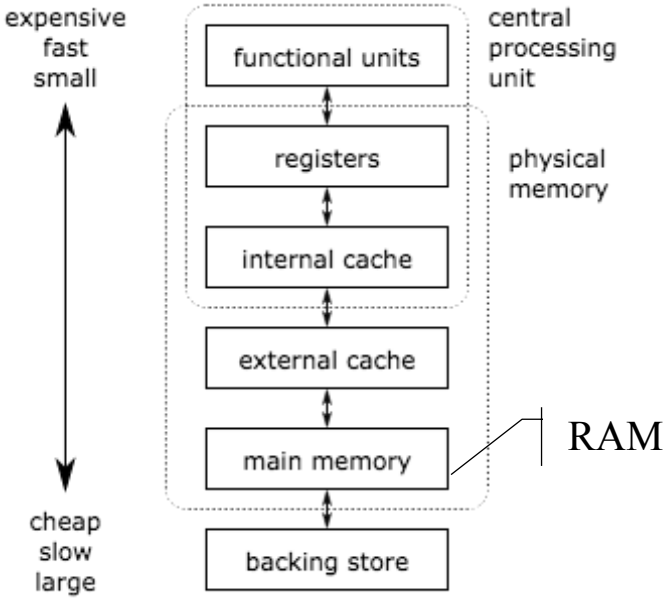
Advanced Programming

MEMORY MODELS

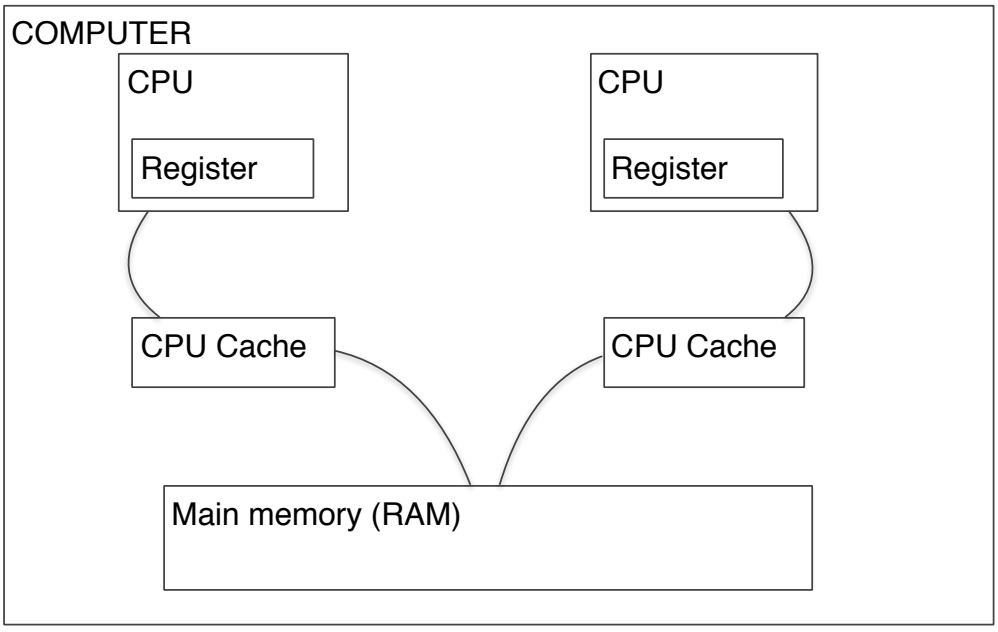
Notes

- …exercises with jdb and javap
- The java debugger jdb must work with both source and class files.
  - Try to run jdb after having cancelled the java file

Storage hierarchy in computer



Computer memory



CPU Registers

- The CPU exposes the results of its computations in the registers
- Registers are CPU internal memory (fast access)

CPU cache

- A processor's **memory cache** is a small piece of fast, but expensive memory used for copies of parts of main memory
- Access to the cache typically takes only a few processor clock cycles, whereas access to main memory may take tens or even hundreds of cycles

Physical memory (RAM)

