Java Workshop Lab File

7 April 2025

1. Write any ten features of java.

Simple: Java is simple and easy to understand. It eliminates complex features of C++ such as pointers, multiple inheritance, and operator overloading. Its syntax is clean and easy to learn, especially for those with knowledge of C or C++.

Object-Oriented: Java is a fully object-oriented language. It uses classes and objects to organize code and supports key principles like encapsulation, inheritance, abstraction, and polymorphism. These principles help in building modular and maintainable applications.

Platform-Independent: Java programs are compiled into bytecode instead of machine code. This bytecode can run on any system that has the Java Virtual Machine (JVM), making Java programs platform-independent. This property is known as "write once, run anywhere."

Secure: Java is designed with security in mind. It avoids direct memory access using pointers, performs runtime checks, and uses bytecode verification to ensure the safety of applications. Java also provides APIs for developing secure applications.

Robust: Java is a robust language due to its strong memory management, exception handling, and compile-time error checking. It uses automatic garbage collection to manage memory and reduces the chances of memory leaks and crashes.

Architecture-Neutral: Java is architecture-neutral because its bytecode is not dependent on any particular hardware or operating system. The same Java application can run on different systems without modification, as long as the JVM is present.

Portable: Java is portable as its code does not rely on system-specific features. Primitive data types have fixed sizes, and the compiled bytecode can be moved from one system to another without needing recompilation.

High Performance: Java offers high performance with the help of the Just-In-Time (JIT) compiler, which converts bytecode into native machine code at runtime. This improves the speed of execution, making Java suitable for many real-time applications.

Multithreaded: Java supports multithreading, allowing multiple threads to run concurrently within a program. It provides built-in support for thread creation, synchronization, and communication, making it ideal for interactive and network-based applications.

Dynamic: Java is dynamic in nature. It supports dynamic loading of classes during runtime and provides reflection APIs to inspect classes, methods, and objects. Java programs can also link to new code and libraries as they run.

2. What is Unicode and Byte Code?

Unicode:
Unicode is a universal character encoding standard used in Java to represent characters from all major writing systems in the world. Each character is assigned a unique number, known as a code point. Unicode allows Java programs to support a wide range of characters and symbols, including letters from different languages, numbers, punctuation marks, and special symbols. Java uses Unicode internally, which means characters in Java (such as in strings and identifiers) are stored and processed using Unicode. This makes Java suitable for international applications.

Bytecode:
Bytecode is the intermediate machine-independent code generated by the Java compiler after a Java program is compiled. It is a set of instructions that can be executed by the Java Virtual Machine (JVM). Unlike source code, which is written in human-readable form, or machine code, which is hardware-specific, bytecode is a portable binary format. This bytecode enables Java's "write once, run anywhere" capability, as the same bytecode can run on any device that has a JVM, regardless of the underlying hardware or operating system.

3. How java is platform independent?

Java is platform independent because the programs written in Java are not directly compiled into machine code specific to one system. Instead, the Java compiler converts the source code (`.java` files) into bytecode (`.class` files), which is an intermediate code. This bytecode is not specific to any particular operating system or hardware. It is designed to be executed by the Java Virtual Machine (JVM).The JVM acts as a layer between the bytecode and the operating system. It reads the bytecode and translates it into machine code appropriate for the system it is running on. Since JVMs are available for almost all platforms (Windows, Linux, Mac, etc.), the same Java bytecode can run on any of them without modification.This concept is often described using the phrase **"write once, run anywhere"**, meaning that once you write and compile a Java program, you can run it on any system that has a JVM installed.

4. What is JDK and JRE ? Differantiate?

JDK (Java Development Kit):
The Java Development Kit, commonly known as JDK, is a complete software package used to develop Java applications. It provides all the essential tools needed by a programmer to write, compile, and debug Java programs. JDK includes the Java compiler (`javac`), which is used to convert Java source code into bytecode. Apart from the compiler, it also contains other development tools such as JavaDoc (for generating documentation), Java Debugger (`jdb`), and various utilities. JDK also includes the Java Runtime Environment (JRE), so it can not only develop but also execute Java programs. Without JDK, it is not possible to write and compile Java applications. It is mainly used by developers during the software development process.

