**Write any ten features of java**

### Ans -: 1. Simple Syntax

Java syntax is very straightforward and very easy to learn. Java removes complex features like pointers and multiple inheritance, which makes it a good choice for beginners.

### 2. Object Oriented

Java is a pure [object-oriented language](https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/). It supports core OOP concepts like,

3. Platform Independent

Java is [platform-independent](https://www.geeksforgeeks.org/java-platform-independent/) because of Java Virtual Machine (JVM).

* When we write Java code, it is first compiled by the compiler and then converted into bytecode (which is platform-independent).
* This byte code can run on any platform which has JVM installed.

4. Interpreted

Java code is not directly executed by the computer. It is first compiled into bytecode. This byte code is then understand by the [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). This enables Java to run on any platform without rewriting code.

5. Scalable

Java can handle both small and large-scale applications. Java provides features like [multithreading](https://www.geeksforgeeks.org/multithreading-in-java/) and[distributed computing](https://www.geeksforgeeks.org/what-is-distributed-computing/) that allows developers to manage loads more easily.

6. Portable

When we write a Java program, the code first get converted into bytecode and this bytecode does not depend on any operating system or any specific computer. We can simply execute this bytecode on any platform with the help of JVM. Since JVMs are available on most devices and that's why we can run the same Java program on different platform

7. Secured and Robust

Java is a reliable programming language because it can catch mistakes early while writing the code and also keeps checking for errors when the program is running. It also has a feature called [exception handling](https://www.geeksforgeeks.org/exceptions-in-java/) that helps deal with unexpected problems smoothly.

8. Memory Management

Memory management in Java is automatically handled by the Java Virtual Machine (JVM).

* Java[garbage collector](https://www.geeksforgeeks.org/garbage-collection-java/) reclaim memory from objects that are no longer needed.
* Memory for objects are allocated in the heap
* Method calls and local variables are stored in the stack.

9. High Performance

Java is faster than old interpreted languages. Java program is first converted into bytecode which is faster than interpreted code. It is slower than fully compiled languages like C or C++ because of interpretation and [JIT](https://www.geeksforgeeks.org/just-in-time-compiler/) compilation process. Java performance is improve with the help of Just-In-Time (JIT) compilation, which makes it faster than many interpreted languages but not as fast as fully compiled languages.

10. Multithreading

[Multithreading](https://www.geeksforgeeks.org/multithreading-in-java/) in Java allows multiple threads to run at the same time.

* It improves CPU utilization and enhancing performance in applications that require concurrent task execution.
* Multithreading is especially important for interactive and high-performance applications, such as games and real-time systems.
* Java provides build in support for managing multiple threads. A [thread](https://www.geeksforgeeks.org/main-thread-java/) is known as the smallest unit of execution within a process.

**What is Unicode and Byte Code?**

Unicode

* Universal Character Set:

Unicode provides a way to represent a wide range of characters, including letters from different languages, symbols, and punctuation marks.

* Code Points:

Each character in Unicode has a unique code point, which is a numerical value representing that character.

* Encodings:

Unicode text is stored and processed as binary data using encodings like UTF-8, UTF-16, and UTF-32.

* UTF-8:

A widely used encoding that uses one byte for ASCII characters and multiple bytes for other characters, making it efficient and backward compatible with ASCII.

* Benefits:

Unicode ensures consistent representation of text across different platforms and languages, preventing display issues and allowing for easier internationalization of software.

Bytecode

* Intermediate Language:

Bytecode is a low-level language that is compiled from a higher-level programming language.

* Virtual Machine Execution:

Bytecode is executed by a virtual machine (VM) like the Java Virtual Machine (JVM).

* Platform Independence:

Because the VM translates the bytecode, it can be run on different platforms without requiring re-compilation.

* Example:

In Java, the source code is compiled into bytecode, which is then executed by the JVM.

* Benefits:

Bytecode enables portability, security, and performance optimization through VM management.

**How java is platform independent?**

Java achieves platform independence through the use of the Java Virtual Machine (JVM). When you compile Java code, it's not translated directly into machine code for a specific operating system or hardware. Instead, it's compiled into bytecode, a platform-neutral intermediate language. The JVM then takes this bytecode and executes it on any device with a compatible JVM, regardless of the underlying operating system or hardware.

Here's a more detailed breakdown:

1. 1. Bytecode Compilation:

Java code is compiled into bytecode, which is a set of instructions that can be executed by the JVM.

