**JAVA CONCEPTS:**

**For jdbc questions:**

**http://www.javatpoint.com/jdbc-interview-questions.**

**If the variable is not a static we can access it by creating a object.**

**Constructor**: it is a block of code sam as method , but the diff is constructor don’t have return type so it has a public modifier. Evry class should have a constructoe if we won’t create a constructor compiler automatically creates a constructor cslled as **default constructor**. **Constructor will** call itself when the object is created.

1)constructor is a member method.

2)constructor name is same as class name**. it will be used to allocate memory**

**Ex: public Name(){ }**

**Oops concepts:**

**Inheritance:** Aquiring the properties of parents class (or) creating a class from existing class. Ther are 3 types of inheritance:

1)single inheritance: A

B aquires the properties of A

B

2)multilevel inheritance:

A

B

C

3) 3)multiple inheritance(MI) : java doesnot supports multiple inheritance but it can support indirectly by using the interface. Java doesnot supports MI because of diamond prblm d aquires properties of b,c,a it will tough for d to choose propertis either a,b,c.

By using the **extends** key word child class can aquire the properties from parent class. Inheritance can also aceesses itself properties

Example:

**P: public** **class** Employee {

**Int id;**

**Double salary;**

**String name;}**

C: **public** **class** Programmer **extends** Employee{

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

{

Programmer programmer = **new** Programmer();

System.***out***.println("salary is...>" + programmer.salary);

}

}

Another class:in this class we can create variables also.

**public** **class** Testing **extends** Programmer {

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

{

Testing testing = **new** Testing();

System.***out***.println("result is" + testing.age);

**int** pno = 684858899;

System.***out***.println(pno);

}

}

**Example2:**

**public** **class** Inht1

{

**int** c;

**public** **void** add(**int** A, **int** B)

{

c = A + B;

System.***out***.println("the addition is:" + c);

}

**public** **void** sub(**int** A, **int** B)

{

c = A - B;

System.***out***.println("the subtraction is:" + c);

}

}

**public** **class** Inht2 **extends** Inht1{

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

**int** x = 10;

**int** y = 20;

Inht2 z = **new** Inht2();

z.add(x, y);

z.sub(x, y);

z.mul(x, y);

}

**public** **void** mul(**int** A, **int** B)

{

c = A \* B;

System.***out***.println("the multiplication is:" + c);

}

}

**CREATE OBJECT:** object is a instance of class, it consists states and behavior,by using new keyword we create object.

Classname name = new classname();

Ex:

public class CreateObject {

public static void main(String[] args){

CreateObject s = new CreateObject();

s.getage(3);

}

public int getage(int age)

{

System.out.println(age);

return age;

}

}

**CALLING METHOD WITH VOID:**

public class CreateObject {

public static void main(String[] args){

CreateObject s = new CreateObject();

s.getage(3);

}

public void getage(int age)

{

System.out.println(age);

}

}

calling static method:

**public class ExampleMinNumber {**

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

**int x = 11;**

**int y = 6;**

**int c = minFunction(x, y);**

**System.out.println(c);**

**}**

**public static int minFunction(int a, int b) {**

**int min;**

**if (a > b)**

**min =b;**

**else**

**min = a;**

**return min;**

**}**

**}**

**calling method with return and no parameter:**

public class Emp{

public static void main(String[] args){

int number = 1;

Emp s = new Emp();

s.Age();

System.out.println(number);

}

Public int Age()

{

int num = 1;

return num;

}}

**public class Emp {**

**calling method with return and parameter:**

**public** **class** Emp1 {

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

**int** x = 20;

**int** y = 30;

**int** z = *ass*(x, y);

System.***out***.println(z);

