Procedural vs OO

Procedural	OO
• Functions	• Objects
• Top-down approach	• Bottom-up approach
• No access specifier	• Access specifiers
• No encapsulation / less secure	• Encapsulation / more secure
• No overloading	• Overloading → better readability

Object

- Entity (Real-world / Runtime)
- Has states (data) and behaviours (functionalities)
- Instance of a class

Class

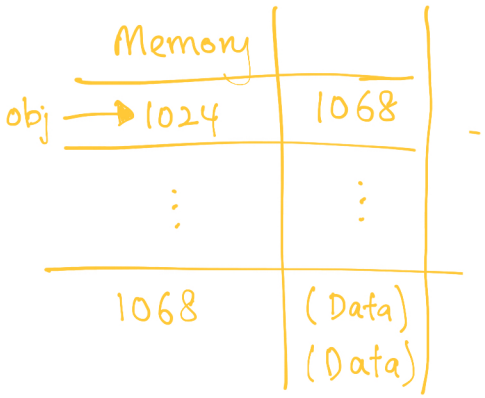
- Blueprint for objects
- Logical entity
- Contains:
  1) Fields (Attributes)
  2) Methods
  3) Constructors

* Creating Objects

1) Create variable ⇒ reference to object's memory
2) Allocate memory

⇓

new keyword ⇒ in heap memory



Constructors

- Initialise object state

Static Keyword

- Belongs to class instead of objects (persist)
- variable : single copy across all objects (shared)
- method :
  1) invoked without creating an instance
  2) may access static variables or methods but not instance variables or methods
  3) may not use this or super

this keyword

- reference variable referring to current object

Accessor and Mutator

- Accessor : Returns value of data property
- Mutator : Changes value of data property ⇒ data consistency

Encapsulation

- Access to private data via public methods
- Hides details and implementation of class from users
  - ↳ knows what a class does and how to call the methods

Object Composition (Aggregation)

- "HAS-A" relationship
- contains an object reference ⇒ reference data type
- used when there's no or limited "IS-A" relationship

Naming Conventions

1. ClassName ⇒ noun
2. InterfaceName ⇒ adjective
3. methodName ⇒ verb
4. variableName ⇒ no initial special char.
5. package.name
6. CONSTANT_NAME
7. ENUM_NAME

Inheritance

- derive new classes from existing classes ⇒ specialisation v.s. generalisation
- inherits attributes and behaviours from parent (except private)
- adds new capabilities in new class
- "IS-A" relationship

* Sub-Class

- defined using extends keyword
- may use super keyword to call parent class

Method Overloading

- same method name, different parameter types or length.
- increases readability

Method Overriding

- subclass contains method of same signature as parent class
- two ways : 1) refinement ⇒ implements super with additional refinement
  2) replacement
- subclass → superclass

Access / Visibility Modifiers

1. public : anywhere in the application
2. default / package : anywhere in package
3. protected : package or anywhere via inheritance only
4. private : only within class

Final keyword

1. stop value change
2. stop method overriding
3. stop inheritance

- improve security ⇒ ensures behaviour
- improve efficiency ⇒ type check @ compile time

Abstract

- do not have implementation
- hides implementation
- shows only functionalities or essential things

Interface

- only static constants and abstract methods
- classes may implement multiple interfaces.
- may be extended ⇒ implements derived also implements base implicitly

Abstract v.s. Interface

Abstract	Interface
• abstract & non-abstract methods	• only abstract methods
• single inheritance	• multiple inheritance
• any variable / constant types	• only static CONSTANTS
• may provide implementation of interface	• cannot provide implementation of abstract class
• may have class members of all visibilities	• class members are "public"

Package

- group of classes by functionality, usability and category
- provides access protection
- removes naming collision across packages

Polymorphism

- object reference referred to different types
  1) stores a subclass object in a superclass variable
  2) invoke method through superclass variable
  3) stored subclass' method is called instead (if overridden)
- facilitates addition of new classes with minimal modifications

Binding

- connecting a method call to a method body
  1) Static binding : object type determined at compile time
  2) dynamic binding : object type determined at run time
- dynamic by default except private, final and static
- actual type instead of referenced type

Upcasting

- derived class assigned to base class variable
- always safe

Downcasting

- type cast from base class to derived class
- requires explicit casting
- will always throw ClassCastException unless checked using "instanceof"

Benefits of Polymorphism

- allows code to ignore type specific details and interact with just base type
- easier to write and understand
- code that deals with the base class may be substituted with any derived class
- easily extended ⇒ minimal modification to existing code

Class Diagram

- static view of an application
- describes attributes, operations and constraints of a class

Association

- belongs or relates to
- unidirection : A holds reference to B only 
- bidirection : A holds ref. to B and vice versa

Aggregation and Composition

- "HAS-A" ⇒ has whole/part relationship
- aggregation : parts may or may not belong to whole 
- composition: whole creates/owns the parts. 

