**OOPs**

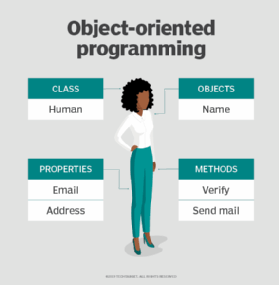
**#Introduction**

OOPs is a **programming paradigm based on the concept of object**.

Object: It is an entity which has *state/property* and *behaviour*.

Anything that object class hold are called *properties/data members*.

Object is an *instance*(variation of object) of class.



**Basic concepts:**

* **Run-time**
  + Running phase of program
  + A stage of programming life cycle
  + Time where program is running with external instructions needed for proper execution.
* **Compile-time**
  + While code is converting into machine language i.e. 0 and 1

**OOPs Concepts:**

* *Objects are real world entities like pen, chair, keyboard, etc*.
* **OOPs is a methodology or paradigm to design a program using classes and object.**
* It simplifies software development and mainteinance by providing some concepts:
  1. **Object**
     1. Physical or logical object that has a behavior like pen, chair, keyboard.
  2. **Class**
     1. **Collection of objects**. It’s an **logical entity**.
     2. **Instance of class is required to access** class objects and use data members and members functions
     3. Example:
        1. Let’s take a car, even though names and brands may be different but it has similarities like 4 wheels and no\_of\_seats.
  3. **Data Members**
     1. Data variables inside the class are data members
     2. This can be of any type **fundamental or user-defined**
  4. **Class Member Function**
     1. Function declaration inside the class
     2. Function definition can be inside or outside the class
     3. Code:

class Cube

{

public:

**int side;** //data member

/\*

Declaring function getVolume

with no argument and return type int.

\*/

**int getVolume();** //member function

};

* 1. **Derived Class (Child class/Subclass)**
     1. Used to inherit properties of base class(parent class)
     2. Properties can be added or modified according to requirement in derived class
     3. **Allows to define virtual method that helps in implementing polymorphism**
  2. **Base Class (Parent class/Superclass)**
     1. This class from which derived class is created through inheritance is called Base class.
  3. **Abstraction**
     1. It means hiding internal data and showing functionality
     2. Basically, it means hiding internal data may be sensitive data and showing outside world only the data is safe to be shown or only required data is shown
     3. Example:
        1. Phone call, we just do calling we won’t show internal working
     4. In C++, it is achieved using abstract class and interface to achieve abstraction
  4. **Polymorphism**
     1. When one task is performed by different ways
     2. Different situations may cause tasks to behave differently
     3. Overloading
        1. Subset of polymorphism
        2. An existing operator is said to be overloaded when it is forces to operate on new data types
  5. **Inheritance**
     1. When one class *inherits the behavior and properties* of parent or super class, it’s called inheritance
     2. It’s done to reuse the code i.e, reusability. So, that a code cannot be repeated if its needed to recreate.
     3. **Sub class**
        1. Class which inherits super class
     4. **Super class, base class**
        1. Class from which sub class inherits
  6. **Encapsulation**
     1. Binding or wrapping pieces of data together into single unit is known as encapsulation.
     2. For example:
        1. Medicine capsule: different medicines are wrapped together
        2. Take a look at a practical illustration of encapsulation: at a company, there are various divisions, including the sales division, the finance division, and the accounts division. All financial transactions are handled by the finance sector, which also maintains records of all financial data. In a similar vein, the sales section is in charge of all tasks relating to sales and maintains a record of each sale. Now, a scenario could occur when, for some reason, a financial official requires all the information on sales for a specific month. Under the umbrella term "sales section," all of the employees who can influence the sales section's data are grouped together.
     3. Dynamic binding
        1. A decision is made in run-time regarding code that will run in response to function call
        2. C++ supports virtual function is used for this

**Advantages of OOPS**

* It makes **development** and **maintenance** easier where in procedural oriental programming, it’s hard to maintain if code grows
* It provides **data hiding** while in pop, code is available for everyone
* Ability to **simulate real world event** effectively. It also provides real world problem’s solution.
* **Faster** and **easier** to execute.
* It provides **clear structure** of the program.
* It provides **DRY principle** i.e, “Don’t repeat yourself”, which makes code easier to maintain, develop and debug the code.
* **Reusability** of code.

**Why C++ is partial OOPs?**

1. **Main function is always outside of the class.**
   1. It means we can program C++, without using any class or objects
   2. Hence, concept of pure OOPs is violated.
2. **C++ supports global variable.**
   1. Global variable can be accessed by any objects within the program and defined outside of it.
   2. Encapsulation is violated here, even though c++ encourages to use encapsulation, classes and objects but ignores global variable.

**Real-World OOPs Examples**

* <https://fdhgjhjhukj.medium.com/object-oriented-programming-with-real-world-example-63f69f917d62>

**Class vs Structure**

| Class | Structure |
| --- | --- |
| * Members of a class are **private by default**. | * Members of a structure are **public by default**. |
| * Member classes/structures of a class are private by default. | * Member classes/structures of a structure are public by default. |
| * It is declared using the **class** keyword. | * It is declared using the **struct** keyword. |
| * It is normally used for **data abstraction** and further **inheritance**. | * It is normally used for the **grouping of data** |

**Code:**

* Point:1

**For class**

class Test {

    // x is private

    int x;

};

int main()

{

    Test t;

    t.x = 20; // compiler error because x

              // is private

    return t.x;

}

**For structure**

struct Test {

    // x is public

    int x;

};

int main()

{

    Test t;

    t.x = 20;

    // works fine because x is public

    std::cout << t.x;

}

* **Point:4**

**For class**

// Base class

class Parent {

public:

    int x;

};

// Subclass inheriting from

// base class (Parent).

class Child : public Parent {

public:

    int y;

};

int main()

{

    Child obj1;

    // An object of class Child has

    // all data members and member

    // functions of class Parent.

    obj1.y = 7;

    obj1.x = 91;

    cout << obj1.y << endl;

    cout << obj1.x << endl;

    return 0;

}

**Q. How many bytes allocated to empty class?**

* **1 byte** for identification and keeping track.

**Access Modifiers**

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as **Data Hiding.**

1. **Private**:
   1. Can only be used inside the class.
   2. Code:

class Circle

{

    // private data member

    private:

        double radius;

    // public member function

    public:

        double  compute\_area()

        {   // member function can access private

            // data member radius

            return 3.14\*radius\*radius;

        }

};

int main()

{

    // creating object of the class

    Circle obj;

    // trying to access private data member

    // directly outside the class

    obj.radius = 1.5; //will show error

    cout << "Area is:" << obj.compute\_area();

    return 0;

1. **Public**:
   1. Can be used inside the class
   2. Can be used outside of the class
   3. Code:

class Circle

{

    public:

        double radius;

        double  compute\_area()

        {

            return 3.14\*radius\*radius;

        }

};

int main()

{

    Circle obj;

    // accessing public datamember outside class

    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";

    cout << "Area is: " << obj.compute\_area();

    return 0;

}

1. **Protected**:
   1. a class and its subclasses have access to the variable.
   2. But not any other classes**.**
   3. Code:

class Parent

{

    // protected data members

    protected:

    int id\_protected;

};

// sub class or derived class from public base class

class Child : public Parent

class Child : public Parent

{

    public:

    void setId(int id)

    {

        // Child class is able to access the inherited

        // protected data members of base class

        id\_protected = id;

    }

    void displayId()

    {

        cout << "id\_protected is: " << id\_protected << endl;

    }

};

// main function

int main() {

    Child obj1;

    // member function of the derived class can

    // access the protected data members of the base class

    obj1.setId(81);

    obj1.displayId();

    return 0;

}

1. **Friend**
   1. Keyword: **friend**
   2. Gets special permission to access private and protected members of classes it’s declared on.
   3. Can be used in any access modifier ie, public, private and protected.
   4. **It’s types**:
      1. **Friend class**

class GFG {

private:

    int private\_variable;

protected:

    int protected\_variable;

public:

    GFG()

    {

        private\_variable = 10;

        protected\_variable = 99;

    }

    // friend class declaration

**friend class F; //syntax friend class name;**

};

// Here, class F is declared as a

// friend inside class GFG. Therefore,

// F is a friend of class GFG. Class F

// can access the private members of

// class GFG.

class F {

public:

    void display(GFG& t)

    {

        cout << "The value of Private Variable = "

