

Home Work [3]

(OOP with Java)

1. How data hiding is accomplished in Java? Explain.

Answer:

Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members). Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.

Data hiding is also known as data encapsulation or information hiding.

<https://www.techopedia.com/definition/14738/data-hiding>

Example:

```
/* File name : RunEncap.java */
public class RunEncap {

    public static void main(String args[]) {
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " +
encap.getAge());
    }
}
```

```
/* File name : EncapTest.java */
public class EncapTest {
    private String name;
    private String idNum;
    private int age;

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }

    public String getIdNum() {
        return idNum;
    }
}
```

```
public void setAge( int newAge) {  
    age = newAge;  
}
```

```
public void setName( String newName) {  
    name = newName;  
}
```

```
public void setIdNum( String newId) {  
    idNum = newId;  
}  
}
```

https://www.tutorialspoint.com/java/java_encapsulation.htm

2. How are data and functions organized in OOP? Explain.

Answer:

OOP languages involve using a series of classes to keep data and functions organized. Every class will contain variables and functions specific to that class that are called from elsewhere in the program where that class is used.

A simple example from Java:

```
1. public class Dog(){  
2.     public enum Breed {  
3.         poodle,husky,german_shepherd,labrador  
4.     }  
5.  
6.     private Breed breed;  
7.     private int age;  
8.  
9.     public Dog(Breed breed,int age){  
10.        this.breed = breed;  
11.        this.age = age;  
12.    }  
13.  
14.    public Breed getBreed(){  
15.        return breed;  
16.    }  
17.    public void setBreed(Breed breed){  
18.        this.breed = breed;  
19.    }  
20.    ...  
21. }
```

This is a Dog class that allows one to set a breed and an age for a dog. One could create as many instances of this class as wanted with each instance being a different dog. Using polymorphism one can create even more organization by having classes that extend other classes such as Animal->Mammal->Dog->Puppy.

<https://www.quora.com/How-are-data-and-functions-organised-in-an-object-oriented-program>

3. What kind of things can become an object in OOP?

Answer:

In OOP(Object Oriented Programming) the things which have some attributes and have some functions will become objects. Here is an example to check that what will become an object:-

Example :-

Let say you are going to develop a software for the School to take attendance on computers ,so you will start programming in object oriented .You will analyse that in attendance what are the things which have attributes and some functionality. These things will come in your mind like Student,Teacher,Register etc.

You will analyse that when attendance is taken the teacher take the attendance ,student speak that I am present and register is used to mark the attendance.

So the Program will have following things:-

```
class Student
{
    string sName ;
    int    sRollNo;
}
class Teacher
{
    string TeacherName;
}

class Register
{
    Student  StudentList[20];
}
```

<http://jswinjava.blogspot.com/2013/08/what-kind-of-things-can-become-objects.html>

OR

OOP was found to make your code more organised, easy to maintain, and to module the real thing in life (everything).

Take this little exemple:

You want to write a program to manage a company so you create a class called employee. This class contains Employee's ID,name, department and then you create class called Company which has company name as an attribute then you plant your employee class in a company class that is called composition.

Now we've moduled the reality (each company have employees) and that is applied to everything.

<https://www.quora.com/What-kind-of-things-can-become-objects-in-oop/answer/Hadi-AL-Halbouni>

4. What is meant by data binding?

Answer:

The process of binding data members and functions in a class is known as, encapsulation. Encapsulation is the powerful feature (concept) of object-oriented programming. With the help of this concept, data is not accessible to the outside world and only those functions which are declared in the class, can access it.

https://www.slideshare.net/Sachin_Kpl/basic-concepts-of-object-oriented-programming

OR

In computer programming, data binding is a general technique that binds data sources from the provider and consumer together and synchronizes them. This is usually done with two data/information sources with different languages as in XML data binding. In UI data binding, data and information objects of the same language but different logic function are bound together (e.g. Java UI elements to Java objects).

In a data binding process, each data change is reflected automatically by the elements that are bound to the data. The term data binding is also used in cases where an outer representation of data in an element changes, and the underlying data is automatically updated to reflect this change.

https://en.wikipedia.org/wiki/Data_binding

5. **What does meant by literals? Shortly explain integer, floating-point, boolean, character and string literals used in Java programming.**

Answer:

Literals

In computer science, a literal is a notation for representing a fixed value in source code. Almost all programming languages have notations for atomic values such as integers, floating-point numbers, and strings, and usually for booleans and characters; some also have notations for elements of enumerated types and compound values such as arrays, records, and objects.

[https://en.wikipedia.org/wiki/Literal_\(computer_programming\)](https://en.wikipedia.org/wiki/Literal_(computer_programming))

Java Literal

Java Literals are syntactic representations of boolean, character, numeric, or string data. Literals provide a means of expressing specific values in your program.

For example, in the following statement, an integer variable named count is declared and assigned an integer value. The literal 0 represents, naturally enough, the value zero.

Numeric literal:

```
int count = 0;
```

Boolean literal:

There are two boolean literals

- true represents a true boolean value
- false represents a false boolean value

There are no other boolean literals, because there are no other boolean values!