}

**public** **static** **int** ass(**int** a, **int** b){

**int** c = a + b;

**return** c;

}

}

**create classes under multiple packages**

**:**

**public class MultiClass {**

**MultiClass(){**

**System.out.println("welcome to world");**

**}**

**void MultiClass\_method(){**

**System.out.println("wlcome");**

**}**

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

**MultiClass s = new MultiClass();**

**MultiClass2 z = new Mclass2();**

**s.MultiClass\_method();**

**z.MultiClass\_method2();**

**}}**

**public class Mclass2{**

**Mclass2() {**

**System.out.println("hello");**

**}**

**void MultiClass\_method2(){**

**System.out.println("world");**

**}**

**}**

: **write code to handle exceptions with try/catch/finally:**

try

{code may result in exception/ any exception may occur

if exception occur catch block will execute i

}

**Catch(will write what exception will occur ex: (ArithmeticException ae))**

{

Her will print a user friendly msg which can be understable by user

}

**Finally**

{

If we want to execute with/without exception will use finally}

Instead of try/catch/finally we can use **throws** which are called exception handlers;

**Throws: we will add throws to methods then the whole logic exception will be handeled by method**.

**example:**

**import** java.io.IOException;

**import** java.util.Scanner;

**public** **class** TryCatch {

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

**try**{

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("enter a number:");

**int** a = s.nextInt();

System.***out***.println("enter a number:");

**int** b = s.nextInt();

**int** c = a / b;

System.***out***.print("the division of two numbers is:"+ c);

}

**catch**(ArithmeticException ae)

{

System.***out***.println("we are handling arithmetic exception");

}

**finally**{

System.***out***.println("exception is mandetory");

}

}

}

**what is final keyword, create final class, final method, final property:**

**final: it is a keyword which can applied to**

**varisble, method and class.**

**Variable:** if we apply final to variable s treated as a constant, its value cannot be change**.**

**Ex: public** **class** Fexample {

**final** **int** a = 100;

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

Fexample s = **new** Fexample();

//here we created object because variable is not static in order to access variable we created object//

s.a =200;//here we get error that fexample.a cannot assigned//

}

} here we cannot change the value of a “a” variable because we declare a as a final in the above.

**Method:** abstract applying to method gives override,

But by appliying final method doent override.

Ex: In order to show we need to create 2 classes with samemethod name, whenwe declare one method as a final and extends it then we cannot override.

**Class1:** public class Example1 {

int a = 100;

public static void main(String[] args) {

// TODO Auto-generated method stub

Example1 s = new Example1();

s.a =200;

}

public final void show(){

System.*out*.println("finalmethod exaple");

}

}

**Class2:**

**public** **class** Example2 **extends** Example1 {

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

}

**public** **void** show(){**// here we get error like cannot override thefinal method from exampl1//**

System.***out***.println("welcome");

}

}

**Class**: on applying final class cannot be subclass and cannot provide inheritance**.**

**Ex**

**Class:1**

**public** **final** **class** Fcexample {

**int** a = 100;

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

// **TODO** Auto-generated method stub

Fcexample s = **new** Fcexample();

s.a =200;

}

**public** **void** show(){

System.***out***.println("finalmethod exaple");

}

}

Class2:

**public** **class** Fcexample2 **extends** Fcexample {//**here we gt the error like fcexample cannot subclass the fcexample//**

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

}

**public** **void** show(){

System.***out***.println("welcome");

}

}

write code for interface and create class to implement that interface:

interface is a template similar to class but methods in class are by default abstract.if u define a interface to a class no need to create body to that method. No need to create object in a class which v created to implement a interface.

**First u need to create a interface next u can create number of classes using implement keyword.**

**Interface: public** **interface** InterfaceExp {

**public** **void** show();

**public** **void** display();

}

**Class1:**

**public class Interfacechild1 implements InterfaceExp {**

**@Override**

**public void show() {**

**// TODO Auto-generated method stub**

**System.out.println("child1show");**

**}**

**@Override**

**public void display() {**

**// TODO Auto-generated method stub**

**System.out.println("child1display");**

**}**

**}**

**Class2:**

**public** **class** Interfacechild2 **implements** InterfaceExp {

@Override

**public** **void** show() {

// **TODO** Auto-generated method stub

System.***out***.println("child2show");