             << t.private\_variable << endl;

        cout << "The value of Protected Variable = "

             << t.protected\_variable;

    }

};

// Driver code

int main()

{

    GFG g;

    F fri;

    fri.display(g);

    return 0;

}

* + 1. **Friend global function**

class base {

private:

    int private\_variable;

protected:

    int protected\_variable;

public:

    base()

    {

        private\_variable = 10;

        protected\_variable = 99;

    }

      // friend function declaration

**friend void friendFunction(base& obj);**

};

// friend function definition

void friendFunction(base& obj)

{

    cout << "Private Variable: " << obj.private\_variable

         << endl;

    cout << "Protected Variable: " << obj.protected\_variable;

}

// driver code

int main()

{

    base object1;

    friendFunction(object1);

    return 0;

}

* + 1. **Friend function**

class base; // forward definition needed

// another class in which function is declared

class anotherClass {

public:

    void memberFunction(base& obj);

};

// base class for which friend is declared

class base {

private:

    int private\_variable;

protected:

    int protected\_variable;

public:

    base()

    {

        private\_variable = 10;

        protected\_variable = 99;

    }

    // friend function declaration

**friend void anotherClass::memberFunction(base&);**

};

// friend function definition

void anotherClass::memberFunction(base& obj)

{

    cout << "Private Variable: " << obj.private\_variable

         << endl;

    cout << "Protected Variable: " << obj.protected\_variable;

}

// driver code

int main()

{

    base object1;

    anotherClass object2;

    object2.memberFunction(object1);

    return 0;

}

**Types of Member functions:**

* **Inline function**
  + This type of function is default in member function
  + They just replace macros of C language
    - While compiling this functions are copies everywhere
* **Const function**
  + If const keyword is used then the value can’t be change
  + Be it variable, function, anything
* **Static function**
  + Static keyword is used
  + Static means anything that holds position
  + When static keyword is used, position of that variable is set and cannot be changed.
  + Static in class, means objects can declare the class using scope resolution operator **::**
  + Multiple copies of variable can’t be created
* **Friend function**
  + Friend keyword is used
  + Not a class member function
  + can access private and protected members functions and data members
* **Simple function**
  + If other member functions are not in used, hence simple function.

**Getters & Setters**

* They used to access/read and modify private data members.
* It is done by creating functions/methods.

**Q. If we add int and char in class, size must be (4 + 1) but it is (4 + 4). Why?**

It is due to Padding (which add empty bytes to between memory addresses to align data in memory) and greedy alignment(the method is compiler dependent, It aligns till the boundary of maximum memory allocated).

**Static and Dynamic allocation**

* Static allocation
  + Hero ramesh;
  + Here, memory is allocated at compile time.
* Dynamic allocation
  + Hero \*kayiro = new Hero;
  + We can point to member by ***(\*b).getHealth() or b->getHealth().***
  + Here, we can allocate memory in run time.

**Constructor**

* Same name as class or structure
* **Special method invoked during object creation.**
* **Example**: student()
* No return type.
* ***Default constructor***
  + If nothing is passed in constructor.
  + No Input parameter but input is in parameterized and copy constructor
  + Code:

class construct {

public:

    int a, b;

    // Default Constructor

    construct()

    {

        a = 10;

        b = 20;

    }

};

int main()

{

    // Default constructor called automatically

    // when the object is created

    construct c;

    cout << "a: " << c.a << endl << "b: " << c.b;

    return 1;

}

* **Parameterized constructor**
  + parameter is passed.
  + Here, default constructor is killed**.**
  + Code**:**

class Point {

private:

    int x, y;

public:

    // Parameterized Constructor

    Point(int x1, int y1)

    {

        x = x1;

        y = y1;

    }

    int getX() { return x; }

    int getY() { return y; }

};

int main()

{

    // Constructor called

    Point p1(10, 15);

    // Access values assigned by constructor

    cout << "p1.x = " << p1.getX()

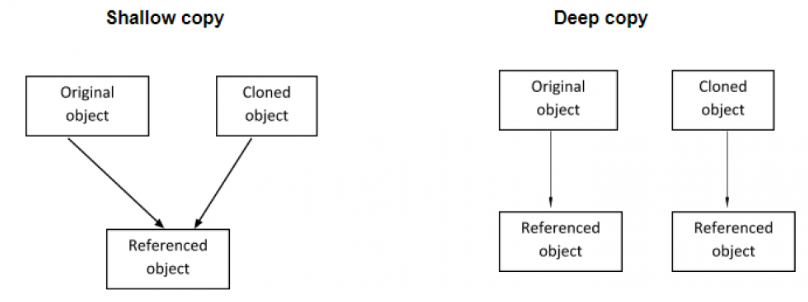
         << ", p1.y = " << p1.getY();

    return 0;

}

* **Copy constructor**
  + constructor is copied.
  + Syntax: Hero suresh(ramesh)
  + Ramesh is copied in suresh.
  + Inbuilit copy constructor is generated.
  + Hero(Hero& temp)
    - Parameter used &
    - If not used Hero call will be infinite.

**Shallow and Deep copy**



* Default copy constructor is shallow copy.
* Shallow copy, pointer points to same copy of the object of class
  + Creating copy of object by copying the member variables as it is.
  + Reflects changes made on original object
  + Stores the copy of original object and points the reference to the object
  + Faster
  + Code:

class box {

private:

    int length;

    int breadth;

    int height;

public:

    void set\_dimensions(int length1, int breadth1,

                        int height1)

    {

        length = length1;

        breadth = breadth1;

        height = height1;

    }

    // Function to display the dimensions

    // of the Box object

    void show\_data()

    {

        cout << " Length = " << length

             << "\n Breadth = " << breadth

             << "\n Height = " << height

             << endl;

    }

};

int main()

{

    // Object of class Box

    box B1, B3;

    // Set dimensions of Box B1

    B1.set\_dimensions(14, 12, 16);

    B1.show\_data();

    // When copying the data of object

    // at the time of initialization

    // then copy is made through

    // COPY CONSTRUCTOR

**box B2 = B1;**

**B2.show\_data();**

    // When copying the data of object **(in deep copy code..we’ll see copy constructor)**

    // after initialization then the

    // copy is done through DEFAULT

    // ASSIGNMENT OPERATOR

    B3 = B1;

    B3.show\_data();

    return 0;

}

* Deep copy, creates copy of each objects inside the class.
  + Stores the copy of original object and recursively copies the objects as well
  + Also copies the memory location as well which is outside the object but handled by objects
  + Slower
  + Code:

class box {

private:

    int length;

    int\* breadth;

    int height;

public:

    // Constructor

    box()

    {

**breadth = new int;**

    }

    void set\_dimension(int len, int brea,

                       int heig)

    {

        length = len;

        \*breadth = brea;

        height = heig;

    }

    void show\_data()

    {

        cout << " Length = " << length

             << "\n Breadth = " << \*breadth

             << "\n Height = " << height

             << endl;

    }

    // Parameterized Constructors for

    // for implementing deep copy

**// this is how copy constructor is made…it is made my compiler automatically but we have to make it if we’re using pointer**

    box(box& sample)

    {

        length = sample.length;

**breadth = new int;**

**\*breadth = \*(sample.breadth);**

        height = sample.height;

    }

    // Destructors

    ~box()

    {

        delete breadth;

    }

};

// Driver Code

int main()

{

    // Object of class first

**box first; //constructor is not called**

    // Set the dimensions

    first.set\_dimension(12, 14, 16);

    // Display the dimensions

    first.show\_data();

    // When the data will be copied then

    // all the resources will also get

    // allocated to the new object

    box second = first;

    // Display the dimensions

    second.show\_data();

    return 0;

}

**Why to use deep copy?**

* If we initiate a member variable with pointer like in code of deep copy… **\*breadth = new int**
* When shallow copy happens
  + new copy object’s pointer variable points to the same value pointed by original object
  + when destructor deletes the original pointer’s value, copied pointer’s value is also affects
* To stop that from happening we initiate copy constructor by ourself and add some lines like
  + box(box& sample)
  + {
  + length = sample.length;
  + **breadth = new int;**
  + **\*breadth = \*(sample.breadth);**
  + height = sample.height;
  + }
  + Here, breadth will create and points to copy of original pointer’s value

**Destructor**

* Memory de-allocation.
* No I/p parameter.
* No return type.
* Syntax: ***~hero()***
* For static allocated objects: Destructor is called automatically.
* For dynamic allocated objects: Destructor can only be called manually.