1. 2. JVM Execution:

The JVM interprets or JIT compiles the bytecode into machine code for the specific platform it is running on.

1. 3. Platform-Neutrality:

Because bytecode is platform-neutral, the same Java program can run on different operating systems and hardware architectures as long as a compatible JVM is available.

1. 4. "Write Once, Run Anywhere":

This means developers can write Java code once and deploy it on multiple platforms without needing to recompile or make changes to the source code.

In essence, the JVM acts as a bridge between the Java bytecode and the specific operating system and hardware, allowing Java programs to run consistently across different environments.

.**What is JDK and JRE? Differentiate?**

The main difference between JDK (Java Development Kit) and JRE (Java Runtime Environment) is their purpose and what they include. JRE is used to run Java applications, while JDK is used to develop Java applications, including compilation, debugging, and development tools. In essence, JDK contains JRE along with additional development tools.

Here's a more detailed breakdown:

JRE (Java Runtime Environment):

* Purpose: Provides the runtime environment needed for Java applications to execute. It includes the Java Virtual Machine (JVM), class libraries, and other supporting files.
* Tools: Does not include any development tools like compilers or debuggers.
* User: Primarily used by end users who want to run Java applications.

JDK (Java Development Kit):

* Purpose: Provides the environment for developers to write, compile, debug, and run Java applications. It includes JRE and development tools.
* Tools: Includes tools for compiling Java code (e.g., javac), debugging, and other development tasks.
* User: Used by developers to create Java applications.

Analogy: Think of JRE as the engine that runs a car, while JDK is the whole toolkit needed to build and maintain the car, including the engine.

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

In JRE, the Class Loader loads .class files (containing bytecode) into memory. The Verifier ensures the bytecode is valid and adheres to Java's security rules. The JIT (Just-In-Time) compiler dynamically compiles bytecode into native machine code at runtime for performance optimization.

Elaboration:

* Class Loader:

This component of the JRE is responsible for dynamically loading class files into the Java Virtual Machine (JVM) at runtime. It handles the loading of bytecode, which is the intermediate language Java code is compiled into.

* Bytecode Verifier (or Class Verifier):

This part of the JRE verifies the integrity and correctness of the bytecode before it is executed. It checks for potential issues like invalid instruction sequences, type errors, or attempts to access protected resources.

* JIT Compiler:

This is a core component of the JVM that improves performance by compiling frequently executed portions of bytecode into native machine code during runtime. This native code is then executed directly by the hardware, resulting in faster execution times for those specific code sections

**Write/compile/execute Hello World program in Java**

Java Data types, Operators and IF - ELSE

Hello World Program in Java

1. Write the program:

Create a text file named HelloWorld.java and add the following code:

Java

public class HelloWorld {  
 public static void main(String[] args) {  
 System.out.println("Hello, World!");  
 }  
}

2. Compile the program:

Open a command prompt or terminal, navigate to the directory where you saved HelloWorld.java, and execute:

Code

javac HelloWorld.java

This command compiles the Java source code into bytecode, creating HelloWorld.class.

3. Execute the program:

In the same command prompt or terminal, execute:

Code

java HelloWorld

This command runs the compiled Java program, and "Hello, World!" will be printed to the console.

Java Data Types

Java has two main categories of data types:

* Primitive Data Types:
  + byte: 8-bit integer
  + short: 16-bit integer
  + int: 32-bit integer
  + long: 64-bit integer
  + float: 32-bit floating-point
  + double: 64-bit floating-point
  + boolean: true or false
  + char: 16-bit Unicode character
* Non-Primitive (Reference) Data Types:
  + String: Represents sequences of characters.
  + Arrays: Collections of elements of the same type.
  + Classes: Blueprints for creating objects.
  + Interfaces: Contracts for classes.

Java Operators

Operators perform operations on variables and values. Common types include:

* Arithmetic Operators: +, -, \*, /, %
* Assignment Operators: =, +=, -=, \*=, /=, %=
* Comparison (Relational) Operators: ==, !=, >, <, >=, <=
* Logical Operators: && (AND), || (OR), ! (NOT)
* Unary Operators: ++ (increment), -- (decrement)

Java if-else Statement

The if-else statement is used for conditional execution.