Integer Literals:

In Java, you may enter integer numbers in several formats:

1. As decimal numbers such as 1995, 51966. Negative decimal numbers such as -42 are actually *expressions* consisting of the integer literal with the unary negation operation - .
2. As octal numbers, using a leading 0 (zero) digit and one or more additional octal digits (digits between 0 and 7), such as 077. Octal numbers may evaluate to negative numbers; for example 037777777770 is actually the decimal value -8.
3. As hexadecimal numbers, using the form 0x (or 0X) followed by one or more hexadecimal digits (digits from 0 to 9, a to f or A to F). For example, 0xCAFEFEBABEL is the long integer 3405691582. Like octal numbers, hexadecimal literals may represent negative numbers.
4. Starting in J2SE 7.0, as binary numbers, using the form 0b (or 0B) followed by one or more binary digits (0 or 1). For example, 0b101010 is the integer 42. Like octal and hex numbers, binary literals may represent negative numbers.

Floating Point Literals:

Floating point numbers are expressed as decimal fractions or as exponential notation:

```
double decimalNumber = 5.0;  
decimalNumber = 89d;
```

```
decimalNumber = 0.5;  
decimalNumber = 10f;  
decimalNumber = 3.14159e0;  
decimalNumber = 2.718281828459045D;  
decimalNumber = 1.0e-6D;
```

Floating point numbers consist of:

1. an optional leading + or - sign, indicating a positive or negative value; if omitted, the value is positive,
2. one of the following number formats
 - *integer digits* (must be followed by either an exponent or a suffix or both, to distinguish it from an integer literal)
 - *integer digits* .
 - *integer digits* . *integer digits*
 - . *integer digits*
3. an optional exponent of the form
 - the exponent indicator e or E
 - an optional exponent sign + or - (the default being a positive exponent)
 - *integer digits* representing the integer exponent value
4. an optional floating point suffix:
 - either f or F indicating a single precision (4 bytes) floating point number, or
 - d or D indicating the number is a double precision floating point number (by default, thus the double precision (8 bytes) is default).

Here, *integer digits* represents one or more of the digits 0 through 9.

Character Literals:

Character literals are constant valued character expressions embedded in a Java program. Java characters are sixteen bit Unicode characters, ranging from 0 to 65535. Character literals are expressed in Java as a single quote, the character, and a closing single quote ('a', '7', '\$', 'π'). Character literals have the type char, an unsigned integer primitive type.

String Literals:

String literals consist of the double quote character (") (ASCII 34, hex 0x22), zero or more characters (including Unicode characters), followed by a terminating double quote character ("), such as: "Ceci est une string."

String literals may not contain unescaped newline or linefeed characters. However, the Java compiler will evaluate compile time expressions, so the following String expression results in a string with three lines of text:

Multi-line string.

```
1 String text = "This is a String literal\n"  
2               + "which spans not one and not two\n"  
3               + "but three lines of text.\n";
```

6. Discuss the scope and lifetime of a variable.

Answer:

Scope:

Scope of a variable refers to in which areas or sections of a program can the variable be accessed and lifetime of a variable refers to how long the variable stays alive in memory.

General convention for a variable's scope is, it is accessible only within the block in which it is declared. A block begins with a left curly brace { and ends with a right curly brace }.

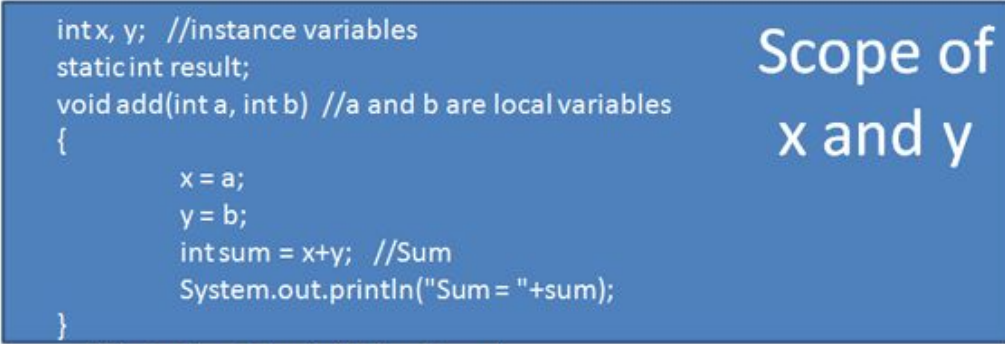
Instance Variables:

A variable which is declared inside a class and outside all the methods and blocks is an instance variable.

General scope of an instance variable is throughout the class except in static methods. Lifetime of an instance variable is until the object stays in memory.

```
class Sample
{
    int x, y; //instance variables
    static int result;
    void add(int a, int b) //a and b are local variables
    {
        x = a;
        y = b;
        int sum = x+y; //Sum
        System.out.println("Sum = "+sum);
    }

    public static void main(String[] args)
    {
        Sample obj = new Sample();
        obj.add(10,20);
    }
}
```



Startertutorials.com

Class Variables:

A variable which is declared inside a class, outside all the blocks and is marked static is known as a class variable.

General scope of a class variable is throughout the class and the lifetime of a class variable is until the end of the program or as long as the class is loaded in memory.

```

class Sample
{
    int x, y;
    static int result; //Class variable
    void add(int a, int b)
    {
        x = a;
        y = b;
        int sum = x+y;
        System.out.println("Sum = "+sum);
    }
    public static void main(String[] args)
    {
        Sample obj = new Sample();
        obj.add(10,20);
    }
}

```

Scope of
result

Startertutorials.com

Local Variables:

All other variables which are not instance and class variables are treated as local variables including the parameters in a method.

