**How does default virtual behavior differ in C++ and Java ?**

**Default virtual behavior of methods is opposite in C++ and Java:**

**Virtual Function** is a member function of the base class which is overridden in the derived class

In C++, class member methods are **non-virtual** by default. They can be made **virtual** by using ***virtual*** keyword. For example, *Base::show()* is **non-virtua**l in following program and program prints *“Base::show() called”*.

|  |
| --- |
| #include<iostream>    using namespace std;    class Base {  public:        // non-virtual by default  **void show()** {           cout<<"Base::show() called";      }  };    class Derived: public Base {  public:  **void show()** {           cout<<"Derived::show() called";      }  };    int main()  {    Derived d;    Base &b = d;    b.show();    getchar();    return 0;  } |

**Output:**

Base::show() called

Adding *virtual* before definition of *Base::show()* makes program print *“Derived::show() called”*

#include <iostream>

using namespace std;

class Base {

public:

// non-virtual by default

virtual void show() {

cout<<"Base::show() called\n";

}

};

class Derived: public Base {

public:

void show() {

cout<<"Derived::show() called\n";

}

};

int main()

{

Derived d;

Base &b = d;

b.show();

return 0;

}

**Output:**

Derived::show() called

In Java, methods are virtual by default and can be made non-virtual by using *final* keyword. For example, in the following java program, *show()* is by default virtual and the program prints *“Derived::show() called”*

|  |
| --- |
| class Base {        // virtual by default      public void show() {         System.out.println("Base::show() called");      }  }    class Derived extends Base {      public void show() {         System.out.println("Derived::show() called");      }  }    public class Main {      public static void main(String[] args) {          Base b = new Derived();;          b.show();      }  } |

*Output:*

Derived::show() called

Being an object-oriented programming language java supports features like inheritance, polymorphism etc. Therefore, OOPs in Java deals with objects, classes, and functions. A virtual function is one of a member function which facilitates [run time polymorphism in Java](https://www.educba.com/runtime-polymorphism-in-java/).

Definition: A virtual function is not any special function, but it is a member function that facilitates the method overriding mechanism. That means, in OOPs, a virtual function of the parent class is a function that can be overridden by a child class which has the same type but with different functionality.

Syntax: For Virtual Function in Java, you should follow the basic syntax of java with annotations. To implement the overriding mechanism for the virtual function, @Override annotation may be used here to specifically point out which virtual function we want to override. Although it is not mandatory.

How Virtual Function Works in Java?

Now let us see how virtual function works. When we call an overridden method of child class through its parent type reference, then the type or reference of the object indicates which method will be invoked. The functionality of the virtual function is overridden by the inherited child class of the same type.

Some Points Regarding Virtual Function:

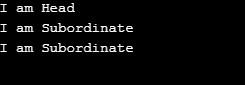
1. Functions of child and parent class must have the same name and have the same parameter.
2. IS-A relationship is mandatory (inheritance).
3. A virtual function cannot be private as we cannot override private methods of the parent class.
4. A virtual function cannot be stated as Final, as we cannot override Final methods.
5. A virtual function cannot be stated as Static, as we cannot override static methods.

Examples of Virtual Function in Java

Example #1

In this example, we will show how the virtual function showMe() is displaying different text depending on to which reference of object it is associated with. When it is associated with the “Head” type, it is showing messages from the parent class. Whereas, when it is associated with the “Subordinate” type, it is showing messages from the child class.

class Head {  
public void showMe() {  
System.out.println("I am Head");  
}  
}  
class Subordinate extends Head {  
@Override  
public void showMe() {  
System.out.println("I am Subordinate ");  
}  
}  
public class VirtualFuntionDemo {  
public static void main(String args[]) {  
Head superObject = new Head();  
superObject.showMe(); //method of super class or parent class is called  
Head subObject = new Subordinate(); // upcasting  
subObject.showMe();//method of sub class or child class is called by Parent reference, this is called "Virtual function"  
Subordinate subObject2 = new Subordinate();  
subObject2.showMe(); //method of sub class or child class is called  
}  
} Output:



Example #2

Let us take an example of the virtual function in the case of multilevel inheritance. In this example we have two levels of inheritance is taken into account. In this example, we will show how the virtual function administration() is displaying different messages depending on which type of object it is associated with. When it is associated with the “State” type, it is showing messages from the parent class. Whereas, when it is associated with the “District” type, it is showing messages from its child class. Again in the second level of inheritance, when it is associated with the “Municipality” type, it is showing messages from its child class of its parent which is “District” class.

Code:

class State{  
void administartion() {  
System.out.println("This is under state govt.");  
}  
}  
class District extends State{  
void administartion(){  
System.out.println("This is under District Magistrate");  
}  
}  
class Municipality extends District{  
void administartion(){  
System.out.println("This is under Mayor ");  
}  
}

public class VirtualFunctionDemo2 {

public static void main(String args[]){

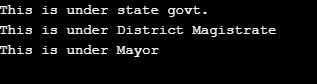
State superObject=new State ();  
State subObject=new District ();  
State sub2Object=new Municipality ();

superObject. administartion ();

subObject.administartion (); // run time polymorphism occurs in virtual function happening in first level of heritance

sub2Object.administartion (); // run time polymorphism occurs in virtual function happening in 2nd level of heritance  
}  
}

Output:



Example #3

Let us take another example of run time polymorphism in case of multilevel inheritance. In this example we have three levels of inheritance is taken into account. In this example, we will show how the virtual function whoami() is displaying different features depending on which type of object it is associated with. When it is associated with the “Cars” type, it is showing messages from the parent class. Whereas, when it is associated with the “SUV” type, it is showing messages from its child class. Again in the second level of inheritance, when it is associated with the “MPV” type, it is showing messages from its child class of its parent which is “SUV” class. Again in the third level of inheritance, when it is associated with the “Hatchback” type, it is showing messages from its child class of its parent which is the “MPV” class.

Code:

class Cars{  
void whoami() {  
System.out.println("This is Car");  
}  
}  
class SUV extends Cars{  
void whoami(){  
System.out.println("This is SUV");  
}  
}  
class MPV extends SUV{  
void whoami(){  
System.out.println("This is MPV");  
}  
}  
class Hatchback extends MPV{  
void whoami(){  
System.out.println("This is hatchback");  
}  
}  
public class VirtualFunctionDemo3 {  
public static void main(String args[]){  
Cars superObject=new Cars();  
Cars subObject=new SUV();  // object of child type : 1st level heritance  
Cars sub2Object=new MPV();  // object of child type : 2nd level heritance  
Cars sub3Object=new Hatchback();  // object of child type : 3rd level heritance  
superObject.whoami();  
subObject.whoami();  //run time polymorphism occurs in virtual function happening in first level of heritance  
sub2Object.whoami(); //run time polymorphism occurs in virtual function  happening in second level of heritance  
sub3Object.whoami(); //run time polymorphism occurs in virtual function  happening in third level of heritance  
}  
}

Output:

