WEEK 4

Memory allocation in java;
Garbage collection: concept, working, types,
advantages finalize () method;

Memory allocation in java

In Java, memory management is the process of allocation and de-allocation of objects, called Memory management. Java does memory management automatically. Java uses an automatic memory management system called a garbage collector. Thus, we are not required to implement memory management logic in our application.

Java memory management divides into two major parts:

1. JVM Memory Structure
2. Garbage Collector

JVM Memory Structure

JVM creates various run time data areas in a heap. These areas are used during the program execution. The memory areas are destroyed when JVM exits, whereas the data areas are destroyed when the thread exits.

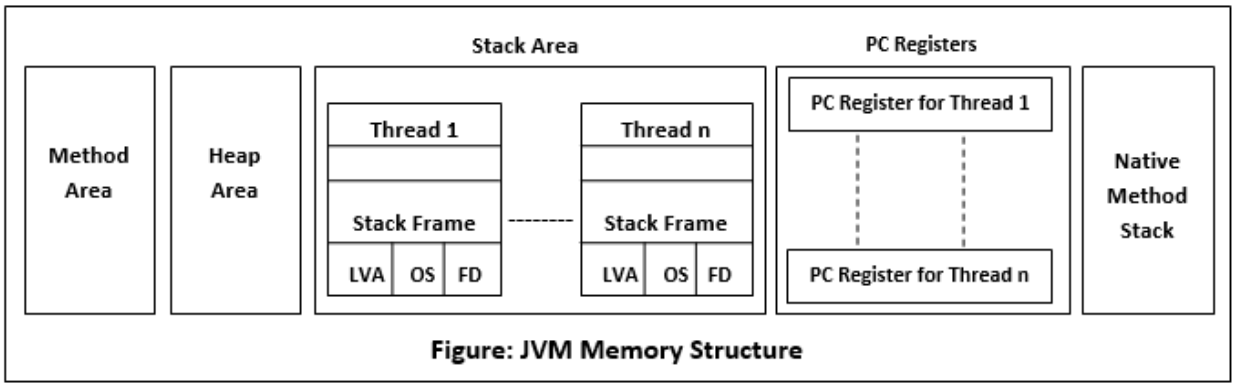


Figure: JVM Memory Structure

Method Area

Method Area is a part of the heap memory which is shared among all the threads. It creates when the JVM starts up. It is used to store class structure, superclass name, interface name, and constructors.

The JVM stores the following kinds of information in the method area:

- A Fully qualified name of a type (ex: String)

- The type's modifiers
- Type's direct superclass name
- A structured list of the fully qualified names of super interfaces.

Heap Area

Heap stores the actual objects. It creates when the JVM starts up. The user can control the heap if needed. It can be of fixed or dynamic size. When you use a new keyword, the JVM creates an instance for the object in a heap. While the reference of that object stores in the stack. There exists only one heap for each running JVM process. When heap becomes full, the garbage is collected. For example:

StringBuilder sb= new StringBuilder();

The above statement creates an object of the StringBuilder class. The object allocates to the heap, and the reference sb allocates to stack. **Heap** is divided into the following parts:

- Young generation
- Survivor space
- Old generation
- Permanent generation
- Code Cache

Stack Area

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. 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.

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.
- 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).
- 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.
- 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.

Native Method Stack

It is also known as C stack. It is a stack for native code written in a language other than Java. Java Native Interface (JNI) calls the native stack. The performance of the native stack depends on the OS.

PC Registers

Each thread has a Program Counter (PC) register associated with it. PC register stores the return address or a native pointer. It also contains the address of the JVM instructions currently being executed.

Stack memory

Stack memory is the space allocated for a process where all the function calls, primitive data types like int, double, etc., and local and reference variables of the functions are stored. Stack memory is always accessed in a Last-In-First-Out (LIFO) manner. In the stack memory, a new memory block is created for every method that is executed. All the primitive variables and references to objects inside the method are stored in this memory block. When the method completes its execution, the memory block is cleared from the stack memory and the stack memory is available for use. The values in the stack exist for as long as the function that created them is running. The size of the stack memory is fixed and cannot grow or shrink once created.

