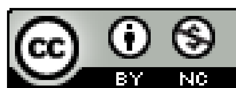
Object Oriented Memory Management (Java and C++)



This work is licensed under a
Creative Commons Attribution-Noncommercial 3.0 License.

Table of Contents

- The model....4
- Code load and execution....5
- Activation Record (AR)....5
  - Contents of the Activation Record....6
  - Abbreviations for AR....6
- Declaration vs. Definition....6
- The scope of a variable....6
- Extent of a Variable....6
- Blocks....6
- Scope Activation Record (SAR)....7
  - Example on SAR....7
- Role of SLs....7

Dynamic Memory Allocation And Handling....8
- Dynamic Vs. Static memory allocation....8
- Dynamic Memory Scope and Extent....8
- Accessing dynamic memory 8
- Classes....9
- Objects....9
  - Object instantiation....9
  - Objects in Memory (Java):....9
  - Objects in Memory (C++)....10
- Memory Management issues (Java)....11
- Memory Management issues (C++)....11
- Methods....12
  - Methods (Java)....12
  - Methods (C++)....13
- Attributes....14
- The null value (Java)....14
- The NULL value (C++)....14
- Parameters....15
  - Parameter Passing (Java)....15
  - Example of parameters passing (Java):....16
  - Example of parameters passing (Java), continued:....17
  - Parameter Passing (C++)....18
  - Example of parameters passing by value (C++):....18
  - Example of parameters passing by reference (C++):....19
- Pointers vs. Parameters (C++):....20
- Previous example using pointers (C++):....20
- Constructors....21
  - Inline initialization....21
  - A constructor’s call (Java)....22
- Class attributes....23
  - Example of class attributes (Java)....23
  - Example of class attributes (C++)....24
- Class Methods....24

Example of Stack/Heap Diagrams in Java:....25
- Code....25
- Stack Diagram....26
- Heap Diagram....27

Introduction

This paper is about a model for memory management during the execution of programs written in Java and C++

It started on March, 2008 as a summary of the lecture notes of both the “Programming Project” and “Software Engineering Project” courses held by professors of the CASE (Center for Applied Software Engineering) of the Free University of Bolzano – Bozen.

The first versions of this publication were only about Java memory management.

Subsequent revisions added information found on other sources. Unfortunately, the author forgot to reference the sources on the document.

On March, 2009 the author began to add the information about C++ programming language. More information from other sources were added, including their attribution.

The biggest source of this document is still the set of presentations of CASE.
The code snippets and their corresponding stack/heap diagrams are copied in full from those of the slides.

The next major revision will contain images not belonging to CASE slides, as well as other code snippets that I could find more clear than those of CASE.

If you find that this document contains information taken from one of your publications, please contact the author, that is willing to either delete them from this document or to add an attribution to your work.

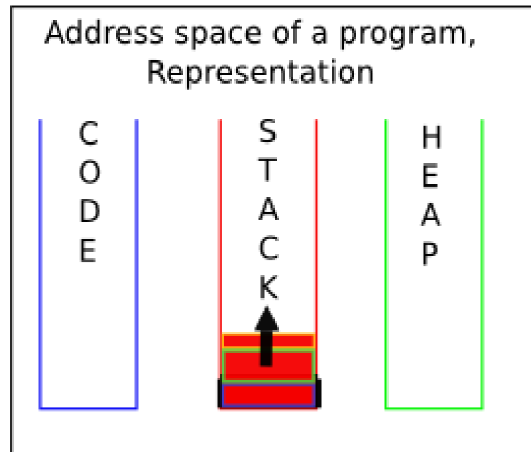
Organization

The first section is about the definition of the model and all of its components.
The second section contains tiny definitions of Object-Oriented paradigm characteristics and terms studied in the publication.
The rest of the paper is divided in micro-chapters. Each micro-chapter is about a characteristics of Object-Oriented paradigm (i.e. class, attribute, object) and starts with a tiny definition, followed by some comments, a little code snippet and the relative memory diagram.
Each micro-chapter covers first Java then C++, as well as their common factors.

The model

In the model chosen for this document, we simplify both Java and C++ programs as **single processes with one thread of execution**.
A program is assigned three independent portions of memory, as address space:

- **Code area** - where code to be executed is stored
- **Dynamic Memory Area a.k.a. Heap**[i] - to store variables and objects allocated dynamically accessed with no restrictions
- **Execution Stack**[ii] - to perform computation, store local variables and perform function call management accessed with a **LIFO** policy (Last In - First Out)



Please note that this model is very simple and minimalist[iii]. In C++ programs, for example, there are *several other memory areas*[iv]

I also assume that on C++ the entire code is loaded into code area, and neglect dynamic loading.

Code load and execution

Code is loaded on class-by-class basis
1. The class containing the main method is loaded
2. All the other classes are loaded, according to the flow of computation

The execution is centered around the execution stack, the order of execution is LIFO
 -> *the last method called is the first to terminate*.

Activation Record (AR)

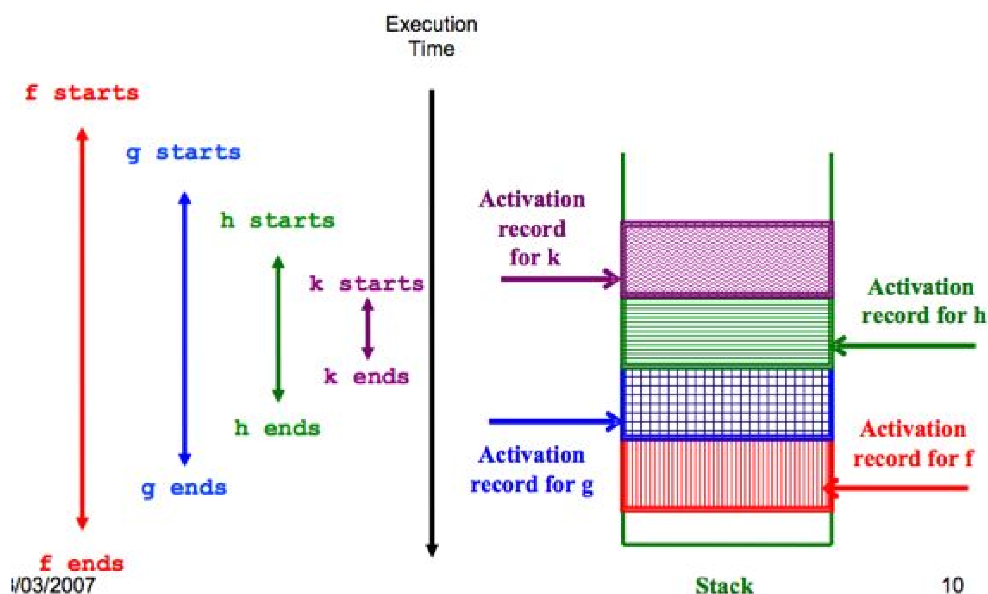
Each time a method is called, all the information specifically needed for the method execution is put on the stack. That information is called the **activation record (AR)** of the method call.
This system allows recursion, since for each call there will be a separate activation record on the stack.
When the call is completed the corresponding AR is destroyed.
Activation records are organized *from bottom to top* in memory diagram.

