

## Procedural vs OO

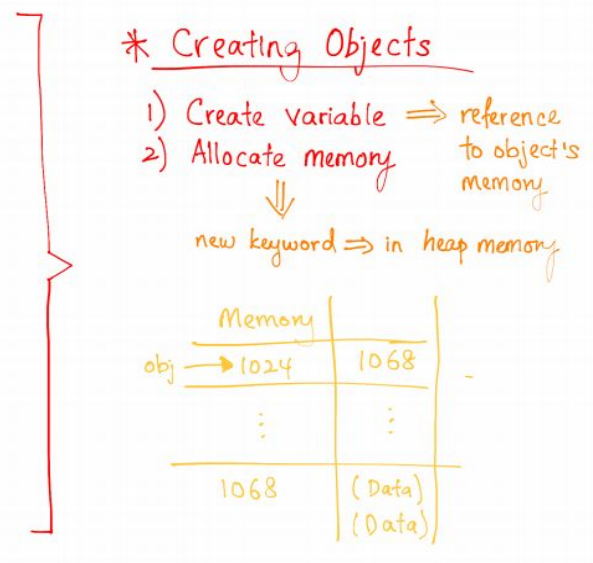
<u>Procedural</u>	<u>OO</u>
• Functions	• Objects
• Top-down approach	• Bottom-up approach
• No access specifier	• Access specifiers
• No encapsulation/less secure	• Encapsulation/more secure
• No overloading	• Overloading → better readability

### Object

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

### Class

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



### 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 method but **not** instance variable or methods
  - 3) may not use this or super

### **This keyword**

- Reference variable referring to current object

### **Accessor and Mutator**

- Accessor : Return 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. CONSTANTS\_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 => enforces 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

<ul style="list-style-type: none"> <li>• May provide implementation of interface</li> </ul>	<ul style="list-style-type: none"> <li>• Cannot provide implementation of abstract class</li> </ul>
<ul style="list-style-type: none"> <li>• May have class members of all visibilities</li> </ul>	<ul style="list-style-type: none"> <li>• Class members are "public"</li> </ul>

### **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 runtime
- Dynamic by default except private, final and static
- Actual type instead of reference 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 substitute with any derived class
- Easily extended => minimal modification to existing code

### **Class Diagram**

- Static View of an application
- Describe attributes, operations and constraints of a class

### **ASSOCIATION**

- Belongs or relates to
- Unidirection : A holds reference to B only →
- Bidirection : A holds reference 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

