**OOPS**

**Object**

Any entity that has state and behavior is known as an object. An Object can be defined as an instance of a class.

Class

*Collection of objects* is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object.

### **Inheritance**

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

### **Polymorphism**

If one task is performed in different ways, it is known as polymorphism.

In Java, we use method overloading and method overriding to achieve polymorphism.

#### **Abstraction**

Hiding internal details and showing functionality is known as abstraction

In Java, we use abstract class and interface to achieve abstraction

### **Encapsulation**

Binding (or wrapping) code and data together into a single unit are known as encapsulation

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

### **Association**

Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

* One to One
* One to Many
* Many to One, and
* Many to Many

1) OOPs makes development and maintenance easier, whereas, in a procedure-oriented programming language, it is not easy to manage if code grows as project size increases.

2) OOPs provides data hiding, whereas, in a procedure-oriented programming language, global data can be accessed from anywhere.

### **Types of Variables**

There are three types of variables in java:

* local variable
* instance variable
* static variable

#### **1) Local Variable**

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

#### **2) Instance Variable**

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as [static](https://www.javatpoint.com/static-keyword-in-java).

It is called an instance variable because its value is instance-specific and is not shared among instances.

***Instance refers to the copy of the object at a particular time whereas object refers to the memory address of the class***. an instance is one occurrence of a class or object. For example, a program may have a class/object named Animal, but there could be many instances of Animal, such as lion, cat, and dog

#### **3) Static variable**

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

**public** **class** A

{

**static** **int** m=100;//static variable

**void** method()

    {

**int** n=90;//local variable

    }

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

    {

**int** data=50;//instance variable

    }

}

**Constructors in Java**

a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class.

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

* The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
* The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.
* Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

 //creating three arg constructor

    Student5(**int** i,String n,**int** a){

    id = i;

    name = n;

    age=a;

    }

**void** display(){System.out.println(id+" "+name+" "+age);}

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

    Student5 s1 = **new** Student5(111,"Karan");

    Student5 s2 = **new** Student5(222,"Aryan",25);

    s1.display();

    s2.display();

   }

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

### Explain the use of **final keyword** in variable, method and class.

In Java, the final keyword is used as defining something as constant /final and represents the non-access modifier.

* **final variable:**
  + When a variable is declared as final in Java, the value can’t be modified once it has been assigned.
  + If any value has not been assigned to that variable, then it can be assigned only by the constructor of the class.
* **final method:**
  + A method declared as final cannot be overridden by its children's classes.
  + A constructor cannot be marked as final because whenever a class is inherited, the constructors are not inherited. Hence, marking it final doesn't make sense. Java throws compilation error saying - modifier final not allowed here
* **final class:**
  + No classes can be inherited from the class declared as final. But that final class can extend other classes for its usage

# **Java static keyword**

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly.

## 1) Java static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer to the common property of all objects
* The static variable gets memory only once in the class area at the time of class loading.

**class** Student{

**int** rollno;//instance variable

   String name;

**static** String college ="ITS";//static variable

   //constructor

   Student(**int** r, String n){

   rollno = r;

   name = n;

   }

   //method to display the values

**void** display (){System.out.println(rollno+" "+name+" "+college);}

}

# **this keyword in Java**

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.

Usage of Java this keyword

Here is given the 6 usage of java this keyword.

* this can be used to refer current class instance variable.
* this can be used to invoke current class method (implicitly)
* this() can be used to invoke current class constructor.
* this can be passed as an argument in the method call.
* this can be passed as argument in the constructor call.
* this can be used to return the current class instance from the method.

**Inheritance** represents the **IS-A relationship** which is also known as a parent-child relationship.

### **Why use inheritance in java**

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

# **Aggregation in Java**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

**class** Employee{

**int** id;

String name;

Address address;//Address is a class

...

}

**Method Overloading**.

multiple methods having same name but different in parameters, it is known as **Method Overloading**.

### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

# **Method Overriding in Java**

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

#### **Rules for Java Method Overriding**

The method must have the same name as in the parent class

The method must have the same parameter as in the parent class.

There must be an IS-A relationship (inheritance).