Example of function calls and activation record usage:

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

void g(){
   h();
}

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



Contents of the Activation Record

One call on the activation record contains the following information:

- Information to **restart the execution** at the end of the call (i.e., after the function returns)
  - Return value (if any) **RV** (*what am I returning?*)
  - Return address **RA** (*where am I returning?*)
  - Pointer to the Stack portion devoted to the calling function **SP** (*where am I from?*)
- Information needed to perform the computation (the **actual arguments** passed to the method)
- Local variables

Abbreviations for AR

AR()	activation record of a function
RV	return value
RA	return address (may either be the address of the code area or the code line in which the function returns)
SP	stack pointer
N/E	Non Existent
@	at memory address
??	not yet determined

Some definitions for the next section

Declaration vs. Definition

By **declaring** an entity we inform the compiler about the ***structure and the behavior*** of the entity itself and bind unique name to it.
By **defining** an entity we instruct the compiler to generate code to **perform memory allocation** for that entity

The scope of a variable

The scope of a variable is a portion of the source code in which that variable is visible

Extent of a Variable

The *extent* (or *lifetime*) describes when in a program's execution a variable has a value. A variable can exist (extent) but be not visible (scope). I.e., global variables hidden by homonymous local variable in a scope.

Blocks

A block is a portion of code enclosed between two special symbols, which mark the beginning and the end of the block

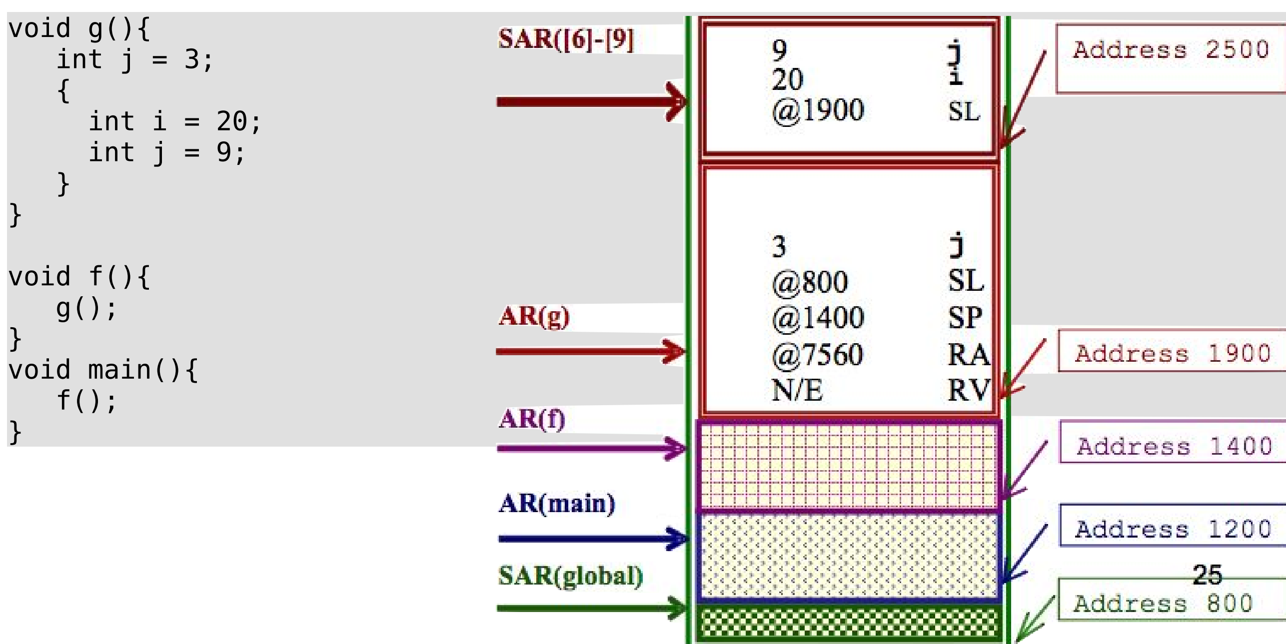
Scope Activation Record (SAR)

For languages that support block, we must extend our model. The **Scope Activation Record (SAR)** is put every time a *new block* is *encountered* in the program flow
The SAR of a block is *analogous to the AR* of a method call, and it holds all the necessary information for visibility rules implementation.
ARs are also SARs, because the body of a method is a block itself.

SARs contain two different kinds of information:

- **Local variables** (declared inside the block)
- **The Static Link SL** (a.k.a SAR link), a *pointer* to the *SAR* of the immediate *enclosing block*; used to access local variables of outer blocks from the current block.

Example on SAR



Role of SL

Each time a variable is used in a scope, but there is no definition of such variable in such scope, the system uses the SL to **reach out for the next enclosing scope** to **find** that **variable**. If it is not there, the SL is used to reach the next enclosing scope, and so on, recursively; until reaching the global scope

SL is totally different from SP:

- **SP** is used to track method instances, and therefore, properly **manage memory**
- **SL** is used **at run-time**, to implement the **language visibility rules**, which are enforced at compile-time, i.e. statically (that's why SL is called *static link)*

Elements of Dynamic Memory Allocation and Handling

Dynamic Vs. Static memory allocation

Stack-based variables have their extent determined by their scope, so the former is constrained by the structure of the code at compile-time .
Sometimes there is a need for the variables with unconstrained extent in order to cope with problems where lifetime of a variable can only be known at run-time.
In this case heap-based variables, whose extent is strictly under control of the programmer, are used. The programmer can create such a variable any time and ask the system to dispose it when it is no longer needed.

Dynamic Memory Scope and Extent

In majority of programming languages, the scope of an entity allocated on the heap is the union of the scopes of all the variables referring to it.
Usually, the extent of dynamic data starts when they are created and lasts until they are destroyed or until the program terminates.

Accessing dynamic memory

Dynamic allocated entities have no name, thus an alternative way of referring to them is required. This is achieved by using special kind of named variables (regular, stack-based)
In Java there are reference variables, whereas in C++ there are both reference and pointer variables (which are conceptually the same).

Object-Oriented Memory Management

Classes

A class represents the general properties and the structure shared by a group of entities: the objects of that class. Classes are units of data abstraction and units of interaction (objects interacting).

Objects

Objects are instances of classes (classes in action). Objects of the same class share the same general structure and behavior.

Object instantiation

The creation of a new object of a given class is called *instantiation*.

