##################################JAVA NOTES##

OOPS - OBJECT ORIENTED PROGRAMMING SYSTEMS

OBJECT
. An instance of a class.
. Objects have some state and behavior.
. Contains address and takes up some space in memory

CLASS
. Blueprint for the object.
. logical entity i.e., does not exists in reality
. does not consume any space
. many objects can be created from a class

CREATING CLASS IN JAVA
```
class className
{
        //create elements (data_types, variables, functions) of class
        data_type var_name; //variable in class
        public void func() //function in class
        {
                //statement
        }
}
```
**** *only main class can be public***
**** *the name of class must be in camelCase***

CREATING THE OBJECT OF CLASS IN JAVA
SYNTAX:
```
className objectName = new className();
objectName.var_name = value; //giving value to the variable of class using object
objectName.func(); //calling the function of the class using object
```

METHODS/FUNCTIONS IN JAVA
. made to reduce repeatative code in a program

SYNTAX FOR CREATING A METHOD/FUNCTION:
```
static return_type methodName(parameters)
{
        //statements
}
data_type var_name = methodName(arguments); //method calling
```

MEHTOD OVERLOADING
. when class has two or more methods with same name but different parameters

PASS BY VALUE OR PASS BY REFERENCE
. when method parameter values are copied to another variable then the copied object is passed it is called pass by value.
. in pass by value, an copy of the parameter is created in the method and any change made to the parameter in the method has no effect on the original variable
. when a reference to actual parameter is passed it is called pass by reference
. in pass by reference, the address of the actual variable is passed to the method and no copy is created so any change made to the parameter in the method changes the original variable.
. JAVA is always pass by value.

CONSTRUCTORS IN JAVA
. similar to method
. no return type
. name of constructor should be same as class name
. called automatically when an object is created.

SYNTAX:

```
class className
{
    className()
    {
        //statements
    }
}
```

NO ARGUMENT CONSTRUCTOR

. constructor which does not accept any parameter.

PARAMETERIZED CONSTRUCTOR

. constructor which accepts some parameters.

DEFAULT CONSTRUCTOR

. constructor which is defined by the program by-default when a class is created.
. it has no arguments.
. if a constructor is made by the user then default constructor cannot be called.

this KEYWORD

. refers the current object or the variable of the class
. used when the variable of the class and parameter of method both are of same name.

CONSTRUCTOR OVERLOADING

. when two or more constructors are present but with different parameters.

ENCAPSULATION

. process of wrapping code and data together into a single unit.
. restricting the access of data
. also known as data hiding as the data is hidden from the rest of the world.

ENCAPSULATION WITH HELP OF PACKAGES AND ACCESS MODIFIERS

JAVA PACKAGES

. group of similar type of classes, interfaces and sub-packages.
. some built-in packages: java, lang, awt, javax, swing, net, io, util, sql, etc
. need to be imported into program using import keyword.

import KEYWORD

. used to import JAVA packages into program

SYNTAX

```
import package_path;
```

. to import all files from a particular package use * at the end of path

ADVANTAGES OF JAVA PACKAGES

. categorizes the classes and interfaces in an easily maintainable form
. provides access protection
. removes naming collision

ACCESS MODIFIERS

. specifies the accessibility or scope of a field, method, constructor or class.
. access modifiers change the access levels of field, method, constructor or class.

TYPES OF ACCESS MODIFIERS

. private : access level only within the class**.** cannot be accessed from outside the class
. default : access level only within the package**.** cannot be accessed from outside the package**.** if access level not specified then it will be default.
. protected : access level only within the package**.** can be access from outside the package by using child class, otherwise not accessible outside the package.
. public : access level is everywhere**.** can be accessed from within class, outside class, within package, outside package.

USE OF ENCAPSULATION
. keep related fields and methods together.
. makes code cleaner and easy to read.
. helps in controlling data field modification**.**
. helps in data hiding.
. helps in decoupling the system components**.**
. reduces the interdependencies of the system components.
. decoupled system can be tested, developed independently.
. code can be updated without affecting the program.

