**Java Solutions**

1. **Java Basics:**
2. **Java**

Java is one of the most popular and commonly used high-level programming languages. It is widely recognized for its performance, platform independence, and security. Java is used as a server-side language for back-end development.

Java Features:

### Inspired by C and C++

Java is inspired by C and C++. The syntax of Java is similar to these languages but the languages are quite different. Java inherits many features from C and C++. Compared to C++, Java code runs a bit slower but it is more portable and offers better security features.

**Simple and Familiar**

Java programming language is simple to learn, understand, read, and write. Java programs are easy to create and implement compared to other programming languages such as C and C++. If you are familiar with the basic principles of programming or the concept of OOP (object-oriented programming), it would be easy to master Java.

### Object-Oriented

Java is a fully object-oriented language, unlike C++ which is semi object-oriented. It supports every OOP concept such as Abstraction, Encapsulation, Inheritance, Polymorphism. Java programs are developed using classes and objects. Another notable feature is that in Java the main() function is defined under a class.

### Platform Independent

Java source code is compiled using Java Compiler. The compiler converts the source code into an intermediate code called the byte code. This code is further converted into machine-dependent form by the JVM (Java Virtual Machine). The JVM can execute byte code on any platform or operating system on which it is present.

### Compiled and Interpreted

Java offers both compilation and interpretation of programs. It combines the power of compiled languages and the flexibility of interpreted languages.

When a Java program is created, the Java compiler (javac) compiles the java source code into byte code. The Java Virtual Machine (JVM) serves as an interpreter that converts byte code to machine code which is portable and can be executed on any operating system.

### Multi-threaded

Java supports multithreading programming. A thread is an independent process to execute a set of statements. The term multi-threaded refers to creating multiple threads to handle multiple tasks at the same time.

Example:

Simple Java code to print hello world

class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello World");

}

}

1. **JVM Architecture:**

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).



*Fig: JVM Architecture*

### Classloader

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1.Bootstrap Classloader

2. **Extension ClassLoader**

**3. System/Application ClassLoader**

### Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

### Heap

It is the runtime data area in which objects are allocated.

### Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

### Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

### Native Method Stack

It contains all the native methods used in the application.

Execution Engine

It contains:

1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

### Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

1. **JDK vs JRE vs JVM**

|  |  |  |
| --- | --- | --- |
| **JDK** | **JRE** | **JVM** |
| The full form of JDK is Java Development Kit. | The full form of JRE is Java Runtime Environment. | The full form of JVM is Java Virtual Machine. |
| JDK is a software development kit to develop applications in Java. | It is a software bundle which provides Java class libraries with necessary components to run Java code. | JVM executes Java byte code and provides an environment for executing it. |
| JDK is platform dependent. | JRE is also platform dependent. | JVM is highly platform dependent. |
| It contains tools for developing, debugging, and monitoring java code. | It contains class libraries and other supporting files that JVM requires to execute the program. | Software development tools are not included in JVM. |
| It is the superset of JRE | It is the subset of JDK. | JVM is a subset of JRE. |
| The JDK enables developers to create Java programs that can be executed and run by the JRE and JVM. | The JRE is the part of Java that creates the JVM. | It is the Java platform component that executes source code. |
| JDK comes with the installer. | JRE only contain environment to execute source code. | JVM bundled in both software JDK and JRE. |

1. **Java OOPS:**
2. **OOPS Concepts**:

1) **Class**

The class is one of the Basic concepts of OOPs which is a group of similar entities. It is only a logical component and not the physical entity. Lets understand this one of the OOPs Concepts with example, if you had a class called “Expensive Cars” it could have objects like Mercedes, BMW, Toyota, etc. Its properties(data) can be price or speed of these cars. While the methods may be performed with these cars are driving, reverse, braking etc.

2) **Object**

An object can be defined as an instance of a class, and there can be multiple instances of a class in a program. An Object is one of the Java OOPs concepts which contains both the data and the function, which operates on the data. For example – chair, bike, marker, pen, table, car, etc.

3) **Inheritance**

Inheritance is one of the Basic Concepts of OOPs in which one object acquires the properties and behaviors of the parent object. It’s creating a parent-child relationship between two classes. It offers robust and natural mechanism for organizing and structure of any software.

4) **Polymorphism**

Polymorphism refers to one of the OOPs concepts in Java which is the ability of a variable, object or function to take on multiple forms. For example, in English, the verb run has a different meaning if you use it with a laptop, a foot race, and business. Here, we understand the meaning of run based on the other words used along with it. The same also applied to Polymorphism.