Java

public class ConditionalExample {  
 public static void main(String[] args) {  
 int number = 10;  
  
 if (number > 0) {  
 System.out.println("The number is positive.");  
 } else {  
 System.out.println("The number is not positive (it's zero or negative).");  
 }  
 }  
}

* The if block executes if the condition (number > 0) is true.
* The else block executes if the condition is false.
* else if can be used for multiple conditions: if (condition1) { ... } else if (condition2) { ... } else { ... }

**What are the two types of Exceptions in Java? What are the differences between them?**

## Checked Exceptions in Java

Checked Exceptions are exceptions that are checked at compile time. If a method throws a checked Exception, then the exception must be handled using a try-catch block and declared the exception in the method signature using the throw keyword.

### Types of Checked Exception

* **Fully Checked Exception:** A checked exception where all its child classes are also checked (e.g., IOException, InterruptedException).
* **Partially Checked Exception:** A checked exception where some of its child classes are unchecked (e.g., Exception).

Checked exceptions represent invalid conditions in areas outside the immediate control of the program like memory, network, file system, etc. Any checked exception is a subclass of Exception. Unlike unchecked exceptions, checked exceptions must be either caught by the caller or listed as part of the method signature using the throws keyword

## Unchecked Exceptions in Java

Unchecked exception are exceptions that are not checked at the compile time. In Java, exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Consider the following Java program. It compiles fine, but it throws an ArithmeticException when run. The compiler allows it to compile becauseArithmeticException is an unchecked exception.

**Example:**Java program to illustrate the Runtime Unchecked Exception.

Note:

* Unchecked exceptions are runtime exceptions that are not required to be caught or declared in a throws clause.
* These exceptions are caused by programming errors, such as attempting to access an index out of bounds in an array or attempting to divide by zero.
* Unchecked exceptions include all subclasses of the RuntimeException class, as well as the Error class and its subclasses.

The separation into checked and unchecked exceptions sounded like a good idea at the time. Over the years, it has introduced more boilerplate and less aesthetically pleasing code patterns than it solved real problems. The typical pattern within the Java ecosystem is to hide the checked exception within an unchecked one.

**What are the Memory Allocations available in Java?**

* Heap Memory:
  + This is the runtime data area where all objects and arrays are allocated dynamically.
  + It is a shared memory area accessible by all threads in the application.
  + Objects stored in the heap have a longer lifetime and are subject to garbage collection when no longer referenced.
* Stack Memory:
  + Each thread in a Java program has its own private stack.
  + This memory is used for storing local variables, method call information (including parameters and return addresses), and primitive data types.
  + Stack frames are pushed onto the stack when a method is called and popped off when the method completes.
  + Stack memory is short-lived and automatically managed by the JVM.
* Method Area (or Class Area/Metaspace in Java 8+):
  + This area stores class-level data, such as bytecode, static variables, and method definitions.
  + It is a logical part of the heap but is often treated as a separate memory region.
* Program Counter (PC) Register:
  + Each thread has its own PC register, which stores the address of the currently executing instruction.
  + This register is crucial for controlling the flow of execution within a thread.
* Native Method Stack:
  + This stack is used for methods written in languages other than Java (e.g., C/C++) that are called by Java code through the Java Native Interface (JNI).
  + It operates similarly to the Java stack but is used for native method calls.

.**Explain final, finally, finalize.**

| final | finally | finalize |
| --- | --- | --- |
| final keyword applies restrictions on variable, method and classes. | The finally block in exception handling is used with try-catch block. | finalize is a method of object class |
| Prevent modification of variables, inheritance of classes, or overriding of methods. | The code that is written inside finally block is always executed after the try-catch block whether an exception occurs or not . | finalize method in Java is used to perform cleanup operations before an object is garbage collected. |
| Variables, methods, and classes. | Only within a try-catch block. | Objects, specifically by overriding the method in a class |
| Executes when declared. | Always executed after try-catch block. | Called by the garbage collector when an object is about to be deleted, but it's not guaranteed to run. |

. **What is a singleton class in Java? And How to break the singleton class object?**

A Singleton class in Java is a class that ensures it has only one instance during the application's lifecycle, and it provides a global point of access to that instance. This is typically achieved with a private constructor and a static method to return the instance. Breaking a singleton class involves circumventing these restrictions, often through techniques like serialization, cloning, or reflection.

Here's a breakdown of the concepts:

What is a Singleton Class?

* Purpose:

Ensures a class has only one instance, and that instance is accessible globally.