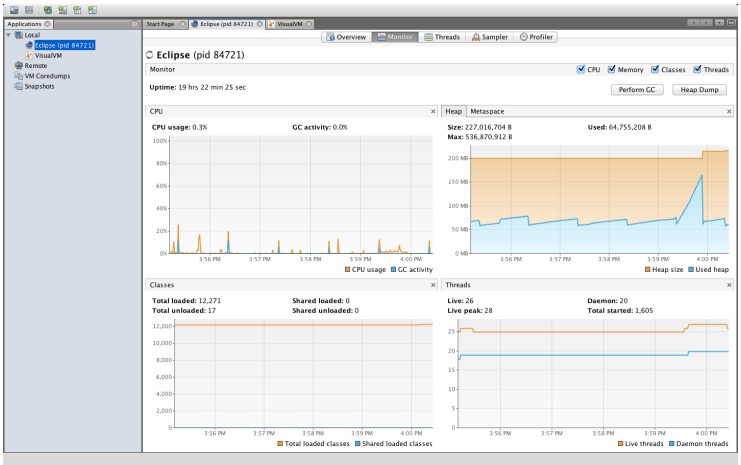
- Physical (main) memory is memory that is wired to directly to the processor, addressable by physical address

Thread

- The OS allocates a copy of the physical memory (called **virtual memory**) for each Java process (called **thread**)
- One can have at least as many threads in parallel as the number of CPUs (modern computers are multi processors and multi cores)

Visualising memory consumption

- Go to your bin folder and search for jvisualvm
- In the shell window type jvisualvm



Run-time memory models

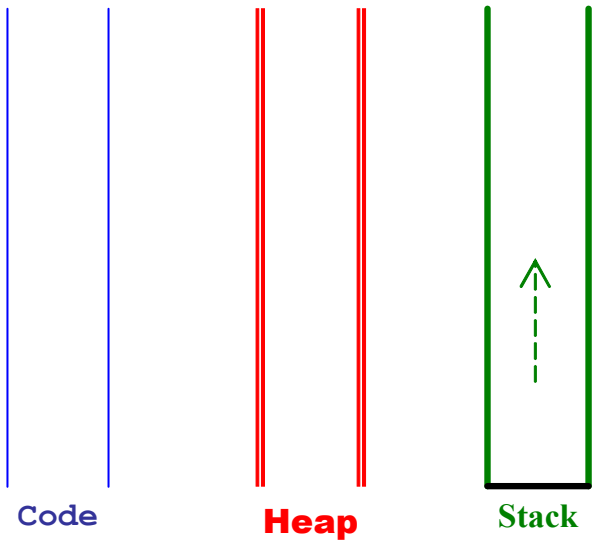
- There are different models of the **program execution**
- In the model we use, the program starts and assigns **three** separate and independent portions of memory (referred to as **address space for** each process thread)

Memory models

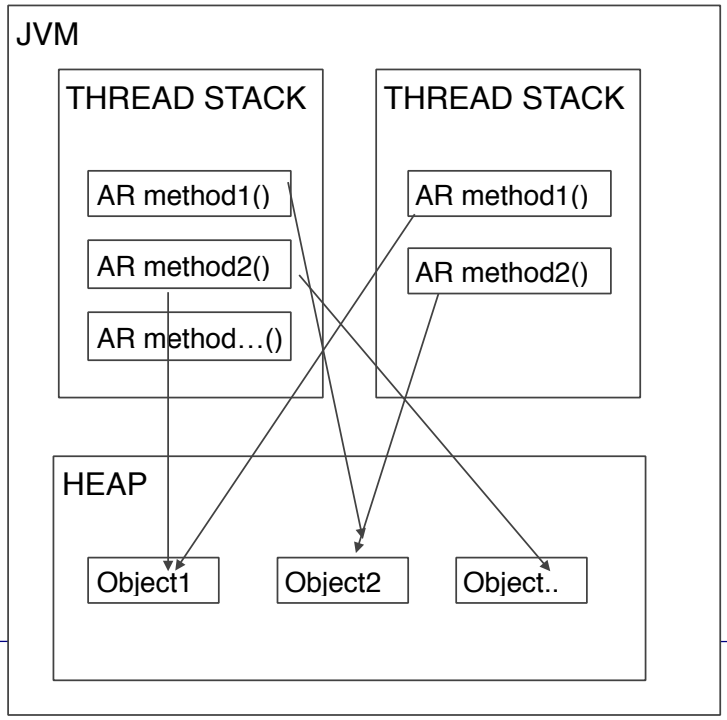
- These are:
  - **code** area
    - where code to be executed is stored
  - **heap**, or dynamic memory area
    - used to store variables and objects allocated dynamically
  - (execution) **stack**
    - used to perform computation,
    - store local variables and
    - perform function call management