5) **Abstraction**

Abstraction is one of the OOP Concepts in Java which is an act of representing essential features without including background details. It is a technique of creating a new data type that is suited for a specific application. Lets understand this one of the OOPs Concepts with example, while driving a car, you do not have to be concerned with its internal working. Here you just need to concern about parts like steering wheel, Gears, accelerator, etc.

6) Encapsulation

Encapsulation is one of the best Java OOPs concepts of wrapping the data and code. In this OOPs concept, the variables of a class are always hidden from other classes. It can only be accessed using the methods of their current class. For example – in school, a student cannot exist without a class.

1. **Keywords**
2. **Constructor**

In Java a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory. It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Example :

package test;

class Test{

//creating a default constructor

Test(){

System.***out***.println("this key is printed ny using constructor");

}

//main method

public static void main(String args[]){

//calling a default constructor

Test b=new Test();

}

}

1. **This**

The this keyword refers to the current object in a method or constructor.

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".

Example:

package test;

public class Test {

int x;

// Constructor with a parameter

public Test(int x) {

this.x = x;

}

// Call the constructor

public static void main(String[] args) {

Test myObj = new Test(5);

System.out.println("Value of x = " + myObj.x);

}

}

this can also be used to:

* Invoke current class constructor
* Invoke current class method
* Return the current class object
* Pass an argument in the method call
* Pass an argument in the constructor call

1. **Final**

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

* variable
* method
* class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

Example:

**class** Bike{

**final** **int** speedlimit=90; //final variable

**void** run(){

  speedlimit=400;

 }

**public** **static** **void** main(String args[]){

 Bike obj=**new**  Bike();

 obj.run();

 }

}

1. **Super**

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

* super can be used to refer immediate parent class instance variable.
* super can be used to invoke immediate parent class method.
* super() can be used to invoke immediate parent class constructor

Example:

**class** Animal{

String color="white";

}

**class** Dog **extends** Animal{

String color="black";

**void** printColor(){

System.out.println(color);//prints color of Dog class

System.out.println(**super**.color);//prints color of Animal class

}

}

**class** TestSuper1{

**public** **static** **void** main(String args[]){

Dog d=**new** Dog();

d.printColor();

}}

1. **Abstract Vs Interface:**

|  |  |
| --- | --- |
| **Abstract** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Test { public abstract void draw(); } | **Example:** public interface Test { void draw(); } |

Example:

For Interface:

interface Pet {

public void test();

}

class Dog implements Pet {

public void test() {

System.out.println("Interface Method Implemented");

}

public static void main(String args[]) {

Pet p = new Dog();

p.test();

}

}

For Abstract:

abstract class Shape {

int b = 20;

abstract public void calculateArea();

}

public class Rectangle extends Shape {

public static void main(String args[]) {

Rectangle obj = new Rectangle();

obj.b = 200;

obj.calculateArea();

}

public void calculateArea() {

System.out.println("Area is " + (b \* b));

}

}

1. **Exception Handling:**
2. **Exception**:

Exception is an abnormal condition.In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.



*Fig: Hierarchy of Java Exceptions*

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

**1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

**2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

**3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

1. **Exception Handling:**

Exception handling is responding to exceptions when a computer program runs. An exception occurs when an unexpected event happens that requires special processing. Examples include a user providing abnormal input, a file system error being encountered when trying to read or write a file, or a program attempting to divide by zero.

Exception handling attempts to gracefully handle these situations so that a program (or worse, an entire system) does not crash. Exception handling can be performed at both the software (as part of the program itself) and hardware levels (using mechanisms built into the design of the CPU).

Example:

class Main {

public static void main(String[] args) {

try {

// code that generate exception

int divideByZero = 5 / 0;

System.out.println("Rest of code in try block");

}

catch (ArithmeticException e) {

System.out.println("ArithmeticException => " + e.getMessage());

}

}

}

1. **Throws Keyword:**

The throws keyword in Java is used to declare exceptions that can occur during the execution of a program. For any method that can throw exceptions, it is mandatory to use the throws keyword to list the exceptions that can be thrown. The throws keyword provides information about the exceptions to the programmer as well as to the caller of the method that throws the exceptions.