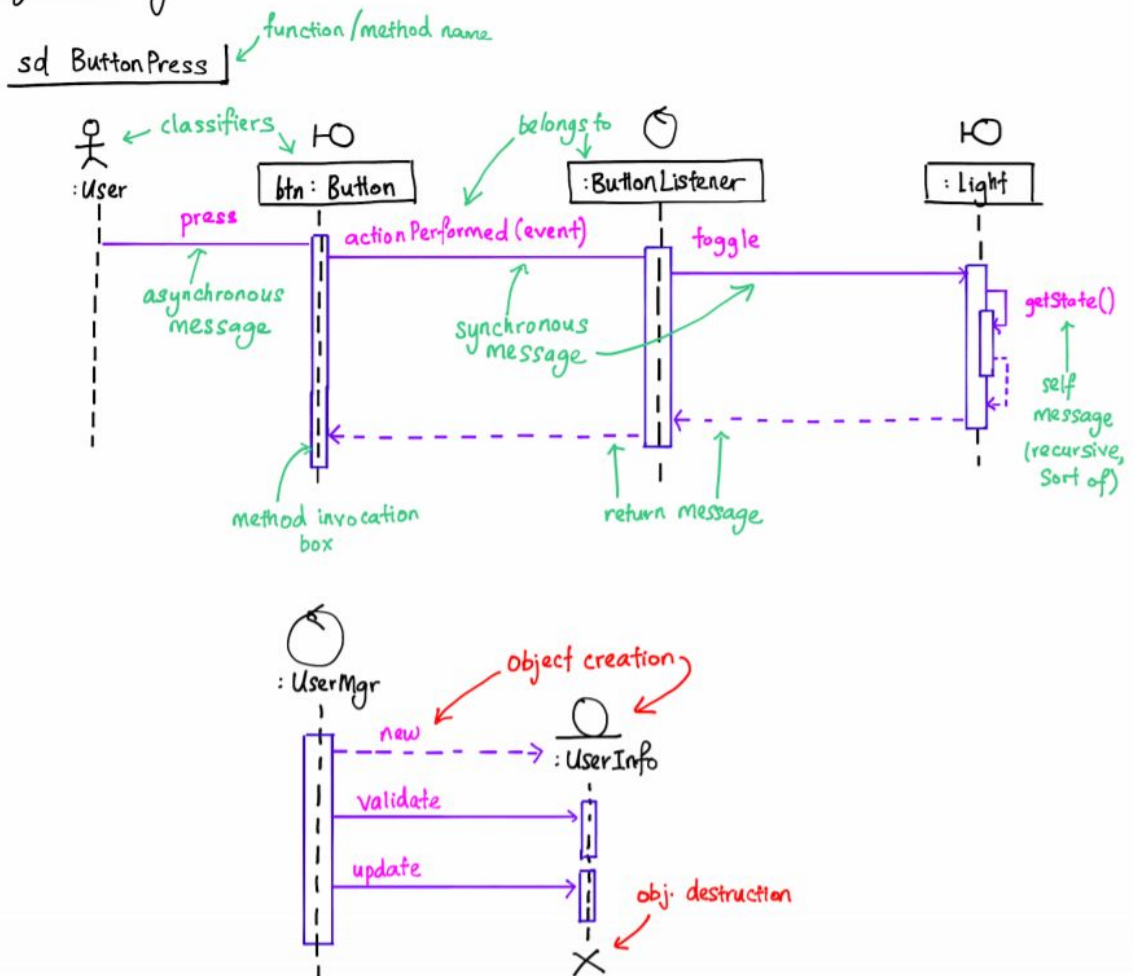
### **Visibility Symbols**

- Public : +
- Protected : #
- Private : -
- Package : ~



## SEQUENCE DIAGRAM EXAMPLE

### Sequence Diagram



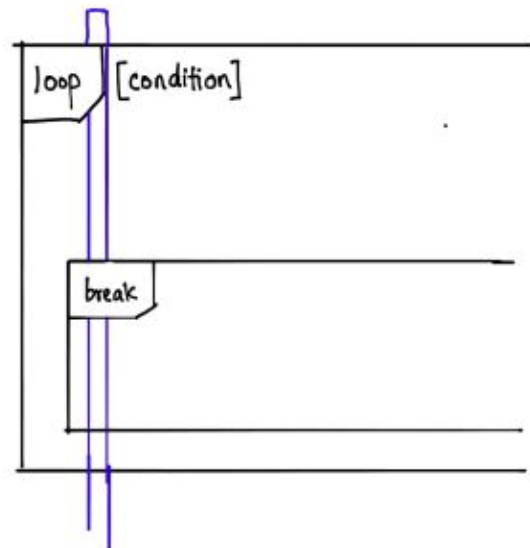
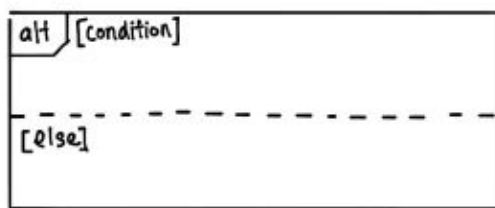
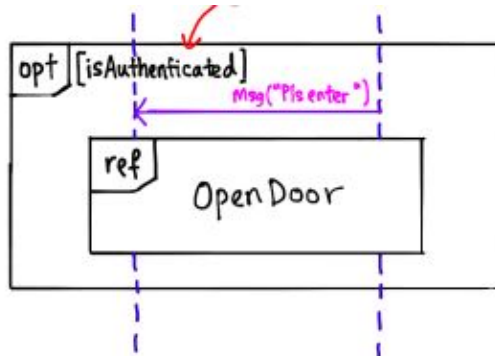
## Reference

- ref on top-left => refer to another singular sequence diagram



## Control Flow

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



## Impt

Diagram to code. Must make sure code is compilable, no missing variables or declaration

### **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 module 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**

1. Make software easier to change
2. Manages dependencies to minimise impact of change
3. Design with reusability, extensibility, maintainability
4. Achieve loose coupling and high cohesion

2-4 are

1. Well-defined
2. Simple
3. Independent

Which are interact via well-defined interfaces

### **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 condition
4. Interface Segregation
  - Classes should not depend on interfaces they dont use
  - Many specific interfaces > one general interface
5. Dependency Injection
  - High level modules independent of low level modules
  - Details depend on abstraction

### Interface/Implementation

1. Header files(.h): function declarations ; interfaces
2. Source code (.cpp) : class/function implementation

### Advantages of header

1. Speed up compile time
2. Oragnies => separate concept, separate files
3. Separates interface from implementation =. Client class only need h.files

### Class definition

```
Class ClassName{
    Private:
        Int _x,_y;
    Public:
        Void set x(int val);
        Int get x(){
            Return _x;
        }
};
Void className= set x (int val) {_x=val};
```

### Destructors

- ~ ClassName()
- Releases allocated memory of the class

### Object Creation

```
Point aPoint; => object created already! Unlike java
Point bPoint(12,34); => no need "=new point(...)"
Point *cPoint = new Point();           must use pointer if using new keyword
Point *dPoint = new Point(12,34);      must use pointer if using new keyword
Delete cPoint,dPoint;                  must use subsequently delete the pointers
```

## Class Inheritance

```
Class DerivedClass: [visibility] BaseClass{
    Public:
        DerivedClass(...) : BaseClass(...) {
        }
}
```