HOW TO ACHIEVE ENCAPSULATION
. Declare all fields of class as private
. Provide public setter and getter methods to read and write variable values
**** *if we use setters and getters then the outside variable will not directly access the fields of the class. They will access them using getter and setter.***

MAKING getters and setters
SYNTAX:

```
public return_type getFunc()
{
        return var;
}
public void setFunc(arguments)
{
        //statements
}
```

DATA HIDING vs ENCAPSULATION
Encapsulation is bundling of related fields and methods together which allows us to achieve data hiding**.** Encapsulation doesn't means data hiding.

static KEYWORD
. related to class
. indicates that the particular member belong to a type itself, rather than to an instance of that type.
. mainly used to help memory management.
. can be applied to variable, method, block and nested class.
** ***static keywords cannot be declared in non-static data type***.

HOW static HELPS IN MEMORY MANAGEMENT
When we use 'static' keyword with any variable or class then it will not have a multiple copy for multiple object**.** It will remain same for every object**.** If a non-static variable is formed then it will have multiple copies for multiple object and its value can change also**.** Therefore, 'static' keyword will help in memory management by not making multiple copies of variables.

BASIC MATH FUNCTIONS
. Math.PI
. Math.max(a,b)
. int.MAX_VALUE

static CLASS
. top-level class cannot be declared static
. only nested class can be declared static
. class within another class is nested class.
. two types of nested class: static and non-static

WHY ARE NESTED CLASS MADE STATIC
One class is nested within another class when both the classes are closely related to each other**.** The nested class is made static so that it can be accessed outside the class such that it seems that the class is actually outside and not nested.

STATIC AND NON-STATIC CLASS

SYNTAX

```
public class className
{
    class nestedNonstatic
    {
        //statements
    }
    static class nestedStatic
    {
        //statements
    }
}
```

** ***to use the object of nonstatic nested class we first need to import the class***

SYNTAX:

```
import className.nestedNonstatic;
        OR
className obj1 = new className();
className.nestedNonstatic obj2 = obj1.new nestedNonstatic();
```

static BLOCKS

. BLOCKS : piece of code kept together

. static blocks means that when the class will run then the first block that will be executed is the static block.

SYNTAX:

```
public class className
{
    static
    {
        //block of code
    }
}
```

EXECUTION ORDER

all static blocks -> main block

INHERITANCE

. inheriting the properties of parent class into child class

. can occur only if the two class have a relation between them.

. it is unidirectional**.** If A can be inherited from B then B cannot be inherited from A

. parent class is also called superclass

. child class is also called subclass

protected KEYWORD

. can be assigned to methods, variable and fields

. can be accessed from within the class, within the subclass, within the package

SYNTAX:

```
child c = new child();
c.func1();
c.func2();
```

. parent class can have multiple child classes but one child class can have only one immediate parent class.

SYNTAX FOR INHERITANCE

```
public class parent
{
        public void func1()
        {
                //statements
        }
        //statements
}
public class child extends parent
{
        public void func2()
        {
                //statements
        }
        //statement
}
```

** ***now child class can use the functions of parent class***

METHOD OVERRIDDING

. if same method is defined in both parent and child class then the method in child class overrides the method in the parent class.

UPCASTING AND DOWNCASTING

IMPLICIT CASTING/UPCASTING

. when we declare the parent class object using child class object.

SYNTAX:

```
child c = new child();
parent p = c;
```

EXPLICIT CASTING/DOWNCASTING

. when we declare the child class object using parent class object.

SYNTAX:

```
child c1 = new child();
parent p  c1;
child c2 = (child)p;
```

instanceof KEYWORD

. returns true if the object is an instance of the class

SYNTAX:

```
c2 instanceof parent;
```

super KEYWORD

. used to refer immediate parent class of the child

SYNTAX:

```
super.func2(); //inside the child class
```

WHY JAVA DOES NOT SUPPORTS MULTIPLE INHERITANCE

because two classes may define different ways of doing the same thing and child class will not be able to choose which one to pick.