Example of Stack Memory in Java

```
public class Main
{
	public static int addOne(int input)
	{
		return input + 1;
	}

	public static int addTwo(int input)
	{
		return input + 2;
	}

	public static void main(String[] args)
	{
		int x = 0;

		x = addOne(x);
		x = addTwo(x);
	}
}
```

Heap memory

Heap memory is used to store the objects that are created during the execution of a Java program. The reference to the objects that are created is stored in stack memory. Heap follows dynamic memory allocation (memory is allocated during execution or runtime) and provides random access, unlike stack, which follows Last-In-First-Out (LIFO) order. The size of heap memory is large when compared to stack. The unused objects in the heap memory are cleared automatically by the Garbage Collector.

The heap memory can be divided into three parts

1) New or Young Generation
2) Old or Tenured Generation
3) Permanent Generation

Example of Stack Memory in Java

```
import java.util.ArrayList;
import java.util.List;

public class HeapMemory
{
	public static void main(String[] args)
	{
		int x = 10;

		List < Integer > list = new ArrayList < > ();
		list.add(1);
		list.add(2);
		list.add(3);
	}
}
```

Differences between Stack vs Heap

Property	Stack Memory	Heap Memory
Size	The size of stack memory is **smaller**	The size of heap memory is **larger**
Order	Stack memory is accessed in **Last-In-First-Out (LIFO)** manner	Heap memory is **dynamically allocated** and does not follow any order
Speed	Access to stack memory is **faster** because of Last-In-First-Out (LIFO) ordering	Access to heap memory is **slower** because it does not follow any order and is allocated dynamically
Resizing	Resizing of variables is **not allowed** in stack	Resizing of variables is **allowed** in a heap
Allocation	Memory is allocated and deallocated automatically when a method starts and completes its execution respectively	Memory is allocated when objects are created and deallocated by the garbage collector when they are no longer in use
Storage	Local variables and object references inside the function are stored in stack	The newly created objects and the JRE classes are stored in a heap
Exception	**StackOverflowError** is thrown when there is no more space left in the stack for new method calls	**OutOfMemoryError** is thrown when there is no space left in a heap to allocate new objects
Thread Safety	Each thread is allocated with a new stack, and it is **thread-safe**	Heap memory is shared across all threads, and it is **not thread-safe**

Garbage Collector:

Garbage Collector Overview

When a program executes in Java, it uses memory in different ways. The heap is a part of memory where objects live. It's the only part of memory that involved in the garbage collection process. It is also known as garbage collectible heap.

All the garbage collection makes sure that the heap has as much free space as possible. The function of the garbage collector is to find and delete the objects that cannot be reached.

Object Allocation

When an object allocates, the JRockit JVM checks the size of the object. It distinguishes between small and large objects. The small and large size depends on the JVM version, heap size, garbage collection strategy, and platform used. The size of an object is usually between 2 to 128 KB.

The small objects are stored in Thread Local Area (TLA) which is a free chunk of the heap. TLA does not synchronize with other threads. When TLA becomes full, it requests for new TLA.

On the other hand, large objects that do not fit inside the TLA directly allocated into the heap. If a thread is using the young space, it directly stored in the old space. The large object requires more synchronization between the threads.

What does Java Garbage Collector?

- JVM controls the garbage collector. JVM decides when to perform the garbage collection.
- We can also request to the JVM to run the garbage collector. But there is no guarantee under any conditions that the JVM will comply.
- JVM runs the garbage collector if it senses that memory is running low.
- When Java program request for the garbage collector, the JVM usually grants the request in short order. It does not make sure that the requests accept.

The point to understand is that "when an object becomes eligible for garbage collection?"