The throws keyword allows exceptions to be propagated in the call stack. When a method declares that it throws an exception, it is not required to handle the exception. The caller of a method that throws exceptions is required to handle the exceptions (or throw them to its caller and so on) so that the flow of the program can be maintained. Only checked exceptions are required to be thrown using the throws keyword. Unchecked exceptions don’t need to be thrown or handled explicitly in code.

**Throw Keyword:**

The throw keyword in Java is used for explicitly throwing a single exception. This can be from within a method or any block of code. Both checked and unchecked exceptions can be thrown using the throw keyword.

When an exception is thrown using the throw keyword, the flow of execution of the program is stopped and the control is transferred to the nearest enclosing try-catch block that matches the type of exception thrown. If no such match is found, the default exception handler terminates the program.

The throw keyword is useful for throwing exceptions based on certain conditions e.g. if a user enters incorrect data. It is also useful for throwing custom exceptions specific to a program or application. Unchecked exceptions can be propagated in the call stack using the throw keyword in a method. Checked exceptions can be propagated using the throw keyword when the method that throws the exception declares it using the throws keyword.

Example:

// Java program that demonstrates the use of throw

class ThrowExcep

{

static void fun()

{

try

{

throw new NullPointerException("demo");

}

catch(NullPointerException e)

{

System.out.println("Caught inside fun().");

throw e; // rethrowing the exception

}

}

public static void main(String args[])

{

try

{

fun();

}

catch(NullPointerException e)

{

System.out.println("Caught in main.");

}

}

}

1. **Java Multi-Threading:**
2. **Multi-Threading:**

Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application. Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Example:

class MultithreadingDemo extends Thread {

public void run()

{

try {

// Displaying the thread that is running

System.out.println(

"Thread " + Thread.currentThread().getId()

+ " is running");

}

catch (Exception e) {

// Throwing an exception

System.out.println("Exception is caught");

}

}

}

// Main Class

public class Multithread {

public static void main(String[] args)

{

int n = 11; // Number of threads

for (int i = 0; i < n; i++) {

MultithreadingDemo object

= new MultithreadingDemo();

object.start();

}

}

}

1. **Define A thread:**

Threads can be created by using two mechanisms :

* Extending the Thread class
* Implementing the Runnable Interface

Thread creation by extending the Thread class:

We create a class that extends the java.lang.Thread class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

class MultithreadingDemo extends Thread {

public void run()

{

try {

// Displaying the thread that is running

System.out.println(

"Thread " + Thread.currentThread().getId()

+ " is running");

}

catch (Exception e) {

// Throwing an exception

System.out.println("Exception is caught");

}

}

}

// Main Class

public class Multithread {

public static void main(String[] args)

{

int n = 11; // Number of threads

for (int i = 0; i < n; i++) {

MultithreadingDemo object

= new MultithreadingDemo();

object.start();

}

}

}

Thread creation by implementing the Runnable Interface:

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

Example:

class MultithreadingDemo implements Runnable {

public void run()

{

try {

// Displaying the thread that is running

System.out.println(

"Thread " + Thread.currentThread().getId()

+ " is running");

}

catch (Exception e) {

// Throwing an exception

System.out.println("Exception is caught");

}

}

}

// Main Class

class Multithread {

public static void main(String[] args)

{

int n = 10; // Number of threads

for (int i = 0; i < n; i++) {

Thread object

= new Thread(new MultithreadingDemo());

object.start();

}

}

}

1. **Synchronization:**

As Java is a multi-threaded language, it supports a very important concept of Synchronization.

The process of allowing only a single thread to access the shared data or resource at a particular point of time is known as Synchronization. This helps us to protect the data from the access by multiple threads. Java provides the mechanism of synchronization using the synchronized blocks.

We declare all synchronized blocks in Java are using a synchronized keyword. A block that is declared with a synchronized keyword ensures that only a single thread executes at a particular time. No other thread can enter into that synchronized block until the thread inside that block completes its execution and exits the block.

