# SQL

## SQL Commands

#### \*Data Manipulation

*Change the state of the data. Most commands belong to this category.*

- Insert, Update, Delete, Call, Lock, Explain Call

#### \*Data Description/Definition

*Defines DB schema* - Create, Drop, Alter, Truncate, Comment, Rename

#### \*Data Control

*Deals with offering or denying permissions.* - Grant, Revoke

#### \*Data Query

*Used to perform queries* - Select

#### \*Transaction Control

*Group a set of tasks into a single execution unit.*

- Begin Transaction, Commit, Savepoint, Rollback, Set Transaction, Set Constraint

## What is Normalization and denormalization?

\*Normalization organizes data in DB to remove redundancy.

WHY Normalize Data?

- Remove redundancy: less storage

- Reduces database complexity

- Faster data writing

- Improves data consistency

\*Denormalization adding precomputed redundant data to an already normalized DB

- Faster data reading / availability

- Less complex queries

- Less compute on read operations

## Explain 1NF, 2NF, 3NF. -

#### 1NF:- Atomicity \* A cell holds single-value attributes. (No multi-value or composite attributes)

\* No repeated rows or columns

\* Must have a primary key

#### 2NF:- Partial Dependency \* All non-key attributes must be dependent on the primary key. No derived attributes.

#### 3NF:- Transitive Dependency \*Non-key attributes dependent on another non-key attributes

#### Boyce Codd Normal Form: - Functional Dependency – the value of one attribute can help determine the value of others.

## Denormalization Use Case

\* Showing inventory prices. Benefits-

* Faster data reading / availability
* Less complex queries
* Less compute on read operations

## Primary key versus foreign key?

* Primary key is the unique identifier of a record in a table.
  + ***Example:*** employee\_id in employees’ table.
* Foreign key refers to the primary key in a different table. Used for enforcing referential integrity.
  + ***Example:*** department\_id in employees table is a foreign key that refers to the department\_id (primary key) in the Departments table.

## Alternate and candidate key?

* Candidate key columns or set of columns that can be potential unique identifiers
  + ***Example:*** employee\_id and national\_id are candidates keys since they can be used to identify an employee uniquely.
* Alternate Key a candidate that is not selected to be a primary key.
  + **Example:** If employee\_id is chosen as the primary key then national id becomes the alternate key

## Window functions?

* \*SQL features that allow you to perform calculations across a set of rows that are related to the current row, without collapsing the result set into a single output row.
* **Example:** SUM(sale\_amount) OVER (PARTITION BY employee\_id) computes the total sales for each employee (employee\_id) without collapsing the rows.

## Explain Ranking Functions?

* ROW\_NUMBER – ranks rows without regard to duplicates
* RANK – assigns the same rank to rows with equal values and skips subsequent ranks.
* DENSE\_RANK - does not skip ranks after ties.

#### Ranking Functions Example Table

| sale\_id | employee\_id | sale\_amount | row\_number | rank | dense\_rank |
| --- | --- | --- | --- | --- | --- |
| 3 | 101 | 700 | 1 | 1 | 1 |
| 1 | 101 | 500 | 2 | 2 | 2 |
| 4 | 103 | 500 | 3 | 2 | 2 |
| 5 | 104 | 400 | 4 | 4 | 3 |
| 2 | 102 | 300 | 5 | 5 | 4 |

## Types of Joins?

#### INNER JOINS – returns records that have matching values in both tables.

* + ***Use Case –*** All employees that are in the Managers table.

#### LEFT JOIN – returns all rows from the left table and the matched from the right.

NULLs are returned if no match exists.

* + ***Use Case –*** returns all employees in manager table. Employees not in the managers table have null values to the right.

#### RIGHT JOIN – opposite of the left join

#### FULL JOIN – returns all rows from the left and right table whether there is a match not.

* + ***Use Case:***Can be used for comparison.

#### CROSS JOIN/CARTESIAN PRODUCT – Each row in the left is combined with all the rows from the second table.

* + ***Use Case:*** Find all possible combinations for test cases or reporting scenarios.