It can also introduce the **DIAMOND PROBLEM**.

SINGLETON PATTERN

. type of design pattern

. used in multi-threaded and database applications

. used for logging, caching, thread pools, configuration settings, etc.

. used when we have to do a time consuming task and we don't want it to repeat

. only single object is made throughout the program.

STEPS TO IMPLEMENT

. Private constructor to restric instantiation of the class from other classes.
. Private static variable of the same class that is the only instance of the class.
. Public static method that returns the instance of the class. It is the global access point for outer world to get the instance of the singleton class.
. making the default constructor of the class private.
. making a private static object to return.
. making a public static function to return the instance or the object.

SYNTAX:

```
public class appConfig
{
        private appConfig()
        {
                //statements
        }
        //statements
}
private static appConfig obj = new appConfig();
public static appConfig getInstance()
{
        //statements
        return obj;
}

appConfig obj1 = appConfig.getInstance(); //calling when we want to get the instance of class
```

** **we can initially initialize the appConfig object as null and then later we can make it whenever we want.**

TWO POINTER ALGORITHM

WHEN TO USE

. when array is sorted.
. when an array is sorted then there only two possiblities can occur
    i. Binary search
    ii. Two pointer algorithm

ALGORITHM

. sort the array
. then take two variable i and j which point to the starting and the ending of the array.
. take a variable x which stores a value with which we want the variable sum to be equal

SYNTAX

```
while(i < j)
{
        if( a[i] + a[j] < x)
        {
                i++;
        }
        else if( a[i] + a[j] > x)
        {
                j--;
        }
        else
        {
                return true;
        }
}
return false
```

POLYMORPHISM
. made of two Greek words: 'poly' : multiple + 'morphism' : forms
. used when we want a object to take multiple forms
. allows us to perform a single action in multiple ways.

TYPES OF POLYMORPHISM
1. RUN-TIME POLYMORPHISM
. also known as dynamic binding or late binding or overriding.
. in this, a method has same parameters but are associated in class and its subclass
. slower execution

2. COMPILE-TIME POLYMORPHISM
. also known as static binding or early binding or overloading
. in this, more than one method share same name with different parameters and return type
. faster execution
****overriding is not possible with variables. it can only be done with methods***

ABSTRACTION
. means reducing complexity from a system
. process of hiding certain details and showing only essential information to user
. helps in reducing programming complexity and effort
. can be achieved using abstract classes or interfaces

ABSTRACT CLASSES
. declared using 'abstract' keyword
. made when we don't want to make the object of the parent class, but want the child class to use the functions of parent class
. the abstract class must be extended and its abstract method must be overridden.
. the class of an abstract method should also be abstract
. abstract class can also contain non-abstract methods. these are called concrete methods.

SYNTAX

```
public abstract class class1
{
        public abstract void func1();
        public abstract void func2();
}
```

ABSTRACTION vs ENCAPSULATION
. Abstraction is hiding unwanted details and showing only the essential information, while Encapsulation is hiding the inner detail or mechanics of how an object does something due to security reasons
. Abstraction allows focussing on what the information object must contain, while Encapsulation means binding of code and data into single unit
. Abstraction solves issues at design level, while Encapsulation solves it at implementation level.

final KEYWORD
. gives restriction in system
. can be used with method, classes and variables.
. if used with variable, it means that the value of variable will not change.
. if used with class, it means that the class will not have any child class.
. if used with methods, it means that the method will not be overriden.

final VARIABLE
. can be initialized only once.
. must be initialized when using inside class, otherwise compile-time error will occur.
. if using as local variable, then final variable can be initialized later.
. final variable should be named in CAPS.
. if not initialized in declaration, it is called blank final variable.
. in reference final variable, internal state of object pointed by reference variable will not change.

