**Index**

**1. OOPs Concepts**

**2. Data Types**

**3. Interfaces**

**3.1 Nested Interface**

**3.2 Marker Interface**

**3.3 Comparator Interface**

**4. Abstract Classes**

**5. Collection**

**6. Exception Handling**

**7. JDK vs JRE vs JVM**

**8. Variable Arguments (Varargs)**

**9. Java Multithreading**

**10. Abstract vs Interface**

**11. Java 8 Features**

**11.1 Functional Interfaces**

**11.2 Lambda Expressions**

**11.3 Stream API**

**11.4 Optional Class**

**11.5 Default and Static Methods in Interfaces**

**11.6 forEach Method**

**11.7 Method References**

**12. Comparable and Comparator**

**13. Jagged Arrays**

**14. Java Memory Management**

**15. How Java Objects are Stored in Memory**

**16. Stack vs Heap Memory Allocation**

**17. Java Virtual Machine (JVM) Stack Area**

**18. Types of Memory Areas Allocated by JVM**

**19. Garbage Collection**

**1.OOPs Concepts :**

**Java OOP (Object Oriented Programming) Concepts**

Object-Oriented Programming or Java OOPs concept refers to programming languages that use objects in programming. They use objects as a primary source to implement what is to happen in the code.

**Java Class**

A Class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

1. Modifiers: A class can be public or have default access.
2. Class name: The class name should begin with the initial setter capitalized by convention.
3. Body: The class body is surrounded by braces, { }.

**Java Object**

An Object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. State: It is represented by the attributes of an object. It also reflects the properties of an object.
2. Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. Identity: It is a unique name given to an object that enables it to interact with other objects.
4. Method: A method is a collection of statements that perform some specific tasks and return the result to the caller. A method can perform some specific tasks without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers.

**Pillars of Java OOPs Concepts**

* 1. **Abstraction in Java**

Abstraction in Java is the process of hiding the implementation details and only showing the essential functionality or features to the user. This helps simplify the system by focusing on what an object does rather than how it does it. The unnecessary details or complexities are not displayed to the user.

Television remote control is an excellent example of abstraction. It simplifies the interaction with a TV by hiding the complexity behind simple buttons and symbols, making it easy without needing to understand the technical details of how the TV functions.

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces. Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

* 1. **Encapsulation**

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared. Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

* 1. **Inheritance**

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using extends keyword. Inheritance is also known as “is-a” relationship.

Superclass: The class whose features are inherited is known as superclass (also known as base or parent class).

Subclass: The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

* 1. **Polymorphism**

It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called polymorphism.

**Types of Polymorphism**

**Polymorphism in Java is mainly of 2 types as mentioned below:**

1. **Method Overloading**
2. **Method Overriding**
3. Method Overloading: Also, known as compile-time polymorphism, is the concept of Polymorphism where more than one method share the same name with different signature (Parameters) in a class. The return type of these methods can or cannot be same.

Method overloading in Java is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding. In Method overloading compared to the parent argument, the child argument will get the highest priority

Different Ways of Method Overloading in Java

Changing the Number of Parameters.

Changing Data Types of the Arguments.

Changing the Order of the Parameters of Methods

1. Method Overriding: Also, known as run-time polymorphism, is the concept of Polymorphism where method in the child class has the same name, return-type and parameters as in parent class. The child class provides the implementation in the method already written.

Overriding in Java occurs when a subclass implements a method which is already defined in the superclass or Base Class. The method in the subclass must have the same signature as in the superclass. It allows the subclass to modify the inherited methods. Method overloading in Java is also known as Run-time Polymorphism, Dynamic Polymorphism, or Late binding.

**2.Java Data Types**

Data types in Java are of different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Java has two categories in which data types are segregated

1. Primitive Data Type: such as boolean, char, int, short, byte, long, float, and double. The Boolean with uppercase B is a wrapper class for the primitive data type boolean in Java.
2. Non-Primitive Data Type or Object Data type:

In Java, non-primitive data types, also known as reference data types, are used to store complex objects rather than simple values. Unlike primitive data types that store the actual values, reference data types store references or memory addresses that point to the location of the object in memory. This distinction is important because it affects how these data types are stored, passed, and manipulated in Java programs.

