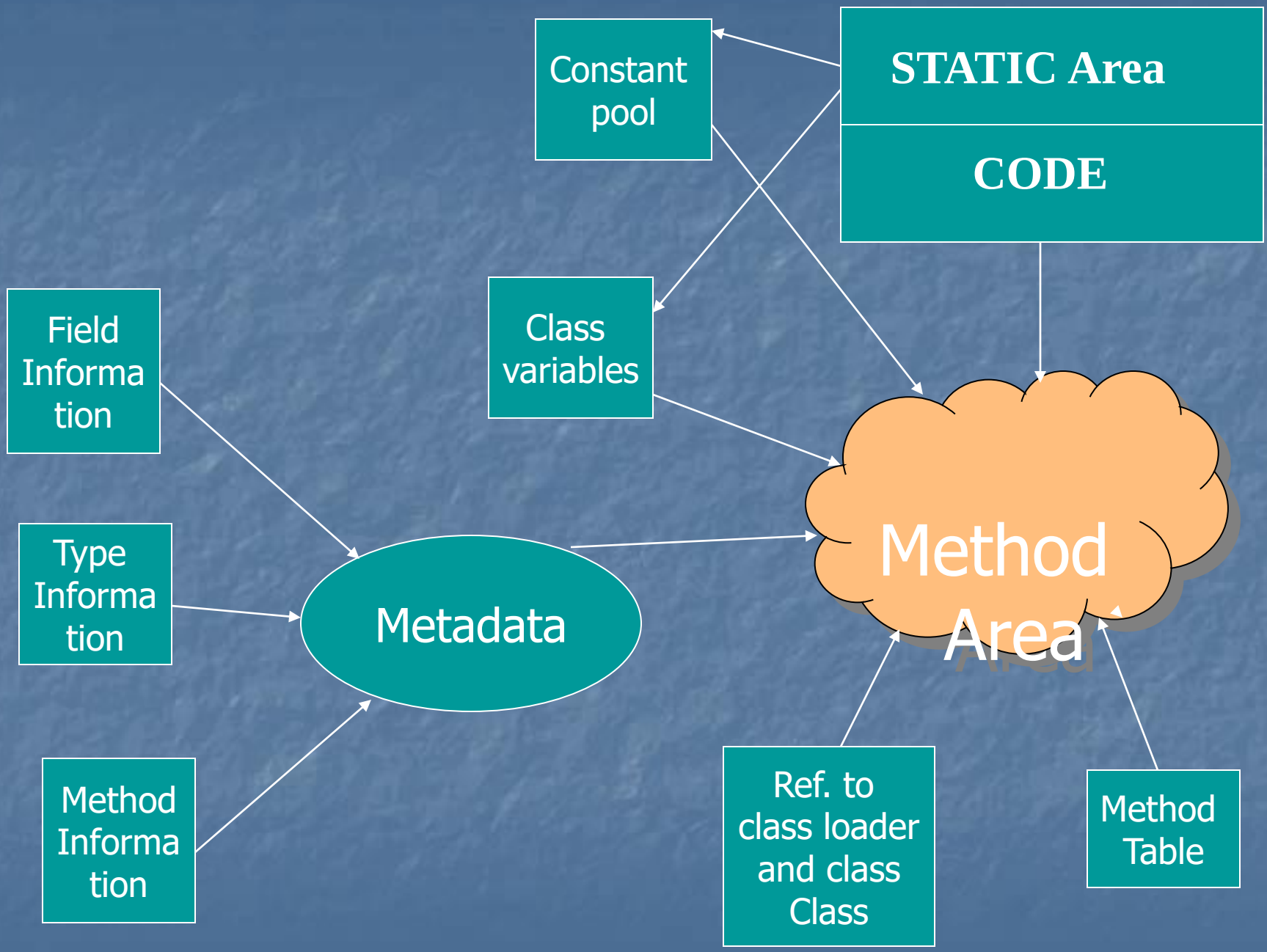

Main
pqr
abc
p
str
s
print
a,b,c
string2

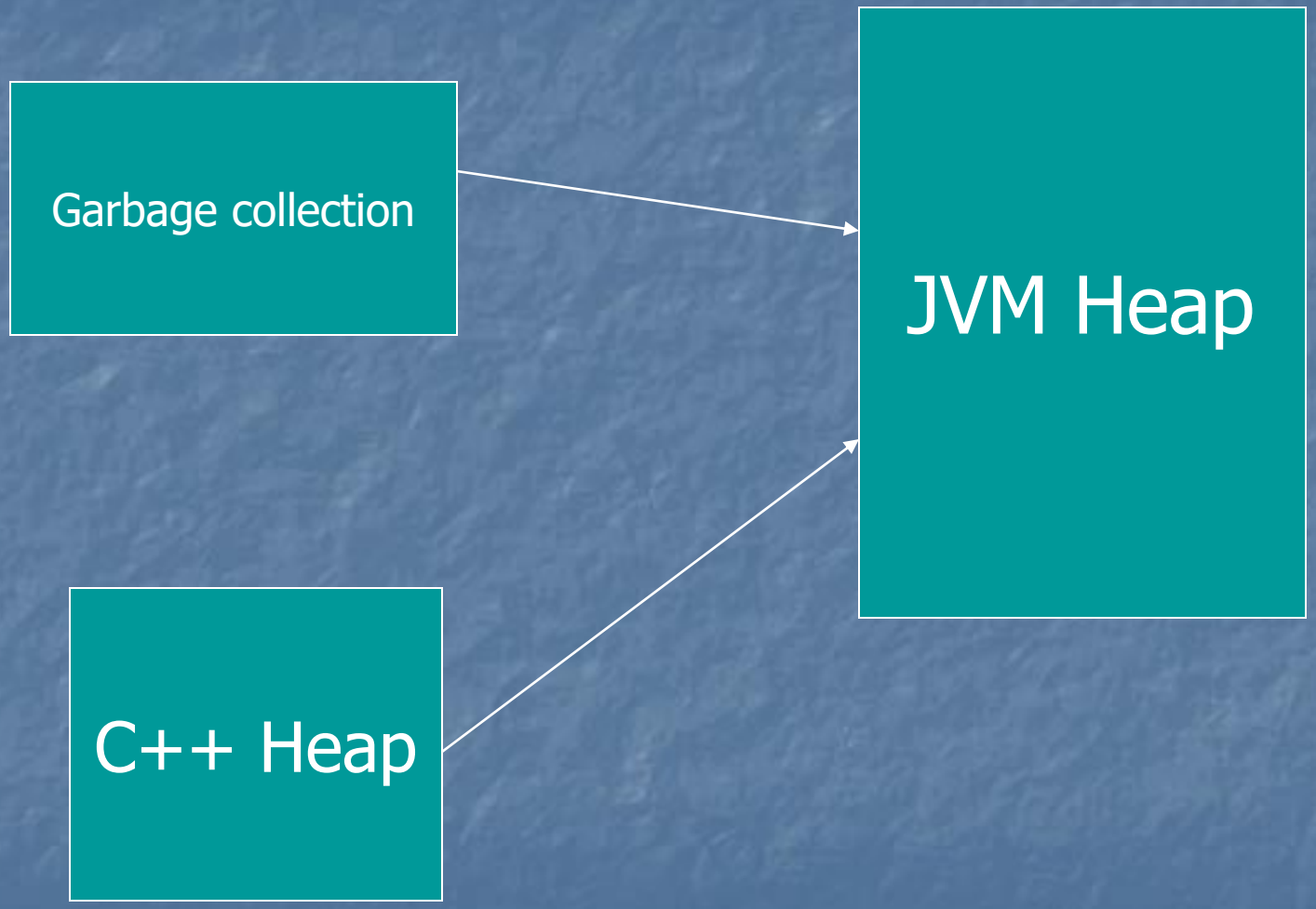

Main
pqr
abc
p
str
s
a,b,c
string2

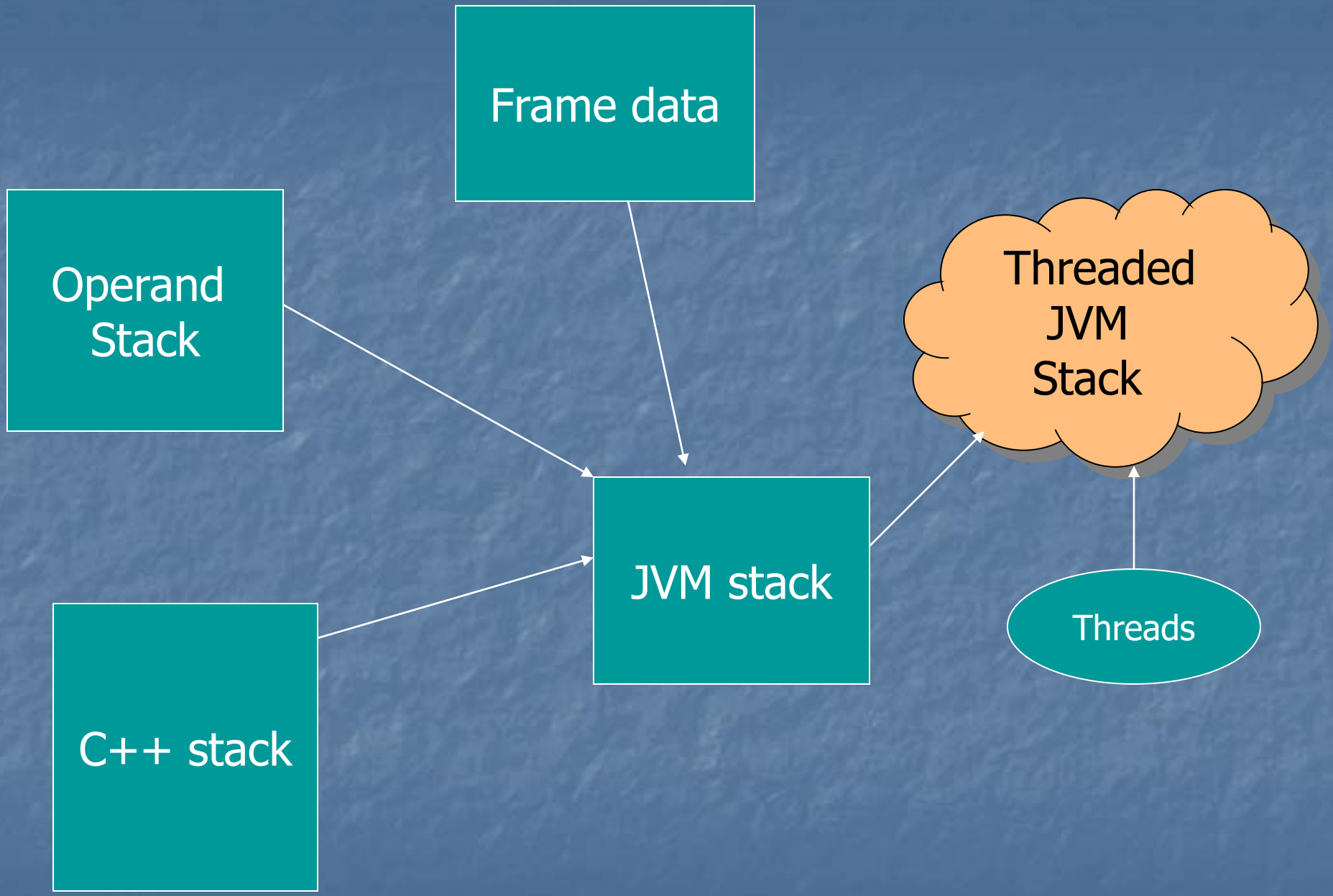

str
s



Relation between C++ memory structure and JVM memory structure


Constant pool
STATIC Area
CODE
Field Information
Class variables
Type Information
Metadata
Method Area
Method Information
Ref. to class loader and class Class
Method Table


Garbage collection
JVM Heap
C++ Heap


Frame data
Operand Stack
Threaded JVM Stack
JVM stack
C++ stack
Threads

Responsibilities of Memory Mgr

- Chop the chunk.
- Allocate Requested number of bytes.
- Provide necessary information to garbage collector.
- Collect the bytes returned by garbage collector.
- Defragment the chunks.