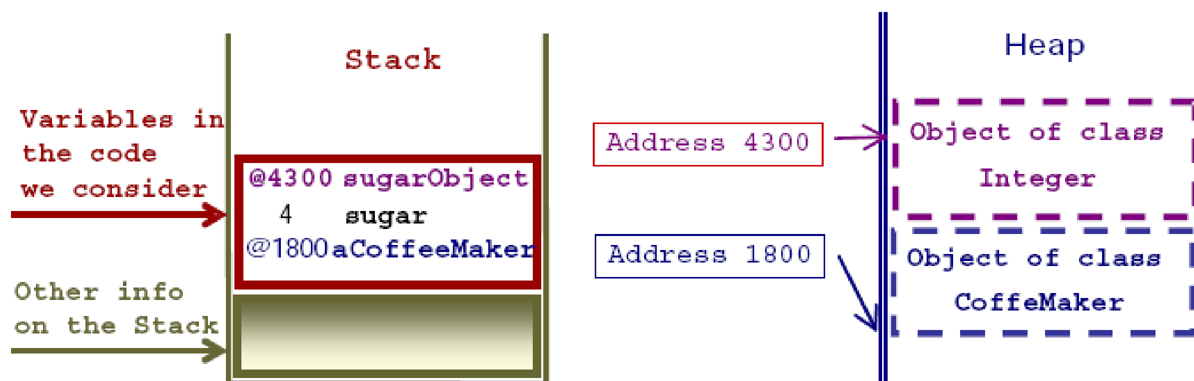
In Java objects are created (allocated) using the keyword new. References to objects are usually initialized with the value null. Objects can only be instantiated in heap.

In C++ classes are truly user defined types. Objects are treated as any other variable and are allocated:

- on the stack, as regular local variables
- on on the heap, like in Java

Objects in Memory (Java):

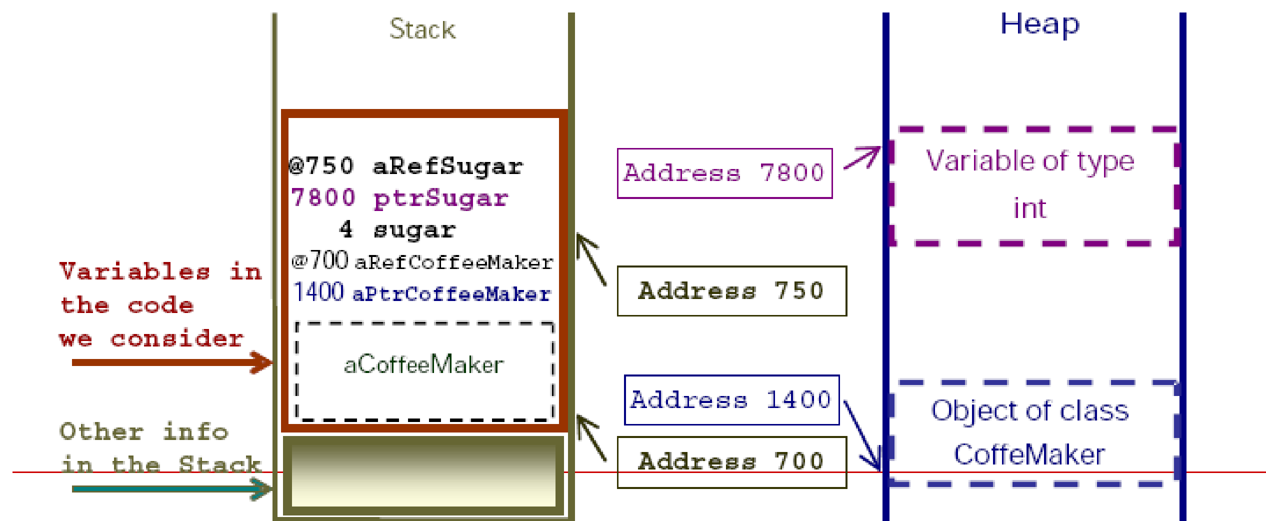
```
public class CoffeeMaker {
      public CoffeeMaker(){}
}
..
..
CoffeeMaker aCoffeeMaker;
aCoffeeMaker = new CoffeeMaker();
int sugar = 4;
Integer sugarObject = new Integer(3);
```



Java objects can be accessed just through reference variables, that hold the address of objects in heap.

Objects in Memory (C++)

```
class CoffeeMaker {
      public:
            CoffeeMaker();
            virtual ~CoffeeMaker();
} ;
..
..
CoffeeMaker aCoffeeMaker;
CoffeeMaker *aPtrCoffeeMaker = new CoffeeMaker();
CoffeeMaker &aRefCoffeeMaker = aCoffeeMaker;
aRefCoffeeMaker = *aPtrCoffeeMaker; // dangerous!
int sugar = 4;
int *ptrSuger = new int;
int &aRefSugar = sugar;
```



In C++ objects of type C can be accessed through pointers of type C*
or references of type C*. Here are listed the main differences between heap and stack in C++[v]:

Stack	Heap
Its size is determined at compile-time	Size determined at run-time
Therefore, it is **less expensive** and **quick**	Therefore, it is **more expensive** and **slower** than stack
The **preferred way** to store objects and variables	To be used **only if needed**: the amount of memory needed is variable and unknown, and may increase rapidly
There is an **AUTOMATIC CLEANUP** of objects when they go out of their scope	Objects **STAY IN MEMORY** even when you don't use them anymore
Programmers don't have to bother to free resources	Therefore, **programmers HAVE TO CLEAN memory** manually

Issues for objects in memory (Java)

Objects with no references pointing to them are considered eligible for automatic garbage collection by the system, which runs periodically and performs the real destruction of the objects. GC is not directly under control of the programmer.

Issues for objects in memory (C++)

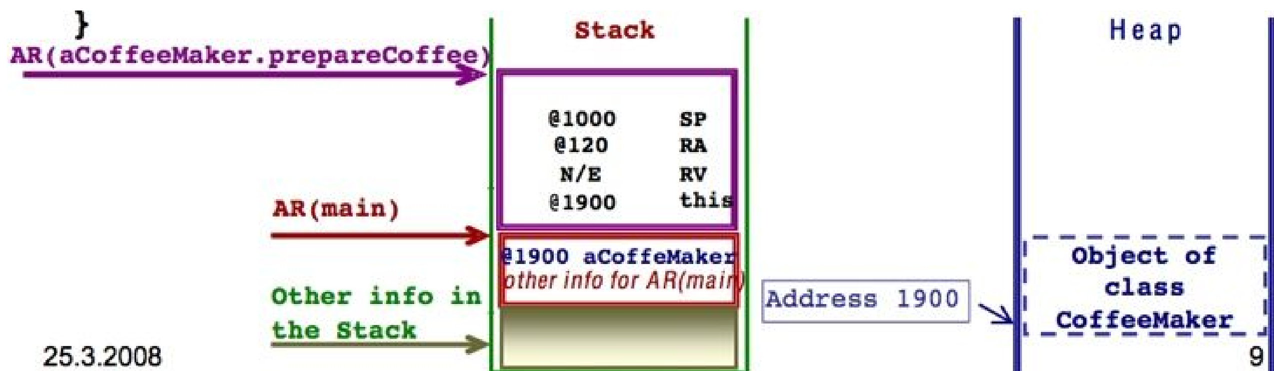
After an object has been created on the heap (with the *new* directive) it survives until someone destroys it explicitly using the *delete* directive.
This could lead to memory leaks in C++: if the programmer forgets to delete objects no longer needed, they remain on the heap as wasted space
On the other hand, local objects, which are allocated on the stack, always have a well-defined lifetime (as any other local variable): such objects are automatically deleted when they go out of scope.