#### SELF JOIN – join a table with itself.

* + ***Use case:*** Compares rows within the same table, i.e. when identifying a hierarchy within the data.

## What is subquery?

\*A query that is nested within another query.

#### Detailed Explanation: Subqueries are used to

1. Add a new column to the main query result.

2. Creating a filter.

3. Creating a consolidated source for selecting data.

## What is corelated subquery?

A subquery in a SQL database query that uses values from the outer query.

#### Detailed Explanation:

I. References a column outside its FROM clause.

ii. Executed repeatedly.

Iii. Can be used in a HAVING clause.

iv. Recommended to qualify column names with a table alias to avoid confusion.

#### Use Case:

Useful when the calculation or condition in the subquery depends on the row being processed in the outer query

## What is CTE?

A CTE is a temporary named result set that you define within a WITH clause, which can be referenced multiple times within the same query.

## What is a derived table?

Derived Table is a subquery that is used in the FROM clause of a query. It is temporary and unnamed, and you can only reference it within the scope of the main query.

## Find third highest employee based on salary?

WITH ranked\_e as

(

Select name, salary,

rank() over (order by salary desc) as ranking

from employees

)

select name, salary

from ranked\_e

where ranking = 3;

## Find third highest employee based on salary per department?

With rankedByDepart as (

select name, salary,

rank () over (partition by department\_id order by salary desc) as ranking

from employees)

select name, salary

from rankedByDepart

where ranking = 3;

## How to find duplicate values in a single column?

Return duplicate column and the count of duplicates

SELECT email, COUNT (\*) AS occurrences

FROM employees

GROUP BY email

HAVING COUNT (\*) > 1;

Return all the details of the duplicate

SELECT \*

FROM employees e

JOIN (

SELECT email

FROM employees

GROUP BY email

HAVING COUNT (\*) > 1

) dups ON e.email = dups.email;

#### Links: Handling Duplicates

Link 1: <https://www.sqlshack.com/finding-duplicates-in-sql/>

Link 2: <https://www.naukri.com/code360/library/how-to-find-duplicate-records-in-sql>

## How to find duplicate values in a multiple column?

SELECT customer\_id, order\_date, product\_id, COUNT(\*) AS occurrences

FROM orders

GROUP BY customer\_id, order\_date, product\_id

HAVING COUNT(\*) > 1;

## What are ACID properties?

#### ATOMICITY – each transaction is viewed as a single unit. If one part of the transaction fails, the whole transaction fails.