Design Choices

- How to allocate bytes
  - Contiguous memory allocation.
  - Non contiguous memory allocation.
- How many bytes to be allocated
  - exact requested number of bytes.
  - Allocate bytes in terms of 2^n (Buddy System).
- How defragmentation to be done
  - Compaction.
  - Buddy system.

JVM Memory Manager

Algorithms Followed

- Noncontiguous Memory Allocation.
- First fit to choose approperiate chunk.
- Buddy system to chop a chunk.
- Buddy system for defragmentation.

How to implement memory manager

- Need of 4 threads
  - Memory allocator.
  - Memory Defragmenter.
  - Chunk cleaner.
  - Data manager.

When VM requests m bytes

- Allocator
  1. Checks for first chunk with size >=m+4.
  2. If available chops it and returns to VM else requests OS for new chunk.
- Data Manager inserts an entry in Memory Allocation Table (for garbage collector).

Starting address	Size requested	Size Allocated

Garbage Collection

- For each entry in MA Table GC moves to starting address+ size requested address, checks the ref. count. If zero returns the index of table entry to Data Manager.
- Data Manager deletes the entry from MAT and adds in Defragment Table.
- For each entry in Defragment Table, Defragmenter combines the consecutive fragments.

Starting Address	Size

Design problem

VM requests an array of bytes

- to store data in Method Area.
- for stack frame.
- for object on Heap.

MM doesn't know the purpose of bytes.

How does MM allocates memory in three different address spaces for different type of requests.

Use Three MMs. Each MM has its own addr. Space. Its responsibility of VM to direct the requests to appropriate MM.

Optimal size of chunk

- For Method Area MM 10K
- For Stack 16K
- For Heap 1K, 10K, 32K

Initial State of Chunk pool

- Trade off between performance and efficient memory usage.
- first and top pointers of each pool is set to null

Chunk Pool cleaner

- similar to Garbage collector.
- For every 5s cleaner returns all chunks more than 5.