}

@Override

**public** **void** display() {

// TOsysoutInterfaceExDO Auto-generated method stub

System.***out***.println("child2display");

}

}

**Testing final class:**

**public** **class** Testing {

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

// **TODO** Auto-generated method stub

Interfacechild1 s = **new** Interfacechild1();

s.show();

s.display();

Interfacechild2 z = **new** Interfacechild2();

z.show();

z.display();

}

}

**write code for creating abstract class:**

**abstract (keyword)**class can have both concrete(complete) and abstatct(incomplete) methods. If class itself have abstract no need to specify abstract inmethod

Abstract can apply to methods and class.

Diff btw normal class and abstract class is:

Abstract class will have definationa and no body.

Normal class will have definition and body hich is said to be concrete:

While creating a method if it doesnot have body then it is said to be error if u add abstract to that w won tget errors,

The main use of abstract method is child cclass must override when th chid class quirs the properties of parentclass.

An 1 abstract class can have both complete and abstract method.

But1 abstract method must haveabstrct class,

To create a abstract class first create abstract class then create child class to call methods in parentclass.

Example:abstract class:

**public** **abstract** **class** AbstractExp {

**public** **void** show()

{

System.***out***.println("display show method");

}

**public** **abstract** **void** display();

}

Create child class:

**public** **class** AbstractExp2 **extends** AbstractExp{

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

// **TODO** Auto-generated method stub

AbstractExp2 s = **new** AbstractExp2();

s.show();

s.display();

}

@Override

**public** **void** display() {

// **TODO** Auto-generated method stub

System.***out***.println("show display method");

}

}

**implement method overloading:**

**polymorphism:**single class with multiple operations.

2 types of polymorphism:

1)compile time poly..::overloading

Same method name with diffrnt arguments and different retuentype i.e, doing same operations for example addition with different datatypes like int, float and double.

Example: method overloading

**public** **class** CompileExp {

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

CompileExp s = **new** CompileExp();

**int** result = s.addnum(10, 20);

System.***out***.println("the addition is integer is.." + result);

**float** floatresult = s.addnum(30.56f, 40.67f);

System.***out***.println("the addition in floatnumber is..." + floatresult);

**double** doubleresult = s.addnum(90.67, 40.78);

System.***out***.println("the addition in doublenumber is..." + doubleresult);

}

**public** **int** addnum(**int** a, **int** b){

**return** a + b;

}

**public** **float** addnum(**float** a, **float** b){

**return** a + b;

}

**public** **double** addnum(**double** a, **double** b){

**return** a + b;

}

}

**2)runtimepolymorphism : overridding**

The method name should be same and returntype and arguments should be same:

Example:parent class:

**public** **class** RuntimeOverloading {

**public** **void** show(){

System.***out***.println("show1");

}

}

Child class:

**public** **class** Runtime1 **extends** RuntimeOverloading {

**public** **void** show(){

System.***out***.println("show2");

}

}

Class to test 2 classes:

**public** **class** Testing {

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

// **TODO** Auto-generated method stub

RuntimeOverloading s = **new** RuntimeOverloading();

s.show();

Runtime1 z = **new** Runtime1();

z.show();

//Runtime1 d = new RuntimeOverloading, we cannot declare runtime1 that is child class is to create object.//

RuntimeOverloading x = **new** Runtime1();//here we used runtimeoverloading to create object and runtime is an instance of class so hre runtimeoverloading acts as a parent class and aquires the child class that is runtime1 properties.//

x.show();

}

}

Output:

Show1

Show2

Show2

ARRAYS: is a group of numbers with similar datatypes.

