



# Recitation - 11

Jahnavi - [jp5867@nyu.edu](mailto:jp5867@nyu.edu)

# Object Oriented Programming:



- Object-Oriented Programming (OOP) is a programming paradigm that focuses on the use of objects to represent real-world entities and concepts. All object have some properties - Data and some functionality to handle that those Properties - Member Functions
- 
- A class is a blueprint for creating objects, while an object is an instance of a class.

For example, a class called "Car" might define properties like "make", "model", and "year", while an object of that class might represent a specific car with those attributes.

## **Properties:**

Encapsulation: bundling data and methods that operate on that data into a single class.

Inheritance: creating a new class by inheriting properties and methods from an existing class.

Polymorphism: ability of objects of different classes to be treated as if they were the same type.

Abstraction: hiding implementation details and focusing on the essential features of an object.

# Styles of OOPS:



## Prototype based OOL:

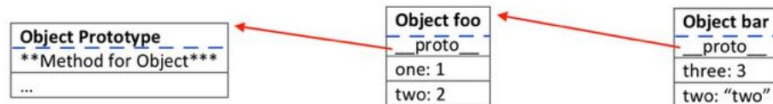
1. They rely on prototypes rather than classes for object creation and inheritance. In this approach, objects are created by cloning or copying existing objects, which act as prototypes.
2. Object is not related to class. It could be created as an empty object or cloned from an existing object (prototype object).
3. Objects inherit directly from other objects through a prototype property. -\_\_proto\_\_ in JavaScript.
4. Changes to prototype object (e.g., assignments) propagate to the clone.
5. Changes to the clone do not propagate back to the prototype

```

var foo = {one: 1, two: 2};
var bar = Object.create( foo ); // bar = clone(foo)
bar.one; // return 1, bar refers foo.a
bar.three = 3; // add new field
bar.two = "this is two"; // add new field two, and shadow proto object
print(bar.two);
bar;
/*
three: 3
two: "this is two"
__proto__:
  name: "foo"
  one: 1
  two: 2
*/

```

The memory map for objects `foo` and `bar` are:



# Styles of OOPS:



Class based OOL:

- A class is always viewed as a template ('blueprint') to create objects.

Components

- Data, in the form of fields, often known as attributes.
- Code, in the form of procedures, often known as methods.

Special methods

- Constructor • Destructor

```
class Point {  
public:  
    Point(double fst, double snd) { // Constructor  
        this->first = fst;  
        this->second = snd;  
    }  
    void print () { // Method  
        cout<<"("<<first<<", "<<second<<")"<<endl;  
    }  
    ~ Point() { } // Destructor  
private:  
    double first; // Attribute  
    double second;  
};
```

# How encapsulation is achieved:



Keyword	C#	C++	Java
<code>private</code>	class	class	class
<code>protected internal</code>	same assembly and derived classes	-	-
<code>protected</code>	derived classes	derived classes	derived classes <i>and/or</i> within same package
<code>package</code>	-	-	within its package
<code>internal</code>	same assembly	-	-
<code>public</code>	everybody	everybody	everybody

# Objects for a class:



A particular instance of a class, where the object can be a combination of variables, functions, and data structures.

## Object allocation

- In Java, all objects are dynamically allocated on the heap.
- In C++, local variable (including object) is allocated on the stack.

Any objects created by using operator new are allocated on the heap.



# Inheritance:



A mechanism to derive a class (i.e. subclass) from another class (i.e. superclass) for a hierarchy of classes that share a set of attributes and methods.

- Inheritance rule
  - Data, all attributes from superclass will be inherited by the subclass.
  - Code, depends on what type of methods you have in the superclass. For example, for non-static method:
    1. Private method will not be inherited.
    2. Other methods will be inherited to subclass.

# Inheritance in C++:




A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions -

1. Constructors, destructors and copy constructors of the base class.
2. Overloaded operators of the base class.
3. The friend functions of the base class.