**A computer screen shot of a diagram

Description automatically generated**

**Primitive Data Types in Java**

**1. boolean Data Type**

The boolean data type represents a logical value that can be either true or false.

Syntax:

boolean booleanVar;

Size: Virtual machine dependent (typically 1 byte, 8 bits)

**2. byte Data Type :**

The byte data type is an 8-bit signed two’s complement integer. The byte data type is useful for saving memory in large arrays.

Syntax:

byte byteVar;

Size: 1 byte (8 bits)

**3. Short Data Type :**

The short data type is a 16-bit signed two’s complement integer. Similar to byte, a short is used when memory savings matter, especially in large arrays where space is constrained.

Syntax:

short shortVar;

Size: 2 bytes (16 bits)

**4. int Data Type :**

It is a 32-bit signed two’s complement integer.

Syntax:

int intVar;

Size: 4 bytes ( 32 bits )

**5. long Data Type :**

The long data type is a 64-bit signed two’s complement integer. It is used when an int is not large enough to hold a value, offering a much broader range.

Syntax:

long longVar;

Size: 8 bytes (64 bits)

**6. float Data Type :**

The float data type is a single-precision 32-bit IEEE 754 floating-point. Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers. The size of the float data type is 4 bytes (32 bits).

Syntax:

float floatVar;

**7. double Data Type:**

The double data type is a double-precision 64-bit IEEE 754 floating-point. For decimal values, this data type is generally the default choice. The size of the double data type is 8 bytes or 64 bits.

Syntax:

double doubleVar;

Size: 8 bytes (64 bits)

**8. char Data Type :**

The char data type is a single 16-bit Unicode character with the size of 2 bytes (16 bits).

Syntax:

char charVar;

Size: 2 bytes (16 bits)

**Non-Primitive (Reference) Data Types**

* 1. **Class**

One common non-primitive data type in Java is the class. Classes are used to create objects, which are instances of the class. A class defines the properties and behaviors of objects, including variables (fields) and methods. For example, you might create a Person class to represent a person, with variables for the person's name, age, and address, and methods to set and get these values.

* 1. **Interface**

Interfaces are another important non-primitive data type in Java. An interface defines a contract for what a class implementing the interface must provide, without specifying how it should be implemented. Interfaces are used to achieve abstraction and multiple inheritance in Java, allowing classes to be more flexible and reusable.

* 1. **Arrays**

Arrays are a fundamental non-primitive data type in Java that allow you to store multiple values of the same type in a single variable. Arrays have a fixed size, which is specified when the array is created, and can be accessed using an index. Arrays are commonly used to store lists of values or to represent matrices and other multi-dimensional data structures.

Array Declaration : type[] arrayName;

Create an Array : arrayName = new type[5];

Assigning values : arrayName[0] = value;

* 1. **Enum**

Java also includes other non-primitive data types, such as enums and collections. Enums are used to define a set of named constants, providing a way to represent a fixed set of values. Collections are a framework of classes and interfaces that provide dynamic data structures such as lists, sets, and maps, which can grow or shrink in size as needed.

**3.Interface in Java**

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java

Since Java 8, we can have default and static methods in an interface

interface {  
// declare constant fields  
// declare methods that abstract   
// default and static methods introduced in java 8 onwards  
}

**3.1.Nested Interface**

We can declare interfaces as members of a class or another interface. Such an interface is called a member interface or nested interface. Interfaces declared outside any class can have only public and default (package-private) access specifiers. In Java, nested interfaces (interfaces declared inside a class or another interface) can be declared with the public, protected, package-private (default), or private access specifiers.

**3.2.Marker Interface**

Marker Interface in Java is an empty interface means having no field or methods. Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

**3.3.Comparator Interface**

Comparator interface in Java is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of the same class. Following function compare obj1 with obj2.