**:: (Scope resolution)**

* Use to access global variable of same name as local variable.
* To define function outside class.
* To access class’s static data members.

**Important keywords**

1. **new**
   1. Working:
      1. requests for memory allocation in heap
      2. If sufficient memory is available, this operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable
   2. **Code:**

class car {

string name;

int num;

public:

car(string a, int n)

{

cout << "Constructor called" << endl;

this ->name = a;

this ->num = n;

}

void enter()

{

cin>>name;

cin>>num;

}

void display()

{

cout << "Name: " << name << endl;

cout << "Num: " << num << endl;

}

};

int main()

{

// Using new keyword

car \*p = new car("Honda", 2017);

p->display();

}

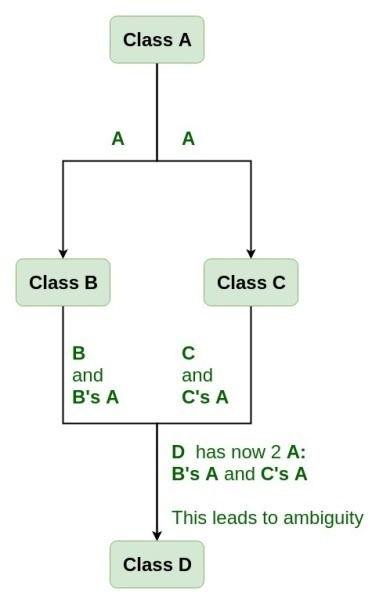
**Output:**

Constructor called

Name: Honda

Num: 2017

1. **Static**
   1. A class level scope
   2. Comes in picture before main()
   3. While class is loading, static can be executed
   4. If we have to run a block once, we can use static.
      1. Static blocks only runs once, during class loading
   5. It can execute a method without creating an object.
   6. To access it:
      1. **Data\_type** *class\_name* :: *var\_name* = **value**;
      2. Ex, **int** *Hero* :: *num* = **24**;
2. **Virtual keyword**
   1. Member function used in base class to be re-defined in derived class.
   2. It is dynamic in nature.
   3. Must be declared dynamically.
   4. If it’s not redefined, statement from base class is used
   5. Should be accessed with pointer or reference of base class type to achieve run-time polymorphism.
   6. It’s called during run-time.
   7. **Pure virtual class**
      1. Its declared in base class
      2. Must be 0 or Null
   8. **Code:** code in polymorphism example.
   9. **Virtual base class:**
      1. Used in virtual inheritance to prevent multiple instances of a given class appearing in inheritance hierarchy when using multiple instances.
      2. For example:



class A {

public:

    void show()

    {

        cout << "Hello form A \n";

    }

};

class B : public A {

};

class C : public A {

};

class D : public B, public C {

};

int main()

{

    D object;

    object.show();

}

**error: request for member 'show' is ambiguous**

**object.show();**

* + 1. To prevent this ambiguity we use virtual keyword for base class while inheriting

class A {

public:

    int a;

    A() // constructor

    {

        a = 10;

    }

};

class B : public virtual A {

};

class C : public virtual A {

};

class D : public B, public C {

};

int main()

{

    D object; // object creation of class d

    cout << "a = " << object.a << endl;

    return 0;

}

* + 1. Explanation:
       1. When base class is specified as virtual base, it acts as a indirect base for more than once without duplication of any other data members.
       2. Single copy of its data members is shared with base classes that use virtual base class.
  1. **Can virtual functions be private?**
     1. Yes, they can be initialized as public and can be done private in derived class.
     2. **Code:**

class base {

public:

    // default base constructor

    base() { std::cout << "base class constructor\n"; }

    // virtual base destructor

    // always use virtual base

    // destructors when you know you

    // will inherit this class

    virtual ~base()

    {

        std::cout << "base class destructor\n";

    }

    // public method in base class

    void show()

    {

        std::cout << "show() called on base class\n";

    }

    // public virtual function in base class,

    // contents of this function are printed when called

    // with base class object when called with base class

    // pointer contents of derived class are printed on

    // screen

    virtual void print()

    {

        std::cout << "print() called on base class\n";

    }

};

class derived : public base {

public:

    // default derived constructor

    derived()

        : base()

    {

        std::cout << "derived class constructor\n";

    }

    // virtual derived destructor

    // always use virtual destructors

    // when inheriting from a

    // base class

    virtual ~derived()

    {

        std::cout << "derived class destructor\n";

    }

private:

    // private virtual function in derived class overwrites

    // base class virtual method contents of this function

    // are printed when called with base class pointer or

    // when called with derived class object

    virtual void print()

    {

        std::cout << "print() called on derived class\n";

    }

};

int main()

{

    std::cout << "printing with base class pointer\n";

    // construct base class pointer with derived class

    // memory

    base\* b\_ptr = new derived();

    // call base class show()

    b\_ptr->show();

    // call virtual print in base class but it is overridden

    // in derived class also note that print() is private

    // member in derived class, still the contents of

    // derived class are printed this code works because

    // base class defines a public interface and drived

    // class overrides it in its implementation

    b\_ptr->print();

    delete b\_ptr;

}

* + 1. **Virtual destructor**
       1. Deleting a derived class using pointer to base class with non-virtual destructor results in undefined behavior even though it works correctly.
       2. But if use virtual destructor in place of destructor it’ll make sure object is deleted correctly.
       3. As per guidelines, if virtual function is declared in the class it should have virtual destructor even if it doesn’t do anything.
       4. Code:

class base {

  public:

    base()

    { cout << "Constructing base\n"; }

    virtual ~base()

    { cout << "Destructing base\n"; }

};

class derived : public base {

  public:

    derived()

    { cout << "Constructing derived\n"; }

    ~derived()

    { cout << "Destructing derived\n"; }

};

int main()

{

  derived \*d = new derived();

  base \*b = d;

  delete b;

  getchar();

  return 0;

}

1. **Abstract**
   1. **Abstract class**
      1. **If it has at least one pure virtual function**
      2. **If derived class doesn’t override virtual function, it become abstract class as well.**
      3. **We can have pointers and references of abstract class types**

#include<iostream>

using namespace std;

class Base

{

public:

virtual void show() = 0;

};

class Derived: public Base

{

public:

void show() { cout << "In Derived \n"; }

};

int main(void)

{

Base \*bp = new Derived();

bp->show();

return 0;

}

* + 1. **Abstract class can have constructors**

**Interface in c++**

* It is defined as describing the behavior of the class without implementing it.
* Describing class in laymen terms
* Must use pure virtual function
* Interface and abstract class are same
* **Code:**
  + class Interface name
  + {
  + public:
  + virtual type function name () =0;
  + virtual type function name(type)=0;
  + ~Interface name ();
  + }

1. **Final**
   1. Value declared by final keyword cannot be modified in the future.
   2. **Variable:**
      1. If a variable declared by final keyword it cannot be modified
      2. **Code:**

// declaring a final variable

class FinalVariable {

        final int var = 50;

        var = 60 //This line would give an error

}

* + 1. **Reference variable:**
       1. You cannot change what object it refers to but you can modify the object
       2. **Code:**

class Reference{

    public int value = 5;

}

class frVariable {

    public static void FinalReference( String args[] ) {

      final Reference example = new Reference(); //declaration

      example.value = 6; // Modifying the object creates no disturbance

      Reference another = new Reference();

      example = another; // Attempting to change the object it refers to, creates an error

         }

}

* 1. **Parameter**
     1. It means value cannot be changed anywhere in the function
     2. **Code:**

class finalParameter {

    public static void example( final int parameter ) {

      parameter = 4; //attempting to reassign a value to a parameter throws an error

    }

}

* 1. **Method**
     1. Final method cannot be overridden or hidden
     2. **Code:**

// declaring a final method

class Base{

    public final void finalMethod(){

        System.out.print("Base");

    }

}

class Derived extends Base{

    public final void finalMethod() { //Overriding the final method throws an error

        System.out.print("Derived");

    }

}

* 1. **Classes**
     1. Final class cannot be subclassed
     2. **Code:**

// declaring a final class

final class FinalClass {

  //...

