Lecture #2 SOLID PRINCIPLES

① S - Single Responsibility

A notification class should ~~have~~ can use email & sms classes' object instead of all the ~~tigt~~ code of email & sms sending.

② O - Open closed - Open for ~~or~~ extension - closed for notification

Have an abstract class as notifier with a virtual function of sending ~~as~~. Inherit and override this virtual function in email and sms child class.
Then in main program, create pointer of notifier and pass ~~the~~ it to the notification class. This way a lot of if else is avoided.

③ L - Liskov Substitution

↑

Similar

↓

Penguin cannot inherit from bird as it cannot fly, It should inherit from FlightlessBird which inherits from Bird.

④ I - Interface Segregation Principle

It is better to have small specific interfaces than a few monolithic ones that try to do everything.

⑤ D - Dependency Inversion

High level modules don't depend on low level modules, they rather depend on abstractions.

Notification class does not depend directly on the email or sms service. It depends on the abstract base class (notifier). We send the abstract class & pass the objects that are already implemented.

S - Functions having diff. responsibilities.
O - If-else cannot be there
L - Look for forced inheritance & overrides
I - Bulky interfaces .. too many func. to implement
D - Object passed as params wherever they are created as an interface type.

```cpp
class NotificationService {
public:
    // Send a message to the user via the specified type ("email" or "sms")
    void sendNotification(const User& user, const std::string& message, const std::string& type) {
        if (type == "email") {
            // Simulate sending an email
            std::cout << "Email to " << user.email << ": " << message << "\n";
        } else if (type == "sms") {
            // Simulate sending an SMS
            std::cout << "SMS to " << user.phone << ": " << message << "\n";
        }
    }
};

int main() {
    User alice{"Alice", "alice@example.com", "1234567890"};
    NotificationService service;
    service.sendNotification(alice, "Welcome, Alice!", "email");
    service.sendNotification(alice, "Your order has shipped.", "sms");
}
```

```cpp
// Abstract base class (interface) for notification senders
struct INotifier {
    virtual ~INotifier() = default;
    virtual void send(const User& user, const std::string& message) = 0;
};

class EmailSender : public INotifier {
public:
    void send(const User& user, const std::string& message) override {
        std::cout << "[Email] To: " << user.email << " | Message: " << message << "\n";
    }
};

class SMSSender : public INotifier {
public:
    void send(const User& user, const std::string& message) override {
        std::cout << "[SMS] To: " << user.phone << " | Message: " << message << "\n";
    }
};
```

```cpp
class NotificationService {
public:
    // Send a notification using any notifier (polymorphism)
    void sendViaChannel(INotifier& notifier, const User& user, const std::string& message) {
        notifier.send(user, message);
    }
};


User alice{"Alice", "alice@example.com", "1234567890"};
NotificationService service;
EmailSender emailSender;
SMSSender smsSender;

service.sendViaChannel(emailSender, alice, "Hello via Email!");
service.sendViaChannel(smsSender, alice, "Hello via SMS!");
```