**4.Abstract Classes**

Abstract Class in Java :

In Java, abstract class is declared with the abstract keyword. It may have both abstract and non-abstract methods (methods with bodies). An abstract is a Java modifier applicable for classes and methods in Java but not for variables.

Java abstract class is a class that cannot be instantiated by itself, it needs to be subclassed by another class to use its properties.

Abstract Methods: Abstract classes can have abstract methods, which are declared without a body. Subclasses must provide concrete implementations for these methods.

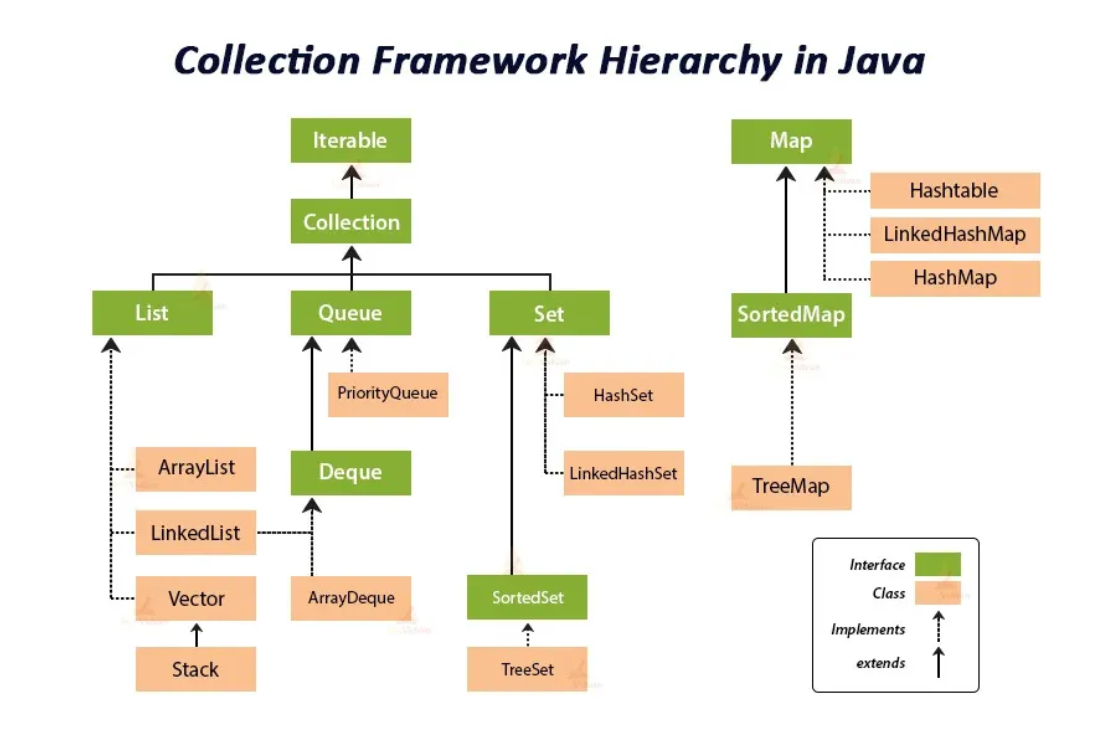
Concrete Methods: Abstract classes can also contain concrete methods with defined behavior. Subclasses inherit these methods along with their implementations.

**5.Collections**

**Collections in Java :**

Any group of individual objects that are represented as a single unit is known as a Java Collection of Objects.

In Java, the Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main “root” interfaces of Java collection classes.

****

**Iterator**

An Iterator in Java is an interface used to traverse elements in a Collection sequentially. It provides methods like hasNext(), next(), and remove() to loop through collections and perform manipulation. An Iterator is a part of the Java Collection Framework, and we can use it with collections like ArrayList, LinkedList, and other classes that implement the Collection interface.

**Types of Cursors in Java**

**There are 3 cursors in Java as mentioned below:**

* Iterator

1. Enumeration
2. ListIterator

**Java List Interface**

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values. List interface is implemented by the classes ArrayList, LinkedList, Vector and Stack.