}

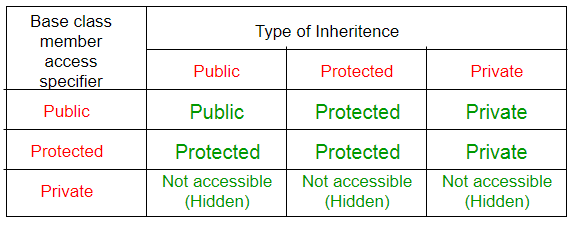
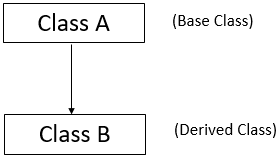
class Subclass extends FinalClass{ //attempting to subclass a final class throws an error

  //...

}

1. **Explicit**
2. **This**
   1. It’s used when the name of instance and local variable is same.
      1. *this* keyword is used to point to current instance variable.
      2. Refers to current object
3. **Const**
4. **Super**

**#Pillar of OOPs**

1. **Encapsulation**
   1. Wraps data members and functions together to manipulate data.
   2. Wrapping of data and information in single unit
   3. It is done to prevent outside misuse.
   4. It is basically information hiding.
      1. Security.
      2. Can make class read-only.
      3. Unit testing.
   5. ***Fully encapsulation***: All data members are private.
   6. **Example**:
      1. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,
      2. The finance section handles all the financial transactions and keeps records of all the data related to finance.
      3. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.
   7. It also leads to data abstraction
   8. In C++, it can be implemented using class and access modifiers
   9. **Advantage:**
      1. **Read-only or write-only**
         1. With the help of getter and setter function we can make the program read/write-only
      2. **Data hiding**
      3. **Easy to test**
         1. Good for unit testing
   10. **Features:**
       1. Creating a class to encapsulate data members and methods into a single unit
       2. Hiding relevant data using access modifiers
       3. We can’t directly call the function inside class, we have to make the object first
       4. Function inside the class must use data members of the class without doing so it can’t be called encapsulation
       5. Increases the security
       6. Helps control the modification of our data members
   11. **Code:** 
       * class Person {
       * private:
       * string name;
       * int age;
       * public:
       * Person(string name, int age) {
       * this->name = name;
       * this->age = age;
       * }
       * void setName(string name) {
       * this->name = name;
       * }
       * string getName() {
       * return name;
       * }
       * void setAge(int age) {
       * this->age = age;
       * }
       * int getAge() {
       * return age;
       * }
       * };
       * int main() {
       * Person person("John Doe", 30);
       * cout << "Name: " << person.getName() << endl;
       * cout << "Age: " << person.getAge() << endl;
       * person.setName("Jane Doe");
       * person.setAge(32);
       * cout << "Name: " << person.getName() << endl;
       * cout << "Age: " << person.getAge() << endl;
       * return 0;
       * }
   12. **Output:**
       * Name: John Doe  
         Age: 30  
         Name: Jane Doe  
         Age: 32
   13. **Another code:**
       * class Encapsulation {
       * private:
       * // Data hidden from outside world
       * int x;
       * public:
       * // Function to set value of
       * // variable x
       * void set(int a) { x = a; }
       * // Function to return value of
       * // variable x
       * int get() { return x; }
       * };
       * // Driver code
       * int main()
       * {
       * Encapsulation obj;
       * obj.set(5);
       * cout << obj.get();
       * return 0;
       * }
   14. **Output:**
       * 5
2. **Inheritance**
   1. It’s a feature or process of reusing or extending parent class without modifying them.
   2. Can add new features on its own.
   3. Producing hierarchical relationship between them.
   4. Creating a child class:
      1. Class class\_name: access\_modifier parent\_name{}
   5. 
   6. **Relationships:**
      1. **Is-A** relationship
         1. Whenever one class inherits another class it’s an Is-A relationship
      2. **Has-A** relationship
         1. Whenever instance of one class is used in another it’s an Has-A relationship
   7. **Types of inheritance**
      1. **Single inheritance**
         1. 
         2. Here, derieved class inherits only one base or parent class.
         3. **Code:**

class Account {

public:

float salary = 60000;

};

class Programmer: public Account {

public:

float bonus = 5000;

};

int main(void) {

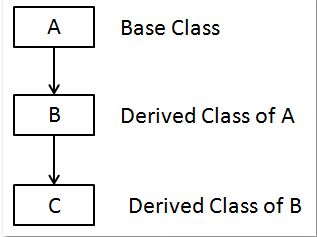
Programmer p1;

cout<<"Salary: "<<p1.salary<<endl;

cout<<"Bonus: "<<p1.bonus<<endl;

return 0;

}

* + 1. **Multi-Level inheritance**
       1. 
       2. Here, we can inherit from more than one classes.
       3. **Code:**

class Animal {

public:

void eat() {

cout<<"Eating..."<<endl;

}

};

class Dog: public Animal

{

public:

void bark(){

cout<<"Barking..."<<endl;

}

};

class BabyDog: public Dog

{

public:

void weep() {

cout<<"Weeping...";

}

};

int main(void) {

BabyDog d1;

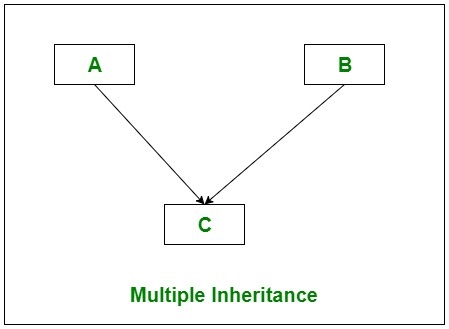
d1.eat();

d1.bark();

d1.weep();

return 0;

}

* + 1. **Multiple Inheritance**
       1. 
       2. Syntax: class name: access\_modifier class\_name1, access\_modifier class\_name2{}
       3. Here, we can inherit from multiple classes.
       4. Code:

class A

{

protected:

int a;

public:

void get\_a(int n)

{

a = n;

}

};

class B

{

protected:

int b;

public:

void get\_b(int n)

{

b = n;

}

};

class C : public A,public B

{

public:

void display()

{

std::cout << "The value of a is : " <<a<< std::endl;

std::cout << "The value of b is : " <<b<< std::endl;

cout<<"Addition of a and b is : "<<a+b;

}

};

int main()

{

C c;

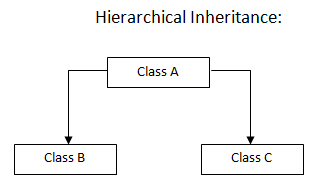
c.get\_a(10);

c.get\_b(20);

c.display();

return 0;

}

* + 1. **Hierarchical Inheritance**
       1. 
       2. One class can be parent for more than one derieved class.
       3. **Code:**

class A

{

    // body of the class A.

}

class B : public A

{

    // body of class B.

}

class C : public A

{

    // body of class C.

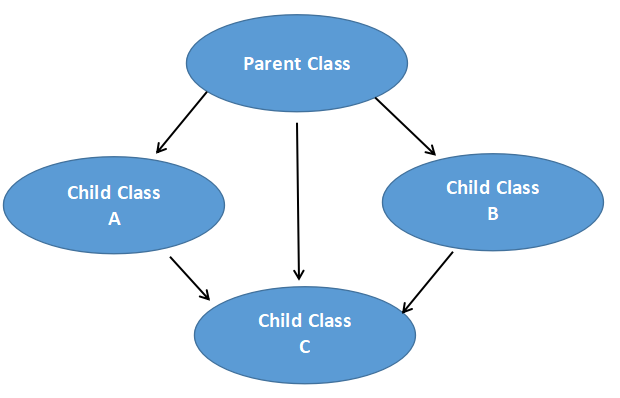
}

class D : public A

{

    // body of class D.

}

* + 1. **Hybrid Inheritance**
       1. 
       2. Here, child class inherits different types of inheritance.
       3. **Code:**

class A

{

protected:

int a;

public:

void get\_a()

{

std::cout << "Enter the value of 'a' : " << std::endl;

cin>>a;

}

};

class B : public A

{

protected:

int b;

public:

void get\_b()

{

std::cout << "Enter the value of 'b' : " << std::endl;

cin>>b;

}

};

class C

{

protected:

int c;

public:

void get\_c()

{

std::cout << "Enter the value of c is : " << std::endl;

cin>>c;

}

};

class D : public B, public C

{

protected:

int d;

public:

void mul()

{

get\_a();

get\_b();

get\_c();

std::cout << "Multiplication of a,b,c is : " <<a\*b\*c<< std::endl;

}

};

int main()

{

D d;

d.mul();

return 0;

}

***Inheritance Ambiguity***

* + - 1. If there if 2 or more class of same name.
      2. We can inherit the class we wanted using ::
  1. **Modes of inheritance**
     1. 1. class ABC : private XYZ              //private derivation  
                    {                }  
        2. class ABC : public XYZ              //public derivation  
                    {               }  
        3. class ABC : protected XYZ              //protected derivation  
                    {              }  
        4. class ABC: XYZ                            //private derivation by default  
        {            }
     2. **public mode**
        1. public member in base class will become public in derived class
        2. protected member will be protected
     3. **Protected mode**
        1. If we derive from protected base class, public as well as protected member, both will be protected in derived class
     4. **Private mode**
        1. All derived members from private base class will be private in derived class
  2. **Disadvantages/Limitations**
     1. Suppose given class is inherited from 10 parent classes, hence it takes too much jumps to get member i.e, it takes time to do the process.
     2. Member in base class is changes, then member in derived class will change as well.