Scope of a local variable is within the block in which it is declared and the lifetime of a local variable is until the control leaves the block in which it is declared.

```

class Sample
{
    int x, y;
    static int result;
    void add(int a, int b) //a and b are local variables
    {
        x = a;
        y = b;
        int sum = x+y; //sum is a local variable
        System.out.println("Sum = "+sum);
    }
    public static void main(String[] args)
    {
        Sample obj = new Sample();
        obj.add(10,20);
    }
}

```

Scope of a and b

Scope of sum

Startertutorials.com

Summary of scope and lifetime of variables

Variable Type	Scope	Lifetime
Instance variable	Throughout the class except in static methods	Until the object is available in the memory
Class variable	Throughout the class	Until the end of the program
Local variable	Within the block in which it is declared	Until the control leaves the block in which it is declared

Startertutorials.com

<https://www.startertutorials.com/corejava/scope-lifetime-variables-java.html>

7. Show and discuss the general form of a java class.

Answer:

Class is the means by which you define objects. A class may contain three types of items : variables, methods and constructors.

Variables represent its state. Class can have static and instance variables. **Methods** provide the logic that constitutes the behavior defined by a class. Class can have static and instance methods.

Constructors initialize the state of a new instance of a class.

General form or Syntax of a Class:

```
class clsName
{
    // instance variable declaration
    type1 varName1 = value1;
    type2 varName2 = value2;
    :
    :
    typeN varNameN = valueN;

    // Constructors
    clsName(cparam1)
    {
        // body of constructor
    }
}
```

```

:
:
clsName(cparamN)
{
    // body of constructor
}

// Methods
rType1 methodName1(mParams1)
{
    // body of method
}
:
:
rTypeN methodNameN(mParamsN)
{
    // body of method
}
}

```

class indicates that a class named **clsName** is being declared. It must follow the java naming conventions for identifiers.

Instance variables named varName1 through varNameN are normal variable declaration syntax. Each variable must be assigned a type shown as type1 through typeN and may be initialized to a value shown as valueN.

Constructors always have the same name as the class. They do not have return values. cparam1 through cparamN are optional parameter lists.

Methods named mthName1 through mthNameN can be included. The return type of the methods are rtype1 through rTypeN and mParamN are an optional parameter lists.

Example of a General Class:

```

class clsSample
{
    // Variables
    static int ctr;
    int i;
    int j;

    // Constructor
    clsSample()
    {
        ctr = 0;
        i = 0;
        j = 0;
    }
    clsSample(int a, int b, int c)
    {
        ctr = a;
        i = b;
    }
}

```

```

        j = c;
    }

    // Methods
    int addition()
    {
        ctr++;
        return i + j;
    }
    int NumberOfInstances()
    {
        return ctr;
    }
}

```

<http://www.dailyfreecode.com/code/general-form-class-1387.aspx>

8. What are the different data types used in Java? Give examples.

Answer:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java –

1. Primitive Data Types
2. Reference/Object Data Types

Primitive Data Types:

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive) ($2^7 - 1$)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer

- Minimum value is -32,768 (-2^{15})
- Maximum value is 32,767 (inclusive) ($2^{15} - 1$)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648 (-2^{31})
- Maximum value is 2,147,483,647 (inclusive) ($2^{31} - 1$)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: int a = 100000, int b = -200000

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808 (-2^{63})
- Maximum value is 9,223,372,036,854,775,807 (inclusive) ($2^{63} - 1$)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: long a = 100000L, long b = -200000L

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: float f1 = 234.5f

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: double d1 = 123.4

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: boolean one = true

char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

Reference Datatypes

- ❖ Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- ❖ Class objects and various type of array variables come under reference datatype.
- ❖ Default value of any reference variable is null.
- ❖ A reference variable can be used to refer any object of the declared type or any compatible type.
- ❖ Example: `Animal animal = new Animal("giraffe");`

https://www.tutorialspoint.com/java/java_basic_datatypes.htm

9. How can you create an initialized array of objects?

Answer:

To create an initialized array of objects first we need to create an array object and assign it to that variable, there are two ways to do this:

- ★ Using new
- ★ Directly initializing the contents of that array

The first way is to use the new operator to create a new instance of an array:

```
String[] names = new String[10];
```

That line creates a new array of Strings with 10 slots (sometimes called elements). When you create a new array object using new, you must indicate how many slots that array will hold. This line does not put actual String objects in the slots-you'll have to do that later.

Array objects can contain primitive types such as integers or booleans, just as they can contain objects:

```
int[] temps = new int[99];
```

When you create an array object using new, all its slots are initialized for you (0 for numeric arrays, false for boolean, '\0' for character arrays, and null for objects). You can then assign actual values or objects to the slots in that array. You can also create an array and initialize its contents at the same time. Instead of using new to create the new array object, enclose the elements of the array inside braces, separated by commas:

```
String[] chiles = { "jalapeno", "anaheim", "serrano", "habanero", "thai" };
```

Each of the elements inside the braces must be of the same type and must be the same type as the variable that holds that array (the Java compiler will complain if they're not). An array the size of the

number of elements you've included will be automatically created for you. This example creates an array of String objects named chiles that contains five elements.

Technical Note
Note that the Java keyword null refers to a null object (and can be used for any object reference). It is not equivalent to zero or the '\0' character as the NULL constant is in C.

http://www.dmc.fmph.uniba.sk/public_html/doc/Java/ch5.htm

10. How can you create prefix and postfix forms of the increment and decrement operators?

Answer:

(In Short)

Java provides two increment and decrement operators which are unary increment (++) and decrement (--) operators. Increment and decrement operators are used to increase or decrease the value of an operand by one, the operand must be a variable, an element of an array, or a field of an object.