**ArrayList**

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

**LinkedList**

In Java, a LinkedList is a class that implements the List interface and represents a linked list data structure. Unlike arrays, which store elements in contiguous memory locations, a linked list stores elements as nodes, where each node contains the element itself and a reference (or pointer) to the next node in the sequence.

This is because each element is stored in its own node, allowing for efficient insertion and deletion operations. However, accessing elements by index in a LinkedList is less efficient compared to arrays, as it requires traversing the list from the beginning or end to reach the desired element.

**Java Set Interface**

The Set Interface is present in java.util package and extends the [Collection interface](https://www.geeksforgeeks.org/collections-in-java-2/). It is an unordered collection of objects in which duplicate values cannot be stored. Allow one null value. Set interface is implemented by the classes HashSet, LinkedHashSet and TreeSet.

**HashSet**

HashSet in Java implements the Set interface of Collections Framework. It stores the elements by using a mechanism called hashing. It doesn’t allow the duplicates and allow one null value. It doesn’t maintain insertion order.

**LinkedHashSet**

LinkedHashSet in Java implements the Set interface of the Collection Framework. It combines the functionality of a HashSet with a LinkedList to maintain the insertion order of elements. It allows one null value.

**TreeSet**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order. Java TreeSet class doesn't allow null elements.

**Map Interface in Java**

In Java, Map Interface is present in the java.util package represents a mapping between a key and a value. Java Map interface is not a subtype of the Collection interface. Here keys are unique, but values can be duplicate. Allows one null key and multiple null values. Primary classes implement these interfaces HashMap, TreeMap, and LinkedHashMap.

**Map.Entry Interface**

Map.Entry is an inner interface of the Map interface which represents a single pair of key-value in a map. It additionally allows one to get a map and to iterate over its entries, convert the map to a set of keys/values, etc. It returns a collection-view of the map, whose elements are of this class. It provides methods to get keys and values**.**

**HashMap**

HashMap stores data in (key, value) pairs. Each key is associated with a value, and you can access the value by using the corresponding key. Allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. Duplicate Elements not allowed in HashMap, if you try to insert the duplicate key in HashMap, it will replace the element of the corresponding key**.**

**LinkedHashMap**

Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

**Hashtable**

Hashtable class, introduced as part of the Java Collections framework, implements a hash table that maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method. The java.util.Hashtable class is a class in Java that provides a key-value data structure, similar to the Map interface.

**6.Exception Handling**

**Java exception handling:**

Java exception handling is a powerful mechanism to handle runtime errors, which helps maintain the normal flow of the application. Here’s a detailed guide to understanding core Java exception handling with examples.

**Types of Exceptions in Java:**

1. **Checked Exceptions**: These are checked at compile-time.

Checked exceptions are the exceptions that are checked at compile-time. This means that the compiler verifies that the code handles these exceptions either by catching them or declaring them in the method signature using the throws keyword example: IOException, SQLException, FileNotFoundException..

1. **Unchecked Exceptions**: These are not checked at compile-time but at runtime.

Unchecked exceptions, also known as runtime exceptions, are not checked at compile-time. These exceptions usually occur due to programming errors, such as logic errors or incorrect assumptions in the code. They do not need to be declared in the method signature using the throws keyword, making it optional to handle them

example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException.

1. **Errors**: These are serious issues that cannot be handled by the program. For example:OutOfMemoryError.

A screenshot of a computer screen

Description automatically generated

**Java Exception Keywords**

Java provides five keywords that are used to handle the exception

**try**: The try block contains code that might throw an exception. If an exception occurs, it is caught by the corresponding catch block.

**catch**: The catch block handles the exception thrown by the try block. It catches the exception and defines how to respond to it.

**finally**: The finally block contains code that is always executed, whether an exception is thrown or not. It is typically used to release resources like file handles or database connections.

**throw**: The throw keyword is used to explicitly throw an exception. It can be used in a method to throw an exception to the caller.