JRE (Java Runtime Environment):
The Java Runtime Environment, or JRE, is a part of the JDK and is required to run Java applications. It provides the necessary environment to execute Java bytecode. JRE contains the Java Virtual Machine (JVM), which interprets and executes the bytecode, as well as a set of core class libraries and supporting files. However, JRE does not include development tools like the compiler or debugger, so it cannot be used to develop Java applications. It is mainly used by end users who only need to run Java programs and do not require any development functionality.

Difference:
The main difference between JDK and JRE is that JDK is used for both development and execution of Java programs, while JRE is used only for running Java programs. JDK includes the JRE as well as additional tools needed for development. On the other hand, JRE contains only the tools required to run Java applications, such as the JVM and standard libraries.

5. What is the task of Class Loader,Verifier,JIT Compiler in JRE.

Class Loader:
The Class Loader is a component of the Java Runtime Environment responsible for loading Java class files into memory when they are needed during program execution. It reads the `.class` files (bytecode) from the file system or network and loads them into the JVM. The Class Loader also ensures that classes are loaded only once and manages the namespace of classes. It plays a crucial role in enabling dynamic loading, meaning classes can be loaded at runtime rather than all at once before execution starts.

Verifier:
The Verifier is a part of the JVM that performs important security and correctness checks on the bytecode loaded by the Class Loader. It verifies that the bytecode adheres to the Java language rules and does not violate access rights or memory constraints. This step ensures that the code is safe to execute and prevents harmful or corrupted bytecode from causing security breaches or system crashes. It checks things like correct method calls, proper use of variables, and that the code does not attempt illegal operations.

JIT Compiler (Just-In-Time Compiler):
The JIT Compiler is a component that improves the performance of Java programs. It converts the bytecode, which is interpreted by the JVM, into native machine code at runtime. This native code runs directly on the hardware, which is faster than interpreting bytecode line-by-line. The JIT compiler compiles frequently executed code sections into machine code, optimizing the execution speed while still keeping the platform independence of Java.

29 April 2025

6. What are the two types of Exceptions in java? What are the differences between them.

Checked Exceptions:
These are exceptions that are checked at compile-time by the Java compiler. When a method can throw a checked exception, it must either handle the exception using a try-catch block or declare it using the throws keyword in the method signature. Checked exceptions represent conditions that a reasonable application might want to catch, such as file not found, input/output errors, or network failures. Examples include IOException, SQLException, and ClassNotFoundException.

Unchecked Exceptions:
Unchecked exceptions are not checked at compile-time but occur during runtime. They are also known as runtime exceptions and are subclasses of RuntimeException. These exceptions usually indicate programming errors such as logic mistakes, incorrect use of APIs, or improper use of null references. Examples include NullPointerException, ArithmeticException, and ArrayIndexOutOfBoundsException. Methods are not required to declare unchecked exceptions.

Differences between Checked and Unchecked Exceptions:

- **Compile-Time Checking:** Checked exceptions are checked by the compiler, whereas unchecked exceptions are not checked during compilation.
- **Handling Requirement:** Checked exceptions must be either caught or declared to be thrown; unchecked exceptions do not have this requirement.
- **Cause:** Checked exceptions often represent external problems outside the program's control; unchecked exceptions usually indicate programming errors.
- **Hierarchy:** Checked exceptions are subclasses of Exception but not RuntimeException; unchecked exceptions are subclasses of RuntimeException.

7. What are the Memory allocations available in Java?

Memory Allocations in Java:

Java manages memory in several areas during the execution of a program. The main types of memory allocation in Java are:

1. Heap Memory:
Heap is the runtime data area where all Java objects and arrays are allocated. It is shared among all threads of a Java application. When you create an object using the `new` keyword, memory for that object is allocated on the heap. The heap is managed by the Garbage Collector, which automatically frees memory occupied by objects that are no longer in use.

2. Stack Memory:
Stack memory is used to store local variables, method parameters, and partial results

during method execution. Each thread in Java has its own stack. Whenever a method is called, a new stack frame is created to hold its variables. When the method execution completes, the stack frame is removed. Stack memory is faster but limited in size compared to heap memory.