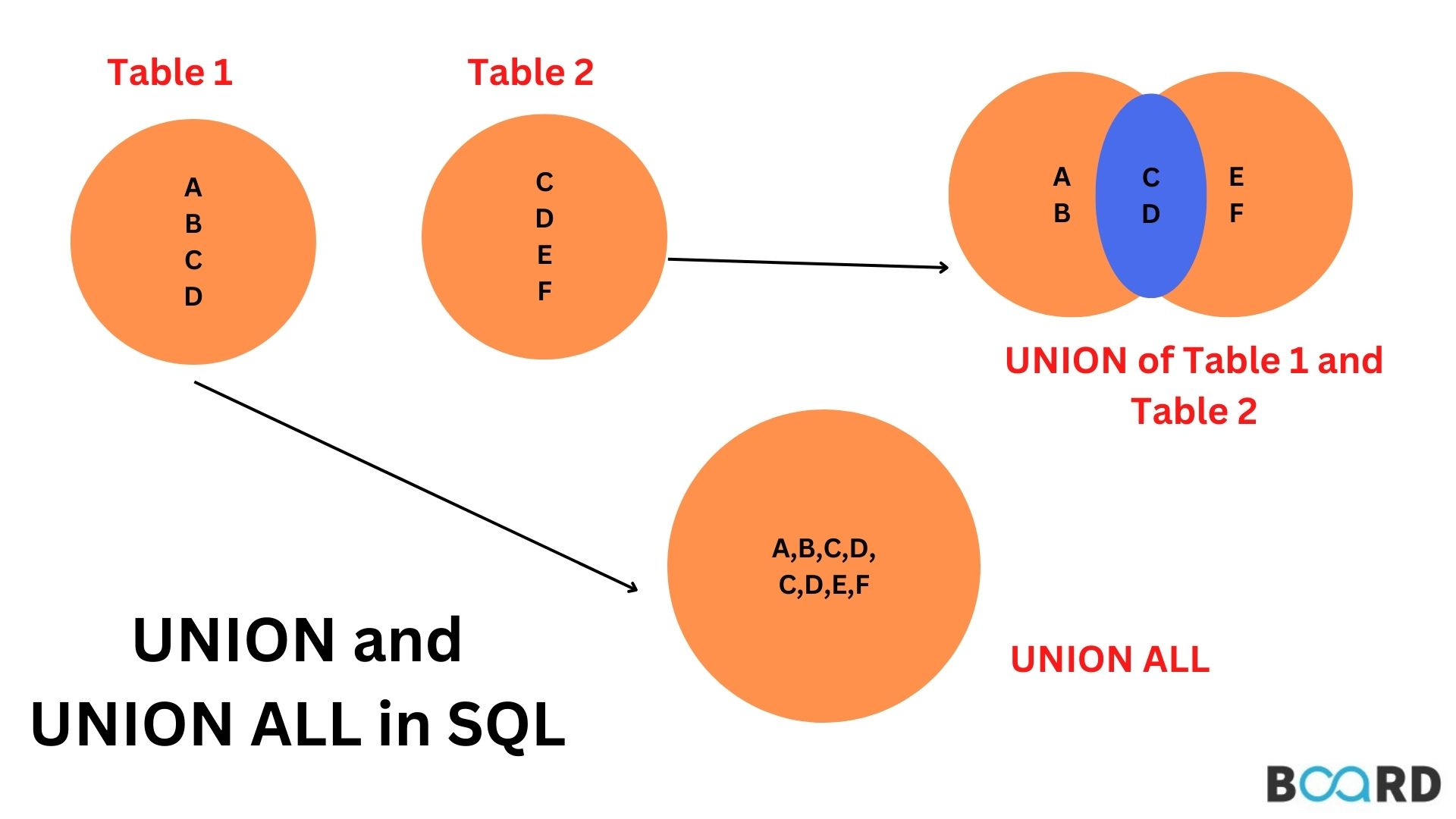
#### CONSISTENCY – transactions must make sure data moves from one valid state to another. A transaction must observe integrity constraints like foreign keys and unique keys.

#### ISOLATION – transactions are executed in isolation from one another. Even if transaction is running concurrently, they should not interfere with each other.

#### DURABILITY - once a transaction is performed its committed permanently. Changes made by a transaction are permanently saved in memory.

## UNION VS UNION ALL

UNION and UNION ALL are set operators in SQL used to concatenate the results of two or more SELECT statements.

The UNION command combines two or more SELECT statements but removes duplicates from the final result set, whereas the UNION ALL statement does not remove the duplicate from the result. UNION ALL simply concatenates all records, including duplicates from the SELECT statement.   


21. Diff between primary key and unique key

### Primary Key: a column (or a combination of columns) that uniquely identifies each row in a table. It enforces both uniqueness and NOT NULL constraints.

A table can only have one primary key

**Unique Key:** a constraint that also ensures the uniqueness of the column or a set of columns, but unlike the primary key, it allows NULL values.

A table can have multiple unique keys.

## Truncate vs Delete

#### DELETE – A DML command that removes a specific row or rows from a table. The WHERE clause can be used to specify which records to delete

#### TRUNCATE – A DDL command that removes all records from a table. TRUNCATE does not use a WHERE clause, and it does not alter the table's structure

## HAVING VS WHERE

#### WHERE - is used to filter rows before any grouping or aggregation takes place. It applies directly to the rows in the table.