SYNTAX:

final data_type VAR_NAME = value;

final METHODS

SYNTAX:

```
public final return_type func()
{
        //statement
}
```

final CLASS

SYNTAX:

```
public final class class1
{
        //statements
}
```

INTERFACES

When we are having two classes which are not related to each other, i.e., they are at same level and there is a third class which wants the properties of both the classes, then interfaces are used.
In this, we make both the classes interfaces and then use multiple interface as JAVA does not supports multiple inheritance but supports multiple interfaces.
Interfaces also supports abstraction

SYNTAX:

```
public abstract interface className1
{
        abstract return_type func1();
}
public abstract interface className2
{
        abstract return_type func2();
}
public class className3 implements className1, className2
{
        //statements
}
```

FUNCTIONS OF INTERFACES

. provide specification that a implemented class must follow
. helps us to achieve abstraction same as abstract classes
. multiple inheritance possible

IMPORTANT POINTS ABOUT INTERFACES

. interfaces cannot be instantiated
. interfaces cannot have constructors as we cannot instantiate them
. interfaces cannot have a method with body
. default attribute of interface if public, static and final
. we can provide access modifiers to interface
. defalut interface methods are implicilty abstract and public as methods don't have body
. interface cannot extend class but can extend another interface
. class implementing interface must provide implementation for all method unless it is an abstract class

UPCASTING IN INTERFACE

SYNTAX:

```
className3 obj1 = new className();
className2 obj2 = obj1;
```

In this we are upcasting obj2 with obj1**.** obj2 will not be able to access the methods present in className1**.** It will only be able to access the methods present in className2.

CHANGES IN INTERFACES IN JAVA 8.0

If we want to give body to a method in interface then we need to use the default keyword in starting.
If we give the body to a method in the interface then there is no need to define it in the implemented class.
To prevent Diamond Problem, we need to define the method in the implemented class.

SYNTAX:

```
default void func()
{
        //statements
}
```

EXCEPTION HANDLING

. error event that can happen during the execution of a program and disrupt its normal flow

. can arise due to wrong input, hardware failure, network failure, database server down

THROWING THE EXCEPTION

. process of creating exception object and handing it to runtime environment

. whenever an error occurs, an exception object is created**.** It contains lot of debugging information like method hierarchy, error line number, type of exception, etc.

. the normal flow of program halts and JRE tries to find who can handle the exception

CATCHING THE EXCEPTION

. process of finding appropriate exception handler and passing the exception object to it

. when runtime environment receives exception object, it finds handler for exception.

. it starts the search in a method where error occured**.** if no handler found, then it moves to caller method and so on.

eg**.** let a process runs from A -> B -> C and an exception occurs at C then the search for handler will move from C -> B -> A.

****handler is a block of code that can process exception object***

try-catch BLOCK

. used to handle exceptions

. if there is any statement after the try-catch block then the statement will run even if an exception has occured.

SYNTAX

```
try
{
        //block of code which you think can give exception
}
catch(exception_type exception_object)
{
        //block of code which you want to run if exception occurs
}
```

finally KEYWORD

. this is placed after all catch blocks.

. it always runs even if any exception is catched or not.

SYNTAX

```
try
{
        //block of code which you think can give exception
}
catch(exception_type exception_object)
{
        //block of code which you want to run if exception occurs
}
finally
{
        //block of code which you want to run always
}
```

throw AND throws KEYWORD

. throw is used to explicitly throw an exception from a method or block of code.

SYNTAX

```
throw exception_object;
```

. throws is used in method signature to indicate this method can throw one of the listed exception**.** the caller to these methods handles the exception using try-catch block.

SYNTAX

```
public static void func1()
{
        try
        {
                func2();
        }
        catch(exception_type exception_object)
        {
                //block of code which you want to run if exception occurs
        }
}
public static void func2() throws exception_type
{
        //statements
}
```

there is a function called Thread.sleep(time in ms) which stops the execution of the program for a particular time. This is always kept within a try-catch block otherwise it will give an error