Example:

public class MyCounter {

private int count = 0;

public synchronized void increment(int value) {

this.count += value;

}

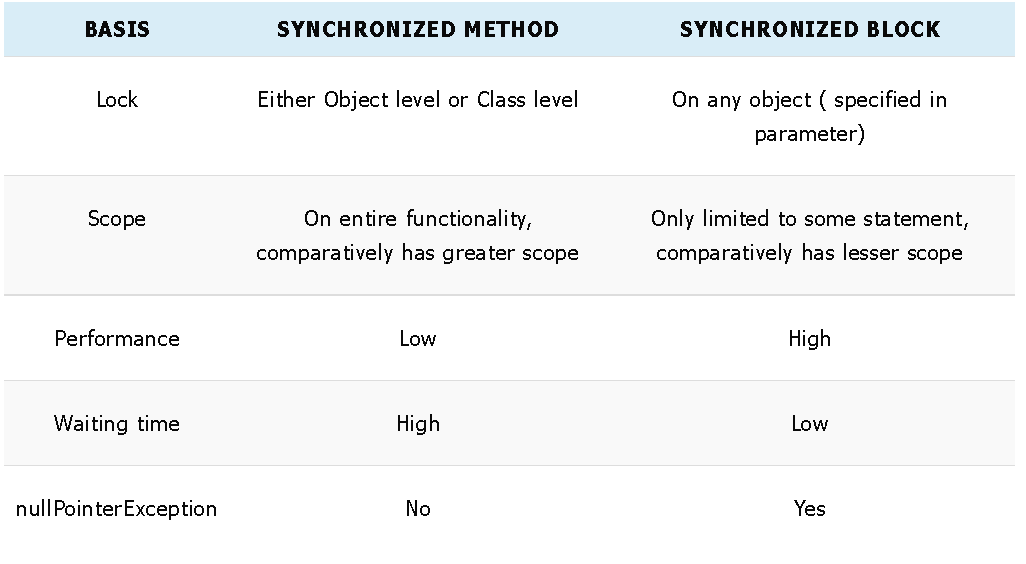
public synchronized void decrement(int value) {

this.count -= value;

}

}

1. **Synchronization Vs Synchronized Block:**



Example:

class SynchronizationExample implements Runnable{

@Override

public void run() {

incrementMe();

unSynchronizedShowMe();

}

synchronized void incrementMe() {

for (int i = 0; i <= 7; i++) {

System. out .println(Thread.currentThread().getName() + " is printing : " +

i);

}

}

void unSynchronizedShowMe()

{

System. out .println("unSynchronized : "+Thread.currentThread().getName());

synchronized (this)

{

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

e.printStackTrace();

}

for(int i = 0; i<5;i++)

System. out .println("printing from synchronized block :

"+Thread.currentThread().getName());

}

}

public static void main(String[] args) {

SynchronizationExample obj = new SynchronizationExample();

Thread threadOne = new Thread(obj,"Child One");

Thread threadtwo = new Thread(obj,"Child two");

threadOne.start();

threadtwo.start();

}

}

1. **Java Programs:**
2. **Fibonacci Series:**

class FibSeries {

public static void main(String[] args) {

int n = 11, a = 0, b = 1;

System.out.println("Fibonacci Series till " + n + " terms:");

for (int i = 1; i <= n; ++i) {

System.out.print(a + ", ");

int c = a+ b;

a= b;

b= c;

}

}

}

1. **Pyramids and Patters:**

public class Main {

public static void main(String[] args) {

int rows = 5, k = 0;

for (int i = 1; i <= rows; ++i, k = 0) {

for (int space = 1; space <= rows - i; ++space) {

System.out.print(" ");

}

while (k != 2 \* i - 1) {

System.out.print("\* ");

++k;

}

System.out.println();

}

}

}

1. **Reverse of A String:**

import java.io.\*;

import java.util.Scanner;

class Test {

public static void main (String[] args) {

String str= "Govardhan", nstr="";

char ch;

System.***out***.print("Original word: ");

System.***out***.println(str);

for (int i=0; i<str.length(); i++)

{

ch= str.charAt(i);

nstr= ch+nstr;

}

System.***out***.println("Reversed word: "+ nstr);

}

}

1. **Check Leap Year:**

public class Test {

public static void main(String[] args) {

int year = 2012;

boolean leap = false;

// if the year is divided by 4

if (year % 4 == 0) {

// if the year is century

if (year % 100 == 0) {

// if year is divided by 400

// then it is a leap year

if (year % 400 == 0)

leap = true;

else

leap = false;

}

// if the year is not century

else

leap = true;

}

else

leap = false;

if (leap)

System.***out***.println(year + " is a leap year.");

else

System.***out***.println(year + " is not a leap year.");

}

}

1. **Multiplication Table:**

public class Test {

public static void main(String[] args) {

int num = 5;

for(int i = 1; i <= 10; ++i)

{

System.***out***.printf("%d \* %d = %d \n", num, i, num \* i);

}

}

}