**class** Vehicle{

**void** run(){

System.out.println("Vehicle is running");}

}

### **without method overriding**

//Creating a child class

**class** Bike **extends** Vehicle {

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

  //creating an instance of child class

  Bike obj = **new** Bike();

  //calling the method with child class instance

  obj.run();

  }

}

### **Example of method overriding**

**class** Vehicle {

  //defining a method

**void** run(){System.out.println("Vehicle is running");}

}

//Creating a child class

**class** Bike2 **extends** Vehicle{

  //defining the same method as in the parent class

**void** run(){System.out.println("Bike is running safely");}

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

  Bike2 obj = **new** Bike2();//creating object

  obj.run();//calling method

  }

}

o/p: Bike is running safely

# **Super Keyword in Java**

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

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

## Usage of Java super Keyword

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

### **Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

**1.**

**abstract** **class** Bike{

**abstract** **void** run();

}

**class** Honda4 **extends** Bike

{

**void** run(){System.out.println("running safely");}

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

 Bike obj = **new** Honda4();

 obj.run();

}

}

o/p: running safely

**2.**

**abstract** **class** Shape{

**abstract** **void** draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

**class** Rectangle **extends** Shape{

**void** draw(){System.out.println("drawing rectangle");}

}

**class** Circle1 **extends** Shape{

**void** draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

**class** TestAbstraction1{

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

Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method

s.draw();

}

}

drawing circle

# **Java 8**

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods

# **Java Lambda Expressions**

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface.

## Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an annotation @FunctionalInterface, which is used to declare an interface as functional interface.

(argument-list) -> {body}

@FunctionalInterface  //It is optional

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample2 {

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

**int** width=10;

        //with lambda

        Drawable d2=()->{

            System.out.println("Drawing "+width);

        };

        d2.draw();

    }

}

2……..

**interface** Sayable{

**public** String say(String name);

}

**public** **class** LambdaExpressionExample4{

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

        // Lambda expression with single parameter.

        Sayable s1=(name)->{

**return** "Hello, "+name;

        };

        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses

        Sayable s2= name ->{

**return** "Hello, "+name;

        };

        System.out.println(s2.say("Sonoo"));

    }

}

Output:

Hello, Sonoo

Hello, Sonoo

# **Java forEach loop**

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

**public** **class** ForEachExample {

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

        List<String> gamesList = **new** ArrayList<String>();

        gamesList.add("Football");

        gamesList.add("Cricket");

        gamesList.add("Chess");

        gamesList.add("Hocky");

        System.out.println("------------Iterating by passing lambda expression--------------");

        gamesList.forEach(games -> System.out.println(games));

    }

}

**Collections**

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet)

**Collectors**

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

**import** java.util.stream.Collectors;

**import** java.util.List;

**import** java.util.ArrayList;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** CollectorsExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList =

                productsList.stream()

                            .map(x->x.price)         // fetching price

                            .collect(Collectors.toList());  // collecting as list

        System.out.println(productPriceList);

    }

}

Output: [25000.0, 30000.0, 28000.0, 28000.0, 90000.0]

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList2 =productsList.stream()

                                     .filter(p -> p.price > 30000)// filtering data

                                     .map(p->p.price)        // fetching price

                                     .collect(Collectors.toList()); // collecting as list

        System.out.println(productPriceList2);

    }

}

**Output:** [90000.0]

Java stream provides a method filter() to filter stream elements on the basis of given predicate.

## Microservices

"Microservices are the small services that work together." Microservice architectural style is an approach to develop a single application as a suite of small services. Each microservice runs its process and communicates with lightweight mechanisms.

* These are the services which are exposed by REST.
* These are small well-chosen deployable units.

Benefits of Microservices

Small Modules

Easier Process Adaption

Increased Security

Open Standards

**Jenkins**

Jenkins is an open source automation tool that allows continuous integration and continuous delivery of software projects. It can build, test, and deploy software across multiple platforms and environments, using various plugins and tools. Jenkins is written in Java and can run on any operating system that supports

**Kubernetes**

an open-source container orchestration platform developed by Google. It provides a powerful set of tools and features to automate the deployment, scaling, and management of containerized applications.

Containers are lightweight, portable units that package the application code along with its dependencies and configurations.

Key features of Kubernetes include:

* Container orchestration
* Service discovery and load balancing: Kubernetes enables services to discover and communicate with each other automatically.
* Scaling: Kubernetes allows you to scale your application up or down depending on demand, ensuring resources are efficiently utilized.