**Why multiple inheritance is not possible in java?**

* To **reduce the complexity** and **simplify** the language
* Example:
  + Suppose there are 3 classes A, B, and C
  + C inherits A and B
  + If A and B have same methods
    - Then there will be ambiguity to call A and B
  + To Java renders complile time error instead of run-time error coz its better this way

**Why multiple inheritance is not possible in java but multiple instances is allowed?**

* Coz instances only tell what it’s doing and not how it’s doing it.

**Simulating Final class in C++/ Making class inheritable**

* We can make any class inheritable by using final keyword in java and c#.
* Code:
  + // The class to be made final
  + class Final;
  + // used to make the Final class final
  + class MakeFinal {
  + private:
  + MakeFinal() { cout << "MakFinal constructor" << endl; }
  + friend class Final;
  + };
  + class Final : virtual MakeFinal {
  + public:
  + Final() { cout << "Final constructor" << endl; }
  + };
  + // Compiler error
  + class Derived : public Final {
  + public:
  + Derived() { cout << "Derived constructor" << endl; }
  + };
  + int main(int argc, char\* argv[])
  + {
  + Derived d;
  + return 0;
  + }
* Output:
  + In constructor 'Derived::Derived()':
  + error: 'MakeFinal::MakeFinal()' is private
* we have to use Final class, makeFinal and friend class
* Hence, Derived class invokes constructor of makeFinal which is private and can’t be called
* It’ll show compiler error
* But in C++ 11 update
  + Code:
    - class Base final {
    - // body
    - };
    - // Compile error because base class is final
    - class Derived : public Base {
    - // body
    - };
    - int main() {
    - return 0;
    - }
  + Output:
    - prog.cpp:8:7: error: cannot derive from ‘final’ base ‘base’ in derived type ‘derive’
    - class derive: public base // compile error because base class is final

**Diamond Problem**

* **diamondproblem.png**
* Person is inherited by student and faculty, then student and faculty is called by TA
* **Code:**

class Person {

// Data members of person

public:

Person(int x) { cout << "Person::Person(int ) called" << endl; }

};

class Faculty : public Person {

// data members of Faculty

public:

Faculty(int x):Person(x) {

cout<<"Faculty::Faculty(int ) called"<< endl;

}

};

class Student : public Person {

// data members of Student

public:

Student(int x):Person(x) {

cout<<"Student::Student(int ) called"<< endl;

}

};

class TA : public Faculty, public Student {

public:

TA(int x):Student(x), Faculty(x) {

cout<<"TA::TA(int ) called"<< endl;

}

};

int main() {

TA ta1(30);

}

* **Output:**

**Person::Person(int ) called**

Faculty::Faculty(int ) called

**Person::Person(int ) called**

Student::Student(int ) called

TA::TA(int ) called

* Means, constructor of person is called twice and destructor of person is also called twice when object *ta1* is destructed,
* So, TA gets 2 copies of person class members
* which creates ambiguity
* To solve this we use ***virtual*** keyword
  + **Code:**

class Person {

public:

Person(int x) { cout << "Person::Person(int ) called" << endl; }

Person() { cout << "Person::Person() called" << endl; }

};

class Faculty : virtual public Person {

public:

Faculty(int x):Person(x) {

cout<<"Faculty::Faculty(int ) called"<< endl;

}

};

class Student : virtual public Person {

public:

Student(int x):Person(x) {

cout<<"Student::Student(int ) called"<< endl;

}

};

class TA : public Faculty, public Student {

public:

TA(int x):Student(x), Faculty(x) {

cout<<"TA::TA(int ) called"<< endl;

}

};

int main() {

TA ta1(30);

}

* + **Output:**

Person::Person() called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

* Now, we can see *person* constructor is called only once.
* That’s coz, only one instance of base class is shared with everyone.

# [What is object slicing?](https://stackoverflow.com/questions/274626/what-is-object-slicing)

# When a derived class object is assigned to a base class object and extra components of derived class is sliced off or not considered to generate the base class, this process is termed as object slicing.

# In simple words, extra components of derived class is sliced off or not considered and priority is given to base class objects.

# Code:

# class Base {

# int x, y;

# };

# class Derived : public Base {

# int z, w;

# };

# int main()

# {

# Derived d;

# // Object Slicing,

# // z and w of d are sliced off

# Base b = d;

# }

# Local class

# Class declared inside a function is called as Local class

# Member functions should be declared inside the class

# Local class can access global variable, classes

# Can access only static variables and enum from the function its declared within

# Can’t have static data member but can have static function

# Code:

# void fun()

# {

# // Local class

# class Test {

# // Body

# };

# Test t; // Fine

# Test\* tp; // Fine

# }

# int main()

# {

# Test t; // Error

# Test\* tp; // Error

# return 0;

# }

**Does overloading work with Inheritance?**

We have a function in base class and another function in derived class having same name as base class. Let’s see what happens then…

**C++**

* Code:
  + class Base
  + {
  + public:
  + int f(int i)
  + {
  + cout << "f(int): ";
  + return i+3;
  + }
  + };
  + class Derived : public Base
  + {
  + public:
  + double f(double d)
  + {
  + cout << "f(double): ";
  + return d+3.3;
  + }
  + };
  + int main()
  + {
  + Derived\* dp = new Derived;
  + cout << dp->f(3) << '\n';
  + cout << dp->f(3.3) << '\n';
  + delete dp;
  + return 0;
  + }
* Output:
  + f(double): 6.3
  + f(double): 6.6
* Instead of f(int): 6 f(double): 6.6 because there is no scope resolution between base and derived class and c++ compiler looks for derived class function and stops and it doesn’t disturbs base class. Hence there is no overloading across scopes and derived class scopes are no exception to this general rule

**Java**

* **Code:**
  + class Base
  + {
  + public int f(int i)
  + {
  + System.out.print("f (int): ");
  + return i+3;
  + }
  + }
  + class Derived extends Base
  + {
  + public double f(double i)
  + {
  + System.out.print("f (double) : ");
  + return i + 3.3;
  + }
  + }
  + class myprogram3
  + {
  + public static void main(String args[])
  + {
  + Derived obj = new Derived();
  + System.out.println(obj.f(3));
  + System.out.println(obj.f(3.3));
  + }
  + }
* Output:
  + f (int): 6
  + f (double): 6.6
* So in Java, overloading works across scopes, contrary to C++. The Java compiler determines the correct version of the overloaded method to be executed at compile time based upon the type of arguments used to call the method and the parameters of the overloaded methods of both these classes that receive the values of arguments used in call and executes that overloaded method.

1. **Polymorphism**
   1. From 2 greek words, poly means many and morphs means forms i.e. many forms
   2. Perform single action in different ways
   3. Real world example:
      1. Suppose a person
      2. Who is father, husband, son, employ
      3. He can have different behavior in different situations
   4. **Code:**

**class Geeks** {

public:

    // Function with 1 int parameter

    void **func**(int x)

    {

        cout << "value of x is " <<

                 x << endl;

    }

    // Function with same name but

    // 1 double parameter

    void **func**(double x)

    {

        cout << "value of x is " <<

                 x << endl;

    }

    // Function with same name and

    // 2 int parameters

    void **func**(int x, int y)

    {

        cout << "value of x and y is " <<

                 x << ", " << y << endl;

    }

};

// Driver code

int main()

{

**Geeks obj1;**

    // Function being called depends

    // on the parameters passed

    // func() is called with int value

    obj1.**func**(7);

    // func() is called with double value

    obj1.**func**(9.132);

    // func() is called with 2 int values

    obj1.**func**(85, 64);

    return 0;

}

* 1. It has 2 types:
     1. **Compile time polymorphism(static polymorphism)**
        1. Can be achieved by function and operator overloading
        2. ***Function overloading***
           1. When there are functions with same name/types but different parameters
           2. Using one function in different ways.
           3. When we use constructors, with different parameters it act as different constructors

This is same as function overloading.

