**Difference between Java vs C++?**

|  |  |  |
| --- | --- | --- |
| **Comparison Index** | **C++** | **Java** |
| Platform-independent | C++ is platform-dependent. | Java is platform-independent. |
| Mainly used for | C++ is mainly used for system programming. | Java is mainly used for application programming. It is widely used in window, web-based, enterprise and mobile applications. |
| Goto | C++ supports goto statement. | Java doesn't support goto statement. |
| Multiple inheritance | C++ supports multiple inheritance. | Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java. |
| Operator Overloading | C++ supports operator overloading. | Java doesn't support operator overloading. |
| Pointers | C++ supports pointers. You can write pointer program in C++. | Java supports pointer internally. But you can't write the pointer program in java. It means java has restricted pointer support in java. |
| Compiler and Interpreter | C++ uses compiler only. | Java uses compiler and interpreter both. |
| Call by Value and Call by reference | C++ supports both call by value and call by reference. | Java supports call by value only. There is no call by reference in java. |
| Structure and Union | C++ supports structures and unions. | Java doesn't support structures and unions. |
| Thread Support | C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support. | Java has built-in thread support. |
| Documentation comment | C++ doesn't support documentation comment. | Java supports documentation comment (/\*\* ... \*/) to create documentation for java source code. |
| Virtual Keyword | C++ supports virtual keyword so that we can decide whether or not override a function. | Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default. |
| unsigned right shift >>> | C++ doesn't support >>> operator. | Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator. |
| Inheritance Tree | C++ creates a new inheritance tree always. | Java uses single inheritance tree always because all classes are the child of Object class in java. Object class is the root of inheritance tree in java. |

**Why java doesn't support pointers?**

**Use of pointers:**

* Pointer Arithmetic (Direct Memory manipulation)
* Pointer to Integer Casts

**Issues with Pointers:**

* Invalid references to Memory Address – Source of bugs
* Dangling pointers (Pointer not pointing to a memory location)
* Makes Garbage Collection difficult in Java

Java has pointers but they are not **explicit**, they are called as **References**. And you cannot use them freely like in C/C++.  
  
In Java, types are divided into primitive types (int, long, short, char, byte, double, float, boolean) and reference types (everything else). Object, String, Object[], etc. are all reference types.

The only *real* use for pointers is direct memory manipulation. Since Java doesn't want you to do that (and in fact its garbage-collected memory management would actively interfere with and be broken by manual memory manipulation), there's no need for explicit pointers.

**How to find 3rd highest salary of employee from table?**

|  |  |  |
| --- | --- | --- |
| MySQL | SQL Server | Oracle |
| select salary  from employee  order by salary desc  limit 3,1  SELECT \*  FROM `employee`  ORDER BY `salary` DESC  LIMIT 1 OFFSET 2; | SELECT TOP 1 salary  FROM  (SELECT TOP 3 salary  FROM employee  ORDER BY salary DESC) AS Comp  ORDER BY salary ASC | SELECT salary  FROM  (SELECT salary  FROM employee  ORDER BY salary DESC  LIMIT 3) AS Comp  ORDER BY salary  LIMIT 1; |

(In the offset use 2 if your DB counts result rows from 1 and not from 0.)

**Explain Autoboxing and Unboxing in Java?**

|  |  |
| --- | --- |
| 1. **class** BoxingExample1{ 2. **public** **static** **void** main(String args[]){ 3. **int** a=50; 4. Integer a2=**new** Integer(a);//Boxing 6. Integer a3=5;//Boxing 8. System.out.println(a2+" "+a3); 9. } 10. } | 1. **class** UnboxingExample1{ 2. **public** **static** **void** main(String args[]){ 3. Integer i=**new** Integer(50); 4. **int** a=i; 6. System.out.println(a); 7. } 8. } |
| Output:50 5 | Output:50 |

Boxing is used so that primitive types can be used along with generics in Java.

Eg. List<T> is a generic type and T always expects a Object but primitive types are values not Objects. So List<int>, List<Float> is not possible in java.

That’s why we have wrapper classes which will wrap primitive types into their corresponding wrapper class and make them as Object.

Integer x = 5; // this is not a value, here x is an object which can be used with Generic types easily.

