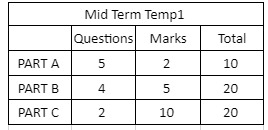
MIDTERM PATTERN

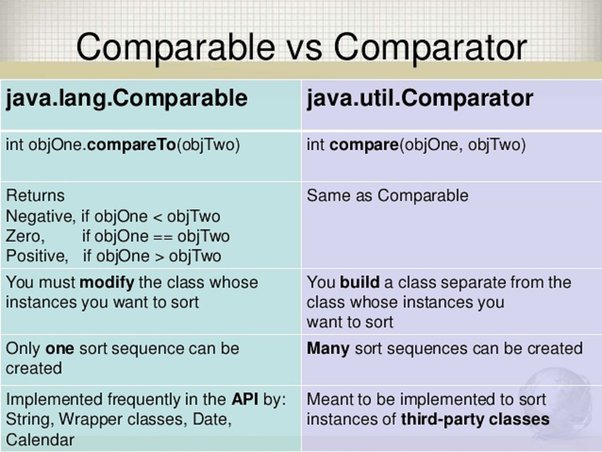


REFER CLASSNOTES AND LABNOTES FOR PARTA AND PARTC

PARTB WILL BE DESCRIPTIVE, LIKE BELOW , U CAN ADD MORE QUESTIONS.

Q. Differentiate Comparable and Comparator.

Ans.



Q. WHAT IS GENERIC PROGRAMMING? WHAT ARE THE ADVANTAGES OF JAVA GENERICS.

ANS - The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

ADVANTAGES-

**Type-safety:** We can hold only a single type of objects in generics. It doesnot allow to store other objects.

**Type casting is not required:** There is no need to typecast the object.

**Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Q. WHAT ARE THE WILDCARD IN JAVA GENERIC?

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

EXAMPLE -

**import** java.util.\*;

**abstract** **void** draw();

}

**abstract** **class** Shape{

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

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

}

**class** Circle **extends** Shape{

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

}

**class** GenericTest{

//creating a method that accepts only child class of Shape

**public** **static** **void** drawShapes(List<? **extends** Shape> lists){

**for**(Shape s:lists){

s.draw();//calling method of Shape class by child class instance

}

}

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

list1.add(**new**List<Rectangle> list1=**new** ArrayList<Rectangle>();

  Rectangle());

List<Circle> list2=**new** ArrayList<Circle>();

list2.add(**new** Circle());

list2.add(**new** Circle());

drawShapes(list1);

drawShapes(list2);

}

}

Q. EXPLAIN THE PURPOSES OF UPPERBOUNDED, LOWERBOUNDED AND UNBOUNDED WILDCARDS.

ANS-

The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

Syntax

List<? **extends** Number>

**?** is a wildcard character.

The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

Syntax

List<? **super** Integer>

Here,

**?** is a wildcard character.

The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

When the given method is implemented by using the functionality provided in the Object class.

When the generic class contains the methods that don't depend on the type parameter.

Example of Unbounded Wildcards

**import** java.util.Arrays;

**import** java.util.List;

**public** **class** UnboundedWildcard {

**public** **static** **void** display(List<?> list)

    {

**for**(Object o:list)

        {

            System.out.println(o);

        }

    }

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

    List<Integer> l1=Arrays.asList(1,2,3);

    System.out.println("displaying the Integer values");

    display(l1);

    List<String> l2=Arrays.asList("One","Two","Three");

      System.out.println("displaying the String values");

        display(l2);

    }

}

Q. WHAT IS ANNOTATION IN JAVA? DRAWTHE HIERARCHIES OF ANNOTATIONS?

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations start with ‘**@**’.

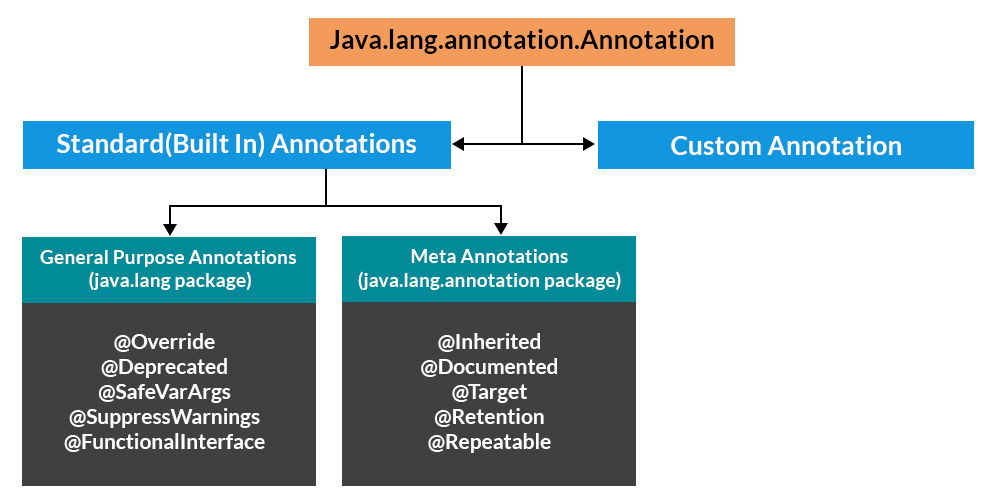
Annotations do not change the action of a compiled program.

Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.

Annotations are not pure comments as they can change the way a program is treated by the compiler.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

HIEARARCHY OF ANNOTATION



Q. WHAT ARE THE CATEGORIES OF ANNOTATIONS IN JAVA?

There are broadly 5 categories of annotations as listed:

Marker Annotations

Single value Annotations

Full Annotations

Type Annotations

Repeating Annotations

**@Override - @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.**

@SuppressWarnings - is used to suppress warnings issued by the compiler. If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

@Deprecated - marks that this method is deprecated so compiler prints warning. It informs

user that it may be removed in the future versions. So, it is better not to use such methods.

Q. EXPLAIN FUNCTIONAL INTERFACE WITH EXAMPLE.

Q. EXPLAIN LAMDA EXPRESSION WITH EXAMPLE

Q. WHAT ARE THE STEPS IN JDBC?

ANS - There are 5 steps to connect any java application with the database using JDBC. These steps are as follows



Register the driver class –

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.Syntax of forName() method

**public** **static** **void** forName(String className)**throws** ClassNotFoundException

**Driver class:**The driver class for the mysql database is **com.mysql.jdbc.Driver**.

Class.forName("com.mysql.jdbc.Driver");

Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

1) **public** **static** Connection getConnection(String url)**throws** SQLException

2) **public** **static** Connection getConnection(String url,String name,String password)  **throws** SQLException

**Connection URL:**The connection URL for the mysql database is **jdbc:mysql://localhost:3306/god** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and DBN is the database name. We may use any database, in such case, we need to replace the god with our database name.

Connection con=DriverManager.getConnection( "jdbc:mysql://localhost:3306/DBN","root","root");

Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.Syntax of createStatement() method

**public** Statement createStatement()**throws** SQLException

Example to create the statement object

Statement stmt=con.createStatement();

Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

**public** ResultSet executeQuery(String sql)**throws** SQLException

Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.Syntax of close() method

**public** **void** close()**throws** SQLException

Q. WHAT ARE THE CLASSES AND INTERFACES IN JDBC ?

*interfaces* of JDBC API –

Driver interface

Connection interface

Statement interface

PreparedStatement interface

CallableStatement interface

ResultSet interface

ResultSetMetaData interface

DatabaseMetaData interface

RowSet interface

classes of JDBC API

DriverManager class

Blob class

Clob class

Q. EXPLAIN ALL 4 TYPES OF JDBC DRIVERS .

ANS –

JDBC Driver is a software component that enables java application to interact with the database.

There are 4 types of JDBC drivers:

JDBC-ODBC bridge driver

Native-API driver (partially java driver)

Network Protocol driver (fully java driver)

Thin driver (fully java driver)

JDBC-ODBC bridge driver (TYPE1)



The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

easy to use.

can be easily connected to any database.

Disadvantages:

Performance degraded because JDBC method call is converted into the ODBC function calls.

Native-API driver (TYPE2)



The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

Advantage:

performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

The Native driver needs to be installed on the each client machine.

The Vendor client library needs to be installed on client machine.

Network Protocol driver (TYPE3)



The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

Advantage:

No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

Network support is required on client machine.

Requires database-specific coding to be done in the middle tier.

Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin driver(TYPE4)



The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.  
Advantage:

Better performance than all other drivers.

No software is required at client side or server side.

Disadvantage:

Drivers depend on the Database.

Q. EXPLAIN SERVLET ARCHITECTURE WITH DIAGRAM.

Q. EXPLAIN JSP ARCHITECTURE WITH DIAGRAM

Q. EXPLAIN 3 TYPES OF SCRIPTING TAGS USED IN JSP ?

ANS-

There are three kinds of scripting elements

Expressions

Scriplets

Declarations

Expressions

Used to write dynamic content back to the browser.