Explanation of highlighted above

-> makes base attributes public/private to users of derived

-> base class constructor => super(...)

```
// Base class
class Shape {
public:
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }

protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape {
public:
    int getArea() {
        return (width * height);
    }
};
```

## 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 without virtual
  - Operates only on pointers and references
  - Automatic virtual in derived classes
- **Pure (abstract) method**
  - Add "=0" at end of declaration
  - Class automatically becomes abstract 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 \* 200[4]; => abstract class  
Zoo[0] = new Dog();
- For(i=0;i<4;i++)  
{  
    if(zoo[i] != NULL)  
    {  
        delete zoo[i];  
    }  
}  
  
delete[ ] zoo; <- no space in between []



## 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

[https://www.tutorialspoint.com/cplusplus/cpp\\_overloading.htm](https://www.tutorialspoint.com/cplusplus/cpp_overloading.htm)

## 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

# Object Oriented Design & Programming

---

## Object-Oriented Model

- In OOP, computation is represented as the interaction among or communication between objects.
- An object is an entity that contains both, the attributes that describe the state of a real-world object and the actions that are associated with the real-world object.
  - The attributes of an object encompasses the data/variables that characterize the state.
  - The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.
  - The behaviour of an object encompasses the methods that represent the services & operations an object provides.
- Messages are requests from an object to another for the receiving object to produce some result.
- A class is a template/blueprint for objects. It contains data properties & methods. An object is a specific instance of a class.
- Abstraction – An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus, provides crisply defined conceptual boundaries.
- Encapsulation – Encapsulation builds a barrier to protect an object's private data. Access to private data can only be done through public methods of the object's class, such as accessors & mutators.
  - Information Hiding – Hides the details of implementation of the class from users.
- Inheritance – A mechanism that defines a new class that inherits the properties and behaviours (methods) of a parent class. Superclass/Base Class (Parent) → Subclass/Derived Class (Child). Any inherited behaviour may be redefined and overridden in the subclass. Avoids duplication of code.
  - Multiple inheritance is when a class inherits from more than one superclass. A problem arises when there is more than one property/method to inherit with the same name.
- Polymorphism – Same message can be sent to different objects with different results. Sending object does not need to know the class of the receiving object or how the object will respond.
- **this** – References the receiver object.
- **static** – Declares a class variable or class method. Applies to the whole class instead of individual objects.
- Object Composition – An object can include other objects as its data member(s). The class will contain object references as its instance variables. "has-a" relationship.

# Inheritance

- An important OO feature that allows us to derive new classes from existing classes by absorbing their attributes and behaviours and adding new capabilities in the new classes. This enables code reuse and can greatly reduce programming effort. “is-a” relationship.
- The superclass is a generalization of the subclasses.
- The subclasses are specializations of the superclass.
- **super** – `super()` calls the superclass’ constructor and `super.X()` can be used to call a superclass’ method.
- Method Overloading – When a method is overloaded, it is designed to perform differently when supplied with different signatures i.e. same method name but different number of parameters or parameter types. *Not a behavior due to inheritance.*
- Method Overriding – A subclass inherits properties and methods from the superclass. When a subclass alters a method from a superclass by defining a method with exactly the same signature, it overrides that method. Can either be a refinement or a replacement of the superclass’ method.
- When a message is sent to an object, the search for a matching method begins at the class of the object → immediate superclass → and so on...
- Visibility Modifiers:
  - `public (+)` – Visible anywhere in an application.
  - `private (-)` – Visible only within that class.
  - `protected (#)` – Visible anywhere within the same package.
- A package contains a set of classes that are grouped together in the same directory. Non-private data can be accessed by any object in the same package.
- Packages allow the same class name to be used in two different packages. For eg: X.Deck & Y.Deck.
- **final method** – When a method is declared as final, it cannot be overridden in subclasses.
- **final class** – When a class is declared as final, it cannot be a superclass.
- This helps improve security by ensuring no change in behaviour & efficiency by reducing compile-time type checking and binding.
- Abstract Classes & Methods (`{abstract}`) – Abstract methods don’t have any implementation in the abstract class. The implementation must be provided by the subclass(es).
  - `public abstract class Rectangle {}`
  - `public abstract double findArea();`
- Multiple inheritance is not supported by Java. However, Java does support implementing multiple interfaces.
- An interface is like an abstract class except it contains only abstract methods and constants (with *static final*). The *abstract* keyword is not needed for an interface in Java.
- A class implementing an interface has to provide an implementation for all the abstract methods. Otherwise, the new class will also be abstract.
- Interfaces can inherit each other as per normal i.e. *extends*.

- An abstract class is a real base class but an interface is not.
- An abstract class can have object attributes (data members) and non-abstract methods but an interface cannot.
- A class *extends* an abstract class, however, it *implements* one or more interfaces.
- Both types cannot be instantiated as objects with *new*.
- A concrete class is a class with implementation for all methods.