**RestTemplate**

Spring RestTemplate is a synchronous REST client performing HTTP requests using a simple template-style API. It uses an underlying HTTP client library,

We will create a Bean for Rest Template to auto wiring the Rest Template object.

**Enabling @Autowired Annotations**

The Spring framework enables automatic dependency injection. In other words, by declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans. This is called Spring bean autowiring.

**Jenkins Pipeline**

Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a suite of plugins which supports implementing and integrating continuous delivery pipelines into Jenkins.

A continuous delivery (CD) pipeline is an automated expression of your process for getting software from version control right through to your users and customers. Every change to your software (committed in source control) goes through a complex process on its way to being released. This process involves building the software in a reliable and repeatable manner, as well as progressing the built software (called a "build") through multiple stages of testing and deployment.

Pipeline provides an extensible set of tools for modeling simple-to-complex delivery pipelines "as code" via the [Pipeline domain-specific language (DSL) syntax](https://www.jenkins.io/doc/book/pipeline/syntax).

The definition of a Jenkins Pipeline is written into a text file (called a [Jenkinsfile](https://www.jenkins.io/doc/book/pipeline/jenkinsfile)) which in turn can be committed to a project’s source control repository.

Jenkins is, fundamentally, an automation engine which supports a number of automation patterns. Pipeline adds a powerful set of automation tools onto Jenkins, supporting use cases that span from simple continuous integration to comprehensive CD pipelines. By modeling a series of related tasks, users can take advantage of the many features of Pipeline:

* **Code**: Pipelines are implemented in code and typically checked into source control, giving teams the ability to edit, review, and iterate upon their delivery pipeline.
* **Durable**: Pipelines can survive both planned and unplanned restarts of the Jenkins controller.
* **Pausable**: Pipelines can optionally stop and wait for human input or approval before continuing the Pipeline run.
* **Versatile**: Pipelines support complex real-world CD requirements, including the ability to fork/join, loop, and perform work in parallel.
* **Extensible**: The Pipeline plugin supports custom extensions to its DSL [[1](https://www.jenkins.io/doc/book/pipeline/#_footnotedef_1)] and multiple options for integration with other plugins.

**Spring Boot**

Spring Boot is a Java-based framework for creating digital services that are testable, easily maintainable and manageable by a small team, fill business needs and can be independently deployed without a larger application."

*Spring is a framework for web applications. The framework has tools and libraries that you can use to create custom applications. Spring Boot is a module of Spring that you can use to create a* **REST APIs***."*

The primary feature of the Spring Framework is **dependency injection**. And develop loosely coupled application.

The primary feature of Spring Boot is **Autoconfiguration**. And develop stand alone application. Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. and **in-memory** database such as **H2**.

* *There are several ways to set up a project in Spring Boot. One way is to use the Spring Initialzr with Eclipse and Maven*
* *Autoconfiguration is an important feature of Spring Boot that was created out of a need to reduce the complexity of configuration in the Spring framework. Autoconfiguration does this by offering different starters*
* *"The exclude feature of the @EnableAutoConfiguration annotation allows you to disable autoconfiguration*

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| To test the Spring project, we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |

Describe some features of Spring Boot.

**Auto-configuration**

**Starter dependencies:** Spring Boot provides a collection of starter dependencies, which are pre-configured dependencies that bundle commonly used libraries and configurations for specific purposes, such as web applications, data access, security, testing, etc.

**Embedded web server**

**Actuator:** Spring Boot Actuator is a set of production-ready features that provide operational insights into your application. It includes endpoints for monitoring application health, metrics, logging, configuration properties, and more. This is invaluable for monitoring and managing your application in production.

**Spring Boot CLI:** Spring Boot Command Line Interface (CLI) allows you to quickly create, test, and run Spring Boot applications using command-line tools

Spring Data REST uses interfaces that extend *JpaRepository and*provides CRUD REST APIs for entities

In a typical Spring-based application, we create three-layer architecture  need to manually write REST APIs ( create, update, delete, get, pagination, sorting etc).