3. Method Area (or PermGen/Metaspace):
The method area stores class-level data such as class definitions, method code, constants, and static variables. It is shared among all threads. In earlier Java versions, this was called PermGen space; from Java 8 onwards, it is replaced by Metaspace which grows dynamically.

4. Program Counter (PC) Register:
Each thread has its own PC register that holds the address of the current instruction being executed. It helps the thread keep track of where it is in the program.

5. Native Method Stack:
This memory area is used for executing native (platform-specific) methods written in languages like C or C++. It supports the native methods called from Java code.

8. Explain final,finally and finalize?

The final keyword in Java is used as a modifier for variables, methods, and classes. It is used to define constants, prevent method overriding, and restrict inheritance.

- When a variable is declared as final, its value cannot be changed once assigned. It becomes a constant.
- When a method is declared as final, it cannot be overridden by subclasses.
- When a class is declared as final, it cannot be extended or subclassed.

```
final int x = 10;  // x cannot be change

final class MyClass {}  // cannot be extended
```

The finally block in Java is used in exception handling to define a block of code that will always be executed, whether an exception is thrown or not. It is typically used to release resources such as files, database connections, or network sockets.

- A finally block is always executed after the try block, and after the catch block if one is present.
- It ensures that cleanup code runs no matter what.

The finalize() method is a protected method defined in the Object class. It is called by the Garbage Collector before an object is removed from memory. It was intended to allow objects to clean up resources before they are destroyed, like closing files or releasing memory.

- You can override the finalize() method in your class to define custom cleanup behavior.

- However, it is generally discouraged to rely on finalize() because it is unpredictable and may not be called promptly.

9. What is a singleton class in java? And how to break singleton class object?

A Singleton Class is a class in Java that allows only one instance (object) to be created for the entire program. It ensures that a class has only one object and provides a global point of access to it.Singleton classes are commonly used for resources like logging, configuration settings, or database connections—where creating multiple instances might lead to errors or waste of memory.To implement a singleton class:

- Make the constructor private to prevent instantiation from outside the class.
- Create a static variable to hold the single instance.
- Provide a public static method (usually called getInstance()) to return that instance.

Although singleton is meant to restrict object creation to one instance, it can still be broken using several methods if not carefully implemented.

1. **Using Reflection:**
   Reflection can access the private constructor and create a new instance.Example:
2. **Using Serialization and Deserialization:**
   When a singleton object is serialized and then deserialized, it creates a new instance.
3. **Using Cloning:**
   If the singleton class implements `Cloneable`, calling `clone()` will create another object.

10. Difference between instance and local variable?

Instance Variables:
Instance variables are variables that are declared inside a class but outside any method, constructor, or block. They belong to the object (instance) of the class. Each object has its own copy of instance variables.

- Declared at the class level (outside methods).
- Memory is allocated when the object is created.
- Can have access modifiers like private, public, etc.
- Automatically initialized to default values (like 0, null, false, etc.) if not explicitly initialized.
- Accessible by all methods in the class (if in scope).
- Their lifetime is as long as the object exists.

```
public class Student

{

int id; // instance variable
String name; // instance variable

}
```

Local Variables:
Local variables are declared inside methods, constructors, or blocks. They are created when the method is called and destroyed when the method exits.

- Declared within a method or block.
- Memory is allocated when the method is called.
- Cannot have access modifiers (like private, public, etc.).
- Must be explicitly initialized before use.
- Only accessible within the method or block in which they are declared.
- Their lifetime ends when the method or block completes execution.

```
public void showDetails()

 {

 int age = 20; // local variable

 String course = "Java"; // local variable

}
```

11. Explain the types of Exceptions in Java?

1. Checked Exceptions:
Checked exceptions are the exceptions that are checked at **compile-time** by the Java compiler. If a method is capable of causing a checked exception, it must either handle the exception using a try-catch block or declare it using the throws keyword. These exceptions are usually caused by external factors, such as file handling, database access, or network issues.

- Examples: IOException, SQLException, ClassNotFoundException, FileNotFoundException
- Must be handled at compile-time, or the program will not compile.