## Polymorphism

- In OOP, polymorphism means the ability of an object reference to refer to different types; knowing which method to apply depends on where it is in the inheritance hierarchy.
- When a program invokes a method through a superclass variable, the appropriate subclass version of the method is called, based on the type of the object reference stored in the superclass variable.
- The same method name & signature can cause different actions to occur, depending on the type of object on which the method is invoked.
- Binding – Refers which method to be called at a given time [i.e. connecting a method call to a method body].
- Static Binding – Occurs when the method call is bound at compile time.
- Dynamic Binding – The selection of the method body to be executed is delayed until execution time (based on the actual object being referred).
  - Java uses this by default for all methods except private, final & static.
- Upcasting – When an object of a derived class is assigned to a variable of a base class (or any ancestor class). However, subclass-only members cannot be referred to by a superclass variable.
- Downcasting – When an object of a base class is assigned to a variable of a derived class. This doesn't make sense in many cases and may be illegal.
- *object instanceof ClassName* – will return true if *object* is an instance of *ClassName* or any descendent class of *ClassName*.
- Benefits of Polymorphism:
  - Simplicity – Code can ignore type-specific details and just interact with the base type of the family. Makes it easier to write and understand the code.
  - Extensibility – New functionality can be added by creating new derived classes without modifying other derived classes.
- Method overriding can be done by overriding methods of a superclass, implementing abstract methods of an abstract class or implementing methods of an interface.

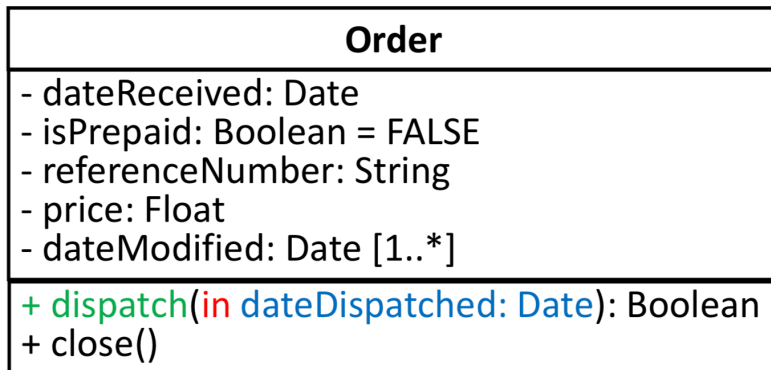
# Design Principles

- Symptoms of Rotting Design:
  - Rigidity – The tendency of software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes.
  - Fragility – The tendency of software to break in many places every time it is changed. Breakage may occur in areas that have no conceptual relationship with the area that was changed.
  - Immobility – The inability to reuse software/module from other projects or from parts of the same project. The module may have too much baggage that it depends on.
- Good design & programming must be easy to read, easy to maintain and modify, efficient, reliable and secure.
- The main design goal of OOD is to make software easier to change i.e. minimise impact of change.
- Class Design Guidelines:
  - Design with Reusability, Extensibility & Maintainability in mind.
  - Achieve loose (low) coupling and high cohesion.
- A modular program has well-defined, conceptually simple and independent units interacting through well-defined interfaces. Achieved through encapsulation, low coupling & high cohesion.
- SOLID Design Principles:
  - Single Responsibility Principle – There should never be more than one reason for a class to change. If the class has more than one responsibility, then the responsibilities become coupled.
  - Open-Closed Principle – A module should be open for extension but closed for modification. We want to be able to change what the modules do, without changing the source code of the modules.
  - Liskov Substitution Principle – Subtypes must be substitutable for their base types. A user of a base class should continue to function if a derivative of that base class is passed to it.
    - Design by Contract: Methods should specify their pre- and post-conditions.
  - Interface Segregation Principle – Many client specific interfaces are better than one general purpose interface. Classes should not depend on interfaces that they do not use.
  - Don't Repeat Yourself – Refactor to eliminate duplicated code and functionality.
  - Dependency Injection Principle – High level modules should not depend upon low level modules. Both should depend upon abstractions. This allows the simple reuse of high level modules.

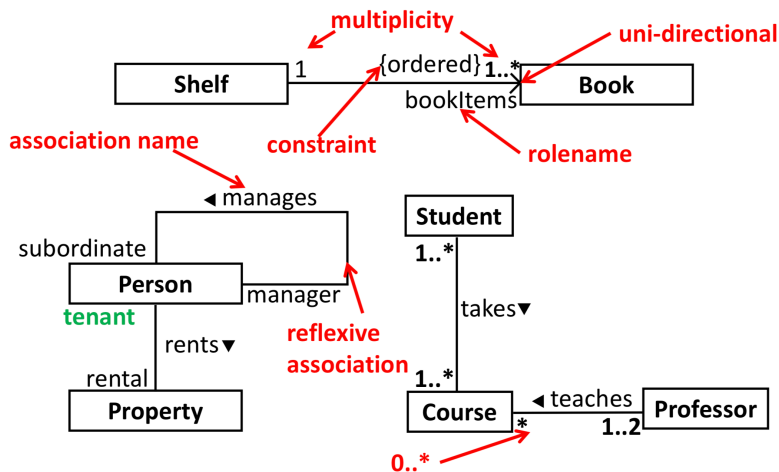
# Object-oriented Design and Programming