Memory models (Run-Time Data Areas)

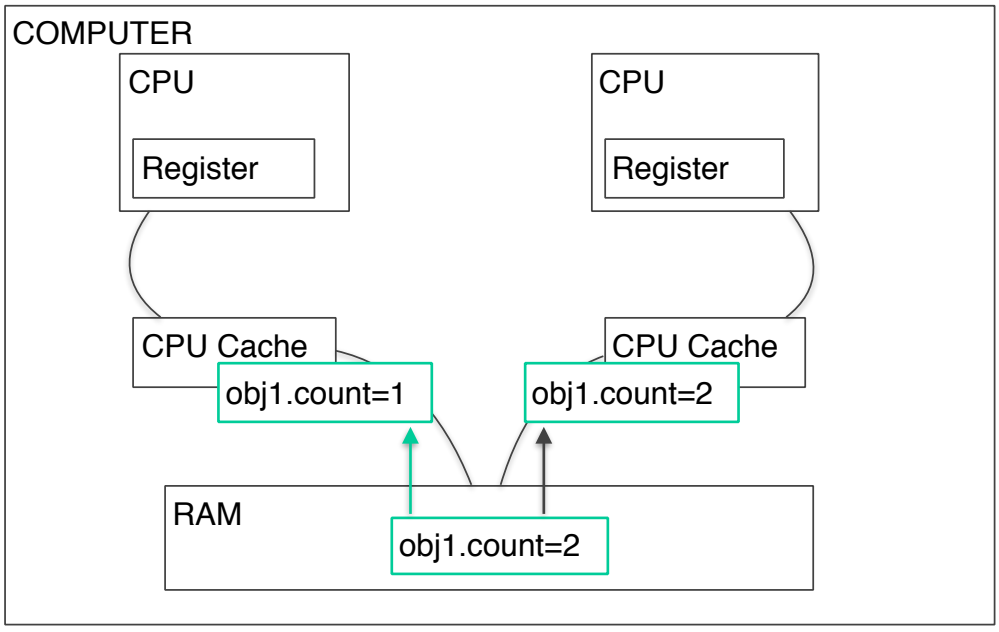
- The code and the heap area can be accessed with no special restriction
- The stack area is accessed using a LIFO (Last In – First Out) policy
- The languages discussed will be "Stack-based"



Stack and heap



Visibility and conflicts



Visibility and conflicts

- Java uses the keyword “volatile” to avoid that two threads that have read the same object from the main memory modify it independently in the cache
- Until the cache memory of the two threads is not flushed back to the RAM nothing happens as the two cache memories are not visible each other
- The conflict starts when the two cache memories are flushed back to the RAM

Volatile

- To solve this problem you can use the **volatile** keyword.
- It makes sure that a given variable declared volatile is read directly from main memory, and always written back to main memory when updated

Synchronised

- Race condition: if two or more threads share an object, and more than one thread updates variables in that shared object
- A synchronized statement or keyword for a code block:
  - only one thread can access the code block
  - all variables accessed inside the block are read from main memory,
  - when the thread exits the block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

Memory management in Java

- Java is organised in classes
- Code is loaded on **class-by-class** basis
  - The first class to be loaded is an executable class that defines a static “main” method (with well defined signature)
- The execution proceeds loading and running new classes when the need arises, according to the flow of the computation