* Implementation:
  + Private Constructor: Prevents external instantiation.
  + Static Instance: A static field that holds the single instance.
  + Static Factory Method: A public static method (like getInstance()) to return the instance, ensuring it's created only once.
* Benefits:
  + Controlled access to a shared resource.
  + Ensures only one object coordinates actions across the system.

How to Break a Singleton Class

* Serialization and Deserialization:
  + Explanation: Serialization converts an object into a stream of bytes, and deserialization recreates the object from the stream. While a singleton class might try to prevent multiple instances, the deserialization process can create a new object, thus breaking the singleton pattern.
  + Mitigation: To prevent this, the singleton class must implement the readResolve() method in the ObjectInputStream to return the existing instance instead of creating a new one.
* Cloning:
  + Explanation: Cloning creates a new object that is a copy of an existing object. If a singleton class implements the Cloneable interface and overrides the clone() method, a new instance can be created via cloning, violating the singleton.
  + Mitigation: To prevent this, the singleton class should not implement Cloneable and should throw CloneNotSupportedException if the clone() method is invoked.
* Reflection:
  + Explanation: Reflection allows access to the internal workings of Java classes, including private fields and constructors. By using reflection, it's possible to bypass the private constructor and create a new instance.
  + Mitigation: While reflection is a powerful tool, it is generally not recommended to break the singleton pattern this way, as it can lead to unexpected behavior and maintainability issues.

**.**

**Differentiate between instance and local variables.**

The primary differences between instance variables and local variables lie in their declaration location, scope, lifetime, and storage location.

Instance Variables:

* Declaration:

Declared inside a class but outside of any method, constructor, or block.

* Scope:

Accessible by all non-static methods within the same class. Their accessibility outside the class depends on their access modifiers (e.g., public, private).

* Lifetime:

Coincides with the lifetime of the object (instance) to which they belong. They are created when an object is created and destroyed when the object is garbage collected.

* Storage:

Stored in the heap memory, with each object having its own distinct copy of the instance variables.

* Initialization:

Can be initialized in a constructor or through initializer blocks. If not explicitly initialized, they receive default values based on their data type (e.g., 0 for numeric types, false for boolean, null for objects).

Local Variables:

* Declaration:

Declared within a method, constructor, or block (e.g., if block, for loop).

* Scope:

Limited to the specific method, constructor, or block in which they are declared. They are not accessible outside that defined scope.

* Lifetime:

Created when the method or block is entered and destroyed when the method or block finishes execution.

* Storage:

Typically stored on the stack memory.

* Initialization:

Must be explicitly initialized before use within their scope. They do not receive default values.

In summary: Instance variables represent the state of an object and persist as long as the object exists, while local variables are temporary variables used for computation within a specific method or block and are destroyed once that execution context ends.

.

**Explain the types of Exceptions in java?**

* Checked Exceptions:
  + These exceptions are checked at compile time, meaning the compiler forces the programmer to handle them.
  + They typically represent external conditions that are beyond the program's control but can be anticipated and handled, such as IOException (e.g., FileNotFoundException) or SQLException.
  + If a method might throw a checked exception, it must either declare the exception using the throws keyword in its signature or handle it using a try-catch block.
* Unchecked Exceptions (Runtime Exceptions):
  + These exceptions are not checked at compile time and typically occur due to programming errors or logical flaws within the code.
  + They are subclasses of RuntimeException and do not need to be explicitly declared or handled, although handling them is still possible.
  + Examples include NullPointerException (accessing a null reference), ArrayIndexOutOfBoundsException (accessing an array index out of bounds), or ArithmeticException (e.g., division by zero).

Additionally, while not strictly a type of exception, Errors are another category of Throwable in Java. Errors represent serious problems that are usually beyond the control of the application and cannot be recovered from, such as OutOfMemoryError or StackOverflowError. Programs typically do not catch or handle Errors.

**Can you call a constructor of a class inside another constructor of same class?**

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

Prerequisite - [Constructors in Java](https://www.geeksforgeeks.org/constructor-in-java/)

Constructor chaining can be done in two ways: 

* Within same class: It can be done using this() keyword for constructors in the same class
* From base class: by using super() keyword to call the constructor from the base class.

Constructor chaining occurs through inheritance. A sub-class constructor's task is to call super class's constructor first. This ensures that the creation of sub class's object starts with the initialization of the data members of the superclass. There could be any number of classes in the inheritance chain. Every constructor calls up the chain till the class at the top is reached.

Why do we need constructor chaining?

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable. 