Every Java program has more than one thread. Each thread has its execution stack. There is a thread to run in Java program that is a main() method. Now we can say that an object is eligible for garbage collection when no live thread can access it. The garbage collector considers that object as eligible for deletion. If a program has a reference variable that refers to an object, that reference variable available to live thread, this object is called reachable.

Here a question arises that "Can a Java application run out of memory?"

The answer is yes. The garbage collection system attempts to objects from the memory when they are not in use. Though, if you are maintaining many live objects, garbage collection does not guarantee that there is enough memory. Only available memory will be managed effectively.

Types of Garbage Collection(GC)

There are five types of garbage collection are as follows:

1. **Serial GC:** It uses the mark and sweeps approach for young and old generations, which is minor and major GC.
2. **Parallel GC:** It is similar to serial GC except that, it spawns N (the number of CPU cores in the system) threads for young generation garbage collection.
3. **Parallel Old GC:** It is similar to parallel GC, except that it uses multiple threads for both generations.

4. **Concurrent Mark Sweep (CMS) Collector:** It does the garbage collection for the old generation. You can limit the number of threads in CMS collector using **XX:ParalleCMSThreads=JVM option**. It is also known as Concurrent Low Pause Collector.

5. **G1 Garbage Collector:** It introduced in Java 7. Its objective is to replace the CMS collector. It is a parallel, concurrent, and CMS collector. There is no young and old generation space. It divides the heap into several equal sized heaps. It first collects the regions with lesser live data.

Java finalize() Method

It is difficult for the programmer to forcefully execute the garbage collector to destroy the object. But Java provides an alternative way to do the same. The Java Object class provides the **finalize()** method that works the same as the destructor. The syntax of the finalize() method is as follows:

Syntax:

```
protected void finalize throws Throwable()
{
//resources to be close
}
```

It is not a destructor but it provides extra security. It ensures the use of external resources like closing the file, etc. before shutting down the program. We can call it by using the method itself or invoking the method **System.runFinalizersOnExit(true)**.

- It is a protected method of the Object class that is defined in the java.lang package.
- It can be called only once.
- We need to call the finalize() method explicitly if we want to override the method.
- The gc() is a method of JVM executed by the Garbage Collector. It invokes when the heap memory is full and requires more memory for new arriving objects.
- Except for the unchecked exceptions, the JVM ignores all the exceptions that occur by the finalize() method.

Example:

```
public class JavafinalizeExample1
{
    public static void main(String[] args)
   {
         JavafinalizeExample1 obj = new JavafinalizeExample1();
         System.out.println(obj.hashCode());
         obj = null;
          // calling garbage collector
```

```
        System.gc();
        System.out.println("end of garbage collection");

    }
    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Output:
2018699554
end of garbage collection
finalize method called

Week 4 Practical's

1) Example of Stack Memory in Java.

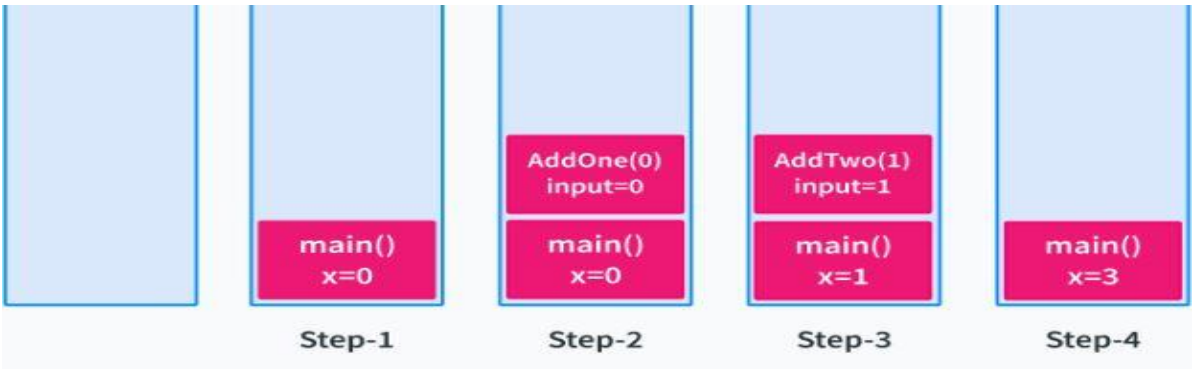
```
public class Main
{
    public static int addOne(int input)
    {
        return input + 1;
    }