# Inheritance in C++:



```
class A {  
    void method_1() { . . . } void method2() {...}  
};  
class B: public A { // class B publicly inherits class A, B <- A  
    // Implicitly inherits method_1 from class A  
    void method_2() {...} // method_2 has been overridden  
    void method3() {...} // New methods just for class B  
};  
// B is subclass of A, A is superclass of B.
```

# Inheritance in Java:



1. A subclass inherits all of the non-private members (fields and methods) of its superclass.
2. A subclass can add its own members (fields and methods) in addition to the ones it inherits from its superclass.
3. If a subclass overrides a method from its superclass, the method in the subclass must have the same signature (method name, parameter types, and return type) as the method in the superclass.
4. Constructors are not inherited, but they can be called using the `super()` keyword.
5. The access modifier for a method in the subclass cannot be more restrictive than the access modifier for the method in the superclass.

For example, if a method in the superclass is declared with the public access modifier, the method in the subclass can be public or protected, but not private.

# Inheritance in Java:

// Base class or Superclass

```
class Animal {  
    protected String name;  
  
    // Constructor  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    // Method to make the animal speak  
    public void speak() {  
        System.out.println("The animal speaks");  
    }  
  
    // Method to print the name of the animal  
    public void printName() {  
        System.out.println("Name: " + name);  
    }  
}
```

// Derived class or Subclass

```
class Dog extends Animal {  
    private String breed;  
  
    // Constructor  
    public Dog(String name, String breed) {  
        // Call the constructor of the superclass  
        super(name);  
        this.breed = breed;  
    }  
  
    // Method to make the dog speak  
    // Overrides the speak() method of the superclass  
    public void speak() {  
        System.out.println("The dog barks");  
    }  
  
    // Method to print the breed of the dog  
    public void printBreed() {  
        System.out.println("Breed: " + breed);  
    }  
}
```

# Liskov substitution principle:



The type of a subclass can extend the type of its superclass by adding new members (attributes and methods).

e.g. method\_3 in class B. (In C++ example)

- Objects of superclass may be replaced with objects of subclass
  - a. That means, objects that belong to the subclass can be used whenever an object of the superclass is expected.
  - b. Due to inheritance features, object for subclass is safe to assign / pass to variable declared as superclass type.

`A *a = new B()`

The reason why dynamic dispatch could be achieved.

# Dynamic dispatch:



1. The process of selecting which implementation of a polymorphic operation to call at the run time.
2. In C++, virtual methods helps in run time polymorphism
3. In Java, every non-static method, except final and private method, is virtual by default.
4. In C++, method calls via pointers (or references) to a base type cannot be dispatched at compile time, the dynamic dispatch is used through virtual function.

```
class A{
    A() { x = 0; z = 0;}
    public int m() { return x; }
    public int x;
    private int z;
}
```

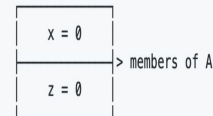
```
class B extends A {
    B() { x = 1; y = 2;}
    @Override
    public int m() { return x + y;}
    public int y;
}
```

```
class Main {
    public static void main(String[] args) {
        A a = new B(); // static: A, dynamic: B
        // int z = a.y; // Not allowed
        System.out.printf("The result of m should be: %d\n", a.m()); // actual call method m in class B
        A act_a = new A(); // sizeof(act_a) == sizeof(a)? No!
        // instanceof checks if a given object is an instance of a given class, superclass or interface
        System.out.println(act_a instanceof A == a instanceof A); // true
        System.out.println(act_a instanceof B == a instanceof B); // false
    }
}
```

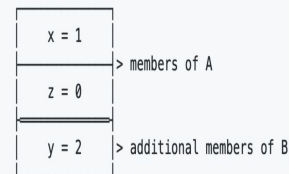
because at compile time y value of B is not accessible,  
since it doesn't know a is of type B.