Methods

The task of a program is accomplished via the iteration of objects, based on the exchange of *messages*. Upon reception of a message, an object replies with a method.
When invoking a method, there is the need to trace which object has invoked it. For this reason, we add “*this*” in the AR of the method.

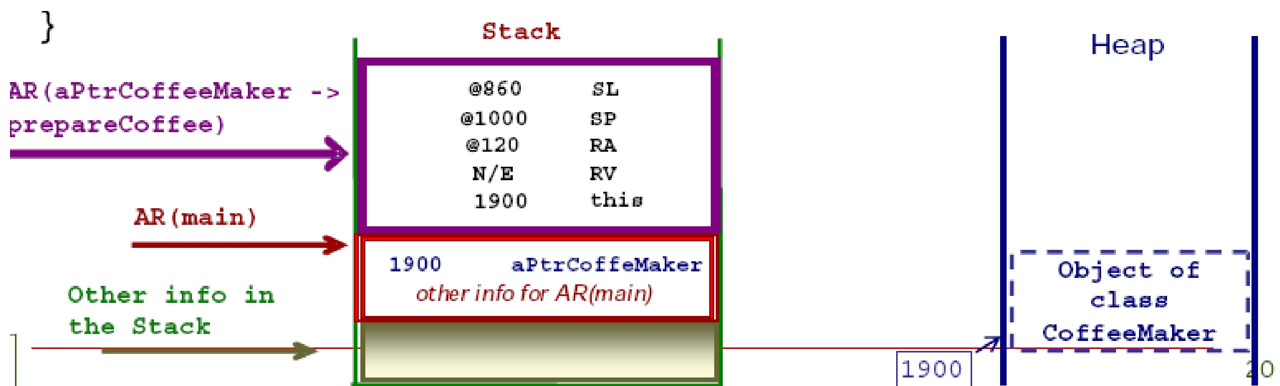
Methods (Java)

```
public class CoffeeMaker {
	public void prepareCoffee() {
	/*Prepare the coffee with a standard amount of sugar*/
	}
	public void prepareCoffeeSweet(int sugarAm){
	/* Prepare the coffee with sugarAm(ount) of sugar */
	}
	void main(...) {
		CoffeeMaker aCoffeeMaker;
		aCoffeeMaker = new CoffeeMaker();
		aCoffeeMaker.prepareCoffee();
	}
}
```



Methods (C++)

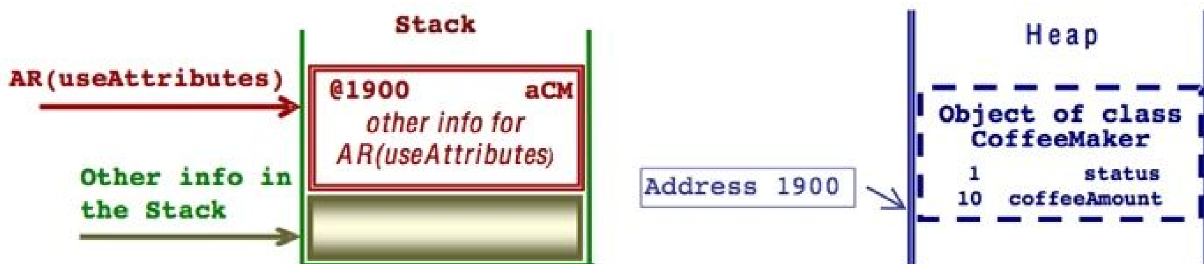
```
class CoffeeMaker {
      public:
            void prepareCoffee() {
            /*Prepare the coffee with a standard amount of sugar*/
            }
            void prepareCoffeeSweet(int sugarAm){
            /* Prepare the coffee with a specified amount of sugar */
            }
      void main(...) {
            CoffeeMaker *aPtrCoffeeMaker;
            aPtrCoffeeMaker = new CoffeeMaker;
            aPtrCoffeeMaker-> prepareCoffee();
      }
}
```



Attributes

To perform useful operations, objects need to carry values. This job is done via attributes. Let's see how they works:

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



The null value (Java)

Null is a special object reference used to signify that a pointer intentionally does not point to (or refer to) an object.
If a program calls an object through a reference not yet initialized, an error occurs (in Java, an exception of type NullPointerException).
If we assign the null value to a reference variable (aCoffeeMaker = null;), the object referenced by the variable will be released and destroyed by the Garbage Collector.

The NULL value (C++)

In C++, NULL is a preprocessor macro used to generate 0. 0 in a pointer context are converted into null pointers[vi] at compile time. Null pointers are not the address of any object[vii].

Parameters

A **formal parameter** is of the form f(x), used to define a function, an **actual parameter** is of the form f(3), used when the function is called.