    public static int addTwo(int input)
    {
        return input + 2;
    }

    public static void main(String[] args)
    {
        int x = 0;

        x = addOne(x);
        x = addTwo(x);
    }
}
```

Output:



2) Example of Heap Memory in Java.

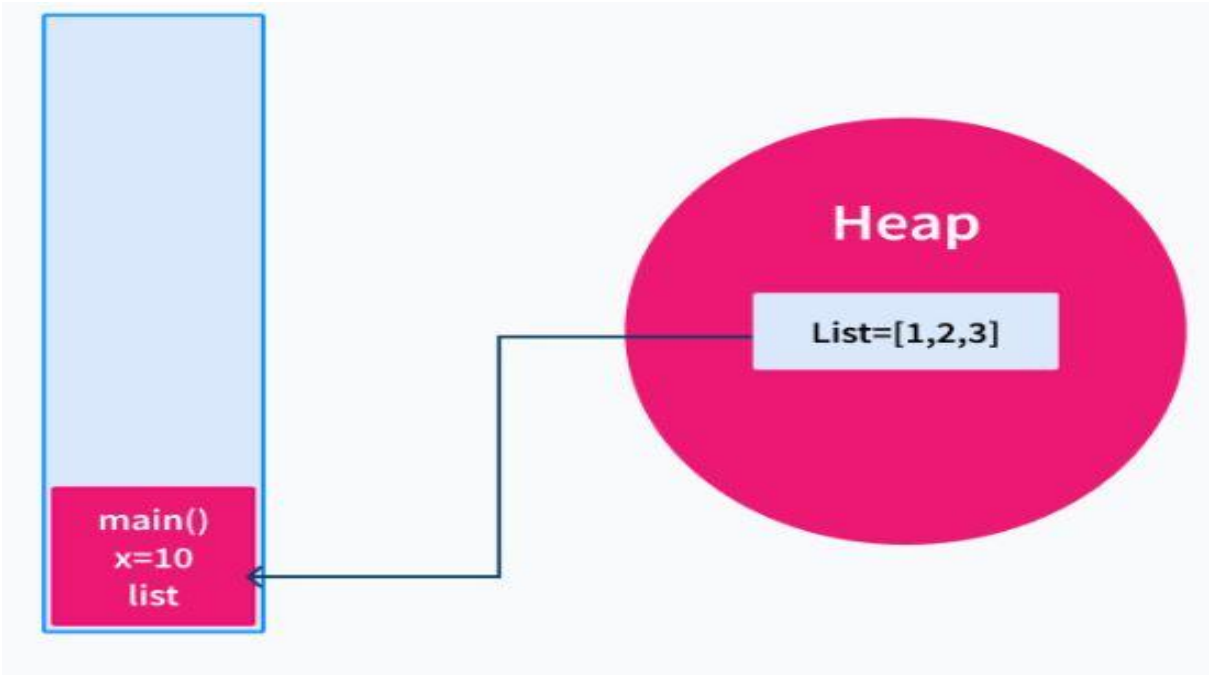
```
import java.util.ArrayList;
import java.util.List;

public class HeapMemory
{
	public static void main(String[] args)
	{
		int x = 10;

		List < Integer > list = new ArrayList < > ();
		list.add(1);
		list.add(2);
		list.add(3);
	}
}
```

Output:

In the above example, the variable x is allocated in the stack, whereas the object list is allocated in the heap. Only the reference to the list object is stored in a stack.


Heap
List=[1,2,3]
main()
x=10
list