#### HAVING - is used to filter groups after aggregation (i.e., after a GROUP BY clause has been applied). It applies to the results of aggregate functions like SUM(), COUNT(), AVG(), etc.

HAVING must be used after: The GROUP BY clause.

## SQL query execution order.

* FROM – Selects and retrieves data from the specified table(s).
* JOIN – Combines data from multiple tables (if any joins are used).
* WHERE – Filters rows based on conditions before aggregation or grouping.
* GROUP BY – Groups rows that have the same values in specified columns into summary rows.
* HAVING – Filters groups based on conditions after aggregation or grouping.
* SELECT – Selects the columns or expressions to be returned in the result set.
* DISTINCT – Removes duplicate rows from the result set (if specified).
* ORDER BY – Orders the result set based on the specified columns.
* LIMIT/OFFSET – Limits the number of rows returned or skips a number of rows before displaying the results.

## What are indexes?

Index – special database objects that store a stored copy of the data that helps database engine find records in a much faster way. Engine does not scan trough all rows

Benefits – Indexes improves the speed of data retrieval operations.

Cons – indexes require additional storage and reduce speed in insertion, update , and deletion because the indexes need to be updated.

## Types of Indexes

#### Clustered Index – defines the physical order of the data in a table usually based on the order of the index. Mostly used for retrieving large ranges of sequential data.

#### Non-clustered Index – does not alter the physical order of the data but implements separate structures with pointers to a row.

Faster for specific lookups when not searching large data sets.

#### Unique Index - No duplicate values are allowed in the indexed column(s). Unique indexes can be either clustered or non-clustered.

Improves read performance while enforcing data integrity.

#### Composite Index (Multi-Column Index): is an index on two or more columns. This is useful for queries that filter by multiple columns at once.

Great for queries that filter on multiple columns simultaneously.

#### Full-Text Index: used to index large text columns, enabling efficient searching of text data for words, phrases, or patterns.

#### Bitmap Index: each distinct value in the column is represented by a bitmap (array of bits).

Bitmap indexes are mostly used in data warehouses or situations where columns have low cardinality.

#### Filtered Index: a non-clustered index that includes only a subset of the rows in a table based on a filter condition (WHERE clause).

Useful for indexing sparse data or frequently queried subsets of data.

## What is surrogate key? Give example where you used it and how.

Surrogate Key: an artificial or system-generated key that serves as the unique identifier for a record in a table, typically used when there is no natural key available, or when natural keys are unsuitable due to their complexity or instability.

Instance: Employee table where the employee\_id keeps on changing based on role changes.

## Ways to optimize SQL query?

#### Denormalizing by adding redundant data into tables (at the cost of additional storage) can speed up read-heavy applications.

#### Avoid cursors (row-by-row operations) that can significantly slow down performance. Use set-based operations instead of cursors whenever possible.

#### Partitioning: Break large tables into smaller, more manageable partitions to improve performance.

#### Batch operations: do batch inserts and updates instead of single query to reduce overheads.

#### Use EXISTS instead of IN: EXISTS is generally faster than IN, especially with large datasets, because EXISTS stops processing once a match is found, while IN continues to check all values.

#### Analyze Execution Plans: Use query execution plans (in SQL Server, MySQL, PostgreSQL) to see how the query is executed and identify bottlenecks.

#### Use Indexes Effectively: Index frequently used columns, avoid over indexing.

#### Use Appropriate Data Types.

#### Limit Data Retrieval.

#### Use Subqueries and CTEs Wisely

#### Optimize JOINs

# Python

## What is init keyword?

It is used for defining a special method called the initializer or constructor in a class. This method is automatically called when an object (an instance of the class) is created.

## What is self keyword?

It is a reference to the current instance of the class. It allows you to access variables, methods, and properties that belong to the class instance.