**throws**: The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

**7.JDK vs JRE vs JVM**

**JDK vs JRE vs JVM:**

* JDK: Java Development Kit is a software development environment used for developing Java applications and applets.
* JRE: JRE stands for Java Runtime Environment, and it provides an environment to run only the Java program onto the system.
* JVM: JVM stands for Java Virtual Machine and is responsible for executing the Java program.

**A screenshot of a computer program

Description automatically generated**

JDK (Java Development Kit)

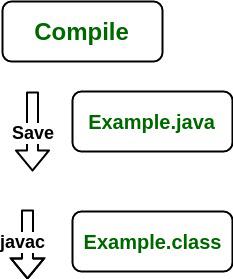
The JDK is a software development kit that provides the environment to develop and execute the java application. It includes two things:

* Development Tools (to provide an environment to develop your java programs)
* JRE (to execute your java program)

Working of JDK

The JDK enables the development and execution of Java programs. Consider the following process:

* Java Source File (e.g., Example.java): You write the Java program in a source file.
* Compilation: The source file is compiled by the Java Compiler (part of JDK) into bytecode, which is stored in a .class file (e.g., Example.class).
* Execution: The bytecode is executed by the JVM (Java Virtual Machine), which interprets the bytecode and runs the Java program.



Note: From above, media operation computing during the compile time can be interpreted.

The following actions occur at runtime as listed below:

* Class Loader
* Byte Code Verifier
* Interpreter
  + Execute the Byte Code
  + Make appropriate calls to the underlying hardware

JRE(Java Runtime Environment)

The JRE is an installation package that provides an environment to only run(not develop) the Java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

Working of JRE

When you run a Java program, the following steps occur:

* Class Loader: The JRE’s class loader loads the .class file containing the bytecode into memory.
* Bytecode Verifier: The bytecode verifier checks the bytecode for security and correctness.
* Interpreter: The JVM interprets the bytecode and executes the program.
* Execution: The program executes, making calls to the underlying hardware and system resources as needed.

JVM (Java Virtual Machine)

The JVM is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an [*interpreter*](https://www.geeksforgeeks.org/compiler-vs-interpreter-2/)*.*

Working of JVM

It is mainly responsible for three activities.

* Loading
* Linking
* Initialization

A diagram of a software system

Description automatically generated

**8.Variable Arguments (Varargs)**

Variable arguments (varargs) in Java allow a method to accept zero or more arguments of a specified type. This feature was introduced in Java 5 and is useful when you want to create methods that can handle a variable number of inputs without explicitly specifying the number of parameters.

Syntax:

public static void methodName(Type... parameterName) {

// Method body

}

Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

* There can be only one variable argument in the method.
* Variable argument (varargs) must be the last argument.

**12. Comparable and Comparator**

**A screenshot of a computer

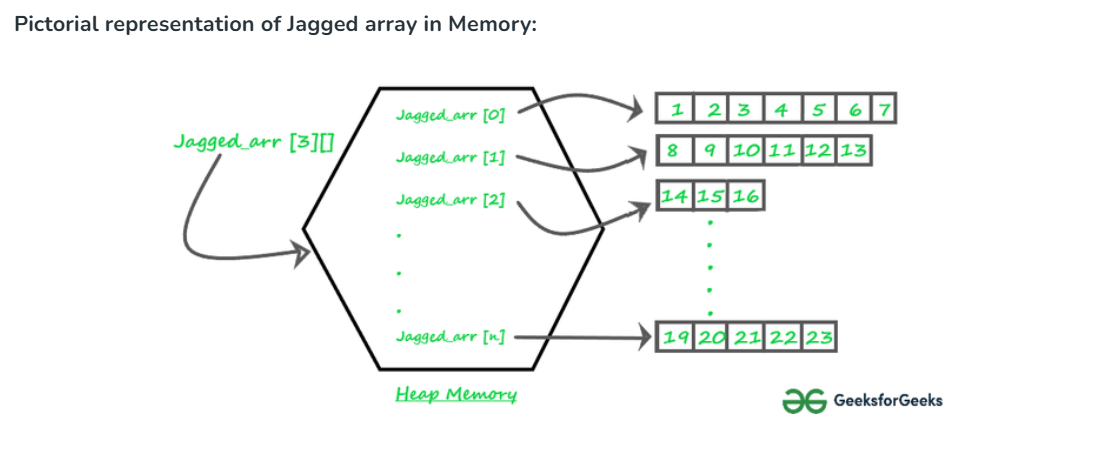
Description automatically generated**

**Comparable Interface:**The Comparable interface is used to define the natural ordering of objects of a class. A class that implements Comparable defines how its objects should be compared to each other.