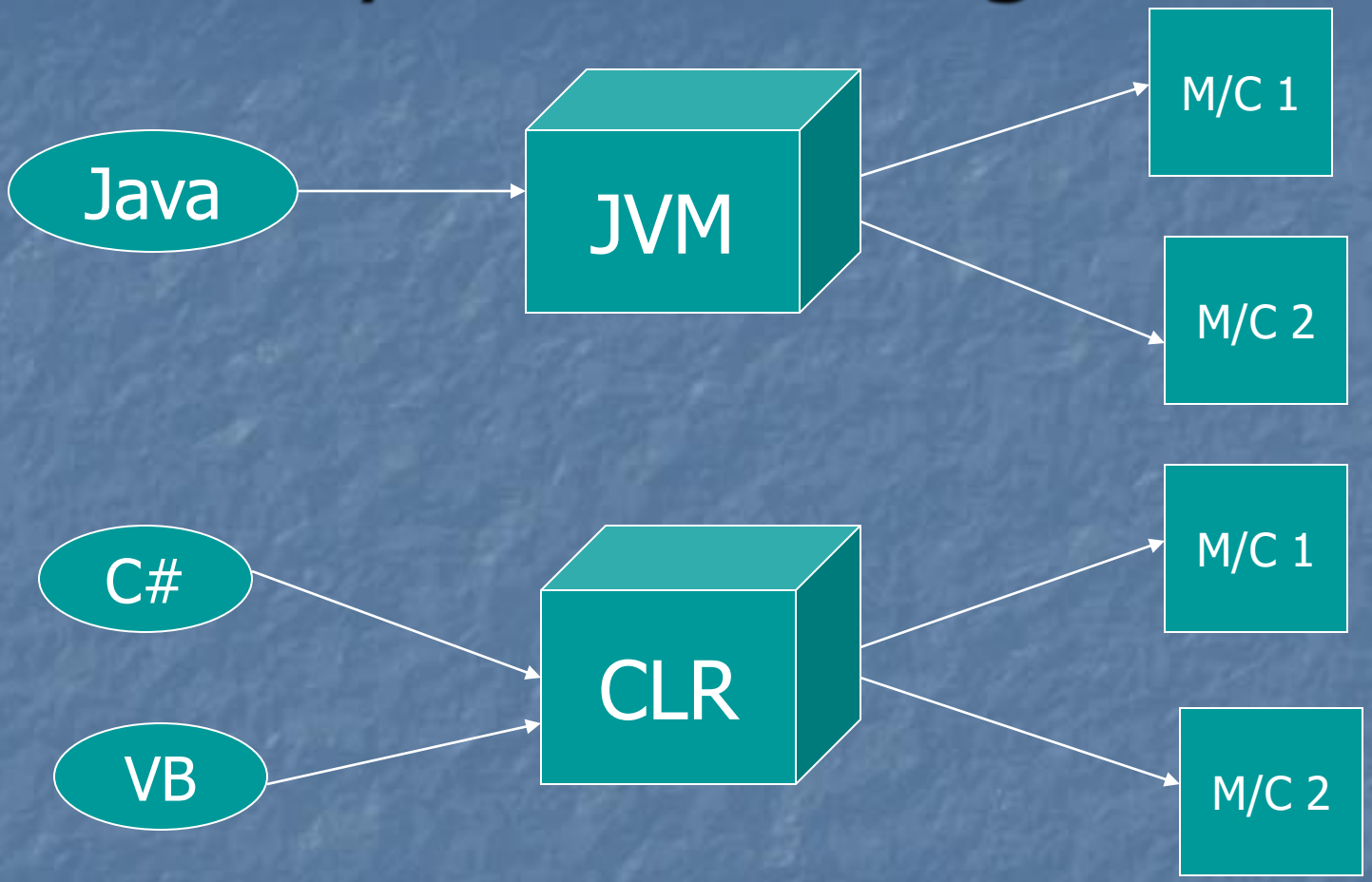
ClR Memory Management

CLR Names to Memory Areas

Method Area	as	Type Area
Stack	as	Roots
Heap	as	Heap, but two heaps
		- Managed heap
		- Unmanaged heap

In JVM entire heap is managed.

Necessity of Unmanaged Heap



Some Languages allow pointers. So to support pointers Unmanaged heap is supported

What is the difference

JVM MM

- Allocation is complex.
- Defragmentation is simple.

ClR MM

- Allocation is simple.
- Defragmentation is complex.

JVM Memory Manager

Algorithms Followed

- Contiguous Memory Allocation.
- Compaction for defragmentation.
- Lazy Defragmenter

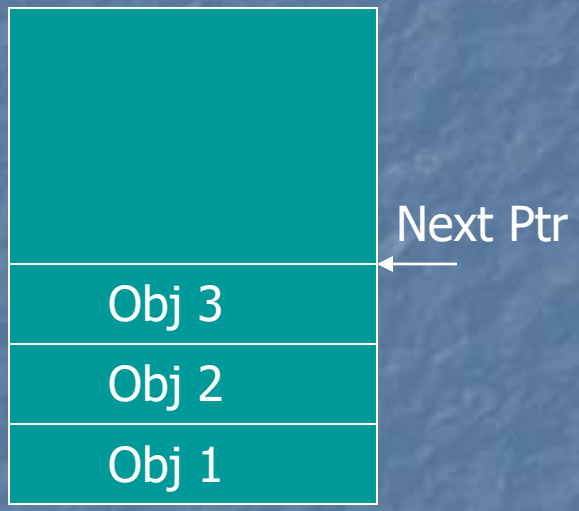
Memory Managers

- Type Memory manager.
- Roots Memory manager.
- Managed heap manager.
- Unmanaged heap manager.