Stack based languages

- The execution is centred around the **execution stack**
- All our algorithms are organised into methods
- The order of execution of methods and functions is LIFO (Last in First Out)

Activation record of a method

- Each time a method / function is called, all the information specifically needed for the method execution is put on the stack
- That information is collectively called the **Activation Record (AR)** of the method call (also called frames)

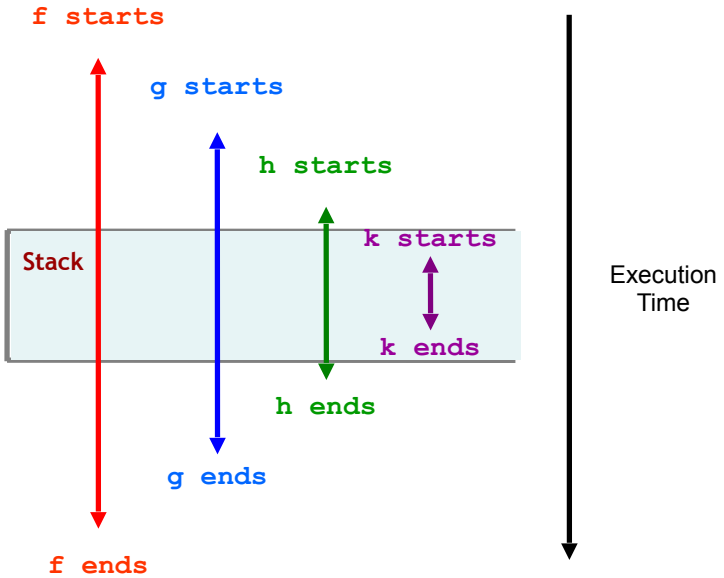
Activation record of a method

- This allows recursion, since for each call there will be a separate activation record on the stack
- When the call is completed (i.e., the method "returns") the corresponding AR is destroyed (i.e., "popped out" of the stack) according to LIFO
- Activation records are organized from bottom to top in memory diagram