The increment and decrement operators increases or decreases the value of an int variable by 1 or of a floating-point (float, double) value by 1.0. The unary increment and decrement operators can also be applied to char variables to step forward or backward one character position in the Unicode sorting sequence. These operators are known as unary operators because they are applied to a single variable.

The increment and decrement operators are used in prefix or postfix manner. If the operator is placed before the variable it's called prefix mode of increment and decrement.

During an assignment of one variable to other the prefix mode of increment and decrement first increments or decrements the variable's value then updated value of the variable is used in assignment. On the contrary, in postfix mode of increment and decrement first variable is used in assignment then the variable is incremented or decremented.

Note that prefix and postfix mode of operations make no difference if they are used in an independent statement, where just the value is incremented or decremented but no assignment is made.

(+Details)

Increment Operator (++)

The ++ operator increments its single operand by one. The behavior of increment operator during an assignment operation depends on its position relative to the operand whether it is used in prefix or postfix mode. When used in prefix mode, it increments the operand and evaluates to the incremented value of that operand. When used in postfix mode, it increments its operand, but evaluates to the value of that operand before it was incremented.

Let's take an example to see the behavior of prefix and postfix form of Java's increment operator.

```
int x = 5, y;

// Demonstrating prefix increment
// first x will be incremented then
// updated value of x will be assigned to y
y = ++x;
System.out.println("y : " + y); //will print y : 6
System.out.println("x : " + x); //will print x : 6

// Demonstrating postfix increment
// first value of x will be assigned to y
// then x will be incremented
y = x++;
System.out.println("y : " + y); //will print y : 6
System.out.println("x : " + x); //will print x : 7

//If increment is made in an independent
//statement, prefix and postfix modes make no difference.
++x;
System.out.println("x : " + x); //will print x : 8

x++;
System.out.println("x : " + x); //will print x : 9
```

Decrement Operator (--)

The -- operator decrements its single operand by one. The behavior of decrement operator during an assignment operation depends on its position relative to the operand whether it is used in prefix or postfix mode. When used in prefix mode, it decrements the operand and evaluates to the decremented value of that operand. When used in postfix mode, it decrements its operand, but evaluates to the value of that operand before it was decremented.

Let's take an example to see the behavior of prefix and postfix form of Java's decrement operator.

```
int x = 5, y;

// Demonstrating prefix decrement
// first x will be decremented then
// updated value of x will be assigned to y
y = --x;
System.out.println("y : " + y); //will print y : 4
```

```

System.out.println("x : " + x); //will print x : 4

// Demonstrating postfix decrement
// first value of x will be assigned to y
// then x will be decremented
y = x--;
System.out.println("y : " + y); //will print y : 4
System.out.println("x : " + x); //will print x : 3

//If decrement is made in an independent
//statement, prefix and postfix modes make no difference.
--x;
System.out.println("x : " + x); //will print x : 2

x--;
System.out.println("x : " + x); //will print x : 1

```

<http://cs-fundamentals.com/java-programming/java-increment-decrement-operators.php>

11. Why is main method static in Java? Explain.

Answer:

There are quite a few reasons about why is main method static in Java:

1. Since the main method is static Java virtual Machine can call it without creating any instance of a class which contains the main method.
2. Since C and C++ also have similar main method which serves as entry point for program execution, following that convention will only help Java.
3. If main method were not declared static than JVM has to create instance of main Class and since constructor can be overloaded and can have arguments there would not be any certain and consistent way for JVM to find main method in Java.
4. Anything which is declared in class in Java comes under reference type and requires object to be created before using them but static method and static data are loaded into separate memory inside JVM called context which is created when a class is loaded. If main method is static than it will be loaded in JVM context and are available to execution.

<https://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html>

12. How does binary operator operate? Explain with example.

Answer:

A binary operator is an operator that operates on two operands and manipulates them to return a result. Operators are represented by special characters or by keywords and provide an easy way to compare numerical values or character strings.

Binary operators are presented in the form:

Operand1 Operator Operand2

Some common binary operators in computing include:

- Equal (==)
- Not equal (!=)
- Less than (<)
- Greater than (>)
- Greater than or equal to (>=)
- Less than or equal to (<=)
- Logical AND (&&)
- Logical OR (||)
- Plus (+)
- Minus (-)
- Multiplication (*)
- Divide (/)

Equal (==) and not-equal (!=) are called equality operators. They produce a result of true (or 1) or false (or 0). This type of operator returns "true" if both operands have the same value, or "false" if they don't have the same value.

For example, the following conditional operation will be performed if the operands are equal:

```
if(operand1 == operand2)
{
    //do the operation
}
```

Greater than (>), less than (<), greater than or equal to (>=) and less than or equal to (<=) are relation operators, which compare two operands and produce a result of either true or false. When two operands are compared, the result depends on the relative location of the two operands.

Logical AND (&&) and logical OR (||) are called logical operators. They compare operands and return a result of either true (1) or false (0). In logical AND, if both operands are true then the result is true. If either one of the operands is false, the result will be false. In logical OR, if both operands are true or either one of the operands is true then the result is true. If both operands are false then the result will be false.