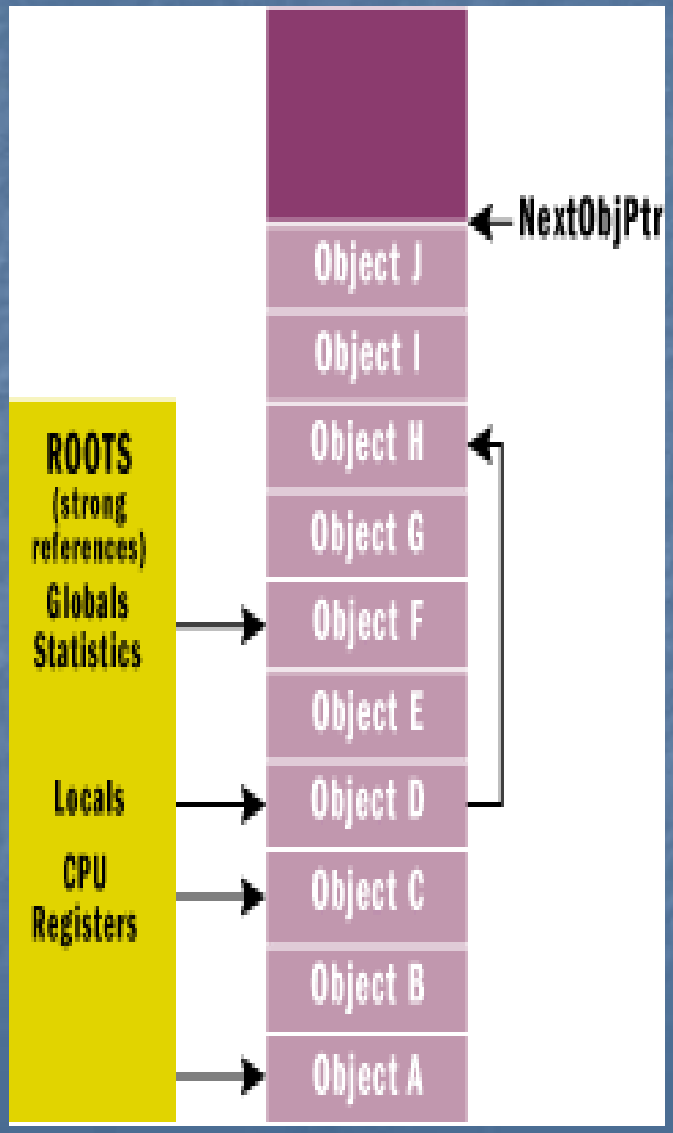
- GC runs.
- No GC.
- GC runs.
- No GC.

To deallocate Unmanaged Memory Finalizers should be used.

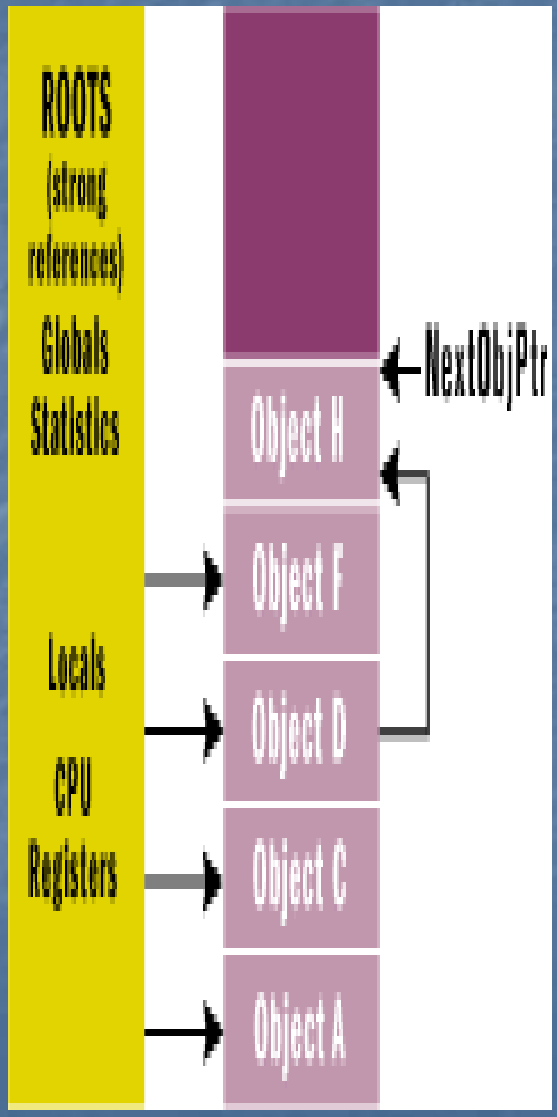
Memory Allocation



Before GC



After GC



Thank You