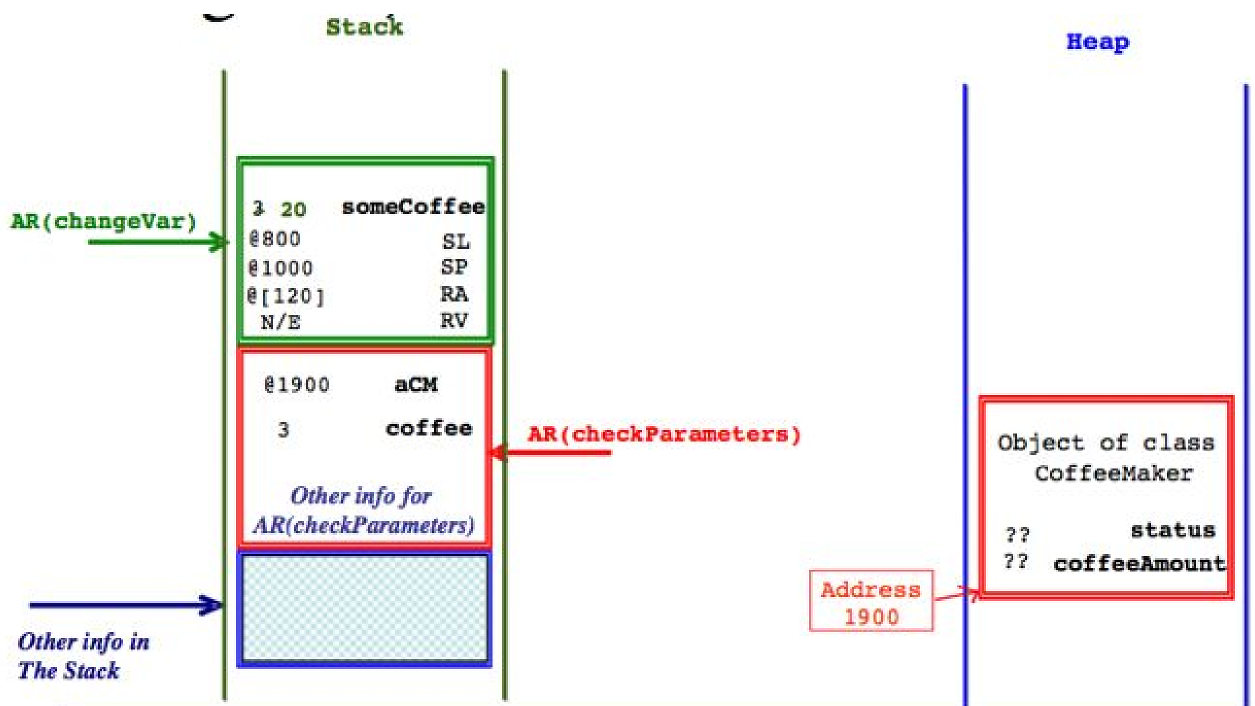
If we *pass* parameters by **value**, the value is copied into the activation record of the called method. This is safe, a formal parameter is an independent copy of an actual parameter.
If we *pass* parameters by **reference** (an alias to the object), the reference is copied into the activation record of the called method.

Parameter Passing (Java)

Parameters in Java are always passed by values. *Only variables are passed as parameters.*

Example of parameters passing (Java)

```
public void changeVar(int someCoffee) {
      someCoffee = 20;
}
public void changeObj(CoffeeMaker cm){
      cm.coffeeAmount = 25;
}
void checkParameters() {
      int coffee = 3;
      CoffeeMaker aCM = new CoffeeMaker();
      changeVar(coffee);
      changeObj(aCM);
}
```



Before the return of changeVar();

When the functions are called, the values of the actual parameters are copied onto the formals.
Therefore, someCoffee gets 3 from coffee and cm gets @1900 from aCM
Then, within changeVar, someCoffee becomes 20. Such modification are local and has no effect on the value of coffee.

Example of parameters passing (Java), continued

```
public void changeVar(int someCoffee) {
      someCoffee = 20;
}
public void changeObj(CoffeeMaker cm){
      cm.coffeeAmount = 25;
}
void checkParameters() {
      int coffee = 3;
      CoffeeMaker aCM = new CoffeeMaker();
      changeVar(coffee);
      changeObj(aCM);
}
```

Before the return of changeObj();

Within changeObj, we use the address 1900 hold in cm to identify our target object, the same referenced by aCM.
We modify our target object assigning 25 to coffeeAmount.
Such modification is global,as there is one object at location 1900.

Parameter Passing (C++)

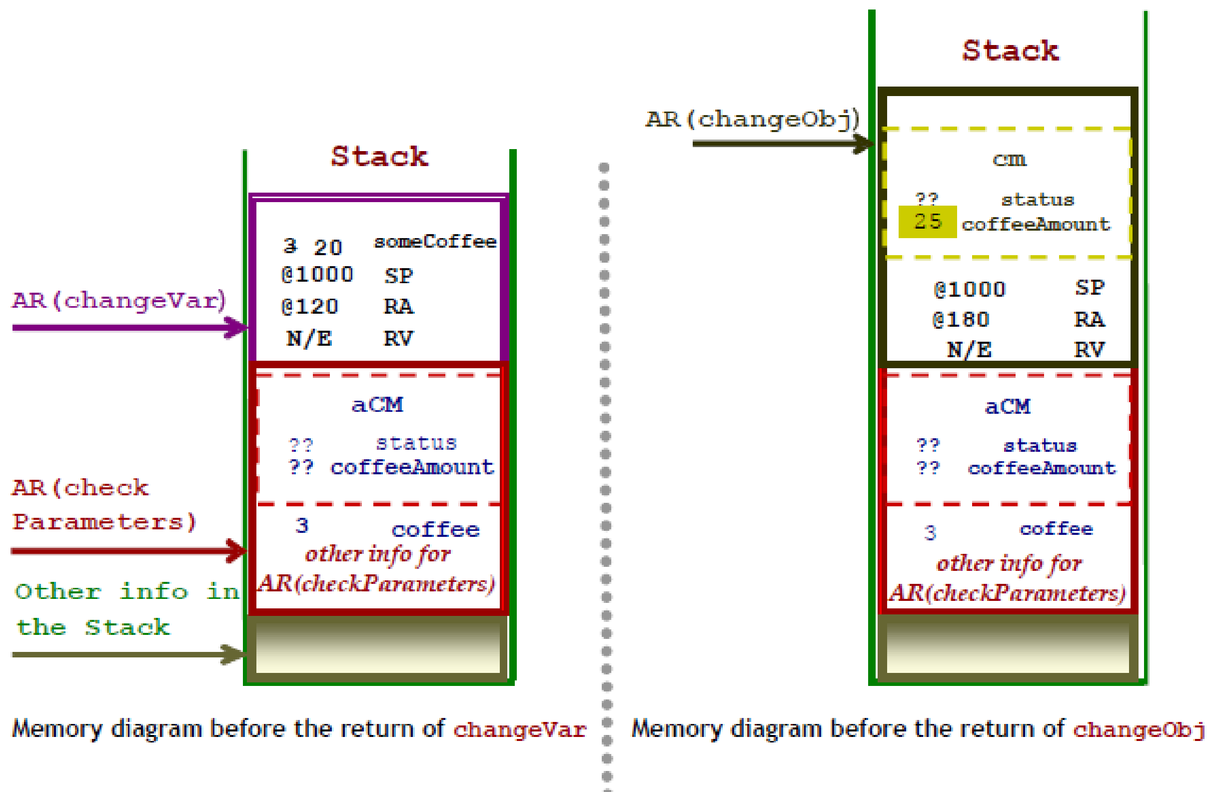
In C++, parameters can be passed either by value or by reference.

When a parameter is passed by value, a copy of the actual parameter is done onto the formal. Objects are also copied.

Passing parameters by reference means that instead of copying the actual parameter into the formal, the formal becomes a reference to the actual. Both the formal and the actual parameter refer to the same object.