<https://www.techopedia.com/definition/23953/binary-operator>

13. In `System.out.println()` what is `System`, `out` and `println`? Explain.

Answer:

System is a final class from the `java.lang` package.

out is a class variable of type `PrintStream` declared in the `System` class.

println is a method of the `PrintStream` class.

<https://www.youth4work.com/Talent/Core-Java/Forum/120020-explain-system-out-println>

More:

<http://net-informations.com/java/cjava/out.htm>

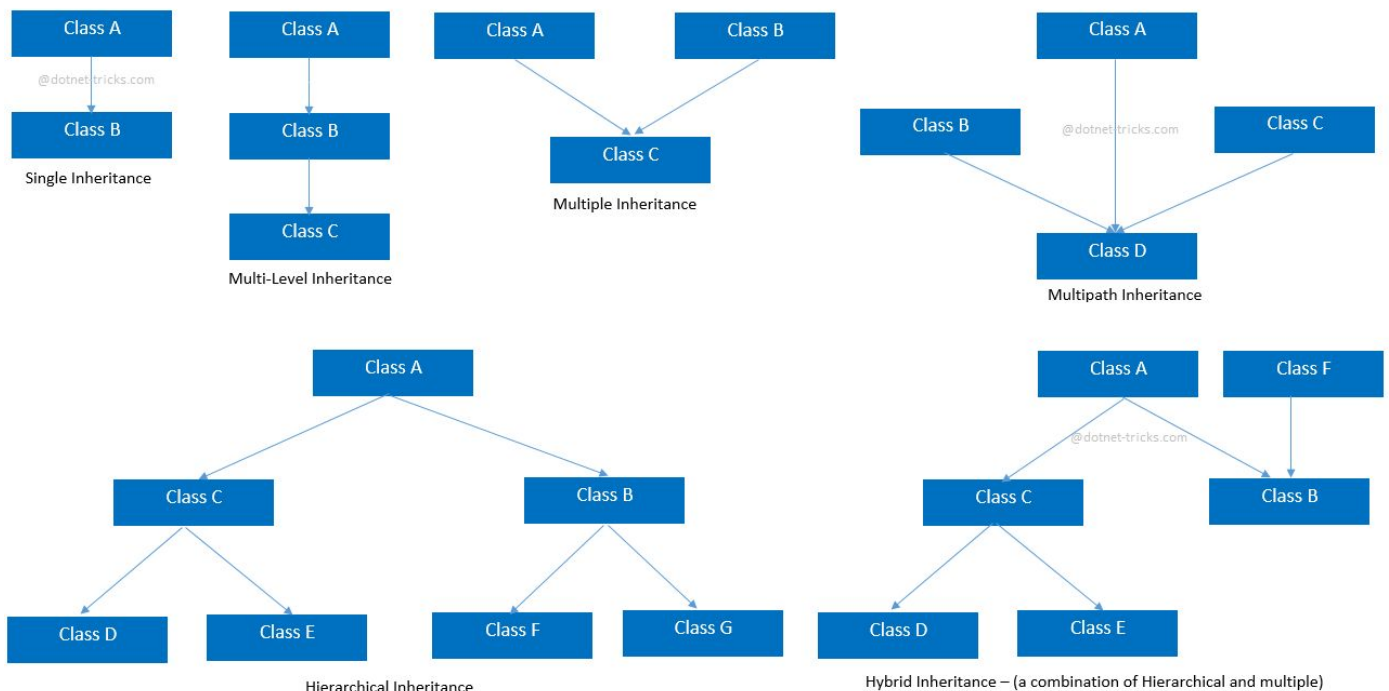
14. Explain different types/forms of inheritance with block diagram and examples.

Answer:

Inheritance is a mechanism of acquiring the features and behaviors of a class by another class. The class whose members are inherited is called the base class, and the class that inherits those members is called the derived class. Inheritance implements the IS-A relationship.

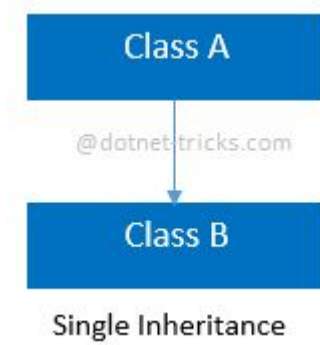
For example, mammal IS-A animal, dog IS-A mammal; Hence dog IS-A animal as well.

OOPs supports the six types of inheritance as given below :



1. **Single inheritance**

In this inheritance, a derived class is created from a single base class.

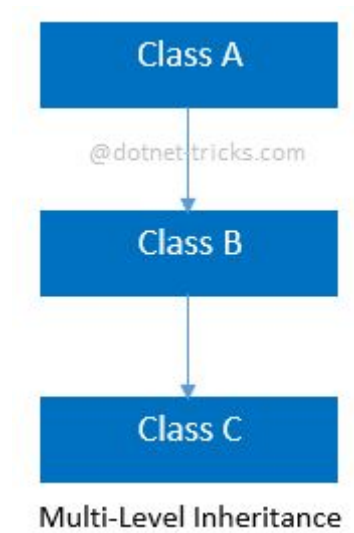


```
//Base Class
class A
{
public void fooA()
{
//TO DO:
}
}

//Derived Class
class B : A
{
public void fooB()
{
//TO DO:
}
}
```

2. Multi-level inheritance

In this inheritance, a derived class is created from another derived class.