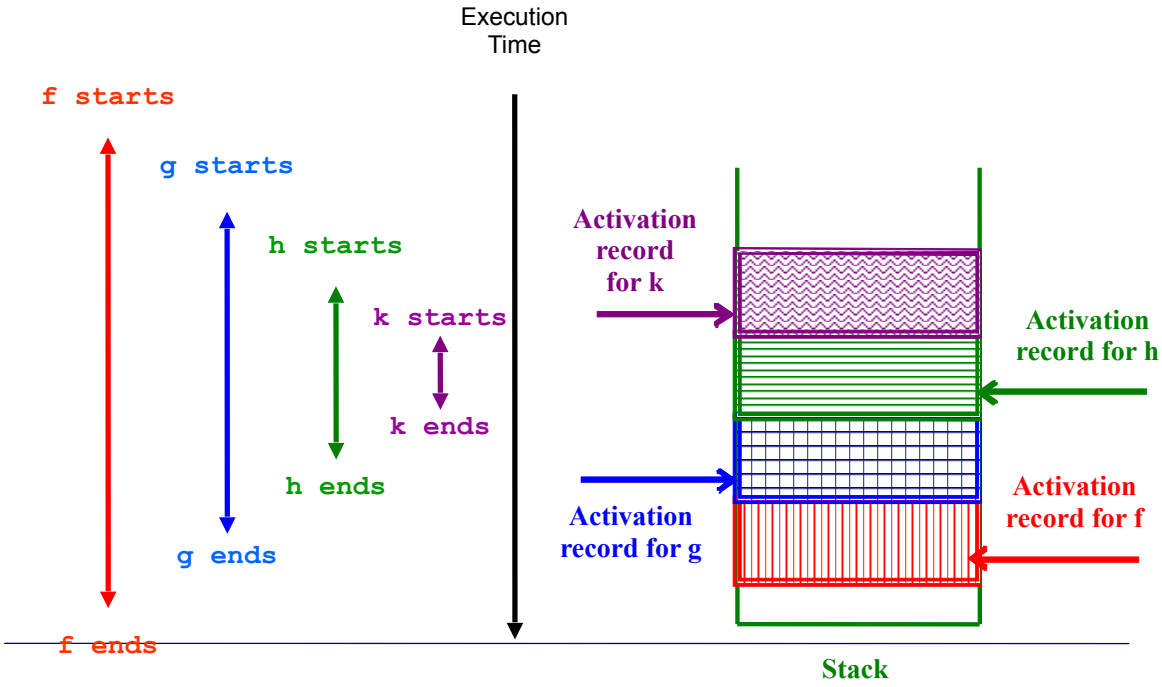
Example

- Each method call results in an AR on stack

```
void f() {
  g() ;
}
void g() {
  h() ;
}
void h() {
  k() ;
}
```



Example



Terminology

- Adding an activation record in the stack is called **winding**
- Removing is called **unwinding**
- **Note:** Exception handling uses the unwinding mechanism

Content of the activation record

- The information stored in the AR for one call are the following:
  - Information to **restart the execution at the end of the call**, i.e. after the function "returns"; these usually are:
    - **Return address**
    - **Pointer to the Stack portion devoted to the calling method**
    - **Return value (if any)**
  - Information needed to **perform the computation** (usually the actual arguments passed to the method in the call – if any)
  - **Local variables (if any)**

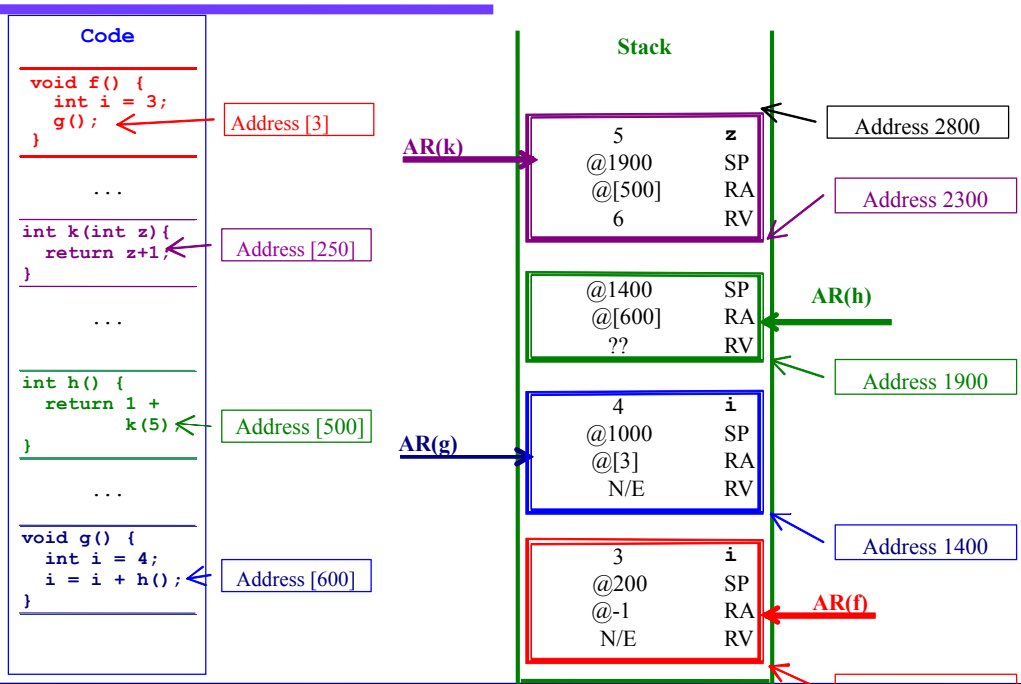
Activation Record: abbreviations

- AR() → activation record of a function
- RV → return value
- RA → return address
- SP → stack pointer
- N/E → Non Existent
- @ → at memory address
- ?? → not yet determined

Example

```
void f() {
        int i = 3;
        g();
    }
void g() {
        int i = 4;
        i = i + h() ;
    }
int h() {
        return 1 + k(5) ;
    }
int k(int z) {
        return z+1;
    }
```


Example



Scope and extent

- To understand how JVM allocates memory for variables we talk of scope and extent

Scope of a variable

- The **scope** of a variable is a portion of the (source) code in which that variable is **visible** in the code

Scope

- Visibility (i.e. scope) is governed by the **structure of the source code** and not by the execution!