Generalisation and Realisation

- "IS-A"
- generalisation : class inheritance 
- realisation : interface implement 

Dependency

- not related ; loosely linked

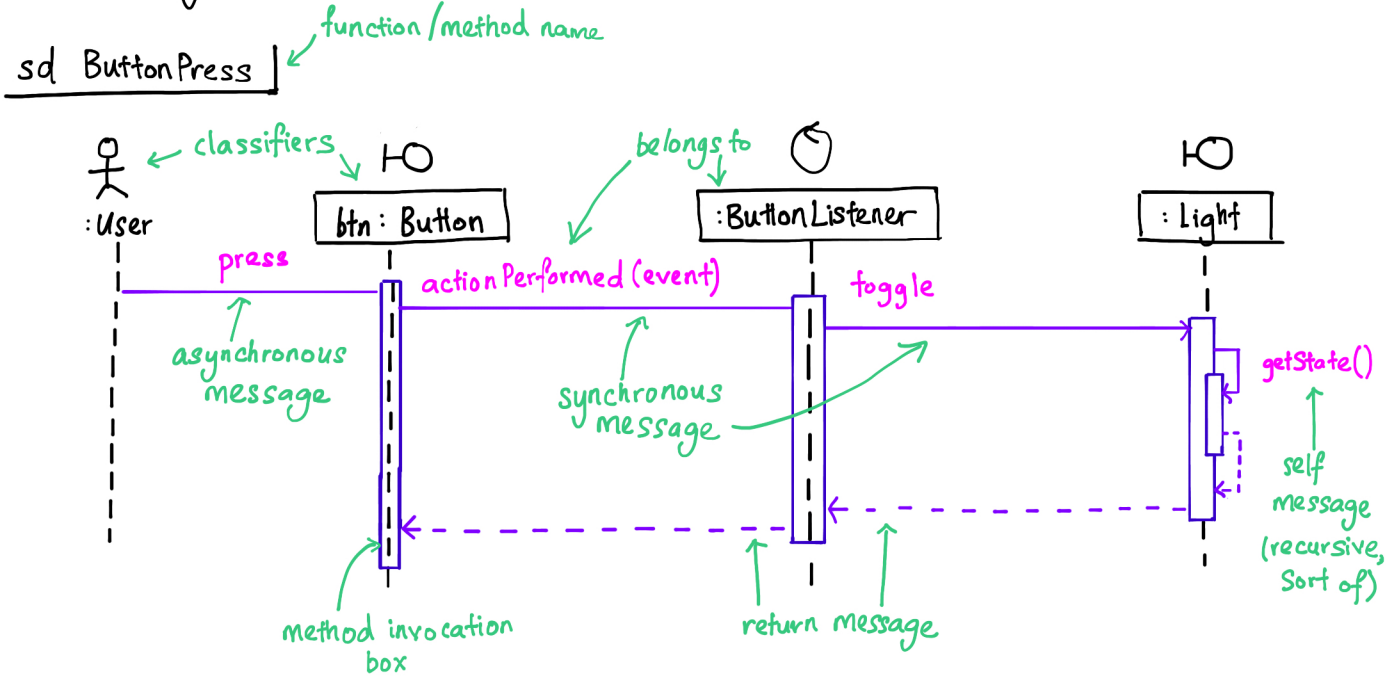
Object Diagram

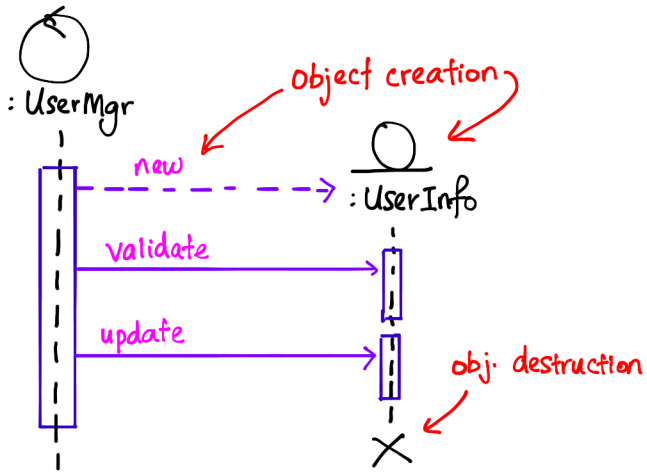
- underline reference and class name.
- shows state of object
- may be anonymous object and class