Example of parameters passing by value (C++)

```
void changeVar(int someCoffee){
      someCoffee = 20;
}
void changeObj(CoffeeMaker cm){
      cm.coffeeAmount = 25;
}
void checkParameters() {
      int coffee = 3;
      CoffeeMaker aCM;
      changeVar(coffee);
      changeObj(aCM);
}
```

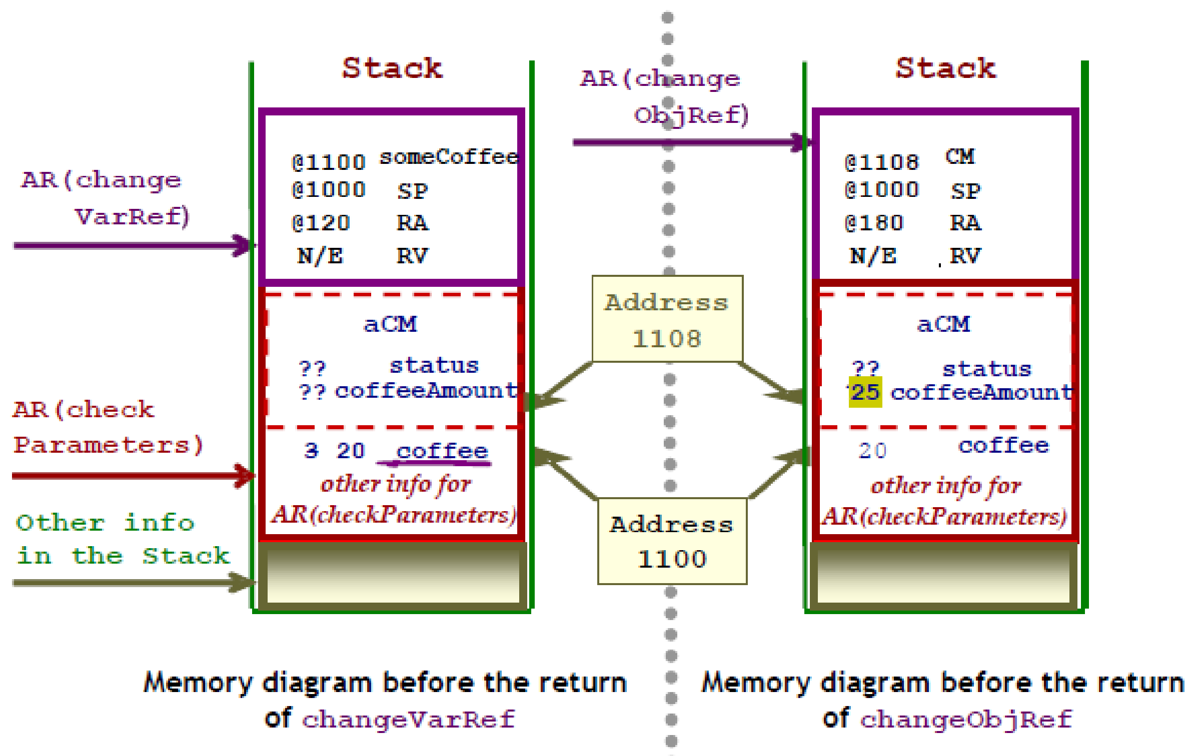


Memory diagram before the return of changeVar

Memory diagram before the return of changeObj

Example of parameters passing by reference (C++)

```
void changeVarRef(int &someCoffee){
      someCoffee = 20;
}
void changeObjRef(CoffeeMaker &CM){
      CM.coffeeAmount = 25;
}
void checkParameters() {
      int coffee = 3;
      CoffeeMaker aCM;
      changeVarRef(coffee);
      changeObjRef(aCM);
}
```



Memory diagram before the return of changeVarRef

Memory diagram before the return of changeObjRef

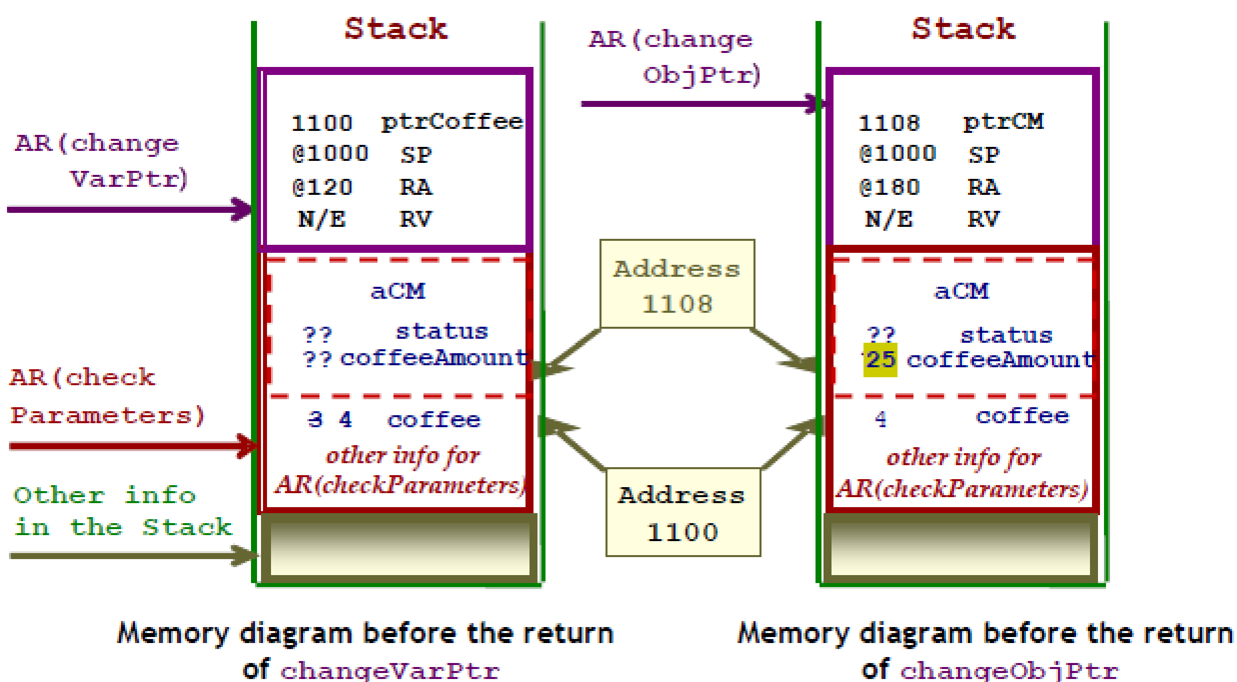
Pointers vs. Parameters (C++)

There are many discussions about the differences between pointers and references in C++. Here is a table taken from one blog article of mine's[viii]:

Reference	Pointer
is an object which **IS AN ALIAS** for another object	is an object that **CONTAINS THE ADRRESS IN MEMORY** of another object
the preferred way of undirectly access objects	you should use it just if you really need it, as it lets you to work in a lower level than a reference does
keeps your code clear	the code is less clear but still understandable
it must be initialized when created	you don't have to initialize it when declared
it references to the one object and only that one, therefore you **can not modify the address referenced**	because it contains an address, it can point to many different objects during lifetime. The **address can be manipulated**
when used, the **address is dereferenced** without using any particular operator	the **address must be dereferenced** using the * operator

Previous example using pointers (C++)

```
void changeVarPtr(int *ptrCoffee){
      (*ptrCoffee)++;
}
void changeObjPtr(CoffeeMaker *ptrCM){
      ptrCM->coffeeAmount = 25;
      // ptrCM->coffeeAmount == (*ptrCM).coffeeAmount
}
void checkParameters() {
      int coffee = 3;
      CoffeeMaker aCM;
      changeVarPtr(&coffee);
      changeObjPtr(&aCM);
}
```



Memory diagram before the return of changeVarPtr

Memory diagram before the return of changeObjPtr

Constructors

A **constructor** in a class is a special block of statements *called when an object is created.*

In Java, it is dynamically constructed on the heap through the keyword “new” and may call other constructors using *this*.
In C++ it may be either on the stack or on the heap, depending whether we are allocating the object on the heap or on the stack.

