OOPS CONCEPT

OBJECT ORIENTED PROGRAMMING

is a methodology to design a program using classes and objects. It simplifies the software development and maintenance.

Class

Class is like a blueprint for an object.
It is user defined data type which holds its own data member and member functions.which can be used by creating an instance of that class.

example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions

Object

It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions.

//In C++ by Default Access Modifiers is Private

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;
```

Creating class in c++

```
class Teacher{

    public: //Access Modifiers, Data Members
    //properties//attributes
    string name;
    string dept;
    string subject;
    double salary;

    //methods //member functions
    void changeDept(string newDept) {
        dept=newDept;
    }
```

```
};

int main() {
    // Write C++ code here
    //creating object in c++
    Teacher t1;
    t1.name="Santosh";
    t1.subject="c++";
    t1.dept="Computer Science";
    t1.salary=25000;
    cout<<t1.name<<endl;

    return 0;
}
```

C++ Syntax (for object):

```
student s = new student();
```

Note : When an object is created using a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created without a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.

```
//Private member access using getter and setter
```

Access Specifiers IMP : The access specifiers are used to define how functions and variables can be accessed outside the class. There are three types of access specifiers:

1. Private: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.

2. Public:Functions and variables declared under public can be accessed from anywhere.

3. Protected:Functions and variable declared protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

//Creating class in c++
class Teacher{
    private: double salary;
    public: //Access Modifiers
    //properties//attributes
    string name;
    string dept;
    string subject;


    //methods //member functions
    void changeDept(string newDept) {
        dept=newDept;
    }

    // setter set private value
    //indirectly access
    void setsalary(double s) {
        salary=s;
    }
    //getter get value
    double getsalary() {
        return salary;
    }

};

int main() {
    // Write C++ code here
    //creating object in c++
    Teacher t1;
    t1.name="Santosh";
    t1.subject="c++";
    t1.dept="Computer Science";
```

```
    t1.setsalary(25000);
    cout<<t1.name<<endl;
    cout<<t1.getsalary()<<endl;

    return 0;
}
```

Output :
Santosh
25000

ENCAPSULATION

Data properties and Data Member Functions
Data Hiding

Encapsulation is the process of combining data and functions into a single unit called class
In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. (Data hiding:

Example

```
Class Teacher {
class Account {
        private:
        double balance;
        string password; //Data Hiding using encapsulation
        public:
        string accountId;
        string username;

    }
};
```

Constructor : Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure.

```
//Creating Constructor
```

Same name as Class Name

Not return type
Only called once at object creation
Memory allocation

There can be two types of constructors in C++.

1. Default constructor : A constructor which has no argument is known as default constructor. It is invoked at the time of creating an object.

2. Parameterized constructor : Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.

3. Copy Constructor : A Copy constructor is an overloaded constructor used to declare and initialize an object from another object. It is of two types - default copy constructor and user defined copy constructor.

```
Class Teacher {
public:
   Teacher() {
      cout<<"Hi I am Constructor\n";
   }
}

int main() {
   // Write C++ code here
   //creating object in c++
   Teacher t1;
   t1.name="Santosh";
   t1.setsalary(25000);
   cout<<t1.name<<endl;
   cout<<t1.getsalary()<<endl;

   return 0;
}
```

Output:
Hi I am Constructor
Santosh

25000

Non parameterised Constructor

```
Teacher() {
        cout<<"Hi I am Constructor\n";
        dept="Computer Science";
    }

Teacher t1;
```

parameterised Constructor

```
//Constructor Overloading
Different Parameter
Polymorphism

Teacher(string n,string d,string s,double sal) {
        name=n;
        dept=d;
        subject=s;
        salary=sal;
    }

void getInfo() {
        cout<<"name: "<<name<<endl;
        cout<<"subject:"<<subject<<endl;
    }

int main() {
    // Write C++ code here
    //creating object in c++

    Teacher t2("Santosh","cse","C++",25000);
    t2.getInfo();

    return 0;
}
```

Output:
name: Santosh
subject:C++

Copy Constructor

```
//Copy Properties of one object into Another

#include<bits/stdc++.h>
#include <iostream>
using namespace std;

//Creating class in c++
class Teacher{
    private: double salary;

    //constructor
    public:
  /*  Teacher() {
        cout<<"Hi I am Constructor\n";
        dept="Computer Science";
    } */
    //Parameterized Constructor

    Teacher(string n,string d,string s,double sal) {
        name=n;
        dept=d;
        subject=s;
        salary=sal;
    }



    public: //Access Modifiers
    //properties//attributes
    string name;
    string dept;
    string subject;

    //methods //member functions
    void changeDept(string newDept) {
        dept=newDept;
    }

    void getInfo() {
        cout<<"name: "<<name<<endl;
        cout<<"subject:"<<subject<<endl;
```

```
    }



};

int main() {
    // Write C++ code here
    //creating object in c++

    Teacher t2("Santosh","cse","C++",25000);
    t2.getInfo();

    Teacher t(t2);
    t.getInfo();

    return 0;
}
```