Visibility Symbols

- public : +
- protected: #
- private : -
- package : ~

Sequence Diagram



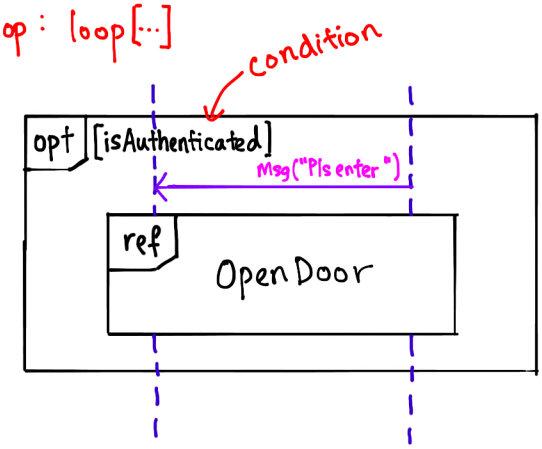


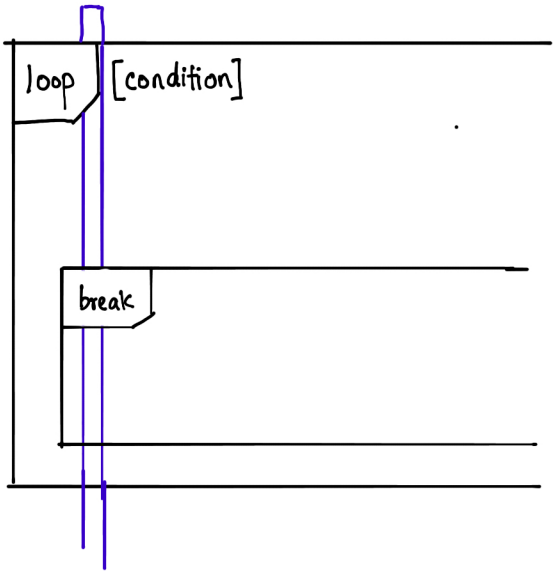
Reference

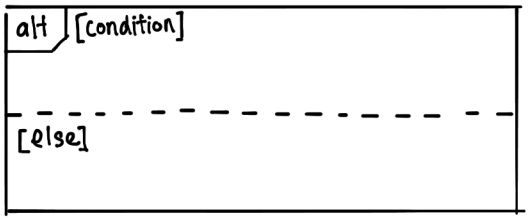
- ref on top-left ⇒ refer to another singular sequence diagram

Control Flow

- if : opt [...]
- loop : loop[...]







* diagram → code ⇒ make sure code is compilable
  - ↳ no missing variables or declaration (esp. global scope)

Modelling OO Applications

1. Identify classes through nouns.
2. Consider boundary classes to handle IO.
3. Consider control classes based on application logic and flow.
4. Add attributes and methods
5. Add class relationship
6. Add multiplicities
7. Identify behaviours and interactions through sequence diagram.
8. Enhance class details.
9. Implement the model
10. Quality testing
11. ???
12. Profit!

Rotting Design

1. Rigid : difficult to change
2. Fragile : breaks when something is changed, usually in areas w/o conceptual relationship
3. Immobile: inability to reuse modules of a project ⇒ too much baggage it depends on

Good Design

1. easy to read, maintain, and modify
2. efficient, reliable, and secure

OO Design Goals

- make software easier to change
- manage dependencies to minimise impact of change
- design with reusability, extensibility, maintainability
- achieve loose coupling and high cohesion