**explain the Java thread lifecycle?**

The Java Thread Lifecycle describes the various states a thread can be in from its creation to its termination. These states are defined by the Thread.State enum and are managed by the Java Virtual Machine (JVM).

* New:

When a thread is instantiated using new Thread(), it is in the NEW state. At this point, the thread object exists, but it has not yet started execution.

* Runnable:

After calling the start() method on a Thread object, the thread enters the RUNNABLE state. In this state, the thread is ready to be executed by the thread scheduler and may or may not be actively running, depending on CPU availability and the scheduler's decisions.

* Blocked:

A thread enters the BLOCKED state when it is waiting to acquire a monitor lock to enter a synchronized block or method. It remains in this state until the lock becomes available.

* Waiting:

A thread enters the WAITING state when it is waiting indefinitely for another thread to perform a specific action. This can occur when methods like Object.wait(), Thread.join(), or LockSupport.park() are called without a timeout.

* Timed Waiting:

Similar to WAITING, a thread in the TIMED\_WAITING state is waiting for another thread to perform an action, but with a specified timeout. This occurs when methods like Thread.sleep(), Object.wait(long timeout), Thread.join(long timeout), or LockSupport.parkNanos()/LockSupport.parkUntil() are invoked. The thread will transition back to RUNNABLE either when the timeout expires or the action occurs.

* Terminated:

A thread enters the TERMINATED state when its run() method completes execution, either normally or due to an unhandled exception. A terminated thread cannot be restarted.

**Can we inherit a Constructor?**

No, constructors are not inherited in most object-oriented programming languages like Java, C#, or C++. While a subclass can call a constructor from its superclass, it does not inherit the superclass's constructors directly. Each class needs to define its own constructors, and the subclass can call the superclass's constructor using the super() keyword or the class name if it's C++.

Here's a more detailed explanation:

* Not Inherited:

Constructors are not considered members of the class that are passed down to subclasses like methods and variables are.

* Subclass Responsibility:

Each class, including subclasses, needs to define its own constructors to handle object initialization.

* super() or Class Name:

In Java, a subclass can call the superclass's constructor using super() at the beginning of its constructor. In C++, the class name is used.

* Constructor Chaining:

This mechanism allows for constructor chaining, where a subclass can call a superclass's constructor, which in turn can call its own superclass's constructor, and so on, ensuring proper initialization across the inheritance hierarchy.

* Default Constructor:

If no constructor is explicitly defined in a class, the compiler may provide a default constructor (no-argument constructor).

* Why not inheritance?

There are arguments that constructors should not be inherited, as they handle the initial state of the object and are often specific to the class they belong to.

In essence, while subclasses can leverage superclass constructors for part of the initialization process, they must define their own constructors to handle the specific requirements of the subclass and its object state.

**How will you implement method overloading in Java?**

Method overloading in Java is implemented by defining multiple methods within the same class that share the same name but have different parameter lists. This allows for a single method name to perform different actions based on the arguments provided during the method call.

The key ways to achieve method overloading in Java are:

* Changing the number of parameters: Define methods with the same name but varying numbers of arguments.

Java

public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
  
 public int add(int a, int b, int c) {  
 return a + b + c;  
 }  
 }

* Changing the data types of parameters: Define methods with the same name and the same number of parameters, but with different data types for those parameters.

Java

public class Printer {  
 public void print(int num) {  
 System.out.println("Printing integer: " + num);  
 }  
  
 public void print(double num) {  
 System.out.println("Printing double: " + num);  
 }  
 }

* Changing the order of data types of parameters: Define methods with the same name and number of parameters, but with a different sequence of data types for those parameters.

Java

public class DataProcessor {  
 public void process(int id, String name) {  
 System.out.println("Processing ID and Name: " + id + ", " + name);  
 }  
  
 public void process(String name, int id) {  
 System.out.println("Processing Name and ID: " + name + ", " + id);  
 }  
 }

Important considerations for method overloading:

* Return type alone is not sufficient:

Method overloading cannot be achieved by simply changing the return type of a method while keeping the same method name and parameter list.

* Compile-time polymorphism:

Method overloading is an example of static polymorphism (or compile-time polymorphism) because the specific method to be invoked is determined at compile time based on the method signature.

**. What is Runtime Polymorphism?**

Runtime polymorphism, also known as dynamic or late binding, is a feature of object-oriented programming where the specific method to be executed is determined at runtime rather than compile time. This means the program decides which version of a method to call based on the actual object being used, not the type of the object's reference. It's typically achieved through method overriding, where a subclass provides its own implementation of a method already defined in its superclass.