This keyword

//point to Current Object

//Object property Access

```
/*this->prop same as *(this).prop

Teacher(string name,string dept,string subject,double salary) {
        this->name=n;
        this->dept=d;
        this->subject=s;
        this->salary=sal;
    }*/
```

Output

```
name: Santosh
subject:C++
name: Santosh
subject:C++
 Teacher ( Teacher &obj) { //pass by reference
    cout<<"I am Custom Copy Constructor\n";
        this->name=obj.name;
        this->dept=obj.dept;
        this->subject=obj.subject;
```

```
        this->salary=obj.salary;

    }

 Teacher t2("Santosh","cse","C++",25000);
    t2.getInfo();

    Teacher t(t2);
    t.getInfo();
```

Output:
name: Santosh
subject:C++
I am Custom Copy Constructor
name: Santosh
subject:C++

Shallow Copy
Copy All Member Values from One Object to Another

//Issue When Dynamic Memory Allocation
Int x=5;
Static Memory Allocation Store Value in Stack
New int[5];
Dynamic memory allocation store value in Heap

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

//Creating class in c++
class Student {
    public:
    string name;
    double* cgpaptr; // pointer of cgpa

    Student(string name,double cgpa) {
        this->name=name;
        cgpaptr=new double;
        *cgpaptr=cgpa;
```

```
    }

    Student(Student &obj) {
        this->name=obj.name;
        this->cgpaptr=obj.cgpaptr;
    }

    void getInfo() {
        cout<<"name: "<<name<<endl;
        cout<<"cgpa: "<<*cgpaptr<<endl;

    }

};

int main() {
    // Write C++ code here
    //creating object in c++
    Student s1("Santosh Kumar",9.08);
    s1.getInfo();

    Student s2(s1);
    *(s2.cgpaptr)=9.2;
    s1.getInfo();


    return 0;
}
```

```
name: Santosh Kumar
cgpa: 9.08
name: Santosh Kumar
cgpa: 9.2
```

Deep Copy

Copies the member values but also makes copies of any dynamically allocated memory that the members point to

Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically. A destructor is defined like a constructor. It must have the same name as class, prefixed with a tilde sign (~).

Deallocate memory

```
 ~Student() {
      cout<<"Hi, I delete everything\n";
      //Dynamically allocated memory delete
      delete cgpaptr;
   }

Student s1("Santosh Kumar",9.08);
   s1.getInfo();
```

Output:
name: Santosh Kumar
cgpa: 9.08
Hi, I delete everything

INHERITANCE

Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can reuse, extend or modify the attributes and behaviors which are defined in other classes.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

It is Possible to inherit attributes and methods from one class to another.

Code Reusability

Example

```
#include<bits/stdc++.h>
```

```
#include <iostream>
using namespace std;

class person {
    public:
    string name;
    int age;

   /* person(string name,int age) {
        this->name=name;
        this->age=age;
    }*/

    person() {

    }
};

class stu : public person {
    public:
    int rollno;

    void getInfo() {
        cout<<"name: "<<name<<endl;
        cout<<"age: "<<age<<endl;
        cout<<"rollno: "<<rollno<<endl;
    }

};

int main() {
    // Write C++ code here
    stu s1;
    s1.name="Santosh";
    s1.age=24;
    s1.rollno=1234;

    s1.getInfo();


    return 0;
}
```

Output ;

name: Santosh
age: 24
rollno: 1234

//In Constructor First Parent class called then child class called