The constructor does not return a type.

In our model we can consider a constructor like any other object method. If we model a code snippet that contains constructors, we must simply remember to represent a constructor's call after the object is created.

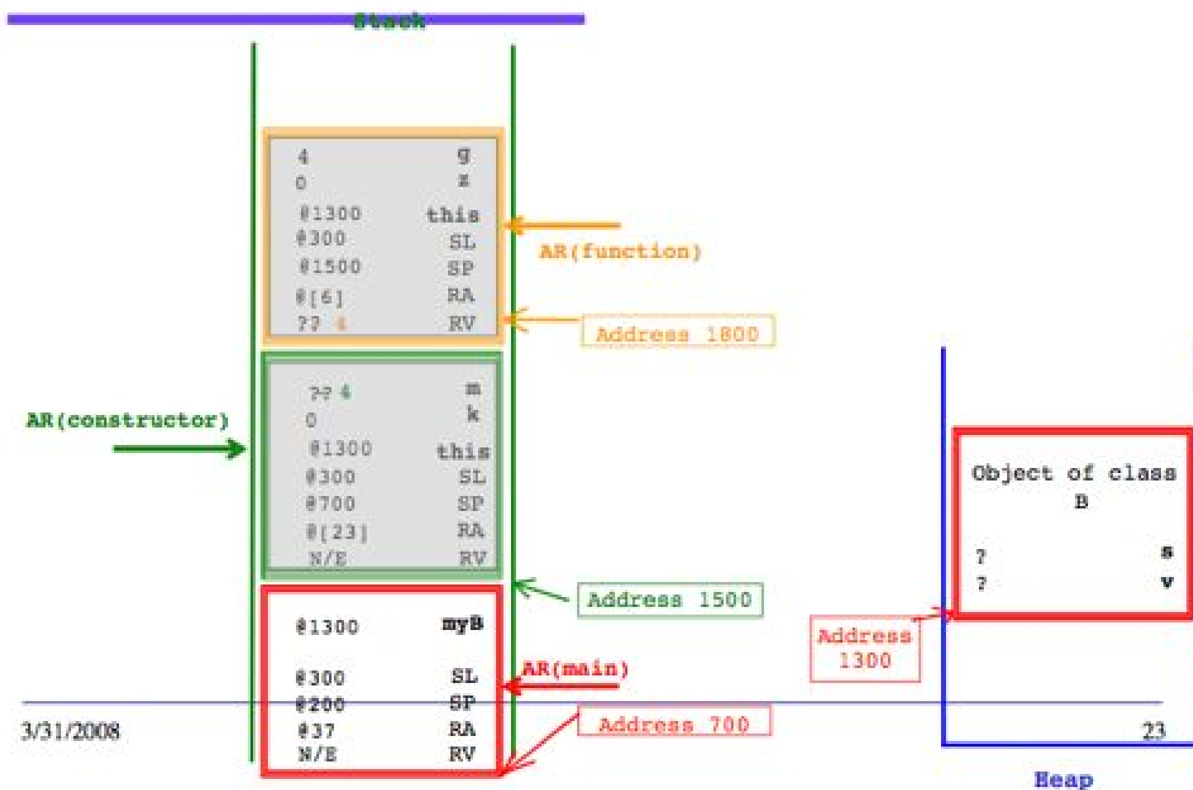
Inline initialization

Java allows the specification of default values of the attributes on the line of their declaration. So when an object is created (with the *new* operator), *first the inline initialization of the values is performed*. Then the constructor is called.

In C++ it is not possible to specify the value of the attributes in the line of their declaration. Attributes may have default values defined in the constructor of the object

A constructor's call (Java)

```
public class B {
      int v;
      int s;
      public B(){
            int k = 0;
            int m = function(k);
      }
      private int function(int z){
            int g = 4;
            return  g+z;
      }
      public static void main (…){
            B myB = new B();
      }
}
```



We don't need an example for C++, as things work in the same way. We must be careful whether the object is instantiated on the stack or on the heap. In the first case, the AR of the constructor will contain a reference to the address in memory of the object on the stack.
In the second case, it will contain a reference to the address of the object on the heap.
In both cases, the reference is of the form this @xxxx

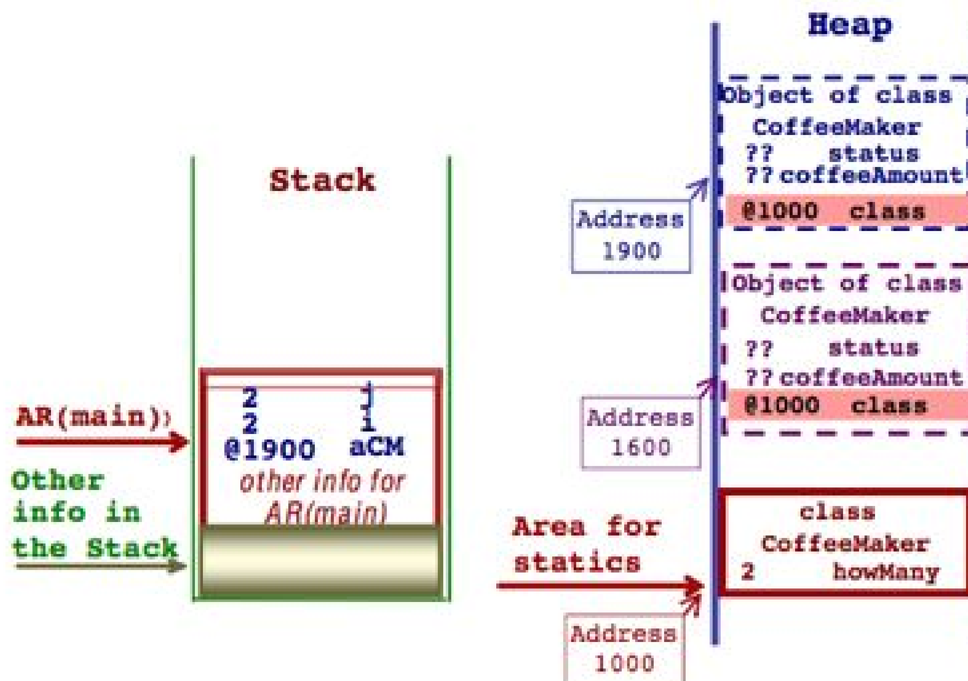
Class attributes

Class attributes are attributes that are shared between all the instances of a class. In both Java and C++ they are known as **static attributes**.
Static attributes are stored in a special area of the heap called “static area”
The extent is from the beginning of the computation to its end, while the scope is the same of the class they belong.
Class attributes cannot be stored in an object, as they are shared among all objects that are instances of the same class