Code Example: demonstrating \_init\_ and self

**class Car:**

def \_\_init\_\_(self, make, model):

# 'self' is used to set the object's attributes

self.make = make

self.model = model

def display\_info(self):

# 'self' is used to access the object's attributes

print(f"This car is a {self.make} {self.model}.")

# Creating an instance of the Car class

my\_car = Car("Toyota", "Corolla")

# Calling a method using the object

my\_car.display\_info() # Output: This car is a Toyota Corolla.

## What is lambda function?

A small, anonymous function defined using the lambda keyword. Unlike regular functions defined with def, lambda functions can be created in a single line, often for simple, short operations.

***Code Example***

add = lambda x, y: x + y

print(add(3, 4)) # Output: 7

## Difference between lambda and normal function?

| Aspect | Lambda Function | Normal Function |
| --- | --- | --- |
| Syntax | Single line, using lambda | Multi-line, using def |
| Naming | Usually, anonymous | Must have a name |
| Return | Implicit return of single expression | Explicit return statement required |
| Complexity | One-liner, single expression | Can have multiple statements, complex |
| Use Cases | Short, quick, throwaway operations | Complex, reusable logic |
| Readability | Less readable for complex logic | More readable, especially for long code |

## What are generators? When to use one?

A special type of iterable, like lists or tuples, that allow you to iterate over a sequence of values, but lazily—meaning they generate values on the fly as you iterate over them, rather than computing and storing all values at once in memory.

#### Large Data Processing: When dealing with large data that can’t fit into memory, such as reading large files or generating large sequences of numbers.

#### Streams of Data: For continuous or infinite data streams (e.g., sensor readings, network data).

#### Performance Optimization: When you want to avoid the overhead of creating large lists or other data structures upfront.

Most generator functions use yield to produce values lazily, resuming state between iterations.

## Is Python a compiled or interpreted language ? What does that mean?

Python is generally considered an interpreted language, meaning that the source code is executed directly by an interpreter, rather than being compiled into machine code ahead of time like in Java or C++.

## What is the difference between list and tuples in Python?

Lists are mutable collection of items while tuples are immutable collections that can serve as hashable entities.

## What is the difference between list and set in Python?

Lists when you need an ordered collection that may have duplicates and requires indexing. Use sets when you need a collection of unique elements and don’t care about the order, especially for fast membership testing and set operations.

| Feature | Lists | Tuples | Sets |
| --- | --- | --- | --- |
| Syntax | Defined using square brackets [] | Defined using parentheses () | Defined using curly braces {} or set() |
| Ordering | Ordered (maintains the order of items) | Ordered (maintains the order of items) | Unordered (does not maintain any order) |
| Mutability | Mutable (can be modified after creation) | Immutable (cannot be modified after creation) | Mutable (can be modified after creation) |
| Duplicates | Allows duplicate elements | Allows duplicate elements | Does not allow duplicate elements |
| Indexing/Slicing | Supports indexing and slicing | Supports indexing and slicing | Does not support indexing or slicing |
| Performance | Slower for membership tests (in) | Slower than lists for membership tests | Faster for membership tests (in) |
| Methods | Many methods available (e.g., append(), remove(), sort()) | Fewer methods available (e.g., count(), index()) | Supports methods like add(), remove(), union() |
| Use Cases | Ideal for collections of items that may change | Ideal for fixed collections of items (e.g., coordinates, records) | Ideal for collections of unique items, membership tests |
| Memory Usage | Generally, more memory-intensive | More memory-efficient | More memory-efficient due to unique elements |
| Hashable | Not hashable (cannot be used as dictionary keys) | Hashable (can be used as dictionary keys) | Not hashable (cannot be used as dictionary keys) |

## When to use dictionary?

Dictionaries are ideal when you need to store data as key-value pairs, perform fast lookups, or manage dynamic data collections. They are versatile and can be used in a wide variety of applications, from simple data storage to more complex data structures like JSON.

## What are decorators? When to use?

A decorator is essentially a function that takes another function as an argument and (when to use) extends or alters its behavior without modifying the original function’s code. Decorators are commonly used in scenarios such as logging, enforcing access control, instrumentation, and more.