## Chapter 6 UML Diagram – Class Diagram

- UML  
Unified modelling language
- Class Diagram  
Attribute specification: name: type [multiplicity] = initial\_value  
Operation (function) specification: name(parameter-list): return type list  
Parameter specification format: direction name: type = default value

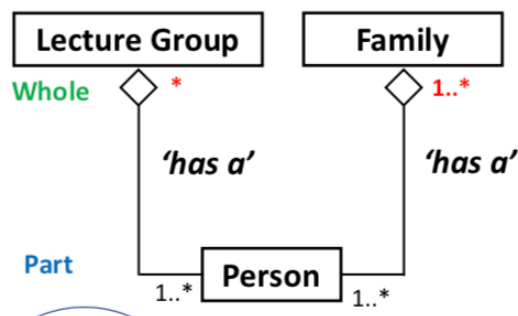


- Association

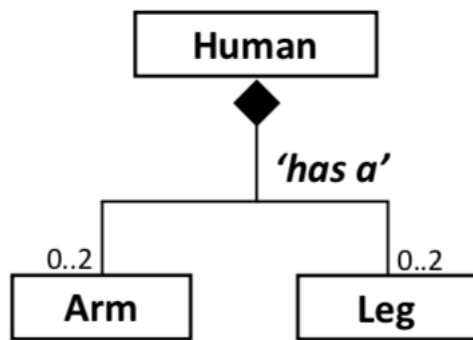


Represented by solid straight line with or without arrows.

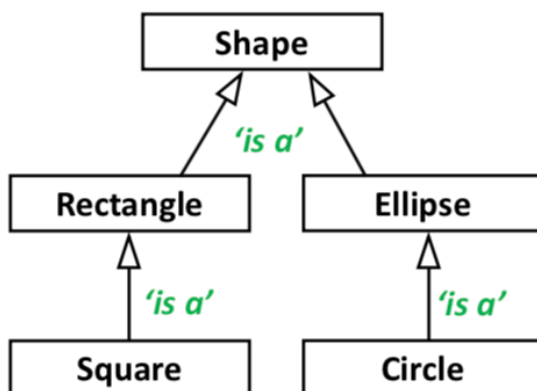
- Aggregation and composition  
Aggregation: whole and parts co-exist



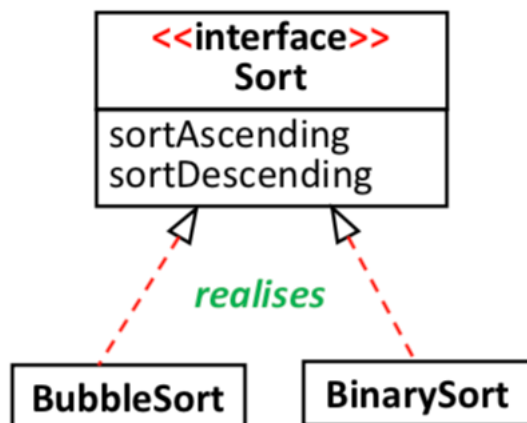
Composition: whole owns/creates parts



- Generalisation and interface realisation  
Generalisation: extends abstract class

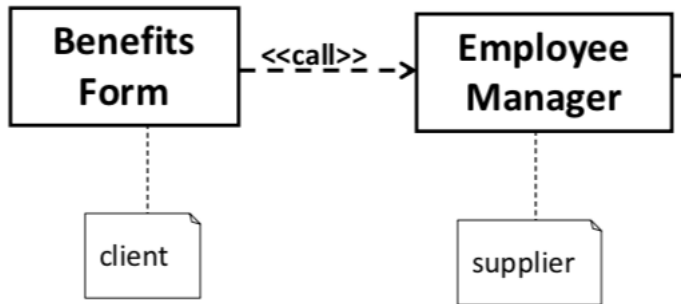


Realization: implement interface



- Dependency  
A dependency is generally shown as a dashed arrow between two model elements. The model element at the tail of the arrow (the **client**) **depends on** the model element at the arrowhead (the **supplier**). The arrow may be labeled with an optional stereotype and an optional name.





- UML Class Diagram to Java code

1. Association

If class A is associated with class B, then class B must be an attribute of class A.

If the association is bi-directional, then two classes have each other as their attributes.

2. Dependency

If class A is dependent on class B, then class B can be a method return type, method parameter type and as a variable type inside the method of class A.

3. Generalization

If class A is the generalization of class B, then class B extends class A.

4. Realization

If class B is the realization of the interface A, then class B implements interface A.