Here's a breakdown:

* Method Overriding:

A subclass implements a method that has the same name and signature as a method in its parent class. This allows the subclass to provide a specialized or different version of the method.

* Dynamic Binding:

The actual method call is resolved at runtime, meaning the JVM (Java Virtual Machine) or the language runtime determines which overridden method to invoke based on the object's actual type.

* "One-Form-Many-Behaviors":

Runtime polymorphism allows a single method call to exhibit different behaviors based on the object's type. For instance, a speak() method in an Animal class could have different implementations in Dog, Cat, or Duck subclasses.

* Benefits:

Runtime polymorphism promotes code flexibility and reusability. It allows you to write generic code that can work with different object types without needing to explicitly know their concrete types.

**How does Garbage Collection work in Java?**

Java's Garbage Collection (GC) is an automatic memory management process within the Java Virtual Machine (JVM) that reclaims memory occupied by objects that are no longer referenced by the running program. This eliminates the need for manual memory deallocation, preventing memory leaks and improving program stability.

The core mechanism typically involves a "mark-and-sweep" algorithm, though various garbage collectors implement this with optimizations:

* Mark Phase:

The garbage collector identifies and marks all "reachable" objects. Reachable objects are those that can be accessed by the running application, starting from "GC roots" (e.g., local variables, static variables, active threads). Objects directly or indirectly referenced from these roots are considered alive.

* Sweep Phase:

The garbage collector then scans the heap memory and identifies all unmarked objects. These unmarked objects are considered "garbage" (unreachable) and their memory is reclaimed, making it available for new object allocations.

* Compact Phase (Optional):

After sweeping, the remaining "live" objects might be scattered throughout the heap, leading to memory fragmentation. Some garbage collectors include a compaction phase, where live objects are moved to a contiguous block of memory, reducing fragmentation and improving memory allocation efficiency.

The JVM triggers garbage collection periodically based on factors like heap memory usage and allocation rates. While System.gc() can be called to suggest a garbage collection run, it is generally not recommended as the JVM manages this process optimally.

**.create a table manufacturers**

Try to insert at least 10 records in the above table

Try to insert at least 2 records with null value

Consider the below tables with estimated columns and then practise below questions.

CUST DTLS

CUST Act DTLS

ACT\_TYPES\_INFO

PROD\_DTLS

EMP

DEPT

86) Fetch all clerks information

87) Display all departments information located at CHICAGO?

88) Display product details manufactured in the current year only?

89) Get the details of cutomers accounts who opened the accounts before thisyear?

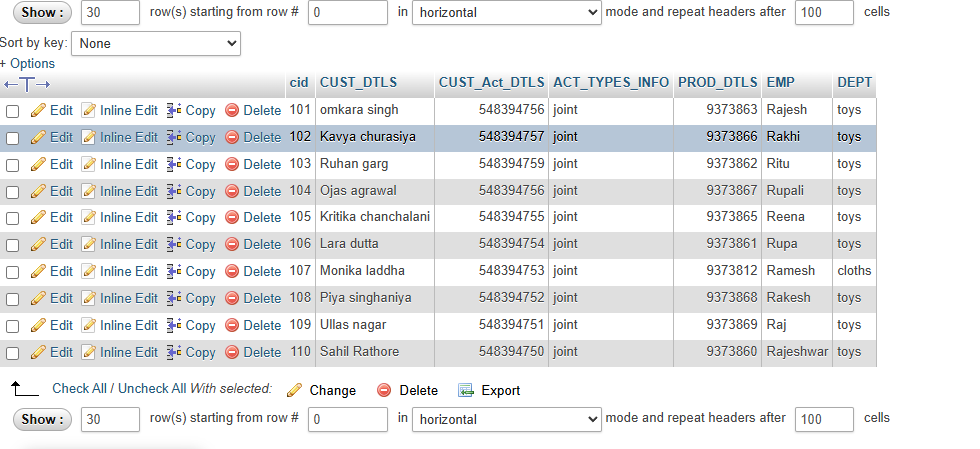
90) Get all SALARY account details?

91) Display customer names and mobile numbers from the city 'Texas'?

select cname, mobile from cust\_dtls where city='Texas';

92) Get the information of Trading account?

93) Display only Expired product details?

select \* from prod\_dtls where exp<sysdate;  
****