**Why we cannot override Static methods in Java?**

Overriding depends on having an instance of a class. A static method is not associated with any instance of a class so the concept is not applicable.

We **can** declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘**No**’.  
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class **hides** the method in the base class. This is also called **method hiding**.

**Can we overload Static methods in Java?**

The answer is ‘Yes’. We can have two or more static methods with same name, but differences in input parameters. For example, consider the following Java program.

1. // filename Test.java
2. **public** **class** Test {
3. **public** **static** **void** foo() {
4. System.**out**.println("Test.foo() called ");
5. }
6. **public** **static** **void** foo(**int** a) {
7. System.**out**.println("Test.foo(int) called ");
8. }
9. **public** **static** **void** main(String args[])
10. {
11. Test.foo();
12. Test.foo(10);
13. }
14. }

**Can we override private members of a class?**

No, a private method cannot be overridden since it is not visible from any other class.

If you try to define same method in subclass, then your method in subclass won’t even have an idea that parent class also has a method with same name since that method is private limited to parent class only.

|  |  |  |
| --- | --- | --- |
| **Property** | **Abstract Class** | **Interface** |
| Methods: | Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also. | Interface can have only abstract methods. |
| Multiple Inheritance | Abstract class doesn't support | Interface supports multiple inheritance. |
| Final Variables: | An abstract class may contain non-final variables. | Variables declared in a Java interface are by default final. |
| Variables | Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| Implementation | Abstract class can provide the implementation of interface. | Interface can’t provide the implementation of abstract class. |
| Inheritance vs Abstraction: | Abstract class can be extended using keyword “extends”. | Interface can be implemented using keyword “implements” |
| Data Members: | Abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| Multiple implementation: | An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only |
| Example | public abstract class Shape{ public abstract void draw(); } | public interface Drawable{ void draw(); } |

**Difference between Mutex and Semaphore?**

|  |  |
| --- | --- |
| A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa. | A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. |
| At any point of time, only one thread can work with the *entire* buffer | The consumer and producer can work on different buffers at the same time. |
| A mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex.  It means there is ownership associated with mutex, and only the owner can release the lock (mutex). | Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal).  For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wake up. |

**Difference between Contiguous and Non-Contiguous Memory Allocation?**

|  |  |  |
| --- | --- | --- |
| **Property** | **Contiguous** | **Non Contiguous** |
| Basic | Allocates consecutive blocks of memory to a process. | Allocates separate blocks of memory to a process. |
| Overheads | Contiguous memory allocation does not have the overhead of address translation while execution of a process. | Noncontiguous memory allocation has overhead of address translation while execution of a process. |
| Execution rate | A process executes faster in contiguous memory allocation | A process executes quite slower comparatively in noncontiguous memory allocation. |
| Solution | The memory space must be divided into the fixed-sized partition and each partition is allocated to a single process only. | Divide the process into several blocks and place them in different parts of the memory according to the availability of memory space available. |
| Table | A table is maintained by operating system which maintains the list of available and occupied partition in the memory space | A table has to be maintained for each process that carries the base addresses of each block which has been acquired by a process in memory. |
| Wastage | More Memory wastage due to Internal fragmentation | Less Memory Wastage as all the memory blocks are utilized. |

**What are ACID properties?**

A transaction is a very small unit of a program and it may contain several lowlevel tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

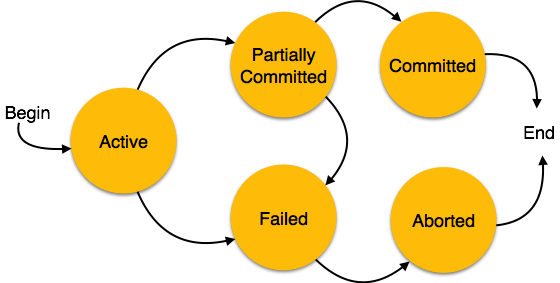
* **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
* **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
* **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
* **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

**What is transaction and what are concurrency control techniques?**

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

Let’s take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account.