One example: logging the execution time of a function.

## What are Iterators?

Iterators are objects that allow you to traverse a container (like a list, tuple, dictionary, or set) without needing to know the underlying structure of the collection. They implement the iterator protocol, which consists of two methods: \_\_iter\_\_() and \_\_next\_\_().

## What is slicing?

Slicing in Python is a powerful feature that allows you to extract a portion (or "slice") of a sequence, such as a list, tuple, or string.

sequence[start:stop:step]

## What is mutable and immutable?

| Feature | Mutable | Immutable |
| --- | --- | --- |
| Definition | Can be changed after creation | Cannot be changed after creation |
| Examples | Lists, Dictionaries, Sets | Strings, Tuples, Frozensets |
| Modification | Changes the original object | Creates a new object |
| Identity | Same identity after modification | May have a different identity after "modification" |

## Is Python single thread or multithread?

### Single-Threaded: By default, many Python programs run in a single-threaded manner, executing one operation at a time in the main thread. This is typical for simple scripts and applications where concurrency is not required.

### Multi-Threaded: Python supports multi-threading using the threading module, which allows you to create and manage multiple threads within a single process. However, due to the Global Interpreter Lock (GIL), only one thread can execute Python bytecode at a time in CPython, which means true parallel execution of CPU-bound tasks is limited.

## What is GIL

The Global Interpreter Lock (GIL) the standard implementation of Python, to protect access to Python objects and prevent multiple threads from executing Python bytecodes simultaneously.

## What you don’t like about python?

Python is slower than other compiled languages like Java and C++ due to its focus on readability over speed, while this can be managed through optimization techniques and using libraries like NumPy. The language still lags behind when scaling for large projects.

## What is list Comprehension?

List comprehension is a concise and efficient way to create lists in Python. It allows you to generate a new list by applying an expression to each item in an existing iterable (like a list, tuple, or string) and can also include optional filtering conditions.

Code Examples

1. squares = [x \*\* 2 for x in range(10)] # Basic comprehension

2. evens = [x for x in range(10) if x % 2 == 0] # Comprehension with condition

3. uppercase\_words = [word.upper() for word in words] # Applying a functions

4. matrix = [[j for j in range(5)] for i in range(3)] #Nested List Comprehension

## What are Dunder methods? Give examples

Dunder methods**, short for "double underscore methods," are special methods in Python that begin and end with double underscores (e.g., \_\_init\_\_, \_\_str\_\_).**

## What does \_init\_ method do?

Constructor method that is called when an object is created**. Check question 1**

## Difference between array and NumPy library.

### Summary of Differences

| Feature | Built-in Array (Lists) | NumPy Arrays |
| --- | --- | --- |
| Data Type Support | Heterogeneous (mixed types) | Homogeneous (same type) |
| Performance | Slower for numerical operations | Faster due to optimization |
| Functionality | Basic operations | Extensive mathematical functions |
| Memory Consumption | Less memory-efficient | More memory-efficient |
| Ease of Use | Simple for basic tasks | More complex but powerful for data analysis |

# Scala

## What is a trait? When do you use it?

Trait is a special kind of class that defines fields and methods which can be reused by different classes.

Traits are similar to Java interfaces but are more powerful because they allow you to define both abstract and concrete methods.

Traits can also contain variables and can be mixed into classes to provide reusable behavior.

#### Traits Use Case:

##### Code reuse: Traits are often used to share functionality across classes without needing to inherit from a base class.

##### Multiple inheritance: Scala traits allow for multiple inheritance of behavior, making them more flexible than classes.

## What is an abstract class?

Abstract class in Scala is a class that cannot be instantiated directly and is intended to be extended by other classes.

Abstract classes can contain both abstract methods (methods without implementation) and concrete methods (methods with implementation).

They are useful when you want to define a common interface or base functionality for other classes but still require subclasses to provide specific implementations.