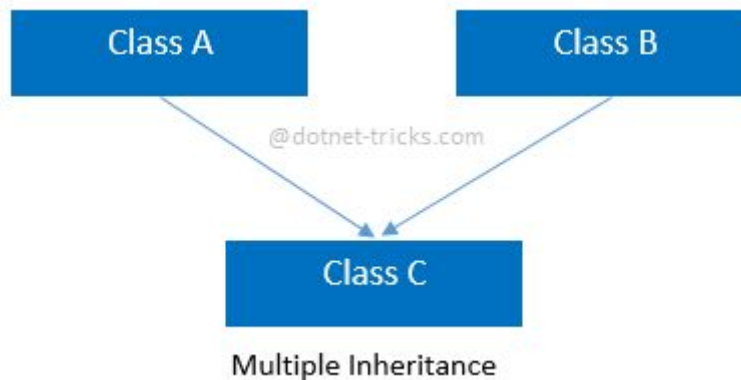
```
//Base Class
class A
{
public void fooA()
{
//TO DO:
}
}

//Derived Class
class B : A
{
public void fooB()
{
//TO DO:
}
}

//Derived Class
class C : B
{
public void fooC()
{
//TO DO:
}
}
```

3. Multiple inheritance

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F# etc.



//Base Class

```
class A
{
public void fooA()
{
//TO DO:
}
}
```

//Base Class

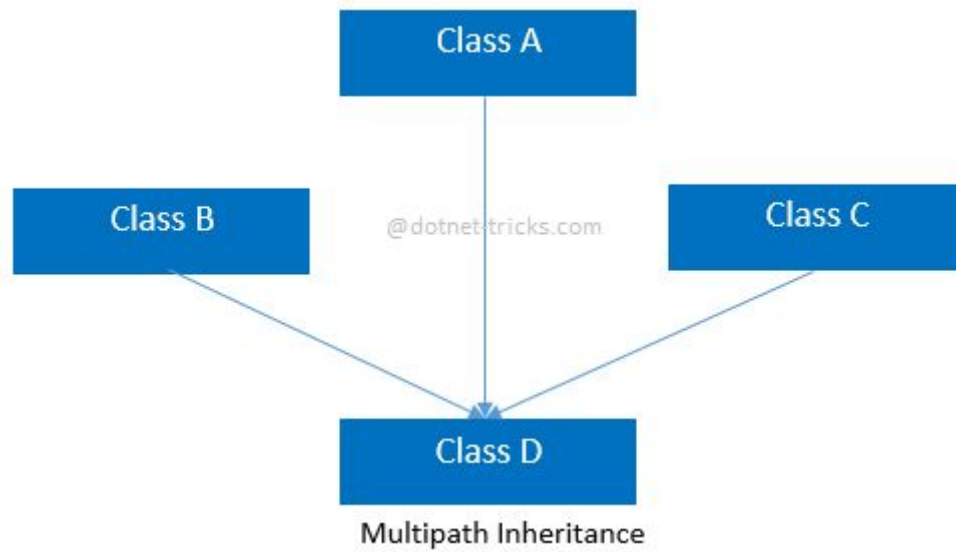
```
class B
{
public void fooB()
{
//TO DO:
}
}
```

//Derived Class

```
class C : A, B
{
public void fooC()
{
//TO DO:
}
}
```

4. Multipath inheritance

In this inheritance, a derived class is created from another derived classes and the same base class of another derived classes. This inheritance is not supported by .NET Languages like C#, F# etc.



```
//Base Class
class A
{
    public void fooA()
    {
        //TO DO:
    }
}
```

```
//Derived Class
class B : A
{
    public void fooB()
    {
        //TO DO:
    }
}
```

```
//Derived Class
class C : A
{
    public void fooC()
    {
        //TO DO:
    }
}
```

```
}
```

```
}
```

```
//Derived Class
```

```
class D : B, A, C
```

```
{
```

```
public void fooD()
```

```
{
```