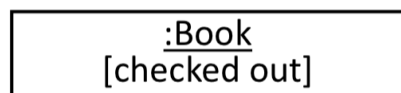
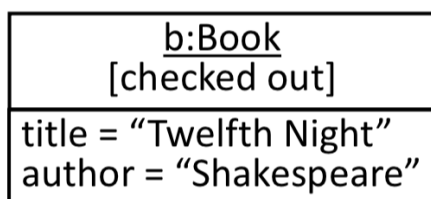
5. Aggregation

Parts are added through reference

6. Composition

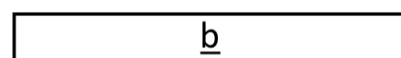
Parts are created in class

- Object Diagram



Anonymous  
object

library.checkout( new Book("Twelfth Night") );



Anonymous  
class

- Class Stereotypes

1. Entity

Persistent information tracked by program/application/system  
e.g. student, course, group

2. Boundary

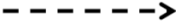




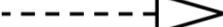
Interaction between system and external – interfaces  
e.g. xxxUI, xxxForm

3. Control

Logic to coordinate and realise use case  
e.g. courseManager

- Summary

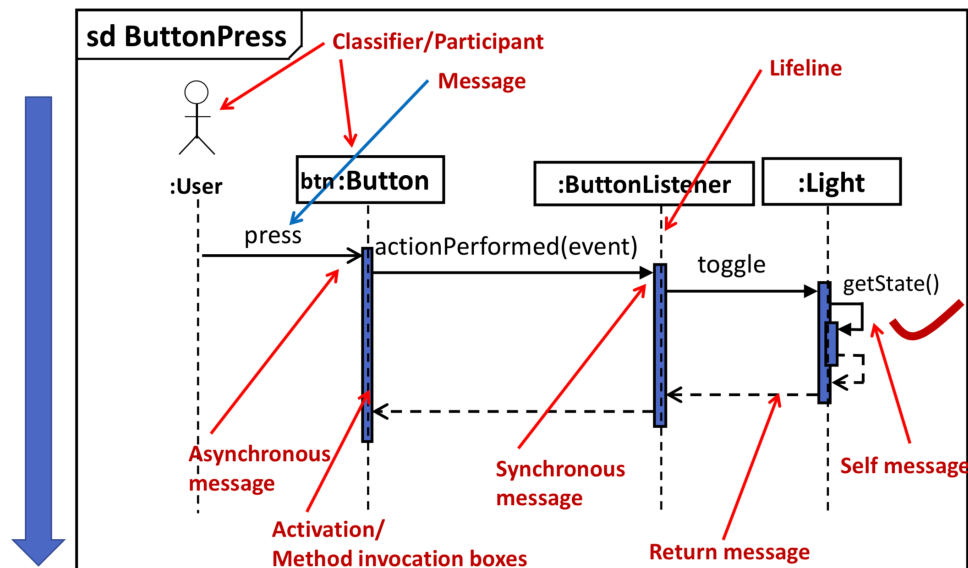
▪ Type of Class relationships

- Dependency 
- Association 
  - Aggregation 
  - Composition 
- Generalisation 
- Realisation 

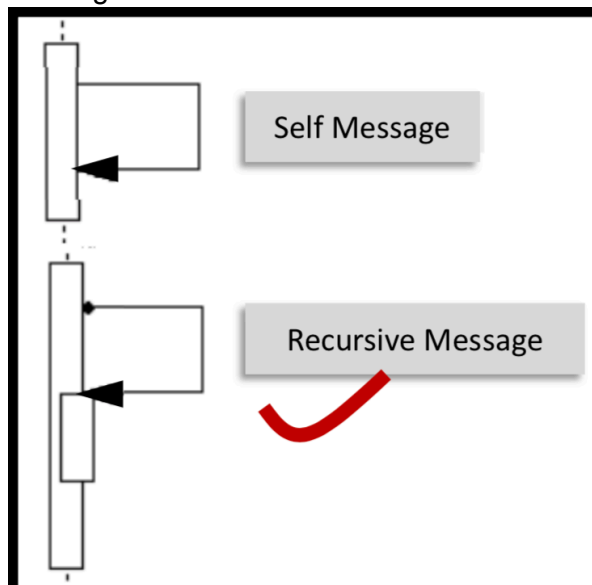
# Object-oriented Design and Programming

