**Core Java**

**Why string class is immutable and write a program**

In java, the string class is immutable, which means that once String object is created it cannot be changed.

String class objects are immutable objects, where the immutable objects are not allowing modifications on their content directly, here the **immutable object data is allowed** **for the modifications** but the modified resultant data will not be stored back in the original object, here the resultant modified data will be stored by creating another new object of the same class.

String is immutable in java because of Security, String Pooling and Memory Efficiency, Caching,Thread Safety and Performance.

**EX:**

public class Main {  
 public static void main(String[] args) {  
 String originalString = "Silpa"**;** System.*out*.println("originalString Before Modify : "+originalString.hashCode())**;** String modifiedString = originalString.concat("Pachuru")**;** System.*out*.println("originalString After Modified : "+modifiedString.hashCode())**;**  
 System.*out*.println("Original String: " + originalString)**;** System.*out*.println("Modified String: " + modifiedString)**;** }  
 }

**O/P:**

originalString Before Modify : 79887655

originalString After Modified : 213972091

Original String: Silpa

Modified String: SilpaPachuru

**Diff b/w wait() & Notify()**

wait() : The purpose of **wait()** method is to keep a running thread in waiting state.

Notify() : The purpose of **notify()** method is to give an activation state to a particular thread which is in waiting state.

**Diff b/w synchronised and Serialized**

|  |  |
| --- | --- |
| **synchronised** | **serialized** |
| Sychronization in java is the process that allows only one thread at a particular time to complete a given task entirely,not allowing more than one thread at a time and allowing other threads after completion of the present thread execution. | Serialization is the process of converting an object's state into a byte stream so that it can be saved to a file, sent over a network, or stored in a database. |
| |  | | --- | | Primarily used for thread synchronization in multi-threaded programs to ensure that only one thread can access a block of code at a time. |  |  | | --- | |  | | |  | | --- | | Primarily used for object persistence or data transmission in I/O operations, such as saving an object to a file or sending it over a network. | |
| Ensures thread safety by preventing concurrent access to shared resources. | Not related to thread safety but used to store or transfer object state. |

**what is Stack**

Stack is a Java class that extends Vector in the Collection framework.

1. It is a class provided by JAVA in its JDK1.0 version.
2. It is a legacy Collection.
3. It is a child class to Vector class.
4. It is able to follow LIFO[Last In First Out] to manage all the elements**.**

**ArrayList vs LinkedList**

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| ArrayList has the internal datastructre is “Resizable Array”. | LinkedList has the datastructre is “Double LinkedList”. |
| It is suggestable for “retrieval operations” | It is suggestable for frequent inseration and deletion operations. |
| ArrayList class implement the interfaces like List,RandomAccess, Serializable and Cloneable. | LinkedList class implements the inerfaces like List, Deque, Serializable and Cloneable. |

**What are the memory management in Java**

The Memory management system in java is used to store the data in the form of objects or in the form of bytecodes.

Memory management in Java is primarily handled by the **Java Virtual Machine (JVM)**, which automatically allocates and deallocates memory through the process of **garbage collection**.

In java, the memory management is the process of allocation and de-allocation of the objects.

Basically, the java is a automatic memory management system.

Java uses a automatic memory management system called as **“Garbage Collector”.**

**JVM contains the following five types of memories.**

1. **Method Area**
2. **Stack memory**
3. **Heap Memory**
4. **PC-Registers**
5. **Native Methods Stack**

**Method Area:**

Method Area will be created at a time of JVM startup. It is a shared memory for all the threads which are executing in the present java applications.

Method Area is able to store all the classes bytecode which are loaded by the class loaders.

**Stack memory:**

Stack memory will be created when a thread is created in the present Java application.

Stack memory is not shared memory, it allows the respective threads only to access the data.

Stack Memory will create a separate stack for each and every thread which is created in the java application.

**Heap Memory:**

Heap Memory will be created by JVM at the time of starting JVM.

Heap Memory is a shared memory for all the threads which are running in the present java application.

Heap Memory is able to create objects which are created by the developers by using the new keyword.

Heap Memory is able to create java.lang.Class objects which are created by JVM when a particular class bytecode is loaded in the method area.

**PC(Program Counter) Registers:**

It is a buffer , it is able to store the address of the current instruction which is being executed.

If the current instruction execution is over then the PC-Register will store the next instruction address value.