If we want to create a CRUD REST APIs for another entity: For example, Department, Project, Company, User, etc. Then, we need to create a controller for all these entities and create REST APIs. **Spring Data REST module is the solution.**Spring Data REST uses interfaces that extend *JpaRepository and*provides CRUD REST APIs for entities for FREE - Helps to minimize the boiler-plate controller layer code.

* what is the use of application properties file

The application.properties file is a configuration file commonly used in Spring Boot applications to externalize configuration settings

**SQL**

CREATE DATABASE TESTDB;

USE TESTDB;

CREATE TABLE CUSTOMERS(

ID INT,

FNAME VARCHAR(25),

LNAME VARCHAR(25),

CITY VARCHAR(25)

);

INSERT INTO CUSTOMERS

VALUES(value1 , value2);

ALTER TABLE CUSTOMERS

ADD EMAIL VARCHAR(366);

ALTER TABLE CUSTOMERS

DROP COLUMN EMAIL;

**BETWEEN**

SELECT \* FROM CUSTOMERS

WHERE PRICE BETWEEN 60 AND 70;

**LIKE**

SELECT \* FROM CUSTOMERS

WHERE CITY LIKE ‘s%’;

**IN**

SELECT \* FROM CUSTOMERS

WHERE CITY IN (‘PARIS’, ‘LONDON’);

**AND OR NOT**

SELECT \* FROM CUSTOMERS

WHERE COUNTRY=’GERMANY’AND CITY=’BERLIN’;

SELECT \* FROM CUSTOMERS

WHERE CITY=’BERLIN’ OR CITY=’MINCHUN’;

SELECT \* FROM CUSTOMERS

WHERE NOT COUNTRY=’GERMANY’;

**ORDER BY**

* Sort customers by country column

SELECT \* FROM CUSTOMERS

ORDER BY COUNTRY DESC ID ASC;

**IS NULL**

SELECT NAME FROM CUSTOMERS

WHERE ADDRESS IS NULL;

* List all customers with a value in address field

SELECT NAME FROM CUSTOMERS

WHERE ADDRESS IS NOT NULL;

**UPDATE**

UPDATE CUSTOMERS

SET CONTACTNAME= ‘ALBERT’ , CITY=’PARIS’

WHERE CUSTOMERID=1;

**DELETE**

DELETE FROM CUSTOMERS

WHERE ID=102 AND NAME=’ALBERT’;

**TOP**

SELECT TOP 3 \* FROM CUSTOMERS; // return top 3 records

SELECT \* FROM CUSTOMERS

LIMIT 3;

**MIN(), MAX()**

SELECT MIN(PRICE) AS SMALLEST PRICE, MAX(PRICE) AS LARGEST PRICE

FROM PRODUCTS;

**COUNT(), AVG(), SUM()**

SELECT COUNT(PRODUCTID), AVG(PRICE),SUM(QUANTITY)

FROM PRODUCTS;

* Return the number of records that have price value set to 18

SELECT COUNT(\*) FROM PRODUCTS

WHERE PRICE =18;

**LIKE**

* START WITH a

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE ‘a%’;

* End with a

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE ‘%a’;

* A in any position

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE ‘%a%’

* A In second position

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE ‘\_a%’;

* Start with a and atleast 3 characters in length

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE ‘a\_\_%’

* Start with a and ends with o

SELECT \* FROM CUSTOMERS

WHERE NAME LIKE ‘a%o’;

* City starting with b s or p

SELECT \* FROM CUSTOMERS

WHERE CITY LIKE ‘[bsp]%’;

* Select products with price in range 20 and 50 and do not show products with id 1 2 3

SELECT \* FROM CUSTOMERS

WHERE PRICE BETWEEN 60 AND 70 AND ID NOT IN(1,2,3);

**JOINS**

SELECT O.ORDERID,

C.CUSTOMERNAME, O.ORDERDATE

FROM ORDERS O

INNER JOIN CUSTOMERS C ON

O.CUSTOMERID=C.CUSTOMERID;

**UNION**

* Combine the result of all select statements
* Every select statement have same no of columns, similar data types , same order
* UNION selects only distinct values but UNION ALL allow duplicate values

SELECT CITY FROM CUSTOMERS

UNION

SELECT CITY FROM SUPPLIERS

ORDER BY CITY;

* Return the GERMAN cities

SELECT CITY FROM CUSTOMERS

WHERE COUNTRY =’GERMANY’