## Chapter 6 UML Diagram – Sequence Diagram

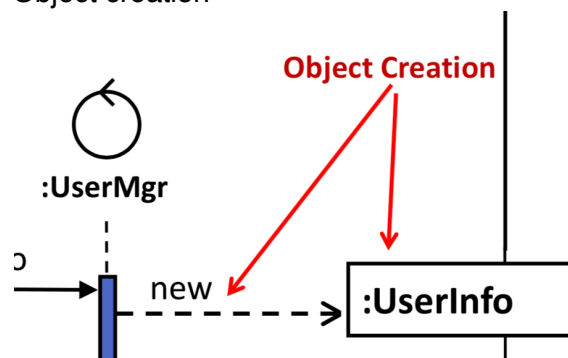
- Sequence Diagram

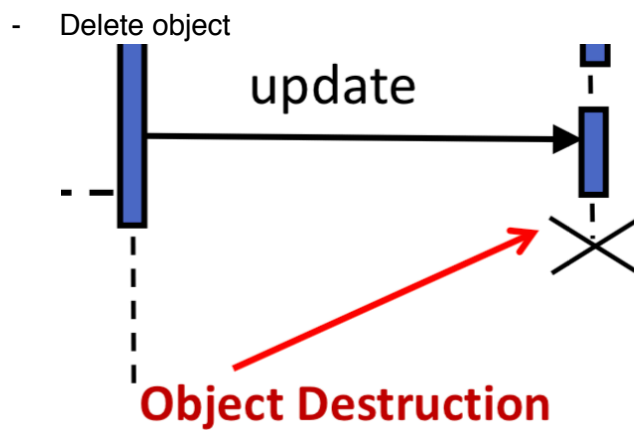
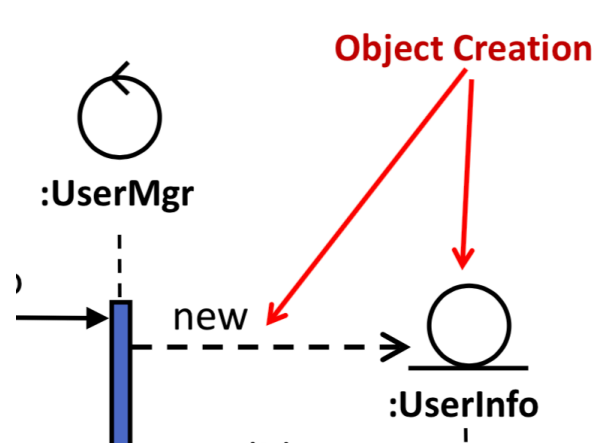


- Message

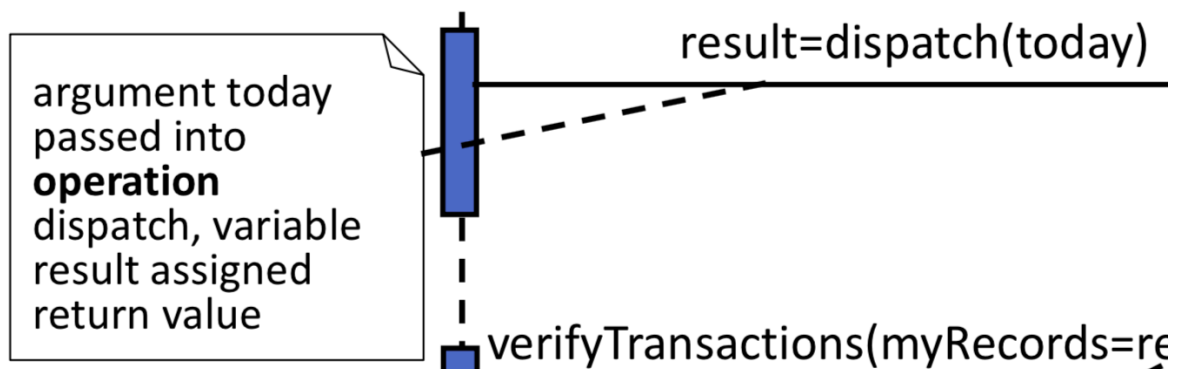


- Object creation





- Message specification



- Interaction fragments
  - ref: refer to another sequence diagram
  - opt: if clause
  - loop: iteration
  - alt: else if

# Object-oriented Design and Programming

## Chapter 8 Modelling OO Application

- Modelling OO Application
  - Entity: persistent information tracked by system
  - Boundary: interaction between actor and system
  - Control: logic to realize use case
  - Add: attributes, methods, class relationship, multiplicities
- Design to codes
  1. Implement the codes for each class in the class diagram
  2. Implement the codes for each functional flow in sequence diagram
  3. Amend class diagram or sequence diagram accordingly if needed