```
class person {
   public:
   string name;
   int age;


   person() {
      cout<<"parent class"<<endl;
       } };

class stu : public person {
   public:
   int rollno;
   stu() {
      cout<<"child class"<<endl;
   }
```

Output:
parent class
child class
name: Santosh
age: 24
rollno: 1234

// Destructor

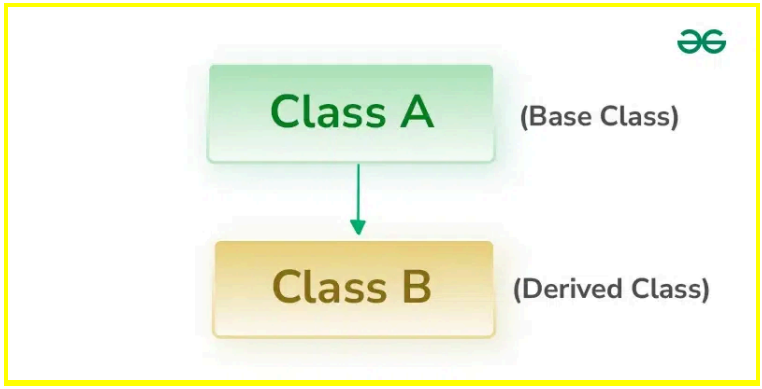
```
~person() {
      cout<<"parent class"<<endl;

   }

~stu() {
      cout<<"child class"<<endl;
   }
```

name: Santosh
age: 24
rollno: 1234
child class
parent class

Types of Inheritance :

1. Single inheritance : When one class inherits another class, it is known as single level inheritance

Person (Parent)
↑
Stu (Child)




```
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// sub class derived from a single base classes
class Car : public Vehicle {
```

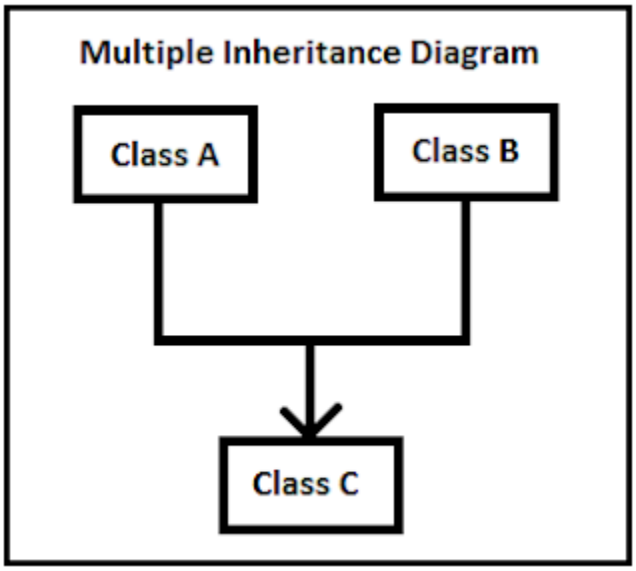
```
public:
    Car() { cout << "This Vehicle is Car\n"; }
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output
This is a Vehicle
This Vehicle is Car

```
//Parameterized
class person {
    public:
    string name;
    int age;
  person(string name,int age) {
        this->name=name;
        this->age=age;
    }
    /*person() {
        cout<<"parent class"<<endl;

    }*/
};
```

```
class stu : public person {
    public:
    int rollno;
    stu(string name ,int age,int rollno) : person(name,age) {
        this->rollno=rollno;
       // cout<<"child class"<<endl;
    }
```

2. Multiple inheritance : **Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.**

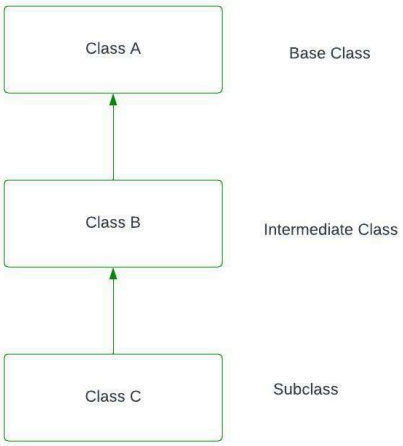



```
#include<iostream>
using namespace std;

class A
{
public:
A() { cout << "A's constructor called" << endl; }
};

class B
{
public:
B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A // Note the order
{
public:
C() { cout << "C's constructor called" << endl; }
};

int main()
{
```

```
        C c;
        return 0;
}
```

Output :
B's constructor called
A's constructor called
C's constructor called

3. Multilevel inheritance : Multilevel inheritance is a process of deriving a class from another derived class.




```
using namespace std;

// single base class
class A {
public:
    int a;
    void get_A_data()
    {
        cout << "Enter value of a: ";
        cin >> a;
    }
};

// derived class from base class
class B : public A {
```

```
public:
    int b;
    void get_B_data()
    {
        cout << "Enter value of b: ";
        cin >> b;
    }
};

// derived from class derive1
class C : public B {
private:
    int c;

public:
    void get_C_data()
    {
        cout << "Enter value of c: ";
        cin >> c;
    }

    // function to print sum
    void sum()
    {
        int ans = a + b + c;
        cout << "sum: " << ans;
    }
};
int main()
{
    // object of sub class
    C obj;

    obj.get_A_data();
    obj.get_B_data();
    obj.get_C_data();
    obj.sum();
    return 0;
}
```