1. well-defined
2. simple

⇒ interact via well-defined interfaces

3. independent

SOLID

1. Single Responsibility
   - each class has only one responsibility
   - each responsibility an axis of change
2. Open-Closed
   - modify module functionality w/o changing source code
   - easily extensible
3. Liskov Substitution
   - subtypes must be substitutable for their base type
   - user of base class should function if a derived class is passed
   - specify pre and post conditions
4. Interface Segregation
   - classes should not depend on interfaces they do not use
   - many specific interfaces > one general interface
5. Dependency Injection
   - high level modules independent of low level modules
   - details depend on abstraction

Interface / Implementation

- header files (.h) : function declarations ; interfaces
- source code (.cpp) : class / function implementation

Advantages of header

- speeds up compile time
- organised ⇒ separate concepts, separate files
- separates interface from implementation. ⇒ client classes only need .h files

Class Definition

```
class ClassName {
    private:
        int _x, _y;
    public:
        void setX(int val);
        int getX() {
            return _x;
        }
};
void ClassName :: setX(int val) { _x = val };
```

Destructors

~ ClassName()

- releases allocated memory for the class

Object Creation

```
Point aPoint;   ⇒ object created already! unlike java
Point bPoint(12,34);   ⇒ no need " = new Point(...)"
Point *cPoint = new Point();
Point *dPoint = new Point(12,34);
```

} must use pointer if using new keyword

```
delete cPoint, dPoint;
```

} must subsequently delete the pointers

Class Inheritance

makes base attributes public / private to users of derived

```
class DerivedClass : [visibility] BaseClass {
    ⋮
  public :
    DerivedClass(...) : BaseClass(...) {
      ⋮
    }
  ⋮
};
```

base class constructor ⇒ super(...)

Default Parameter

- function parameter with default value assigned
- must be the rightmost parameters
- overloading not allowed

Reference

- alias to real variable
- defined using "&"
- cannot be NULL
- examples :

```
int ix;
int &rx = ix;
int *p = &ix;
int &q = *p;
```

Dynamic Initialisation

- initialisation in between function signature and body after a colon
- ClassA(const int x): _x(x) {}
- initialisation done before code execution in function body

Polymorphism

- virtual keyword
  - forces method evaluation based on object type
  - no dynamic binding w/o virtual
  - operates only on pointers and references
  - automatic virtual in derived classes
- pure (abstract) method
  - add "=0" at end of declaration
  - class automatically becomes extract class
  - example: virtual void methodA() = 0;

Downcast

- use "dynamic_cast"
- Type *t = dynamic_cast<Type *>(varName);
- *t = NULL if fail. ⇒ check for NULL
- only applicable to pointers

Array of Objects

- Cat *cats = new Cat[5]; ⇒ concrete class

  delete [] cats;
- Mammal **zoo = new Mammal*[4]; OR Mammal *zoo[4]; ⇒ abstract class

  zoo[0] = new Dog();
- for (i=0; i<4; i++) {

    if (zoo[i] != NULL) {

      delete zoo[i];

    }

  }

  delete [] zoo;

Operator Overloading

```
class Complex {
    double _real, _imag;
    public:
        Complex() {
            _real = 0.0;
            _imag = 0.0;
        }
        Complex(const double real, const double imag) {
            _real = real;
            _imag = imag;
        }
        Complex operator+(const Complex op) {
            double real = _real + op._real;
            double imag = _imag + op._imag;
            return (Complex(real, imag));
        }
        Complex operator*(const Complex op) {
            ...
        }
};
```

} may be implemented as standalone function

Friend

- allows non-member functions to access private data of a class
-
```
class ClassA {
    ⋮
    friend class SomeClass;
    friend int someMethod();
}
```

Java v.s. C++

- everything must be in a class ; no global function or data
- no scope resolution operator "::" ; dot for everything
- non-primitive types created only via "new"
- object references automatically initialised as null before assignment
- no pointers
- no destructors
- single-rooted hierarchy (from class Object)
- super grants access to only one level up in the hierarchy
- interface keyword → pure abstract class
- no virtual keyword → all non-static methods uses dynamic binding ⇒ less efficient
- no multiple inheritance
- no operator overloading

Summary

- C a subset of C++
- Header files for faster compilation, more organised codes, interface-implementation separation
- Object created on stack without "new"
- Created objects need to be deleted manually
- Uses dynamic initialisation
- Allows multiple inheritance
- Uses scope resolution operator "::" to explicitly identify class to use
- provides reference (&) as alternative to pointer (*)
- dynamic binding only for virtual functions
- allows default parameters in functions.
- allows operator overloading