Liskov Substitution Principle

Objects of the Super class can be should be replaceable with objects of the sub classes without altering the correctness of program.

In simple terms, it means that sub class should be able to used whenever superclass is expected, and it should not introduce the error or break the program.

Lets take an example to understand the above with a **Flying Bird Example.**

Suppose we have to design a bird hierarchy in which we need to implement birds like *Penguin*, *Crow*, etc.

```
Class Bird
{
   public:
       virtual void fly()=0;
};

class Crow : public Bird
{
  public:
     void fly() {
       cout<<”Crow is flying”<<endl;
};

class Penguin : public Bird
{
   public:
       void fly() {
        throw runtime_error(“Penguin can’t fly”);
      }
};
```

In the above example, we have a class called *Bird* which is a abstract class having a pure virtual function called *fly()*

We also have two subclasses *Crow* and *Penguin* that is inheriting Bird class. Since Penguins can’t fly the method *fly*() is throwing runtime_error.

Problem in above approach
Now let's say we have a function that takes a **Bird** object and calls its **fly**() function.

```
Void makeBirdFly(Bird& bird) {
   bird.fly();
}
```

If we call this method with *Crow* object, it will work as expected but if we call this method with *Penguin* object, it will throw a run time error. This **violates the Liskov Substitution Principle** because substituting an *Penguin* for a *Bird* results in unexpected behavior.

There are many disadvantage if we don't follow Liskov Substitution principle

- **It will be very difficult to maintain the hierarchies between the classes. Lot of run time issues and strange behavior might happen.**
- **Unit tests of the super class will never be passed for the sub class. This will make code difficult to test.**

Solution to above

To fix this violation we can use appropriate hierarchy.

We can have intermediate super class called **FlyingBird** that represents the birds that can fly.

```
Class Bird
{
  public:
    virtual bool canFly()=0;
};

class FlyingBird : public Bird
{
   public:
      bool canFly() {
        return true;
    }
     virtual void fly()=0;
};

class Crow : public FlyingBird
{
  public:
     void fly() {
       cout<<"Crow can fly"<<endl;
};
```

```
class Penguin : public Bird
{
   public:
      bool canFly() {
          return false;
      }
};

void makeBirdFly(FlyingBird& bird)
{
     bird.fly();
}
```

Now, if we call *makeBirdFly*() with *Crow* object, it will work as expected.

If we try to call it with *Penguin* object, it will give us Compile time error since *Penguin* does not inherit from *FlyingBird*.

This ensures that only a bird that can fly can be passed to *makeBirdFly*() function, which prevents from giving unexpected behavior, which maintains the Liskov Substitution Principle.