Blocks

- A block is a portion of code enclosed between two special symbols, which mark the beginning and the end of the block
- In Java blocks are marked by curly braces

{ <this is a block> }

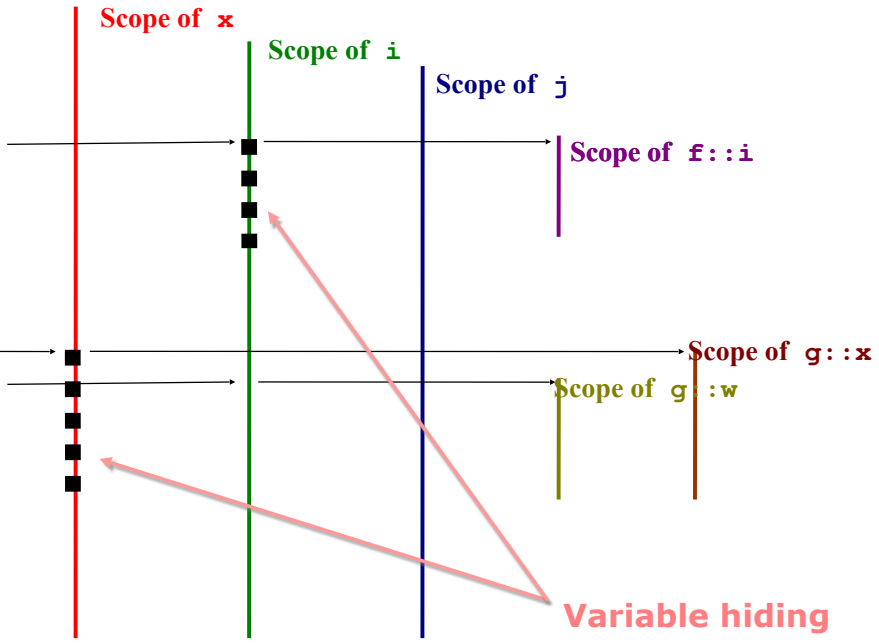
- Block can be nested
- We use block to understand the scope of variables!

Scope & Blocks

- A variable is visible
  - In the block it is defined
    - Starting from the line of definition
  - In all the inner blocks unless a variable of the same name is declared within (homonymous variables)
- Global variables
  - Defined outside the scope of any block
- Hiding a variable
  - A **homonymous variable** declared within a block makes invisible a variable of the same name declared outside the block

Example: variable scopes

```
[1] int x=5;
[2] int i=4;
[3] int j=3;
[4] void f() {
[5]     int i = 3;
[6]     x = x + 2;
[7]     j = j + x;
[8]     g();
[9] }
[10] void g() {
[11]    int x;
[12]    int w = 4;
[13]    x = 7 + w;
[14]    j = j + 1;
[15]    i = i + x;
[16]}
[17] void main() {
[18]    f();
[19] }
```



Use of “this”

- The most common reason for using the “this” keyword is because a field is shadowed by a method or constructor parameter

```
int count = 0;
public void myMethod (int count){
    this.count = count;
}
```


Example

if a field is shadowed by a method or constructor parameter than you can simply change the names, or …

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int a, int b) {
    x = a;
    y = b;
    }
}
```

inside the constructor x is a local copy of the constructor's first argument. To refer to the Point field x, the constructor must use this.x.

```
public class Point {
    public int x = 0;
    public int y = 0;
    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Scope Activation Record (SAR)

- **Scope Activation Record (SAR)** is a memory block in the stack that contains block information for **variable visibility**
- ARs are also SARs, because the body of a function is a block itself, thus information on visibility must be added to ARs.