Output:

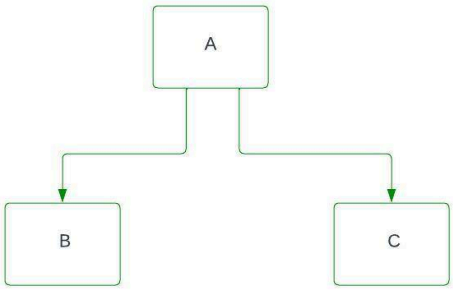
Enter value of a: 4

Enter value of b: 5

Enter value of c: 9

sum: 18

4. Hierarchical inheritance : Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

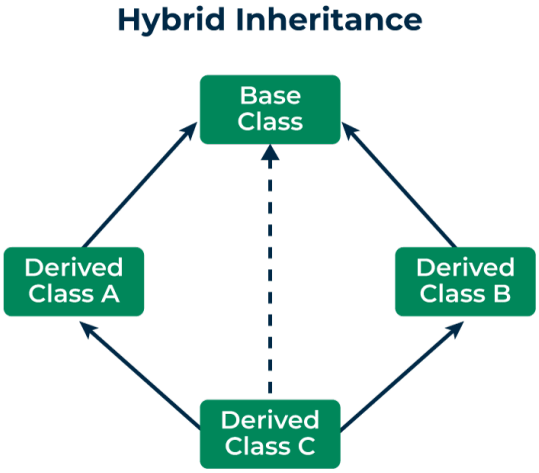



```
using namespace std;

class A   //superclass A
{
  public:
  void show_A() {
    cout<<"class A"<<endl;
  }
};
class B : public A   //subclass B
{
  public:
  void show_B() {
    cout<<"class B"<<endl;
  }
};

class C : public A   //subclass C
{
  public:
  void show_C() {
```

```
    cout<<"class C"<<endl;
  }
};

int main() {
  B b;  // b is object of class B
  cout<<"calling from B: "<<endl;
  b.show_B();
  b.show_A();

  C c;  // c is object of class C
  cout<<"calling from C: "<<endl;
  c.show_C();
  c.show_A();
  return 0;
}
```

Output:
calling from B:
class B
class A
calling from C:
class C
class A

5. Hybrid inheritance : Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.



```
using namespace std;

// Base class
class Person {
protected:
    string name;

public:
    Person(const string& name)
        : name(name)
    {
    }
    void display() { cout << "\nName: " << name << endl; }
};

// Derived class 1: Employee (Single Inheritance)
class Employee : public Person {
protected:
    int employeeId;

public:
    Employee(const string& name, int id)
        : Person(name)
        , employeeId(id)
    {
    }
    void displayEmployee()
    {
        display();
        cout << "Employee ID: " << employeeId << endl;
        cout << "Method inside Derived Class Employee"
             << endl;
    }
};

// Derived class 2: Student (Single Inheritance)
class Student : public Person {
protected:
    int studentId;

public:
    Student(const string& name, int id)
        : Person(name)
        , studentId(id)
```

```
    {
    }
    void displayStudent()
    {
        display();
        cout << "Student ID: " << studentId << endl;
        cout << "Method inside Derived Class Student"
             << endl;
    }
};

// Derived class 3: StudentIntern (Multiple Inheritance)
class StudentIntern : public Employee, public Student {
public:
    StudentIntern(const string& name, int empId, int stuId)
        : Employee(name, empId)
        , Student(name, stuId)
    {
    }
    void displayStudentIntern()
    {
        cout << "Methods inside Derived Class "
                "StudentIntern : "
             << endl;
        displayEmployee();
        displayStudent();
    }
};

// driver code
int main()
{
    StudentIntern SI("Riya", 67537, 2215);
    SI.displayStudentIntern();

    return 0;
}
```

Output
Methods inside Derived Class StudentIntern :

Name: Riya
Employee ID: 67537
Method inside Derived Class Employee

Name: Riya
Student ID: 2215
Method inside Derived Class Student

Modes of Inheritance in C++
There are 3 modes of inheritance:

Public Mode
Protected Mode
Private Mode

ABSTRACTION

In simple terms, it is hiding the unnecessary details & showing only the essential parts/functionalities to the user.

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

class shape { //abstract class
virtual void draw()=0;
};

class circle: public shape {
   public:
   void draw() {
      cout<<"drawing a circle";
   }
};

int main() {
   // Write C++ code here
   //std::cout << "Try programiz.pro";
   circle c1;
   c1.draw();

   return 0;
}
```

Output:
drawing a circle

Static Keyword

Static variables

It is created and initialised once for lifetime of the program in Function

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

void fun() {
    static int x=0;
    cout<<"x: "<<x<<endl;
    x++;
}


int main() {
    // Write C++ code here
    //std::cout << "Try programiz.pro";
    fun();
    fun();
    fun();
    return 0;
}
```

Output:
x: 0
x: 1
x: 2

```
class ABC {
    public:
    ABC() {
        cout<<"Constructor\n";
    }
    ~ABC() {
        cout<<"Destructor\n";
    }
};
```

```
int main() {
    // Write C++ code here
    if(true) {
        static ABC obj;
    }
    cout<<"end of main function\n";
    return 0;
}
```

Output:
Constructor
end of main function
Destructor

POLYMORPHISM

Poly means ‘many’ and morphism means ‘forms’.

Boy Father,Son
Carbon Graphite,Diamond,C02
Rahul Hobbies dance,singer,Coder

Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data.

A point shape needs only two coordinates (assuming it's in a two-dimensional space of course). A circle needs a center and radius. A square or rectangle needs two coordinates for the top left and bottom right corners and (possibly) a rotation. An irregular polygon needs a series of lines

Types of Polymorphism

Compile Time Polymorphism (Static):
Function Overloading and Operator Overloading.

The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Example - Method Overloading

Constructor Overloading

Example

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

class student {
   public:
   string name;
   student() {
      cout<<"non parameterized\n";
   }

   student(string name) {
      this->name=name;
      cout<<"Parameterized\n";
   }
};

int main() {
   // Write C++ code here
   //student s1;
   student s1("santosh");
   return 0;
}
```

Output:

```
Parameterized

student s1;
non parameterized
```

Function Overloading

Method Overloading : Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality.

Method overloading can be possible on the following basis:

1. The return type of the overloaded function.

2. The type of the parameters passed to the function.
3. The number of parameters passed to the function.

Compile time statically
Same name different parameter

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

class print {
   public:
   void show(int x) {
      cout<<"int: "<<x<<endl;
   }
   void show(char ch) {
      cout<<"char: "<<ch<<endl;
   }
};

int main() {
   // Write C++ code here
   print p1;
   p1.show(100);

   return 0;
}
```

Output
int: 100
char: S

Operator Overloading

Run Time Polymorphism

Function Overiding and Virtual Functions.

Runtime Polymorphism : Runtime polymorphism is also known as dynamic polymorphism. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class.

Hence,
the child class overrides the method of the parent class. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

Function Overiding
Inheritance

```
#include<bits/stdc++.h>
#include <iostream>
using namespace std;

class parent {
   public:
   void getInfo() {
      cout<<"Parent Class\n";
   }
};

class child: public parent {
   public:
   void getInfo() {
      cout<<"Child class\n";
   }
};

int main() {
   // Write C++ code here
   child c1;
   c1.getInfo();

   return 0;
}
```

Output:
Child class

Child class override parent class

Virtual Function

Dynamic in nature

Defined by keyword “virtual” inside a base class and always declared with a base class and overridden in a child class.

Virtual Function is called during runtime

Member Function

Friend Function : Friend function acts as a friend of the class. It can access the private and protected members of the class.
. A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
Friend function uses objects as arguments.

```
#include <bits/stdc++.h>
using namespace std;
class A{
inta=2;
intb=4;
public:
// friend function
friend int mul(A k){
return (k.a * k.b);
} };
}
// Output : 8
```

Aggregation : It is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents the HAS-A relationship.

```
int main(){
 A obj;
 int res = mul(obj);
 cout << res << endl;
 return 0;
```

● Delete is used to release a unit of memory, delete[] is used to release an array. ● Virtual inheritance facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.

● Function overloading: Function overloading is defined as we can have more than one version of the same function. The versions of a function will have different signatures meaning that they have a different set of parameters.

```
int main()
 int res = Add :: add(); // accessing the function inside namespace
 cout << res;
```

Operator overloading: Operator overloading is defined as the standard operator can be redefined so that it has a different meaning when applied to the instances of a class.

● Overloading is static Binding, whereas Overriding is dynamic Binding. Overloading is nothing but the same method with different arguments, and it may or may not return the same value in the same class itself. Overriding is the same method name with the same arguments and return types associated with the class and its child class.