## What is the difference between java interface and a scala trait?

| Feature | Java Interface | Scala Trait |
| --- | --- | --- |
| Abstract Methods | Yes | Yes |
| Concrete Methods | Only with default (Java 8+) | Yes |
| Fields (State) | No (only static final constants) | Yes (can define variables) |
| Multiple Inheritance | Yes (through implements) | Yes (through extends and with) |
| Constructor Parameters | No | Yes (Scala 2.12+ allows it) |
| Use Case | Defining contracts | Reusable behavior and state |

Scala traits offer more flexibility than Java interfaces by allowing concrete methods, state, and multiple inheritance, making them powerful tools for code reuse. Java interfaces, especially after Java 8, are becoming more flexible with default methods, but they are still more limited compared to Scala traits.

## What is a singleton?

A singleton in Scala is a class that has only one instance.

In Scala, the most common way to define a singleton object is using the object keyword. Unlike regular classes, you cannot create multiple instances of an object defined with object. Scala automatically ensures that there is only a single instance of the object during the lifetime of the application.

## What is a higher order function?

A higher-order function is a function that does one or both of the following:

1. Takes one or more functions as arguments.
2. Returns a function as its result.

## What is a closure function?

The return value is dependent on external/free variable usually defined outside the function.

## What is a companion object? What are the advantages ? example

## Nil vs Null vs null vs Nothing vs None vs Unit

Nil: Use for empty lists.

* Null: Type of the null literal applies to reference types.
* null: Literal for the absence of a value, avoid in idiomatic Scala.
* Nothing: Bottom type, used to signal non-termination or the absence of any value.
* None: Use with Option to represent the absence of a value.
* Unit: Use for functions that don't return meaningful values, like in side-effecting operations.

## What is pure function?

A pure function gives same output given constant input. Its independent from the environment.

## What is SBT and how have you used it?

SBT (Simple Build Tool) is a build tool used for Scala and Java projects, similar to tools like Maven or Gradle.

It manages project dependencies, compiles code, runs tests, and packages applications.

SBT is specifically designed for Scala projects, integrating closely with Scala's language features and ecosystem.

### SBT Use Cases

1. Managing Dependencies: I have used SBT to manage dependencies by specifying the libraries in the build.sbt file. For example, adding a library like Akka, Spark, or Scalatest for testing:
2. Compiling and Running Scala Code: SBT simplifies compiling and running Scala programs. I typically use the compile and run tasks:
3. Testing: I have written unit tests using frameworks like Scalatest and executed them with the test task:
4. Custom Tasks: For larger projects, I've defined custom tasks in build.sbt for repetitive tasks such as packaging or generating documentation.
5. Handling Multi-Module Projects: In projects with multiple sub-modules, I’ve used SBT’s multi-project build feature to define and compile related modules in one project.

## What is currying?

Currying in Scala (and functional programming in general) is the process of transforming a function that takes multiple arguments into a sequence of functions, each taking a single argument.

In simpler terms, a curried function doesn't take all of its arguments at once, but rather takes them one by one, returning a new function for each argument.

##### Currying Use Cases

- Currying transforms a function with multiple parameters into a series of functions, each taking one argument at a time.

- It enables partial application and function composition, making functional programming more flexible and expressive.

- Currying improves readability and reuse by allowing functions to be applied in stages.

## Difference between currying and higher-order functions

Currying is about transforming functions to accept one argument at a time, enabling partial application.

Higher-Order Functions are functions that operate on other functions, either taking them as parameters or returning them.

Both concepts enhance functional programming by improving modularity, abstraction, and code reuse.

## Difference between var and val?

Var – used to declare mutable variable

Val – used to declare immutable values

## What is case class?

A case class in Scala is a special type of class that comes with built-in functionality, making it easier to create immutable data structures.

Case classes are commonly used for modeling data and simplifying operations like pattern matching, equality checks, and immutability.