Example of class attributes (Java)

```
public class CoffeeMaker {
...
public int coffeeAmount;
      public int status;
      public static int howMany = 0;

public CoffeeMaker() {
      howMany++;
}
...
public static void main(String argv[]){
      CoffeeMaker aCM;
      aCM = new CoffeeMaker();
      aCM = new CoffeeMaker();
      int i =aCM.howMany;
      int j =CoffeeMaker.howMany;
}
...
```

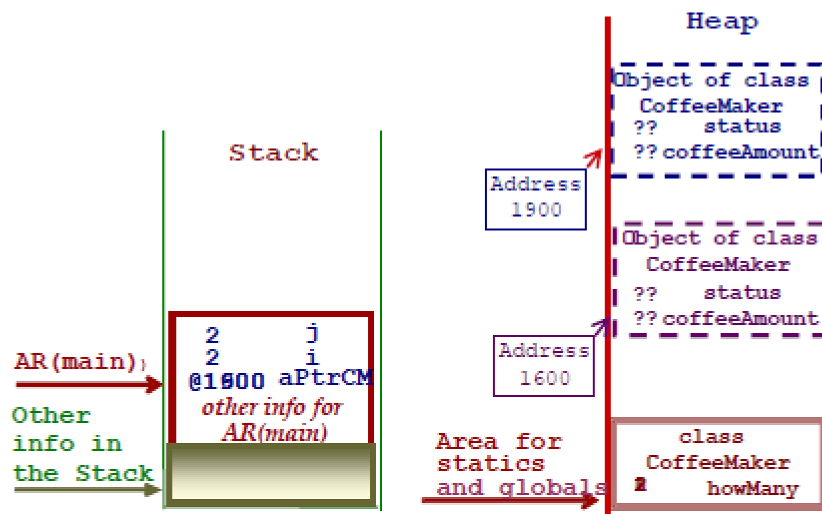


Example of class attributes (C++)

```
class CoffeeMaker {
...
   public:
      int coffeeAmount;
      int status;
      static int howMany;
      CoffeeMaker() {
            howMany++;
      }
...
};

int CoffeeMaker::howMany = 0;

int main(){
      CoffeeMaker *aPtrCM;
      aPtrCM = new CoffeeMaker;
      aPtrCM = new CoffeeMaker;
      int i= aptrCM->howMany;
      int j= CoffeeMaker::howMany;
}
```



Class Methods

Class Methods are methods that can be invoked by a class using the keyword **static**.
These methods are associated directly with the class.
Static method is used in most cases when there is stand alone utility method that should be made available without requiring the overhead of instantiation. A class method cannot access object attributes. It does not require a reference to “this” in its AR.

Example of Stack/Heap Diagrams in Java:

Code

```
[1] public class A {
[2]   static int a = 2;
[3]   int b = 3;
[4] public A(){
[5]      a=0;
[6]      b=0;
[7]    }
[8] public int call(int z){
[9]   int f = z*10;
[10]   {
[11]      int f = 10;
[12]      int a = 10 + f;
[13]      f++;
[14]   }
[15]   return f;
[16] }
[17] public int getOperand(int b){
[18]    int c = this.call(b);
[19]    return c;
[20]}
[21] public static int getCode(){
[22]    return a;
[23]}
[24] public static void main (){
[25]   int x = 1;
[26]   int w = x*3;
[27]   {
[28]      A firstA = new A();
[29]      x = A.getCode();
[30]      x++;
[31]      int w = 5;
[32]      w+=3;
[33]   }
[34]   A secondA = new A();
[35]   int y = secondA.getOperand(5);
[36]   w = secondA.call(4);
[37] }
[38]}
```

Global is at address 10
Main returns at address 20
Draw the memory model once execution reaches line [37]
For all exercises:
The AR of the main method starts at address 100 and the increment for each new activation
record is 100.

Stack Diagram

(S)ARs	(S)ARs values	(S)ARs variables	**ADDRESS**
	20	a	
	10 11	f	
	@200	SL	
SAR [10]-[14]	@200	SP	**300**
	40	f	
	4	z	
	@12000	this	
	@10	SL	
	@100	SP	
	@[36]	RA	
AR A::call(4)	?? 40	RV	**200**
	20	a	
	10 11	f	
	@300	SL	
SAR [10]-[14]	@300	SP	**400**
	50	f	
	5	z	
	@12000	this	
	@10	SL	
	@200	SP	
	@[18]	RA	
AR A::call(5)	?? 50	RV	**300**
	? 50	c	
	5	b	
	@12000	this	
	@10	SL	
	@100	SP	
	@[35]	RA	
AR A::getOperand(5)	?? 50	RV	**200**
	@12000	this	
	@10	SL	
	@100	SP	
	@[34]	RA	
AR A::A()	N/E	RV	**200**
	@10	SL	
	@300	SP	
	@[29]	RA	
AR A::getCode()	? 0	RV	**300**
	@11000	this	
	@10	SL	
	@200	SP	
	@[28]	RA	
AR A::A()	N/E	RV	**300**
	5 8	w	
	@11000	firstA	

	@100	SL	
SAR [27]-[33]	@100	SP	**200**
	7 50	y	
	@12000	secondA	
	3 40	w	
	1 0 1	x	
	@10	SL	
	@10	SP	
	@20	RA	
AR A::main()	N/E	RV	**100**
Global			**10**

Heap Diagram

HEAP ITEMS	HEAP values	HEAP variables	**ADDRESS**
	3 0	b	
OBJECT CLASS A	@10000	Class	**12000**
	3 0	b	
OBJECT CLASS A	@10000	Class	**11000**
STATIC AREA CLASS A	2 0 0	a	**10000**

i http://en.wikipedia.org/wiki/Dynamic_memory_allocation
ii http://en.wikipedia.org/wiki/Call_stack
iii http://en.wikipedia.org/wiki/Process_(computing)#Representation
iv http://www.gotw.ca/gotw/009.htm
v http://bd-things.net/heap-vs-stack-in-c/
vi http://en.wikipedia.org/wiki/Pointer_(computing)#The_null_pointer
vii http://blog.xploiter.com/programming-related-content/cc-null-pointers/
viii http://bd-things.net/reference-vs-pointer/