**Native Methods Stack:**

This memory is the same as the Stack memory, but it is able to provide a very good environment to execute and to trace the native methods.

**Checked & unchecked**

|  |  |
| --- | --- |
| **Checked** | **Unchecked** |
| Checked Exception is an exception recognized at **compilation time** by the compiler[but not occurred at compilation, it will occur at runtime only]. | Unchecked Exception is an exception recognized at **runtime** by the JVM. |
| All the exception classes Except RuntimeException and its subclasses, Error and its subclasses are the examples for the Checked exceptions. | RuntimeException and its subclasses, Error and its subclasses are the examples for the Unchecked Exceptions. |

**Exception Hierarchy**

The **Exception Hierarchy** in Java is a structured set of classes that represent various types of exceptions and errors that can occur during the execution of a program.

The Java exception hierarchy is a system that categorizes different types of exceptions using inheritance

**The hierarchy is made up of the following classes:**

**Throwable** :

The Throwable class is the superclass for all exceptions and errors in Java. It contains methods like getMessage(), printStackTrace(), and getCause() for handling error reporting.It is the root class for all exceptions and errors.

**Errors** : It represent severe issues that are generally unrecoverable.

**Exceptions :** These are conditions that can be handled, divided into checked (e.g., IOException) and unchecked (e.g., NullPointerException) exceptions.

**Types of exceptions names**

In java, the exceptions are mainly classified into two types. They are

1. **Checked Excetions :** These exceptions are checked at Compile time
2. **Unchecked Exceptions :** These exceptions are not checked at complie time, it will checked at “Runtime”.

**Note:** Basically, the exceptions are occurred at run time only, but some exceptions are checked at compile time and some exceptions are checked at run time.

**Checked Exceptions:**

* IOException
* SQLException
* ClassNotFoundException
* InterruptedException
* FileNotFoundException

**Unchecked Exceptions (Runtime Exceptions):**

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException
* ClassCastException
* NumberFormatException

**Features of Java 8**

The features of java 8 are following bellow:-

* **Functional Interface:** If any interface has exactly only one abstract method then that interface id called “Functional Interface”.

In Java applications, if any interface has exactly one abstract method then that interface is default functional interface,but if we want to make an interface as a functional interface then we must use **@FunctionalInterface** annotation, where @FunctionalInterface annotation allows only one abstract method, it does not allow 0 number of abstract methods and more than one abstract methods.

* **Lambda Expression :**

In general, in Java applications, we will prepare Lambda expressions for the abstract method which is available in the Functional Interface, so to access the Lambda Expressions we have to use the Functional Interface reference variable.

**The main use of lambda expression is:**

To provide a implementation of functional interface

Less coding

It is very useful in collection library. It helps to iterate, filter and extract data from collection.

* **Static Methods in Interface:**

In java, upto java 1.7 version, interfaces are able to allows only abstract methods, but from java 1.8 version interfaces are able to allow the Static Methods.

If we provide the static methods inside the interfaces then we are able to access those static methods by using the respective interface name only, not possible to access with interface reference variable, implementation class name, implementation class reference variable.

* **Method Reference:**

The main purpose of the Method Reference is to execute a particular method when we access a function interface method.

**Syntax:**

If the method is an instance method:

FunctionalInterface ref = new ClassName()::MethodName;

If the method is a static method:

FunctionalInterface ref = ClassName::MethodName;

* **Constructor Reference:**

By using Constructor reference we are able to execute a particular class constructor when we access the functional interface method.

* **Predicates:**

In java applications, if we want to perform a test on the basis of a particular value and if we want to return a boolean value there we will use Predicates.

Predicate is a functional interface, it contains the following method.

public interface Predicate{

public boolean **test**(T t);

}

* **Date-Time API(Application Programming Interface):**

The main purpose of Date-Time API is to represent date and time values in java applications.

DateTime API is available in the following predefined classes in a package **java.time.**

LocalDate

LocalTime

LocalDateTime

**can we overload Lambda expression**

No, we cannot overload the lamda expression because of the lamda expression provides the Implementation of functional interface. If an interface contains only one abstract method is called “functional interface”.

Method Overloading occurs when a class having multiple methods with same name but different parameters but in lambda expression has only one abstract method. So, we cannot overload the lambda expression.

**internal working of hashmap**

**Top of Form**

**Bottom of Form**

**solid principles**

**git commands:**

**git status**

**git add .**

**git commit -m”msg"**

**git push**