We just have to play with parameters.

* + - * 1. **Code:**

Above polymorphism code…

* + - 1. ***Operator overloading***
         1. For example:

‘+’ operator can add numbers and it can add strings,

* + - * 1. Operators that can’t be overloaded:

‘**.**’ , member access or dot operator

‘**? :**’ , ternary operator

‘**::**’ , scoper resolution operator

‘**\***’ , pointer to member operator

‘**sizeof’** and ‘**typeid’** , size and type of object respectively

* + - * 1. Can be used to add real and imaginary no.

**Code**

class Complex {

private:

    int real, imag;

public:

**Complex(int r = 0,**

**int i = 0)**

**{**

**real = r;**

**imag = i;**

**}**

    // This is automatically called

    // when '+' is used with between

    // two Complex objects

**Complex operator+(Complex const& obj)**

**{**

**Complex res;**

**res.real = real + obj.real;**

**res.imag = imag + obj.imag;**

**return res;**

**}**

    void print()

    {

      cout << real << " + i" <<

              imag << endl;

    }

};

int main()

{

**Complex c1(10, 5), c2(2, 4)**;

    // An example call to "operator+"

**Complex c3 = c1 + c2**;

    c3.print();

}

* + 1. **Run time polymorphism(dynamic polymorphism)(dynamic method dispatch)**
       1. Changing the values of function (base class) in the calling function(derived class).
       2. It can be achieved by function overriding and virtual function
       3. Late binding or dynamic binding is it’s another name
       4. Here, function call is resolved at runtime.
       5. ***Function overriding***
          1. Here, definition of member function in base class is overridden in derived class.
          2. Code:

class base {

public:

**virtual void print()**

    {

        cout << "print base class" <<

                 endl;

    }

**void show()**

    {

      cout << "show base class" <<

               endl;

    }

};

**class derived : public base** {

public:

    // print () is already virtual function in

    // derived class, we could also declared as

    // virtual void print () explicitly

**void print**()

    {

        cout << "print derived class" <<

                 endl;

    }

**void show**()

    {

      cout << "show derived class" <<

               endl;

    }

};

// Driver code

int main()

{

**base\* bptr;**

**derived d;**

**bptr = &d;**

**// Virtual function, binded at**

**// runtime (Runtime polymorphism)**

**bptr->print();**

    // Non-virtual function, binded

    // at compile time

**bptr->show();**

    return 0;

}

* + - 1. ***Virtual function***
         1. Details in virtual keyword discussed above.
         2. **Code**: Function overriding is an example

**Upcasting**

* + - 1. When reference variable of parent class calls the object of child class.
      2. Code:

class A{}

class B extends A{}

**A a=new B();//**upcasting

**Inheritance vs Polymorphism**

| **S.NO** | **Inheritance** | **Polymorphism** |
| --- | --- | --- |
| 1. | Inheritance is one in which a new class is created (derived class) that inherits the features from the already existing class(Base class). | Whereas polymorphism is that which can be defined in multiple forms. |
| 2. | It is basically applied to classes. | Whereas it is basically applied to functions or methods. |
| 3. | Inheritance supports the concept of reusability and reduces code length in object-oriented programming. | Polymorphism allows the object to decide which form of the function to implement at compile-time (overloading) as well as run-time (overriding). |
| 4. | Inheritance can be single, hybrid, multiple, hierarchical and multilevel inheritance. | Whereas it can be compiled-time polymorphism (overload) as well as run-time polymorphism (overriding). |
| 5. | It is used in pattern designing. | While it is also used in pattern designing. |
| 6. | **Example :**  The class bike can be inherit from the class of two-wheel vehicles, which is turn could be a subclass of vehicles. | **Example :**  The class bike can have method name set\_color(), which changes the bike’s color based on the name of color you have entered. |

1. **Abstraction**
   1. Displaying only essential part and hiding details
   2. Implementation hiding.
   3. Showing only required part.
   4. Mostly done by interfaces rather than abstract classes
   5. **Real-life Example:**
      1. A man driving a bike, he knows that accelerator increases the speed but doesn’t knows how it happens
   6. **Types of abstraction:**
      1. **Data abstraction**
         1. Definition of abstraction
      2. **Control abstraction**
         1. Displaying only essential part of implementation and hiding unnecessary information
   7. **Implementation of Abstraction**
      1. **C++**
         1. **Abstraction using Class**
            1. We can implement abstraction using class
            2. Here, we can group data
            3. Using access modifier we can decide what to show outside world and what not to show
         2. **Abstraction using Header**
            1. For example, we use pow(), for calculating power
            2. We use pow(), using header Math.h but we don’t know how it is implemented inside Math.h
         3. **Abstraction using Access Modifier**
            1. With Public and private features of access modifier we can decide what to show and what not to show
            2. It is the most imp. Method to do abstraction
      2. **Java**
         1. **Abstract class**
            1. Using keyword, abstract
            2. Can contain abstract or concrete method
            3. Should be inherited from the base class
            4. If there is no implementation of method in class, use abstract class
            5. It is public, static and final by default
            6. Code:

abstract class Car

{

//abstract method

abstract void start();

//non-abstract method

public void stop()

{

System.out.println("The car engine is not started.");

}

}

//inherit abstract class

public class Owner extends Car

{

//defining the body of the abstract method of the abstract class

void start()

{

System.out.println("The car engine has been started.");

}

public static void main(String[] args)

{

Owner obj = new Owner();

//calling abstract method

obj.start();

//calling non-abstract method

obj.stop();

}

* + - * 1. **Output**

The car engine has been started.

The car engine is not started.

* + - 1. **Interface**
         1. Only contain empty method and variable
         2. Collection of empty methods and static variable
         3. Helps in achieving 100% abstraction
         4. Interfaces are public and abstract by default
         5. Doesn’t have a constructor
         6. Helps in achieving multiple inheritance
         7. **Features**

Achieve complete abstraction

Using multiple interface leads to achieve multiple inheritance

Helps in loosing **coupling**

**Coupling** means, modification in class leads to modification in another class

* + - * 1. To use an interface java provides a keyword implements
        2. Code:
* interface CarStart
* {

void start();

* }
* interface CarStop
* {

void stop();

* }
* public class Car implements CarStart, CarStop
* {

public void start()

{

System.out.println("The car engine has been started.");

}

public void stop()

{

System.out.println("The car engine has been stopped.");

}

public static void main(String args[])

{

Car c = new Car();

c.start();

c.stop();

}

* }
  + - * 1. Output:

The car engine has been started.

The car engine has been stopped.

**Abstract class vs Interface**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

* 1. **Advantages:**
     1. Avoids writing low language code(code without abstraction)
     2. Avoids code duplication and increases reusability
     3. Increases security
     4. Reducing complexity as well as reusability of the code
        1. Increases readability

**Abstraction vs Encapsulation**

| **Abstraction** | **Encapsulation** |
| --- | --- |
| Abstraction is the process or method of gaining the information. | While encapsulation is the process or method to contain the information. |
| In abstraction, problems are solved at the design or interface level. | While in encapsulation, problems are solved at the implementation level. |
| Abstraction is the method of hiding the unwanted information. | Whereas encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside. |
| We can implement abstraction using abstract class and interfaces. | Whereas encapsulation can be implemented using by access modifier i.e. private, protected and public. |
| In abstraction, implementation complexities are hidden using abstract classes and interfaces. | While in encapsulation, the data is hidden using methods of getters and setters. |
| The objects that help to perform abstraction are encapsulated. | Whereas the objects that result in encapsulation need not be abstracted. |
| Abstraction provides access to specific part of data. | Encapsulation hides data and the user can not access same directly (data hiding. |
| Abstraction focus is on “what” should be done. | Encapsulation focus is on “How” it should be done. |