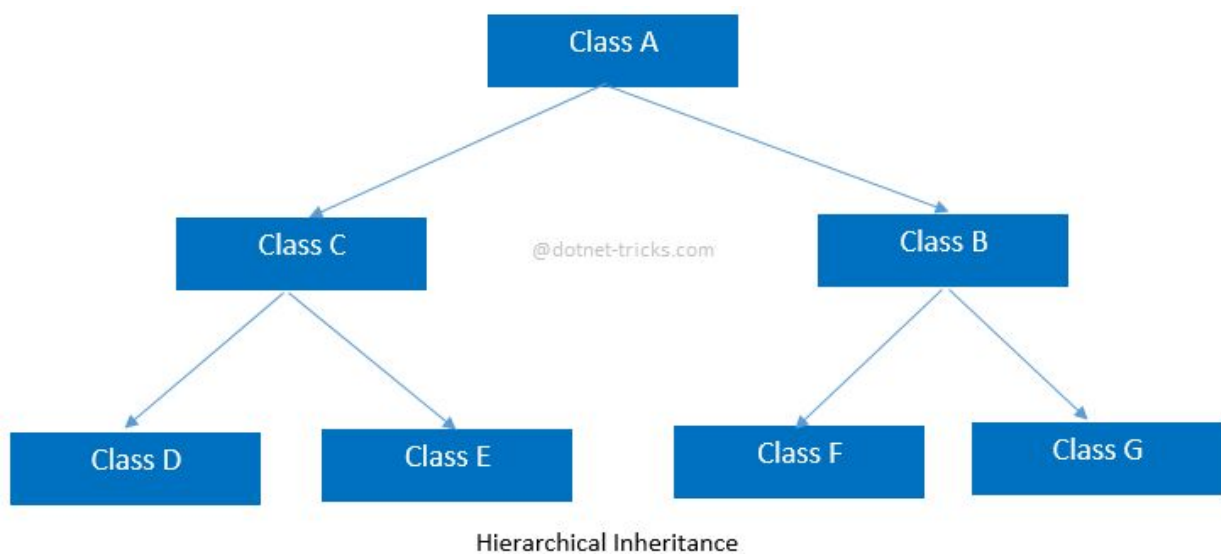
```
//TO DO:
```

```
}
```

```
}
```

5. Hierarchical inheritance

In this inheritance, more than one derived classes are created from a single base.



```
//Base Class
```

```
class A
```

```
{
```

```
public void fooA()
```

```
{
```

```
//TO DO:
```

```
}
```

```
}
```

```
//Derived Class
```

```
class B : A
```

```
{  
public void fooB()  
{  
    //TO DO:  
}  
}  
  
//Derived Class  
class C : A  
{  
public void fooC()  
{  
    //TO DO:  
}  
}  
  
//Derived Class  
class D : C  
{  
public void fooD()  
{  
    //TO DO:  
}  
}  
  
//Derived Class  
class E : C  
{  
public void fooE()  
{  
    //TO DO:  
}  
}  
  
//Derived Class  
class F : B  
{  
public void fooF()  
{
```



```

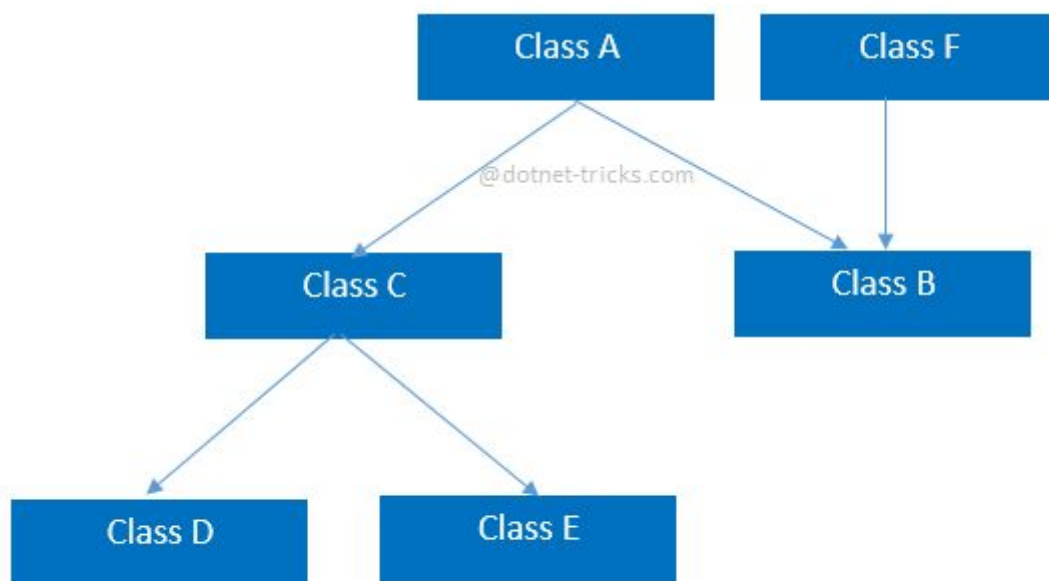
//TO DO:
}
}

//Derived Class
class G :B
{
public void fooG()
{
//TO DO:
}
}

```

6. Hybrid inheritance

This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritance.



Hybrid Inheritance – (a combination of Hierarchical and multiple)

```

//Base Class
class A
{
public void fooA()
{
//TO DO:
}
}

```

```
//Base Class  
  
class F  
{  
public void fooF()  
{  
//TO DO:  
}  
}
```

```
//Derived Class  
  
class B : A, F  
{  
public void fooB()  
{  
//TO DO:  
}  
}
```

```
//Derived Class  
  
class C : A  
{  
public void fooC()  
{  
//TO DO:  
}  
}
```

```
//Derived Class  
  
class D : C  
{  
public void fooD()  
{  
//TO DO:  
}  
}
```

```
//Derived Class  
  
class E : C
```

```

{
public void fooE()
{
//TO DO:
}
}

```

<https://www.dotnettricks.com/learn/oops/understanding-inheritance-and-different-types-of-inheritance>

For only in case of Java you can follow this website:

<https://beginnersbook.com/2013/05/java-inheritance-types/>

15. Describe a scenario in which multi-level inheritance can cause ambiguity. And how this ambiguity can be solved.

Answer:

(Not found yet)

16. Write a short program that will use Java I/O library to write n random numbers to a file.

Answer:

```

public class One {

    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub

        String str = new String();
        Random rng = new Random();

        Scanner enter1 = new Scanner(System.in);
        int n=enter1.nextInt();

        for (int i = 0; i < n; ++i)
            str= str + (rng.nextInt(10) + " ");

        try{
            FileWriter fw=new FileWriter("D:\\testout.txt");
            fw.write(str.toString());
            fw.close();
        }
    }
}

```

```

        System.out.println("Success...");
    } catch (Exception e) {
        System.out.println(e);
    }
}

```

17. How is polymorphism achieved at (a) compile time and (b) run time?

Answer:

Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.