States of a Transaction:



Concurrency control protocols can be broadly divided into two categories −

* Lock based protocols
* Time stamp based protocols

Major Protocols

* Locking (e.g., [**Two-phase locking**](https://en.wikipedia.org/wiki/Two-phase_locking) - 2PL) - Controlling access to data by [locks](https://en.wikipedia.org/wiki/Lock_(computer_science)) assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release.
* **Serialization**[**graph checking**](https://en.wikipedia.org/wiki/Serializability#Testing_conflict_serializability) (also called Serializability, or Conflict, or Precedence graph checking) - Checking for [cycles](https://en.wikipedia.org/wiki/Cycle_(graph_theory)) in the schedule's [graph](https://en.wikipedia.org/wiki/Directed_graph) and breaking them by aborts.
* [**Timestamp ordering**](https://en.wikipedia.org/wiki/Timestamp-based_concurrency_control) (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.
* [**Commitment ordering**](https://en.wikipedia.org/wiki/Commitment_ordering) (or Commit ordering; CO) - Controlling or checking transactions' chronological order of commit events to be compatible with their respective precedence order.

**What are smart pointers?**

Smart pointer is a wrapper class over a pointer with operator like \* and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can’t like automatic destruction (yes, we don’t have to explicitly use delete), reference counting and more.

The idea is to make a class with a pointer, destructor and overloaded operators like \* and ->. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically deleted (or reference count can be decremented).

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;    class SmartPtr  {  int \*ptr; // Actual pointer  public:  explicit SmartPtr(int \*p = NULL) { ptr = p; }    // Destructor  ~SmartPtr() { delete(ptr); }    // Overloading dereferencing operator  int &operator \*() { return \*ptr; }  };    int main()  {  SmartPtr ptr(new int());  \*ptr = 20;  cout << \*ptr;    destructor does delete ptr.    return 0;  } | Output:  20  // We don't need to call delete ptr: when the object  // ptr goes out of scope, destructor for it is automatically  // called and |

**Difference between StringBuffer and StringBuilder?**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **String** | **StringBuffer** | **StringBuilder** |
| Storage Area | Constant String Pool | Heap | Heap |
| Modifiable | No (immutable) | Yes( mutable ) | Yes( mutable ) |
| Thread Safe/ Synchronized | Yes | Yes, Synchronized | No, Non Synchronized |
| Performance | Fast | Very Slow | Fast |

|  |  |
| --- | --- |
| 1. **public** **class** BufferTest{ 2. **public** **static** **void** main(String[] args){ 3. StringBuffer buffer=**new** StringBuffer("hello"); 4. buffer.append("java"); 5. System.out.println(buffer); 6. } 7. } | 1. **public** **class** BuilderTest{ 2. **public** **static** **void** main(String[] args){ 3. StringBuilder builder=**new** StringBuilder("hello"); 4. builder.append("java"); 5. System.out.println(builder); 6. } 7. } |

**Java Program to Remove Duplicates from Array?**

|  |
| --- |
| import java.util.LinkedHashSet;  import java.util.Set;    public class RemoveDuplicatesJavaExample  {      static void removeDuplicates(int[] arrayWithDuplicates)      {          System.out.println("Array With Duplicates : ");            for (int i = 0; i < arrayWithDuplicates.length; i++)          {              System.out.print(arrayWithDuplicates[i]+"\t");          }            Set<Integer> set = new LinkedHashSet<Integer>();      //Use HashSet if you don't want insertion order            for (int i = 0; i < arrayWithDuplicates.length; i++)          {              set.add(arrayWithDuplicates[i]);          }            //Converting set into an array            Object[] arrayWithoutDuplicates = set.toArray();            //Printing arrayWithoutDuplicates            System.out.println();            System.out.println("Array Without Duplicates : ");            for (int i = 0; i < arrayWithoutDuplicates.length; i++)          {              System.out.print(arrayWithoutDuplicates[i]+"\t");          }            System.out.println();            System.out.println("==============================");      }        public static void main(String[] args)      {          removeDuplicates(new int[] {15, 21, 11, 21, 51, 21, 11});            removeDuplicates(new int[] {7, 3, 21, 7, 34, 18, 3, 21});      }  } |