**#Concepts & Interview Questions**

**Some important terms**

1. **Static and Dynamic binding**
   * Object have a type:
     + **Code:**
       - class Animal{}
         * class Dog extends Animal{
         * public static void main(String args[]){
         * Dog d1=new Dog();
       - }
       - }
     + Here, d1 is an instance of Dog but also an instance of Animal
   * **Static binding**
     + When compiler determines the type of object in compiler time its static binding
     + If there is private, public or static method in class, its static binding
     + **Code:**
       - class Dog{
       - private void eat(){System.out.println("dog is eating...");}
       - public static void main(String args[]){
       - Dog d1=new Dog();
       - d1.eat();
       - }
       - }
   * **Dynamic binding**
     + When type of object is determined in the run-time, it’s a dynamic binding
     + **Code:**
       - class Animal{
       - void eat(){System.out.println("animal is eating...");}
       - }
       - class Dog extends Animal{
       - void eat(){System.out.println("dog is eating...");}
       - public static void main(String args[]){
       - Animal a=new Dog();
       - a.eat();
       - }
       - }
     + Output:
       - Dog is eating
     + Here, compiler can’t determine a is instance of animal but also an instance of Dog, so compiler doesn’t know its type only base type
2. **Exception handling**
   * There are 2 types of exception
     + Asynchronous
     + Synchronous
   * While executing c++ code, different errors can occur due to wrong code, wrong input or any other unforeseeable errors.
   * Then, c++ generally generates a message(c++ will throw an error/exception)
   * It consist of try-catch pair and throw
   * **Try:**
     + Represent block that may can throw an exception
   * **Catch:**
     + Represent a block that is executed after some exception is thrown
   * **Throw**
     + Used to throw an exception
     + Can be used to list exception that function throws but it can’t handle it itself
     + It can create an exception too
   * **Why use exception handling?**
     + Separation of error handling from normal code
     + Functions can choose to handle any exception they choose
     + Grouping of exception
   * **Code:**
     + int main()
     + {
     + int x = -1;
     + // Some code
     + cout << "Before try \n";
     + **try** {
     + cout << "Inside try \n";
     + if (x < 0)
     + {
     + **throw x;**
     + cout << "After throw (Never executed) \n";
     + }
     + }
     + **catch (int x )** {
     + cout << "Exception Caught \n";
     + }
     + cout << "After catch (Will be executed) \n";
     + return 0;
     + }
   * **Output**
     + Before try
     + Inside try
     + Exception Caught
     + After catch (Will be executed)
   * Special type of catch(…)
     + If there is no catch for try, then catch(…) gets executed
     + **Code:**
       - #include <iostream>
       - using namespace std;
       - int main()
       - {
       - try {
       - throw 10;
       - }
       - catch (char \*excp) {
       - cout << "Caught " << excp;
       - }
       - catch (...) {
       - cout << "Default Exception\n";
       - }
       - return 0;
       - }
     + **Output:**
       - Default Exception
   * If there is no catch of any type after try block, then program gets terminated.
   * All exceptions are stored in ***Standard Exception Class***
   * Unlike java, c++ doesn’t check whether an exception is thrown or not, but it’s a good practice to specify uncaught exceptions
   * **Java finally block**
     + It is used to close the connection, etc
     + It runs even if the there is no error/exception in the program
     + **Code:**

 class TestFinallyBlock {

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

   try{

 //below code do not throw any exception

    int data=25/5;

    System.out.println(data);

   }

 //catch won't be executed

   catch(NullPointerException e){

 System.out.println(e);

 }

 //executed regardless of exception occurred or not

  finally {

 System.out.println("finally block is always executed");

 }

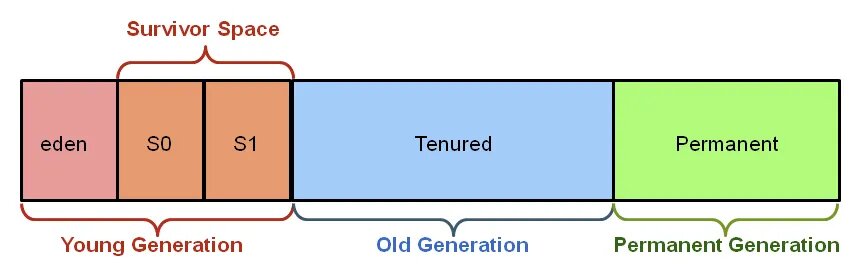


 System.out.println("rest of phe code...");

   }

 }

1. **Garbage Collection**
   * It is a process in Java by which java performs automatic memory management.
   * Java programs are compiled into bytecode that can be run in Java Virtual Machine (JVM).
   * When program is running in JVM, objects are created in heap where some memory is dedicated to them.
   * But eventually some programs will no longer in use and not needed.
   * So, the garbage collector finds that unused programs and destroys/removes them.
   * **In C/C++**
     + We have to destroy the objects using ~destructor()
     + But if we don’t do this
     + Eventually we’ll not be able to create any objects
     + Results in OutOfMemoryErrors
     + But, in java programmer doesn’t have to worry about destroying unused objects explicitly, garbage collector does this work
   * Garbage collection is an automatic process
   * Its main objective is to free memory in heap
   * It looks for objects in the program that is not in-use
     + In-use objects maintains a reference to the program where it is pointed
     + But memory used by objects not in use are reclaimed
   * And, this removal of object is done by garbage collector and not by programmer
   * **Types of activities by java garbage collection**
     + **Minor or incremental garbage collection**
       - **It occurs when unreachable objects are removed from the young generation heap**
     + **Major or full garbage collection**
       - **It occurs when objects survive minor garbage collection and copied to old/permanent generation heap are removed**
     + **NOTE:**

****

* + - * **Young generation:**
        + **Its high level, where all new objects start out**
      * **Old generation:**
        + **Here, long lived objects lie**

1. **Association**
   * Relationship between objects
   * Multiplicity between objects
     + Like, one to one, one to many, many to one, many to many
   * Example:
     + A student and faculty
2. **Aggregation**
   * Special case of association
   * Directional association between object
   * Direction points out which object contains which object
   * It’s a “Has-A” relationship
   * Example:
     + A class contains student
3. **Composition**
   * Special type of aggregation
   * In specific manner, composition is a restricted aggregation
   * Here, a contained class can’t remain without container class
   * Example:
     + There is a student class inside room class
     + And, student class can’t exist without room class
4. **Difference between Aggregation and Composition**
   * **Example**
     + A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition.
5. **Message Passing**
   * **What is message?**
     + It is a form of request sent by a object to perform some function specific action to another object
     + Communication between objects
     + Message is an abstract entity that doesn’t hold any information of the function, it just invokes the function
   * **Working:**
     + Creating class and objects
     + Message is passed to an object that invokes the function of another object
     + Communication between objects happens with the help of messages
   * **Difference in method call and message passing?**
     + Message passing is an abstract version of the term method call
     + Technically, there is no difference
   * **Code:**
     + class A
     + {
     + public void Methodname(Object obj)
     + {
     + // Method does something which you assigned to do
     + }
     + }
     + class B
     + {
     + Object obj1 = new Object();
     + A a = new A();
     + a.Method1(obj1);
     + }
     + Parameter (obj1) in method1 is said to be message passing
   * **Message Design Pattern**
     + The message designing pattern is used to create a middle layer to provide abstraction while transferring message between sender and receiver
     + This makes the system/working seem easier as it hides the logic from end user.
     + **Strategy Pattern**
       - This allows the interchangeability of algorithm during run-time.
       - This is possible by encapsulating the family of algorithm and making them interchangeable.
6. **Finalize()**
   * A java function
   * Method of object class
   * Called before an object is garbage collected
   * Overrides to dispose system resource, perform clean-up activities and minimize memory leaks.
   * Syntax:
     + protected void finalize() throws Throwable
     + here, Throwable raises an exception
   * Code:
     + public class JavafinalizeExample1 {
     + public static void main(String[] args)
     + {
     + JavafinalizeExample1 obj = new JavafinalizeExample1();
     + System.out.println(obj.hashCode());
     + obj = null;
     + // calling garbage collector
     + System.gc();
     + System.out.println("end of garbage collection");
     + }
     + @Override
     + protected void finalize()
     + {
     + System.out.println("finalize method called");
     + }
     + }
   * **Output:**