Method Overloading: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

Example of static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```

class SimpleCalculator
{
    int add(int a, int b)
    {
        return a+b;
    }
    int add(int a, int b, int c)
    {
        return a+b+c;
    }
}

public class Demo
{
    public static void main(String args[])
    {
        SimpleCalculator obj = new SimpleCalculator();
        System.out.println(obj.add(10, 20));
        System.out.println(obj.add(10, 20, 30));
    }
}

```

Output:

```

30
60

```

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that's why it is called runtime polymorphism.

Example

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

```
class ABC{
    public void myMethod() {
        System.out.println("Overridden Method");
    }
}
public class XYZ extends ABC{

    public void myMethod() {
        System.out.println("Overriding Method");
    }
    public static void main(String args[]){
        ABC obj = new XYZ();
        obj.myMethod();
    }
}
```

Output:
Overriding Method

When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time. Since both the classes, child class and parent class have the same method myMethod. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

<https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/>

18. Why OOP is effective over structured programming? Explain.

Answer:

OOP is a methodology to help you manage the complexity of large software systems. It's actually a layer atop of procedural (or structured) programming.

The methodology works because it allows you to break down your code into smaller and more manageable units. An object is really just a refinement of the module concept, except that the module's data (or state) is hidden and all the functions (or methods) that are permitted to act on the data are collected in one place and tightly bound. Objects are usually very small compared to modules, which can be enormous.

The refinement comes in the form of inheritance and polymorphism which help to encourage code reusability.

<https://www.quora.com/How-is-Object-oriented-programming-more-effective-then-structured-programming/answer/Richard-Kenneth-Eng>

Structured Programming	Object Oriented Programming
Structured Programming is designed which focuses on process / logical structure and then data required for that process.	Object Oriented Programming is designed which focuses on data .
Structured programming follows top-down approach .	Object oriented programming follows bottom-up approach .
Structured Programming is also known as Modular Programming and a subset of procedural programming language .	<u>Object</u> Oriented Programming supports inheritance, encapsulation, abstraction, polymorphism , etc.
In Structured Programming, Programs are divided into small self contained functions .	In Object Oriented Programming, Programs are divided into small entities called objects .
Structured Programming is less secure as there is no way of data hiding .	Object Oriented Programming is more secure as having data hiding feature.
Structured Programming can solve moderately complex programs.	Object Oriented Programming can solve any complex programs.
Structured Programming provides less reusability , more function dependency.	Object Oriented Programming provides more reusability, less function dependency .
Less abstraction and less flexibility.	More abstraction and more flexibility .

<https://freefeast.info/difference-between/difference-between-structured-programming-and-object-oriented-programming-structured-programming-vs-object-oriented-programming/>

19. Write a fragment of code that make use of the shorthand operators like += and -=

Answer:

```
public class Example {
    public static void main(String[] args) {
        int j, p, q;
        j = 5;
        p = 1; q = 2;

        p += j;
        q -= j;
    }
}
```

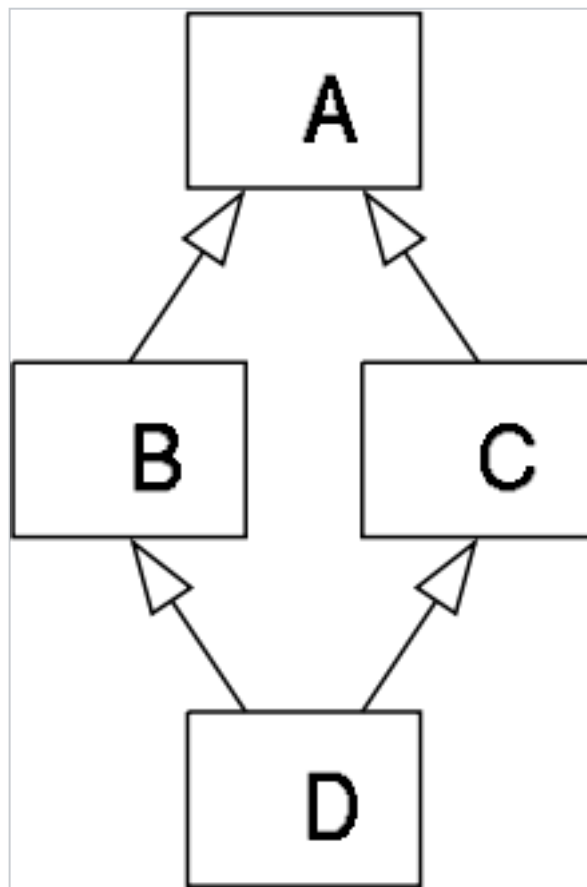
```
        System.out.println("p = " + p);  
        System.out.println("q = " + q);  
    }  
}
```

Output:

p = 6
q = -3

20. What is the ambiguity that arises in multiple inheritance? How it can be overcome? Explain with example.

Answer:



A diamond class inheritance diagram.

The "diamond problem" (sometimes referred to as the "deadly diamond of death") is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation.

Languages have different ways of dealing with these problems of repeated inheritance.

Java 8 introduces default methods on interfaces. If A,B,C are interfaces, B,C can each provide a different implementation to an abstract method of A, causing the diamond problem. Either class D must reimplement the method (the body of which can simply forward the call to one of the super implementations), or the ambiguity will be rejected as a compile error. Prior to Java 8, Java was not subject to the Diamond problem risk because it did not support multiple inheritance.

https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem