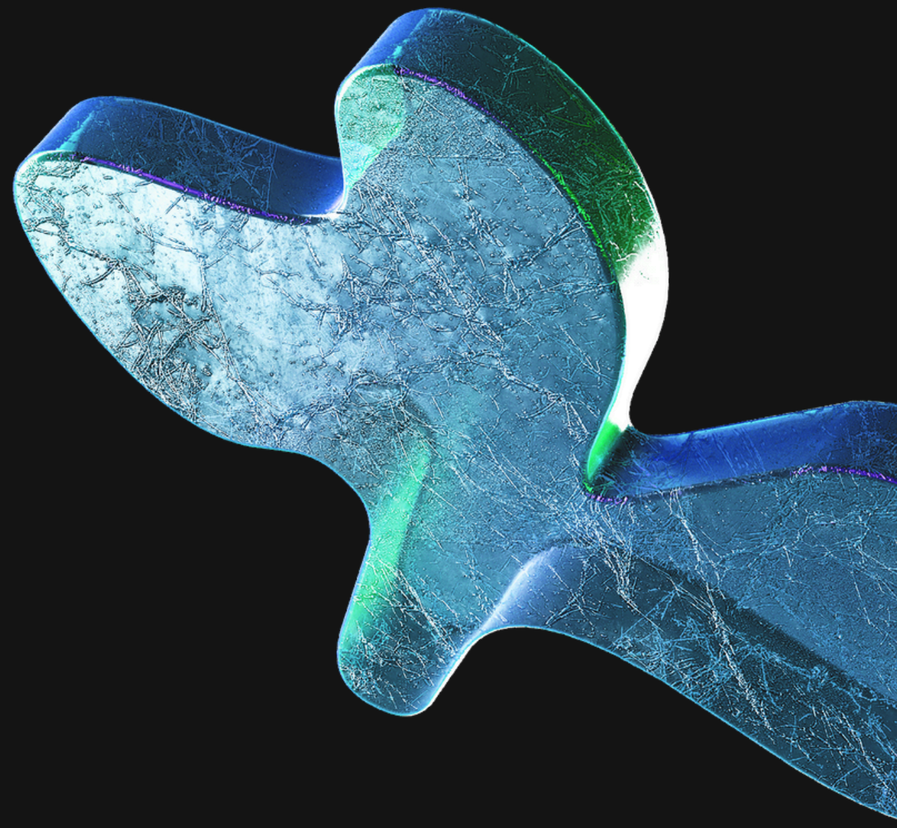
OOP Concept

PRINCIPLES



NGUYENKELVIN

Encapsulation

- Bundling or tying data and its methods together that operate on that data so that they are grouped together within a class.
- Encapsulation has been done with the purpose of preventing anything or anyone outside of our class to be able to directly access our data and to interact with it and to modify it.
- It is used in hiding data.
- Setter and getter method are used in order to access the data while keeping the members private in class.

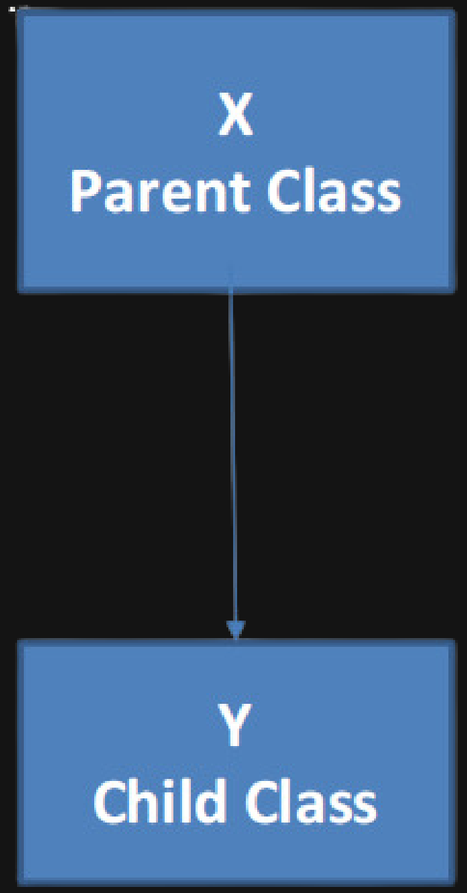
Abstraction

- Abstraction means hiding complexity while showing the functionality.
- Implementing a contract which is actually an abstract class. So function inside a abstract class is mandatory to implement in order to execute a contract.

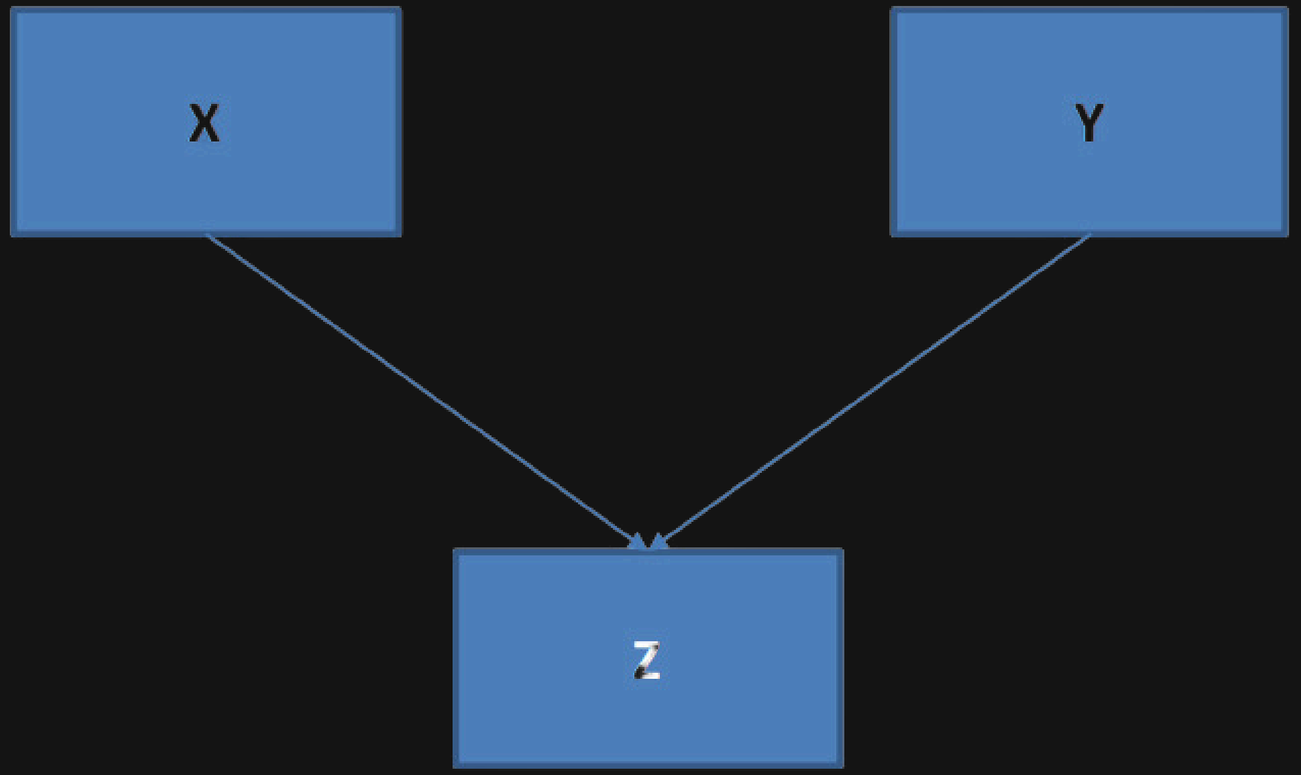
Inheritance

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

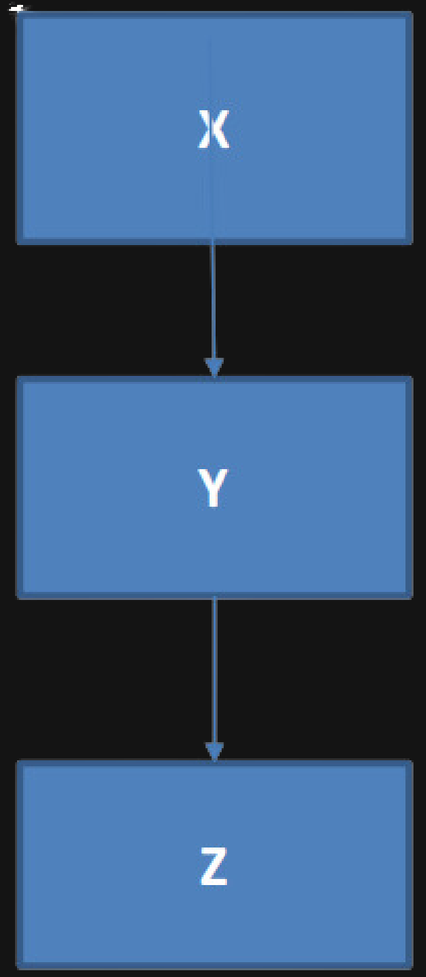
Single Inheritance



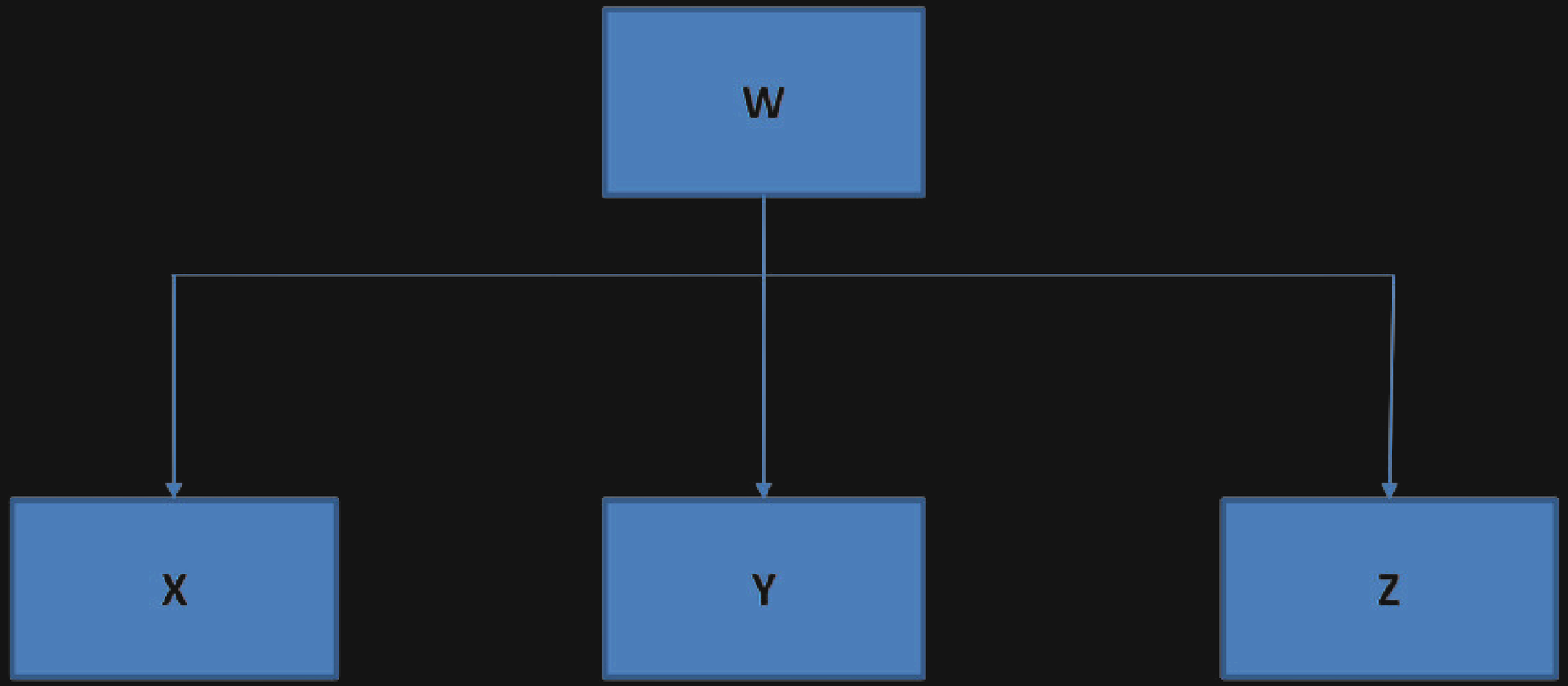
Multiple Inheritance



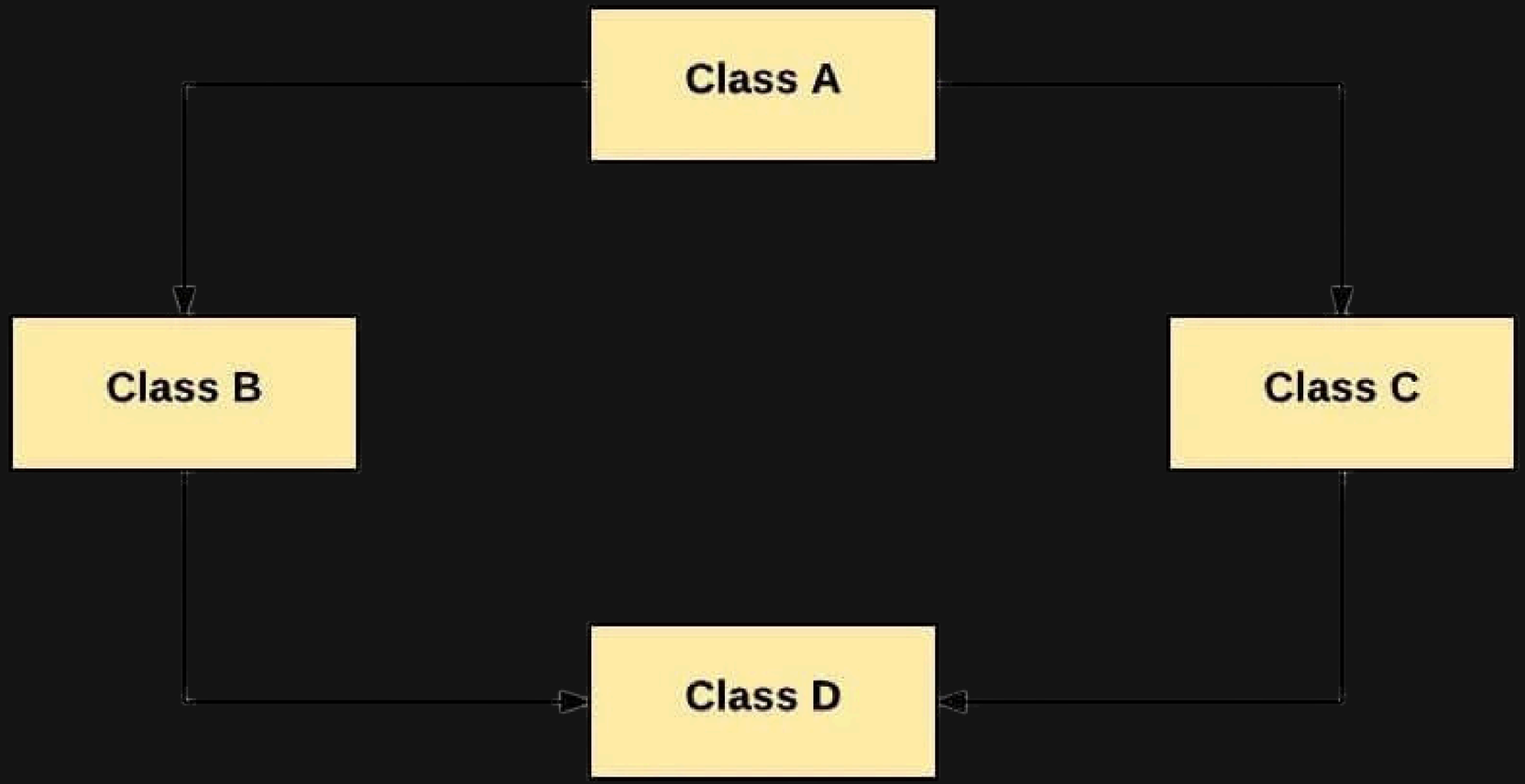
Multilevel Inheritance



Hierarchical Inheritance



Hybrid Inheritance



Polymorphism

- Polymorphism is an ability of an entity to have various forms.
- It permits to execute a single activity in distinct ways.
- It is of two types:
  - Compile time Polymorphism (Function Overloading, Operator Overloading)
  - Runtime Polymorphism (Overriding, Abstract Classes and Interfaces)

Polymorphism

- Function Overloading

```
Example ex;
ex.print(10);
ex.print(3.14);
ex.print("Hello");
```

```
class Example {
public:
    void print(int i) {
        std::cout << "Printing integer: " << i << std::endl;
    }

    void print(double d) {
        std::cout << "Printing double: " << d << std::endl;
    }

    void print(const std::string& str) {
        std::cout << "Printing string: " << str << std::endl;
    }
};
```

Polymorphism

- Operator Overloading

```
Complex a(2, 3);
Complex b(1, 4);
Complex c = a + b;
c.display();
```

```
class Complex {
private:
    double real, imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    Complex operator + (const Complex& c) const {
        return Complex(real + c.real, imag + c.imag);
    }

    void display() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};
```

Polymorphism

- Overriding

```
int main() {
    Base* basePtr = new Derived();
    basePtr->show();
    return 0;
}
```

```
class Base {
public:
    virtual void show() {
        std::cout << "Base class show()" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show()" << std::endl;
    }
};
```

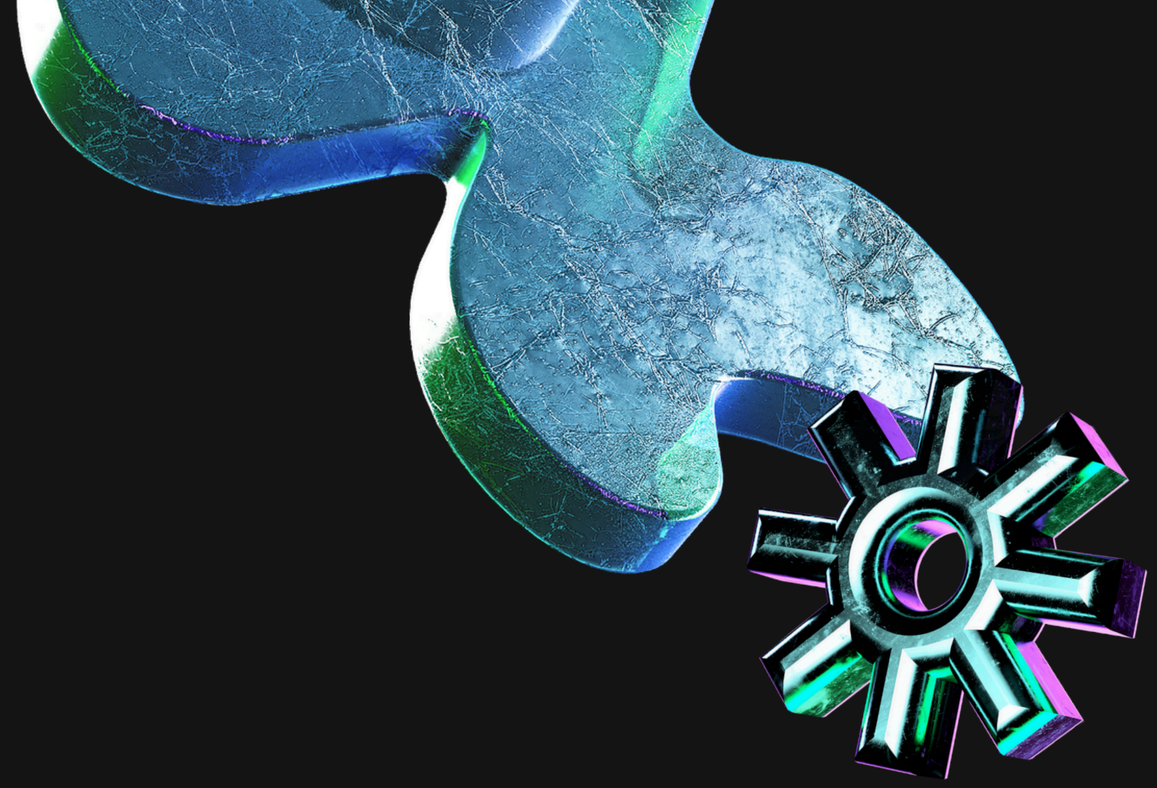
Polymorphism

- Abstract Classes and Interfaces

```
class AbstractBase {
public:
    virtual void pureVirtualFunction() = 0;
};

class Derived : public AbstractBase {
public:
   void pureVirtualFunction() override {
     std::cout << "Derived class implementation" << std::endl;
  }
};
```

```
int main() {
    Derived derivedObj;
    derivedObj.pureVirtualFunction();
    return 0;
}
```



THANK
FOR WATCHING

NGUYENKELVIN