2. Unchecked Exceptions:
Unchecked exceptions are not checked at compile-time. They occur during **runtime** and are usually due to programming errors, such as accessing an array out of bounds or dividing by zero. These exceptions are subclasses of RuntimeException.

- Examples: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException
- Not required to be caught or declared in method signatures.

3. Errors (a special category):
Though not technically exceptions, **Errors** are serious problems that a program should not try to handle. They are issues related to the Java Virtual Machine (JVM), such as memory errors or stack overflow.

- Examples: OutOfMemoryError, StackOverflowError, VirtualMachineError
- These are not meant to be caught or handled by the application code.

12. Explain the Java thread lifecycle?

In Java, a thread goes through various stages in its lifetime. These stages represent the state of the thread from its creation to its termination. The Java thread lifecycle consists of the following five main states:

1. **New:**
   When a thread object is created using the `Thread` class or by implementing the `Runnable` interface, it is said to be in the **new** state. At this point, the thread is not yet started and has not begun execution.
   Thread t = new Thread(); // New state
2. **Runnable:**
   After calling the start() method on the thread object, the thread enters the **runnable** state. In this state, the thread is ready to run and waiting for the CPU to assign it time for execution.t.start(); // Thread is now in runnable state
3. **Running:**
   When the thread scheduler picks the thread from the runnable pool, the thread starts executing. At this point, it is in the **running** state. Only one thread runs at a time per processor.
4. **Blocked / Waiting / Timed Waiting:**
   These are intermediate states where a thread is **not eligible to run** temporarily:

- **Blocked:** A thread is waiting to acquire a lock (used in synchronization).
- **Waiting:** A thread is waiting indefinitely for another thread to perform a specific action.
- **Timed Waiting:** A thread is waiting for a specified period of time.
  Example methods causing these states:
  sleep(), wait(), join(), yield()

5. **Terminated (Dead):**
   A thread enters the **terminated** or **dead** state when it has completed its execution or is forcefully stopped due to an exception.

13. Can we inherit a constructor?

In Java, constructors are **not inherited** by subclasses. This is because constructors are special methods that are used to initialize objects of a class. They have the same name as the class and are not members that can be inherited like methods or variables.

When a subclass is created, it **does not automatically get the constructor** of its parent class. However, a subclass can call the parent class constructor using the `super()` keyword.

14. How you implement method overloading in Java?

Method overloading is a feature in Java that allows a class to have **more than one method with the same name** but with **different parameter lists**. It increases the readability and flexibility of the code.Method overloading can be done in the following ways:

- By changing the **number of parameters**.
- By changing the **type of parameters**.
- By changing the **order of parameters** (if types are different)

15. What is runtime polymorphism?

Runtime polymorphism is a feature in Java that allows a method to behave differently based on the object that invokes it. It is also known as **dynamic method dispatch**, and it occurs **during program execution (runtime)**.It is achieved when a **parent class reference** is used to refer to a **child class object**, and the method that gets executed is determined at **runtime**, based on the actual object.Runtime polymorphism is achieved through **method overriding**.

- A subclass provides its own version of a method that is already defined in its superclass.
- The method in the subclass must have the **same name**, **return type**, and **parameters** as in the superclass.

16. How does garbage collection work in Java?

Garbage Collection (GC) in Java is the process by which the **Java Virtual Machine (JVM)** automatically removes **unused objects** from memory (heap). It helps to **free up memory** by destroying objects that are no longer reachable, thereby preventing memory leaks and improving performance.In Java, memory is allocated to objects dynamically at runtime.

- If objects are not freed after use, they occupy memory unnecessarily.
- Garbage collection ensures that memory used by **objects with no references** is reclaimed.

The **JVM** keeps track of all objects created in memory.

- When there are **no references** pointing to an object, it is considered **eligible for garbage collection**.
- The garbage collector automatically identifies and **deletes** such objects.
- The exact time when garbage collection occurs is **not predictable**. It is managed by the JVM

Garbage collection in Java is an automatic process that improves memory management by removing unused objects. It increases application efficiency and reliability without burdening the programmer with manual deallocation.