**Method: compare**

**13. Jagged Arrays:**

A jagged array in Java is an array of arrays where the inner arrays can have different lengths. Unlike a multidimensional array where each sub-array has the same length, a jagged array allows for varying lengths, making it flexible and memory-efficient when dealing with collections of varying sizes.



Syntax:

To declare and initialize a jagged array, you can use the following syntax:

int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[2]; // First sub-array of length 2

jaggedArray[1] = new int[3]; // Second sub-array of length 3

jaggedArray[2] = new int[4]; // Third sub-array of length 4

**14. Java Memory Management:**

Java memory management involves the allocation and deallocation of memory to ensure efficient utilization of resources. The Java Virtual Machine (JVM) handles memory management through different areas of memory and garbage collection mechanisms.

**Memory Areas:**

1. Heap Area

* Heap is a shared runtime data area where objects and arrays are stored. It is created when the JVM starts.
* The memory in the heap is allocated for all the class instances and arrays.
* Heap can be of fixed or dynamic size depending upon the system’s configuration.
* JVM allows user to adjust the heap size. When the new keyword is used the object is allocated in the heap and its reference is stored in the stack.
* There exists one and only one heap for a running JVM process.

*Scanner sc = new Scanner(System.in)*

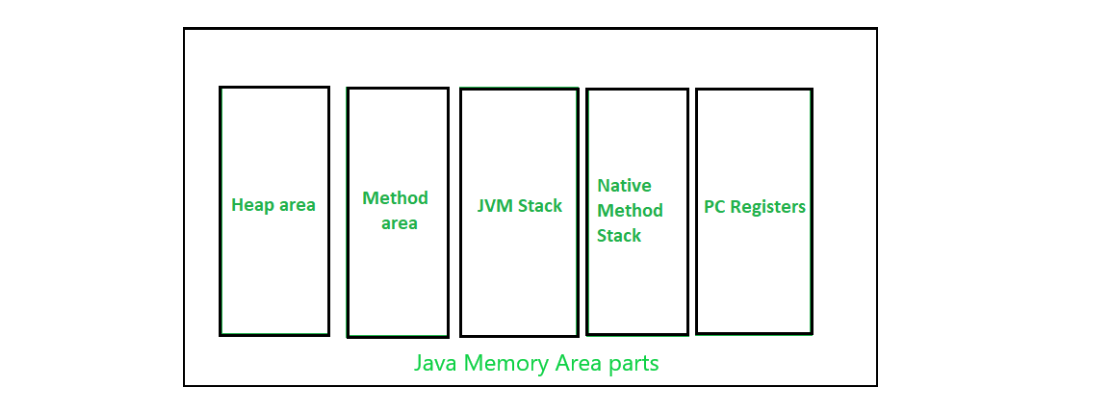
Here, the Scanner object is stored in the heap and the reference sc is stored in the stack

Note: Garbage collection in heap area is mandatory.

2. Method Area

* Method area is a logical part of the heap, and it is created when the JVM starts.
* Method area is used to store class-level information such as class structures, Method bytecode, Static variables, Constant pool, Interfaces.
* Method area can be of fixed or dynamic size depending on the system’s configuration.

*Note: Though method area is logically a part of heap, it may or may not be garbage collected even if garbage collection is compulsory in heap area.*



3. JVM Stacks

* A stack is created when a thread is created, and the JVM stack is used to store method execution data, including local variables, method arguments, and return addresses
* Each Thread has its own stack, ensuring thread safety.
* Stacks size can be either fixed or dynamic, and it can be set when the stack is created.
* The memory for stack needs not to be contiguous.
* Once a method completes execution, its associated stack frame is removed automatically.