# Object-oriented Design and Programming

## Chapter 9 Design Principles

- Symptoms of rotting design
  1. Rigidity
  2. Fragility
  3. Immobility
- OO design goals
  1. Easy to change
  2. Easy to add functionality
  3. Manage dependencies between classes and packages of classes to minimize impact of change on other parts of the software
  4. Minimize reasons that modules or packages might be forced to change because of another module or package it depends on
- Class design guidelines  
Manage: coupling and cohesion (reusability, extensibility, maintainability)  
Achieve: low coupling and high cohesion
- Designing for a change  
A modular program has well-defined conceptually simple and independent units interacting through well-defined interfaces.
  - o Encapsulation
  - o Low coupling
  - o High cohesionInteraction via interfaces promotes loose coupling.
- Solid Design Principles
  1. Single Responsibility Principle  
Avoid creating god class, each class should have only one responsibility which is an axis of change
  2. Open-closed principle  
A module should be open for extension but closed for modifications. We want to be able to change what the modules do, without changing the source code of the modules. Abstraction is the key.
  3. Liskov Substitution Principle  
Subtypes must be substitutable for their base types. A user of a base class should continue to function properly if a derivative of that base class is passed to it.
    - o Design by contract
      - a. Methods should specify the pre and post conditions. What must be true before and what must be true after execution.
      - b. Restating the LSP in terms of contracts, the derived class is substitutable for its base class if its pre-condition are no stronger than the base class methods and post conditions are no weaker than the base methods.
  4. Interface Segregation Principle  
Many client specific interfaces are better than one general purpose interface.

5. Dependency Injection Principle

A high level modules should not depend upon low level modules. Both should depend upon abstractions.

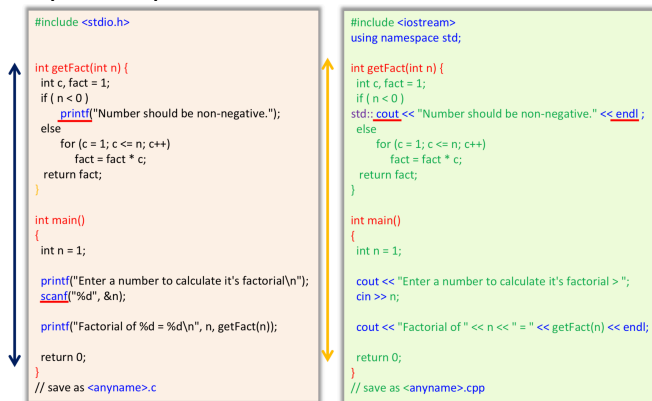
Abstractions should not depend upon details. Details should depend upon abstraction.

Inversion of control.

# Object-oriented Design and Programming

## Chapter 8 OO Concept in C++

- Simple comparison



- Interface/ Implementation

Header files(.h): used for function declaration, declare the interface

Source code(.cpp): used for defining class/ function implementation

Speeds up compile time as not everything needs to be compiled every time you make little change. Separate interface from implementation.

- C++ structure

Private and public attributes/methods are defined separately.

Scope operator: "::"

Dynamic initialization: `className(arg1, arg2): par1(arg1), par2(arg2){}`

Destructor: `~className()`

Declaration and instantiation are done at the same time. E.g. `Point aPoint;` (create using stack)

Create using heap: `Point *bPoint = new Point()`

Delete all objects after use

- Class inheritance

Class `derived_class_name`: `[visibility-mode] base_class_name{};`

Visibility-mode = private(default), public

Private inheritance: public members of the base class become private in derived class

Public inheritance: public members remain public

Can use base class constructor : `base_class(args)`

Support multiple inheritance: `: class1, class 2`

If an object is destroyed, the destructor of the corresponding class is invoked. If the class is a subclass, its superclass will also be destroyed recursively.

- OO and non-OO features

Overloading just like java. If return type is different, then it is not overloading.

Reference:

- o Alias to real variable or object
- o Ampersand & is used to define a reference
- o Cannot be null

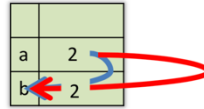


```
#include <iostream>
using namespace std;
```

```
void addIt(int a) { a += a; }
void doubleIt(int &a) { a *= 2; }
void tripleIt(int *a) { *a *= 3; }
```

```
int main() {
    int b = 2;
    cout << "b is now " << b << endl;
    addIt(b);
    cout << "after addIt(int a), b is " << b << endl;
    → doubleIt(b);
    cout << "after doubleIt(int &a), b is " << b << endl;
    tripleIt(&b);
    cout << "after tripleIt(int *a), b is " << b << endl;
    // cin >> b;
}
```

**OUTPUT:**  
b is now 2  
after addIt(int a), b is 2  
after doubleIt(int &a), b is 4  
after tripleIt(int \*a), b is 12



## - Polymorphism

Virtual: force method evaluation to be based on object type rather than reference type

Without virtual: no polymorphic (no dynamic binding)

Virtual function only operates on pointers and references

If a method is virtual in a class, it is automatically virtual in all derived class.

Pure virtual: abstract method

Virtual void area() = 0;

The class becomes abstract class. Cannot be instantiated.

Safe down-cast

Use dynamic\_cast, type \*t = dynamic\_cast<Type\*>(variable)

Return null if the conversion is not possible

Only applicable to pointers

## - Operator overloading

*// standalone function*

Complex **operator +**(Complex op1, Complex op2) {  
 double real = op1.real() + op2.real();  
 double imag = op1.imag() + op2.imag();  
 return(Complex(real, imag));  
}

## - Friend

Friend allows non-member function access to private data of a class.

## - String class

Need to #include<string>