Ex: for strings

**import** java.util.Arrays;

**public** **class** ArrayExample {

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

String[] arrayList = {"text1", "text2", "text3", "text4"};

System.***out***.println(Arrays.*toString*(arrayList));

}}

Ex for int:

**public** **class** Array2 {

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

**int**[] no = {1, 2, 3, 4};

**for**(**int** i=0; i < no.length; i++)

{

System.***out***.println(no[i] + " ");

}

}

}

* ARRAYLIST: it is a dynamically growable component which can accept any kind ofobject.
* It allows duplicates.
* it gives us back the same type of insertion.
* It is index based.it allows nulls.
* To search element = list.get(0(index val));
* Creating object:

ArrayList studentlist = new ArrayList();

Example:

import java.util.ArrayList;

import java.util.Iterator;

public class ArrayListExample {

public static void main(String[] args){

ArrayList numberList = new ArrayList();

numberList.add("1");

numberList.add("2");

numberList.add("3");

numberList.add("4");

numberList.add("5");

System.out.println(“the num of element are:” + numberList.size());

Iterator iterator = numberList.iterator();//instead of using forloop collection framework will gives Iterator//

while(iterator.hasNext())

{

System.out.println("the elements in list are..>" + iterator.next());

}

System.out.println("the 3rd element is" + numberList.get(4));

}

}

**write code to retrieve items Hashset:**

Hashset allows any kind of object, gives the back as same type of insertion. But it doesnot allow the duplicates.it allows only 1 null value because it doesnot allows duplicates.it is the best performance amng all because it takes less time.

Ex: **import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** Hashsetexample {

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

{

HashSet numberSet = **new** HashSet();

numberSet.add("1");

numberSet.add("2");

numberSet.add("3");

numberSet.add("4");

System.***out***.println("my elements are:..>" + numberSet.size());

Iterator iterator = numberSet.iterator();

**while**(iterator.hasNext()){

System.***out***.println("the elements are..>" + iterator.next());

}

}

}

**write code to retrieve items Hashmap:** it allows the objects in the form of key/values. Keys shouldnot be duplicate, values can be duplicate, it allows only one null key.it is notsyncronised so this takes less time among all collectframes.

Ex:

**Linked list**: allows any kind of object,guarantees order, allows duplicates, node form(it stores its value and address of next value), good for insertion and deletion.

Linkdlist numberlist = new linkedlist();it have null values.

Timecomplexity = nlogn

Set:it doesnot allows duplicates

**Linkedhashset**: object any kind, 1 null value, no duplicates, guarantee order,

**Treeset:** object anykind, no duplicates, 1 null value, guarantee oeder and sorting as well.

**Map**: object in the form of key/value. Put(key, value)

Get(key, value)

**Hashtable**:object kry/value, key cannot be duplicate, value can be. Not even 1 null value, it is synchronized.

**Treemap**: sorting order, allows 1 null value, key/value.

**write code to connect to JDBC to get rows from employee table:**

**there are steps to connect to jdbc:**

1. Get a connection to db
2. Create a stmnt
3. Execute a sql query
4. Process the result.

**Difference btw string, stringbuffer and string builder:**

**String:** group of characters, string is a class final class.it is a immutable.

Syntax: string s = “new”;

S = s + “worls”;//when we add another word to string it will create another memory location instead of updating the old location which is called immutable/in order to avoid this we use string buffer.

Ex:

**public** **class** StringExample {

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

String s = "hello";

s = s + "world";

System.***out***.println(s);

}

}

Output:helloworld.

**Stringbuffer:**is a synchronized (it takes extra performance) it is a mutable

Syntax: stringbuffer s = new stringbuffer(“hello”);

s.append(‘world”);/here instead of creating new memory location it will update old location//

ex: **public** **class** StringBuilderExp {

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

StringBuffer s = **new** StringBuffer("hello");

s.append("world");

System.***out***.println(s);

}

}

Op:helloworld

**Stringbuilder**:is not a synchronized, is mutable.

Syntax: stringbuilder s = new stringbuilder(“hello”);

Ex:

**public** **class** StringBuilderExp {

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

StringBuilder s = **new** StringBuilder("hello");

s.append("world");

System.***out***.println(s);

String result = s.substring(1);//here it will giveelloworld

System.***out***.println(result);

String result1 = s.substring(2, 6);//here I will give 110w it stops before index6//

System.***out***.println(result1);

}

}