UNION

SELECT CITY FROM SUPPLIERS

WHERE COUNTRY =’GERMANY’

ORDER BY CITY;

**GROUP BY**

Number of customers in each country

SELECT COUNT(CUSTOMERID) , COUNTRY

FROM CUSTOMERS

GROUP BY COUNTRY

ORDER BY COUNT(CUSTOMERID) DESC;

* 2 TABLES

ORDER AND SHIPPER

SHIPPER ID IS THE FOREIGN KEY

List the number of orders sent by each shipper

SELECT S.SHIPPERNAME,

COUNT(O.ORDERID) AS NO OF ORDERS

FROM ORDERS O

INNER JOIN SHIPPER S ON

O.SHIPPERID = S.SHIPPERID

GROUP BY SHIPPERNAME;

**HAVING**

* List the number of customers in each country and include country having more than 5

Select COUNTRY , COUNT(CUSTOMERID)

FROM CUSTOMERS

GROUP BY COUNTRY

HAVING COUNT(CUSTOMERID)>5;

* List if the employees David and ram have registered more than 25 orders

2 tables order and employee common employid

SELECT EMPLOYEE.LASTNAME,

COUNT(ORDER.ORDERID) AS NO

FROM ORDERS

INNER JOIN EMPLOYEES ON

EMPLOYEE.EMPLOYEEID=ORDERS.EMPLOYEEID

WHERE LAST NAME=’DAVID’ OR LAST NAME=’RAM’

GROUP BY LASTNAME

HAVING COUNT(ORDER.ORDERID) > 25;

* List suppliers with product prize less than 20

SELECT SUPPLIERNAME

FROM SUPPLIERS

WHERE EXISTS (SELECT PRODUCT NAME

FROM PRODUCTS

WHERE PRODUCT.SUPPLIER.ID=SUPPLIERS.SUPPLIERID AND PRIZE <20);

**ANY**

* List the product name if it finds any records in the order table has quantity=10

SELECT PRODUCT NAME FROM

PRODUCTS

WHERE PRODUCTID =ANY(

SELECT PRODUCTID FROM ORDER

WHERE QUANTITY=10);

**ALL**

SELECT ALL NAME

FROM PRODUCTS

WHERE TRUE;

**SELECT INTO** - copies contents to another table

Select \* INTO CUSTOMERBACKUP

FROM CUSTOMERS;

**INSERT INTO**

INSERT INTO CUSTOMERS

(NAME, CITY,COUNTRY)

SELECT SUPPNAME, CITY, COUNTRY

FROM SUPPLIERS

WHERE COUNTRY =’GERMANY’;

FOREIGN KEY

CREATE TABLE ORDERS(

ORDERID INT NOT NULL,

PERSONID INT,

PRIMARY KEY (ORDERID),

FOREIGN KEY (PERSONID)

REFERENCES PERSONS(PERSONID));

**VIEW**

CREATE VIEW BRAZIL CUSTOMERS AS

SELECT NAME, ID

FROM CUSTOMERS

WHERE COUNTRY=’BRAZIL’;

**MERGING**

TARGET, SOURCE

2 TABLES -- PRODUCT LIST, UPDATED LIST

* UPDATE PRODUCT LIST BASED ON UPDATED LIST

MERGE target\_table AS TARGET

USING source\_table AS SOURCE

ON CONDITION

WHEN MATCHED

THEN UPDATE

WHEN NOT MATCHED BY TARGET

THEN INSERT

WHEN NOT MATCHED BY SOURCE

THEN DELETE

EG:

MERGE PRODUCT\_LIST AS TARGET

USING UPDATE\_LIST AS SOURCE

ON TARGET.PID=SOURCE.PID

WHEN MATCHED

AND TARGET.PNAME<>SOURCE.PNAME

OR TARGET.P\_PRICE <> SOURCE.P\_PRICE

THEN UPDATE

SET TARGET.PNAME=SOURCE.PNAME,

TARGET.P\_PRICE=SOURCE.P\_PRICE

WHEN NOT MATCHED BY TARGET

THEN INSERT (PID,PNAME,P\_PRICE)

VALUES(SOURCE.PID, SOURCE.PNAME, SOURCE.P\_PRICE)

WHEN NOT MATCHED BY SOURCE

THEN DELETE