If the output of expression is Java primitive the value is printed back to the browser

If the output is an object then the result of calling toString on the object is output to the browser

Embedded in <%= and %> delimiters

Example:

<%=“Fred”+ “ “ + “Flintstone %>

prints “Fred Flintstone” to the browser

<%=Math.sqrt(100)%>

prints 10 to the browser

Scriptlets

Used to embed java code in JSP pages.

Inserts one to many lines of Java code

Embedded in <% and %> delimiters

Example:

<%

int x = 5;

int y = 7;

int z = x + y;

%>

Declarations

Declarations are used to define methods & instance variables

Do not produce any output that is sent to client

Embedded in <%! and %> delimiters

Example:

<%!

Public void jspDestroy() {

System.out.println(“JSP Destroyed”);

}

Public void jspInit() {

System.out.println(“JSP Loaded”);

}

int myVar = 123;

%>

Q. WHAT ARE THE SERVLET IMPLICIT OBJECTS AND THEIR SCOPES.

Q. WHAT ARE THE JSP IMPLICIT OBJECTS AND THEIR SCOPES.

Q. WHAT ARE THE JSP DIRECTIVES ?

The jsp specification defines three directives

Page: provder information about page, such as scripting language that is used, content type, or buffer size

Include – used to include the content of external files

Taglib – used to import custom actions defined in tag libraries

Page Directives

Page directive sets page properties used during translation

JSP Page can have any number of directives

Import directive can only occur once

Embedded in <%@ and %> delimiters

Include Directive

Used to insert template text and JSP code during the translation phase.

The content of the included file specified by the directive is included in the including JSP page

Example

<%@ include file=“included.jsp” %>

Taglib Directive

The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides means for identifying the custom tags in your JSP page.

Syntax- <%@ taglib uri = "uri" prefix = "prefixOfTag" >

Where uri is address of the page to be loaded. Prefix is a variable to hold that page.

Q. WHAT ARE THE JSP ACTIONS ?

Processed during the request processing phase.

As opposed to JSP directives which are processed during translation

Standard actions should be supported by J2EE compliant web servers

Custom actions can be created using tag libraries

The different actions are

Include action - Include action used for including resources in a JSP page

<jsp:include page=“inlcudedPage.jsp”>

Includes the the output of includedPage.jsp into the page where this is included.

Forward action - Forwards the response to other web specification resources

Example

<jsp:forward page=“Forwarded.html”>

Forwards the request to Forwarded.html

Param action - Used in conjunction with Include & Forward actions to include additional request parameters to the included or forwarded resource

Example

<jsp:forward page=“Param2.jsp”>

<jsp:param name=“FirstName” value=“Sanjay”>

</jsp:forward>

This will result in the forwarded resource having an additional parameter FirstName with a value of Sanjay

getProperty action - getProperty is used in conjunction with useBean to get property values of the bean defined by the useBean action

Example (getProperty)

<jsp:getProperty name=“myBean” property=“firstName” />

Name corresponds to the id value in the useBean

Property refers to the name of the bean property

setProperty action - setProperty is used to set bean properties

Example (setProperty)

<jsp:setProperty name=“myBean” property=“firstName” value=“Sanjay”/>

Sets the name property of myBean to SanjayExample (setProperty)

useBean action - The jsp:useBean action tag is used to locate or instantiate a bean class.

Example –

<jsp:useBean id=”objectname” scope=”request|application|session” class=”packagename.classname” beanname=”packagename.classname”

|<% =expression>

</jsp:useBean>

plugIn action - The **plugin** action is used to insert Java components into a JSP page.

Q. EXPLAIN MVC ARCHITECTURE



VIEW IS JSP/SERVLET

MODEL IS POJO+SERVICE+REPOSITORY)

CONTROLLER IS SERVLET/JSP

MVC, which stands for Model-View-Controller, is a software architectural pattern commonly used in the development of user interfaces, particularly for web applications. It separates the application into three interconnected components:

**Model**: The Model represents the data and business logic of the application. It is responsible for managing the application's state, responding to requests for information, and updating itself accordingly. In simpler terms, the Model encapsulates the data and the rules that govern how the data can be manipulated. It does not directly interact with the user interface.

**View**: The View is responsible for presenting the data to the user and handling user interactions. It displays the information from the Model in a way that is visually understandable to the user. Views can include various components such as HTML templates, CSS stylesheets, and JavaScript code to create an interactive user interface. In the MVC pattern, Views are passive and only display information; they do not handle data manipulation or business logic.

**Controller**: The Controller acts as an intermediary between the Model and the View. It receives input from the user via the View, processes that input (often by interacting with the Model), and then updates the View accordingly. Controllers are responsible for interpreting user actions, invoking appropriate methods on the Model, and selecting the correct View to render the updated information back to the user. They essentially control the flow of the application.