4. Native Method Stacks

* Native method stack is also known as C stacks.
* Native method stacks are not written in Java language
* This memory is allocated for each thread when it is created and can have either a fixed or dynamic size.
* Native method stacks handle the execution of native methods that interact with the Java code.

5. Program Counter (PC) Registers

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

**15.How Java Objects are Stored in Memory**

Java objects are stored in memory primarily on the heap. The Java Virtual Machine (JVM) manages memory allocation for objects, ensuring efficient use of resources and enabling features like garbage collection. Here’s a detailed look at how Java objects are stored and managed in memory.

**1. Object Creation and Storage**

When you create an object in Java using the new keyword, memory is allocated on the heap. The heap is a shared memory area where all objects and their instance variables are stored.

Example: MemoryExample

**2. Stack Memory**

Stack memory is used for method execution and stores local variables and references to objects in heap memory. Each thread has its own stack, which contains stack frames for each method call. These frames hold method-specific information such as local variables, method parameters, and the return address.

Example: StackMemoryExample

**3. Method Area**

The method area is a shared memory space that stores class-level information such as class definitions, static variables, and constant pool data. The method area is part of the heap and is used by the JVM to store runtime constant pool, method data, and the code for methods and constructors.

Example: MethodAreaExample

**4. Garbage Collection**

Garbage collection is the process of automatically identifying and reclaiming memory occupied by objects that are no longer reachable or in use. The JVM handles garbage collection to free up heap memory and prevent memory leaks.

Example: GarbageCollectionExample

Java objects are stored in heap memory, while references to these objects and local variables are stored in stack memory. The JVM manages memory allocation and deallocation, with garbage collection playing a crucial role in reclaiming memory. Understanding how objects are stored and managed in memory helps in writing efficient and robust Java applications.

**16.Stack vs Heap Memory Allocation**

Stack memory is a region of memory that stores method call frames, local variables, and method parameters. It operates in a Last-In-First-Out (LIFO) manner, where the most recently added frame is the first to be removed. Stack memory is limited in size, automatically managed, and thread-specific, meaning each thread has its own stack.

**Key Points:**

* It’s a temporary memory allocation scheme where the data members are accessible only if the method() that contained them is currently running.
* It allocates or de-allocates the memory automatically as soon as the corresponding method completes its execution.
* Memory allocation and de-allocation are faster as compared to Heap-memory allocation.
* Stack memory has less storage space as compared to Heap-memory.

Heap memory is a region of memory used for dynamic memory allocation. It stores objects and instance variables, and memory is managed by the garbage collector to free up space occupied by objects that are no longer in use. Heap memory is shared among all threads, allowing for global access to objects.

**Key Points:**

* We receive the corresponding error message if Heap-space is entirely full,  java. lang.OutOfMemoryError by JVM.
* This memory allocation scheme is different from the Stack-space allocation, here no automatic de-allocation feature is provided. We need to use a Garbage collector to remove the old unused objects in order to use the memory efficiently.
* The size of the Heap-memory is quite larger as compared to the Stack-memory.
* Heap memory is accessible or exists as long as the whole application(or java program) runs.

A diagram of a stock exchange

Description automatically generated

A white sheet with black text

Description automatically generated

**Key Differences Between Stack and Heap Allocations** 

1. In a stack, the allocation and de-allocation are automatically done by the compiler whereas, in heap, it needs to be done by the programmer manually.
2. Handling the Heap frame is costlier than handling the stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.

4. A stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.

5. Accessing the time of heap takes is more than a stack.

**17.Java Virtual Machine (JVM) Stack Area**

The JVM stack area is a region of memory used to store stack frames, which hold local variables, method call information, and partial results. Each thread in a Java application has its own JVM stack, created when the thread is started. The JVM stack is an essential part of method execution, and it operates in a Last-In-First-Out (LIFO) manner.