## Why/when to use case class? Example

1. Data Modeling: Case classes are great for representing domain models or entities.
2. Pattern Matching: They work perfectly with Scala’s powerful pattern matching mechanism, making it easy to destructure objects and write concise matching logic.
3. Immutability: If your data should remain constant once created, case classes provide built-in immutability by default.
4. Functional Programming: Case classes fit well in functional programming paradigms, where immutability and value equality are fundamental.
5. Simplicity: You need a quick and convenient way to create data classes with less boilerplate code for equality checks, string representations, and copy operations.

### Example Use Case: Modeling a Domain Entity

Let’s say you are working on a library management system. You need to model a Book entity with attributes like title, author, and year. Since books in the system will be immutable (no need to modify their data after creation), and you want to compare books based on their content (not references), a case class would be a perfect fit.

### Why Case Class Is Useful in This Example:

1. Equality by Value: When comparing book1 and book3, the comparison is based on their content (title, author, and year) rather than memory references.
2. Immutability: Once a book is created, its data cannot be altered, which is often desired in systems where data consistency is important.
3. Pattern Matching: You can easily extract and use the fields of the book using pattern matching.
4. Boilerplate-Free: Scala automatically generates methods like equals, hashCode, toString, and copy, reducing the amount of code you need to write.
5. Data Representation: Case classes are perfect for modeling simple data entities (like Book) in an elegant and concise way.

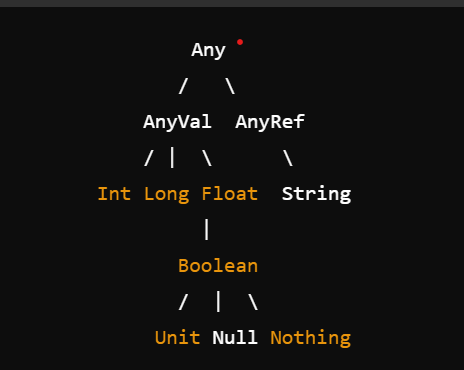
## Difference between case class and normal class?

| Feature | Case Class | Normal Class |
| --- | --- | --- |
| Immutability | Immutable by default | Can be mutable |
| Method Generation | Automatically generates equals, hashCode, toString, and copy methods | Requires manual implementation |
| Pattern Matching | Supports pattern matching | Does not support pattern matching directly |
| Instantiation | Can be instantiated without new | Requires new keyword |
| Use Case | Best for modeling immutable data and entities | Best for complex objects with behavior and state |

#### Use case class vs normal class

* Use case classes when you need simple data structures with immutability, automatic equality checks, and pattern matching.
* Use normal classes when you need more complex behavior, mutable state, or a class that requires more customized implementations of methods.

## Scala type hierarchy?



Any: The root type of all types in Scala.

AnyVal: Base for value types (e.g., Int, Double).

AnyRef: Base for reference types (e.g., classes, collections).

Null: Subtype of all reference types, representing "no value."

Nothing: Subtype of all types, representing no value or computation that doesn't terminate.

Unit: Represents "no value," similar to void in other languages.

## What are partially applied functions?

partially applied functions in Scala are functions that take fewer arguments than they are defined to accept, resulting in a new function that can take the remaining arguments later. This concept allows you to create more specific functions from general ones, promoting code reuse and flexibility.

### Key Features of Partially Applied Functions

a. Function Definition: You define a function that takes multiple parameters, but you call it with fewer parameters, resulting in a new function that expects the remaining parameters.

b. Closure: The partially applied function "closes over" the arguments that were provided, retaining their values for use when the new function is eventually called.

c. Higher-Order Functions: Partially applied functions are often used in conjunction with higher-order functions, where functions can be passed as parameters or returned as results.

## What is tail recursion.

Tail recursion is a form of recursion where the recursive call is the last operation performed in the function.

**In other words**, if a function is tail-recursive, the result of the recursive call is immediately returned as the final result of the function. This allows the compiler or interpreter to optimize the recursive calls, preventing stack overflow errors and reducing the amount of memory used for function calls.