2018699554

end of garbage collection

finalize method called

1. **Enums**
   * Enumerations
   * User-defined data type that can be assigned with limited values
   * Starts with 0 same as arrays
   * It is a data type and its values cannot be defined as int, float, etc
   * Enumeration types of values known as enumerator.
   * Creation:
     + enumerated-type-name variable-name = value;
       - Default value is 0 but it can be explicitly changed.
     + *enum enumerated-type-name{value1=1, value2, value3};*
   * **Code:**
     + // C++ Program to Demonstrate the Functioning of Enumerators
     + // with an example of Gender
     + #include <bits/stdc++.h>
     + using namespace std;
     + int main()
     + {
     + // Defining enum Gender
     + enum Gender { Male, Female };
     + // Creating Gender type variable
     + Gender gender = Male;
     + switch (gender) {
     + case Male:
     + cout << "Gender is Male";
     + break;
     + case Female:
     + cout << "Gender is Female";
     + break;
     + default:
     + cout << "Value can be Male or Female";
     + }
     + return 0;
     + }
     + **Output:**
       - Gender is Male
   * **Code:**
     + enum year {
     + Jan,
     + Feb,
     + Mar,
     + Apr,
     + May,
     + Jun,
     + Jul,
     + Aug,
     + Sep,
     + Oct,
     + Nov,
     + Dec
     + };
     + // Driver Code
     + int main()
     + {
     + int i;
     + // Traversing the year enum
     + for (i = Jan; i <= Dec; i++)
     + cout << i << " ";
     + return 0;
     + }
     + **Output:**
       - 11
2. **Tokens**
   * Smallest element of the program that is meaningful
     + **They are:**
       - Operators
         * Unary(=+, =-, ++, --, sizeof(), &, \*, ..)
         * Binary(==, !=, &&, ||, ..)
         * Ternary(x == y ? z : x)

condition ? if-true : else-false

* + - * Keywords (31 in c and 63 in c++)
      * Identifiers
      * String
      * Special symbols
      * constants

1. **Manipulators**
   * It is a function that is used to modify I/O stream, but not changing the value of variables.
   * It is done using insertion(<<) and extraction(>>)
   * Used to manipulate data display
   * To access manipulators we’ve to use ***iomanip.h***
   * **Types of manipulators**
     + **Manipulators without argument**
       - Endl
         * Ostream, used to entering a new line after a line.
         * Also flushes the output stream
       - We
         * Istream, used to remove white spaces from the string
       - Ends
         * Ostream, used to end a line with null
       - Flush
         * Ostream, it is used to flush the stream, i.e. forcing all the output written on the screen or in the file. Without o/p output will be the same but may not appear in real time.
       - **Code:**
         * int main()
         * {
         * istringstream str("           Programmer");
         * string line;
         * // Ignore all the whitespace in string
         * // str before the first word.
         * getline(str >> std::ws, line);
         * // you can also write str>>ws
         * // After printing the output it will automatically
         * // write a new line in the output stream.
         * cout << line << endl;
         * // without flush, the output will be the same.
         * cout << "only a test" << flush;
         * // Use of ends Manipulator
         * cout << "\na";
         * // NULL character will be added in the Output
         * cout << "b" << ends;
         * cout << "c" << endl;
         * return 0;
         * }
       - **Output:**
         * Programmer
         * only a test
         * abc
     + **Manipulators with parameter**
       - Setprecision()
         * Controls the no. of digits to show after decimal
       - SetFill(), etc

**Questions**

1. **Is array primitive type or object in Java?**
   1. While initializing array we use ***new*** operator, and we know initializing any variable with ***new*** operator is object. hence java in an object
2. **How many instances can be created of abstract class?**
   1. Zero!
      1. We cannot create instance of an abstract coz it doesn’t have an implementation
      2. An abstract class is a template or empty structure.
3. **Error vs Exception**
   1. Both comes under java.lang.Throwable
   2. **Error**
      1. It is generally described in java.lang.Error package
      2. It represents trouble which cannot be handled and usually refers as catastrophic failures e.g. running out of memory
      3. It’s not possible to recover from error
      4. They are unchecked
   3. **Exception**
      1. It is generally described in java.lang.Exception package
      2. It indicates condition which a reasonable condition might want to catch
      3. Its possible to recover
      4. They are both checked and unchecked
         1. **Checked exceptions**
            1. Checked by compilers at compile time
            2. Can be handled by throws keyword
            3. E.g. IOException, InterruptedException
            4. **Code:**

// Java Program to Illustrate Checked Exceptions

// Where FileNotFoundException occurred

// Importing I/O classes

import java.io.\*;

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

{

// Reading file from path in local directory

FileReader file = new FileReader("C:\\test\\a.txt");

// Creating object as one of ways of taking input

BufferedReader fileInput = new BufferedReader(file);

// Printing first 3 lines of file "C:\test\a.txt"

for (int counter = 0; counter < 3; counter++)

System.out.println(fileInput.readLine());

// Closing file connections

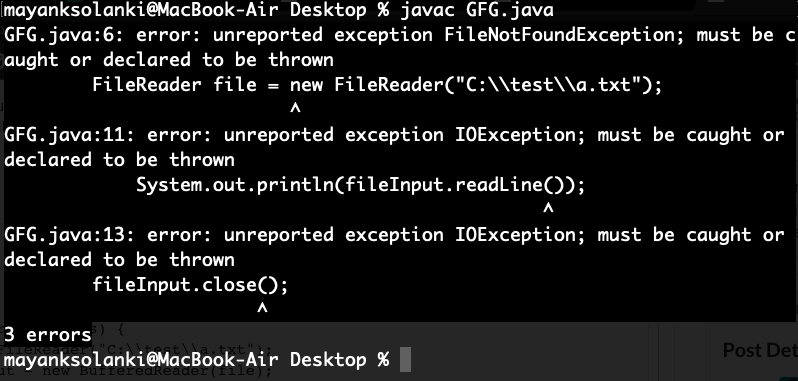
// using close() method

fileInput.close();

}

}

**Output:**



* + - * 1. After using try, catch, throws

// Java Program to Illustrate Checked Exceptions

// Where FileNotFoundException does not occur

// Importing I/O classes

import java.io.\*;

// Main class

class GFG {

// Main driver method

public static void main(String[] args)

throws IOException

{

// Creating a file and reading from local repository

FileReader file = new FileReader("C:\\test\\a.txt");

// Reading content inside a file

BufferedReader fileInput = new BufferedReader(file);

// Printing first 3 lines of file "C:\test\a.txt"

for (int counter = 0; counter < 3; counter++)

System.out.println(fileInput.readLine());

// Closing all file connections

// using close() method

// Good practice to avoid any memory leakage

fileInput.close();

}

}

**Output:**

First three lines of file "C:\test\a.txt"

* + - 1. **Unchecked exceptions**
         1. Are not checked by compilers
         2. In c++, all the exceptions are unchecked. Here, programmers have to be civilized to check it.
         3. In java, exceptions under error and RuntimeException classes comed under unchecked error

E.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException

* + - * 1. Code:

// Java Program to Illustrate Un-checked Exceptions

// Main class

class GFG {

// Main driver method

public static void main(String args[])

{

// Here we are dividing by 0

// which will not be caught at compile time

// as there is no mistake but caught at runtime

// because it is mathematically incorrect

int x = 0;

int y = 10;

int z = y / x;

}

}

* + - * 1. **Output:**

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Main.main(Main.java:5)

Java Result: 1

1. **Is it possible for a class to inherit constructor of its base class?**
   1. It’s all or none, you can inherit a class and you’ll get all the constructors.
   2. If you want to inherit only one constructor make a class having one constructor and one destructor and inherit it.
2. **Cohesion vs Coupling**
   * **Cohesion** refers to what the class (or module) can do.
     + **Low cohesion**
       - It would mean that the class does a great variety of actions - it is broad, unfocused on what it should do.
     + **High cohesion**
       - It means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.
     + Example of Low Cohesion:
       - * -------------------
         * | Staff |
         * -------------------
         * | checkEmail() |
         * | sendEmail() |
         * | emailValidate() |
         * | PrintLetter() |
         * -------------------
         * Example of High Cohesion:
         * ----------------------------
         * | Staff |
         * ----------------------------
         * | -salary |
         * | -emailAddr |
         * ----------------------------
         * | setSalary(newSalary) |
         * | getSalary() |
         * | setEmailAddr(newEmail) |
         * | getEmailAddr() |
         * ----------------------------
   * **Coupling**
     + It refers to how related or dependent two classes/modules are toward each other.
     + **low coupled classes**
       - changing something major in one class should not affect the other.
     + **High coupling classes**
       - It would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.
   * P.S. Good software design has **high cohesion** and **low coupling**.