A close-up of a diagram

Description automatically generated

**Key Characteristics:**

* **Thread-Specific**: Each thread has its own JVM stack, which is not shared with other threads.
* **Stack Frames**: The stack is composed of frames, each corresponding to a method call. When a method is invoked, a new frame is pushed onto the stack, and when the method completes, the frame is popped off.
* **Automatic Memory Management**: Memory is automatically allocated and deallocated as methods are called and return.
* **Scope and Lifetime**: Variables in a stack frame are accessible only within the method call, and their lifetime is limited to the duration of the method call.

**Stack Frames:**

Each stack frame contains:

* **Local Variables**: Method parameters and local variables.
* **Operand Stack**: Used for intermediate calculations and method return values.
* **Frame Data**: Includes the return address and additional information for method execution.

**18. Types of Memory Areas Allocated by JVM**

The JVM allocates memory into 5 distinct areas to carry out its operations. These areas are:

1. Class(Method) Area
2. Heap
3. Stack
4. Program Counter Register (PC Register)
5. Native Method Stack

**A diagram of a software system

Description automatically generated**

1. Class (Method) Area

The Class (Method) Area is a memory block in the JVM that stores important information about classes. It includes:

* Class Code: The bytecode that defines the class itself.
* Variable Code: This includes static variables and runtime constants associated with the class.
* Method Code: The bytecode for all the methods defined in the class, including constructors.

*Note: Here method means the function which is written inside the class*

The class area stores class-level data of every class such as the runtime constant pool, field and method data and the code for methods, enabling the JVM to execute Java programs effectively.

2. Heap

The Heap Area is the memory block in the JVM where all objects are created and stored. It is used to allocate memory for:

* Objects: Instances of classes created during the execution of a program.
* Class Interfaces: Memory for class interfaces is also allocated here.
* Arrays: Since arrays are considered objects in Java, their memory is allocated in the heap as well.

*Note: Static Methods and Variables were previous stored in Class Area (Till Java 8). But, in current versions of Java static variables and methods are stored in Heap Memory.*

The Heap Area is crucial for dynamic memory allocation, allowing the JVM to allocate memory for objects at runtime.

3. Stack

* In Java, each thread has its own stack called the Run-Time Stack, created when the thread starts.
* The JVM only performs two operations directly on Java stacks. It pushes and pops frames.
* Every method call performed by the thread is stored in the corresponding run-time stack. The stack area is used to store Method call information, Local variables, Method parameters, Return Address
* After completing all method calls the stack becomes empty and that empty stack is destroyed by the JVM just before terminating the thread.

4. Program Counter Register

* Each JVM thread that carries out the task of a specific method has a program counter register associated with it.
* The non-native method has a PC that stores the address of the available JVM instruction whereas, in a native method, the value of the program counter is undefined.
* PC register is capable of storing the return address or a native pointer on some specific platform.

5. Native Method Stack

Also called C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when it’s created And it can be of a fixed or dynamic nature.

**19. Garbage Collection**

Garbage collection in Java is an automatic memory management process that helps Java programs run efficiently. Java programs compile to bytecode that can be run on a Java Virtual Machine (JVM). When Java programs run on the JVM, objects in the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

Working of Garbage Collection

* Java garbage collection is an automatic process that manages memory in the heap.
* It identifies which objects are still in use (referenced) and which are not in use (unreferenced).
* Unreferenced objects can be deleted to free up memory.
* The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Types of Activities in Java Garbage Collection

Two types of garbage collection activity usually happen in Java. These are:

1. Minor or incremental Garbage Collection: This occurs when unreachable objects in the Young Generation heap memory are removed.
2. Major or Full Garbage Collection: This happens when objects that survived minor garbage collection are removed from the Old Generation heap memory. It occurs less frequently than minor garbage collection.

**Benefits of Garbage Collection**

1. **Automatic Memory Management**: Reduces the burden on developers to manually manage memory, preventing common issues like memory leaks and dangling pointers.
2. **Improved Performance**: Optimizes memory utilization and application performance by reclaiming unused memory.
3. **Simplified Development**: Allows developers to focus on writing business logic without worrying about low-level memory management.