**How To Remove Duplicate Elements From An Array In Java Without Using Collection API?**

|  |
| --- |
| import java.util.Arrays;    public class RemoveDuplicatesJavaExample  {      static void removeDuplicates(int[] arrayWithDuplicates)      {          System.out.println("Array With Duplicates : ");            for (int i = 0; i < arrayWithDuplicates.length; i++)          {              System.out.print(arrayWithDuplicates[i]+"\t");          }            //Assuming all elements in input array are unique            int noOfUniqueElements = arrayWithDuplicates.length;            //Comparing each element with all other elements            for (int i = 0; i < noOfUniqueElements; i++)          {              for (int j = i+1; j < noOfUniqueElements; j++)              {                  //If any two elements are found equal                    if(arrayWithDuplicates[i] == arrayWithDuplicates[j])                  {                      //Replace duplicate element with last unique element                        arrayWithDuplicates[j] = arrayWithDuplicates[noOfUniqueElements-1];                        //Decrementing noOfUniqueElements                        noOfUniqueElements--;                        //Decrementing j                        j--;                  }              }          }            //Copying only unique elements of arrayWithDuplicates into arrayWithoutDuplicates            int[] arrayWithoutDuplicates = Arrays.copyOf(arrayWithDuplicates, noOfUniqueElements);            //Printing arrayWithoutDuplicates            System.out.println();            System.out.println("Array Without Duplicates : ");            for (int i = 0; i < arrayWithoutDuplicates.length; i++)          {              System.out.print(arrayWithoutDuplicates[i]+"\t");          }            System.out.println();            System.out.println("==============================");      }        public static void main(String[] args)      {          removeDuplicates(new int[] {4, 3, 2, 4, 9, 2});            removeDuplicates(new int[] {1, 2, 1, 2, 1, 2});            removeDuplicates(new int[] {15, 21, 11, 21, 51, 21, 11});            removeDuplicates(new int[] {7, 3, 21, 7, 34, 18, 3, 21});      }  } |

**Write a Matching parenthesis program in Java?**

|  |
| --- |
| public static boolean CheckParentesis(String str)  {  if (str.isEmpty())  return true;  Stack<Character> stack = new Stack<Character>();  for (int i = 0; i < str.length(); i++)  {  char current = str.charAt(i);  if (current == '{' || current == '(' || current == '[')  {  stack.push(current);  }  if (current == '}' || current == ')' || current == ']')  {  if (stack.isEmpty())  return false;  char last = stack.peek();  if (current == '}' && last == '{' || current == ')' && last == '(' || current == ']' && last == '[')  stack.pop();  else  return false;  }  }  return stack.isEmpty();  } |

**Search a particular string in File using Java?**

|  |
| --- |
| Scanner txtscan = new Scanner(new File("filename.txt"));  while(txtscan.hasNextLine()){  String str = txtscan.nextLine();  if(str.indexOf("word") != -1){  System.out.println("EXISTS");  }  } |

**Java Program to Find factors of a number using For Loop?**

|  |
| --- |
| package SimplerPrograms;    import java.util.Scanner;    public class FactorsOfNumberUsingFor {  private static Scanner sc;    public static void main(String[] args) {  int Number, i;  sc = new Scanner(System.in);    System.out.println("Please Enter any number to Find Factors: ");  Number = sc.nextInt();    for(i = 1; i <= Number; i++) {  if(Number%i == 0) {  System.out.format(" %d ", i);  }  }  }  } |

**Difference between final, finalize () and finally?**

|  |  |  |
| --- | --- | --- |
| Final is a keyword. | Finally is a block. | Finalize is a method. |
| Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 1. **class** FinalExample{ 2. **public** **static** **void** main(String[] args){ 3. **final** **int** x=100; 4. x=200;//Compile Time Error 5. }} | 1. **class** FinallyExample{ 2. **public** **static** **void** main(String[] args){ 3. **try**{ 4. **int** x=300; 5. }**catch**(Exception e){System.out.println(e);} 6. **finally**{System.out.println("finally block is executed");} 7. }} | 1. **class** FinalizeExample{ 2. **public** **void** finalize(){System.out.println("finalize called");} 3. **public** **static** **void** main(String[] args){ 4. FinalizeExample f1=**new** FinalizeExample(); 5. FinalizeExample f2=**new** FinalizeExample(); 6. f1=**null**; 7. f2=**null**; 8. System.gc(); 9. }} |