Scope Activation Record

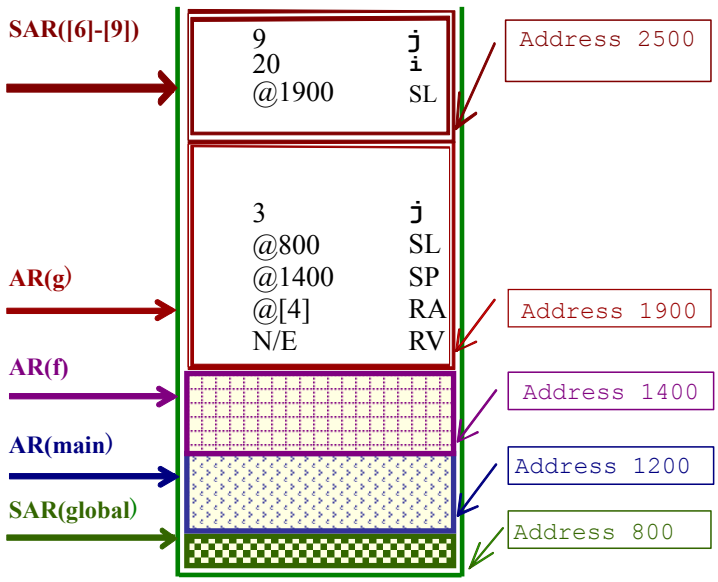
- SARs contain, at least, two different kinds of information (ARs contain more) :
  - local variables (local to the block itself)
  - the Static Link (SL)

Scope Activation Record

- The **SAR link** is a pointer to the SAR of the immediate enclosing block and it is used to **access local variables of outer blocks** (in a recursive fashion) from the current block

Example of SAR

```
[1] /* Some global definitions
[2] */
[3]void f() {
[4]     g();
[5] }
[6]void g(){
[7]     int j=3;
[8]     if(j==3){
[9]         int i=20;
[10]        int j=9;
[11]     }
[12]}
...
[27] void main() {
[28]       f();
[29] }
```



SLs

- Each time a variable is used in a block, but there is no definition of such variable in such block, the system uses the SL to reach out for the next enclosing scope to find that variable recursively(i.e., if it is not there, the SL is used to reach the next enclosing scope, and so on) until reaching the global scope

Extent of a variable

- The extent of a variable (i.e., lifetime) is the time during which the variable exists in memory
- The extent of a variable is the time during which it **exists on the stack or on the heap**, i.e. the time during which there is some memory allocated for it

Extent of a variable

- The extent of a variable is the time-domain concept
- The scope is a structure concept
- A variable can exist (Extent) but not visible (Scope)
  - e.g., global variable hidden by homonymous local variable in a block but exist for the whole execution

Static/Dynamic memory allocation

- There are two types of memory allocations of variables:
  - Stack-based variables
  - Heap-based variables
- The definition of scope and extent varies in the two cases

Variables and References

- Stack-based variables are all created on the Stack statically
  - e.g., object reference variables or primitive variables in methods
- Heap-Based variables are created on the Heap dynamically
  - e.g., object reference variables in classes

Example

AR of constructors are omitted in this example

```
void useAttributes() {
  CoffeeMaker aCM = new CoffeeMaker();
  aCM.status = 1;
  aCM.coffeeAmount = 10;
}
```

```
public class CoffeeMaker{
   int status = 0;
   int coffeeAmount = 0;
}
```

Stack-based variables

- Stack-based variables (statically allocated):
  - the **extent** is determined by **scope**
- Stack variables begin when their definition is encountered in the code and end at the end of the scope in which it is defined

Heap-based variables

- Heap-based variables are the objects
  - Their memory is dynamically allocated in the heap
  - Their **extent** is under control of programmers, (unconstrained **extent)**

Reference variables

- Objects have no name
- References to objects are stack-based variables:
  - In Java, these are called reference variables

Heap-based variables

- The scope of heap-based variable is the **union of the scopes of all the variables referring to it**
- The extent of an entity allocated on the heap starts **when they are created and lasts until they are destroyed** or until the program terminates

Example

- CoffeeMaker aCoffeeMaker;
- aCoffeeMaker = new CoffeeMaker();
- int sugar = 4;
- Integer sugarObject = new Integer(3);