Members of B are available to a only at run time, not compile time

A Instance (act\_a):



B Instance (a):





# Virtual Method Table:

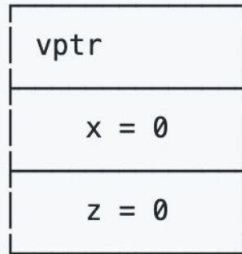


It is a table that contains pointers to the virtual functions of a class, which are functions that can be overridden by subclasses.

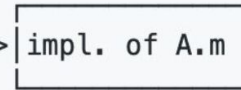
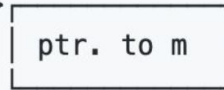
1. Each class has its own vtable which is shared by all instances of that class.
2. When a class has one or more virtual functions, the compiler automatically adds a virtual pointer to the class's layout, which points to the VMT for the class. When an object of the class is created, its virtual pointer is initialized to point to the VMT for that class.
3. A vtable for subclass is created by copying the vtable from superclass and changing any pointers of overridden (virtual) methods to the new implementations

## Objects' memory map in memory

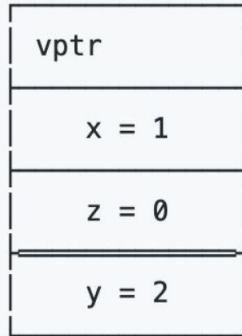
A Instance (act\_a):



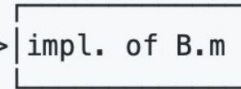
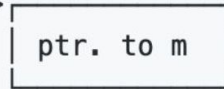
A vtable:



B Instance (a):



B vtable:



# Example:

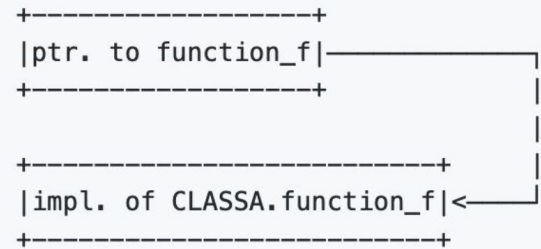
```
#include <iostream>

using namespace std;
class CLASSA {
public:
    virtual void function_f () {}
    void function_g() {}
};

int main()
{
    CLASSA local_a;           // stack object
    local_a.function_f();     // callq    __ZN6CLASSA10function_fEv, normal method call
    local_a.function_g();     // callq    __ZN6CLASSA10function_gEv, normal method call
    CLASSA *dyn_a = new CLASSA(); // heap object
    dyn_a->function_f();      // callq    *(%rdx), use vtable
    dyn_a->function_g();      // callq    __ZN6CLASSA10function_gEv, normal method call
    CLASSA *ptr_a = &local_a;
    ptr_a->function_f();      // callq    *(%rdx), use vtable
    ptr_a->function_g();      // callq    __ZN6CLASSA10function_gEv, normal method call
    return 0;
}
```

since we are not using a pointer

## CLASSA's vTable



## vtable inside assembly code

```
__ZTV6CLASSA: # Vtable for CLASSA
    .quad    0
    .quad    __ZTI6CLASSA
    .quad    __ZN6CLASSA10function_fEv
```

`dyn_a->function_f()` and `ptr_a->function_f()`  
uses vtable

```

class A {
    private A m1(){
        System.out.println("A.m1()");
        return new A();
    }
    public void m2(){
        System.out.println("A.m2()");
    }
    public A m3() {
        System.out.println("A.m3()");
        return this;
    }
    public A m4() {
        return this.m1();
    }
}

```

```

class B extends A {
    private B m1(){
        System.out.println("B.m1()");
        return new B();
    }

    @Override
    public void m2() {
        System.out.println("B.m2()");
    }
}

```

```

class Main {
    public static void main(String[] args) {
        A a1 = new B();

        A a2 = a1.m4();
        // B's m1 not on vtable, m1 of A used by m4()
        // Private method cannot be overridden

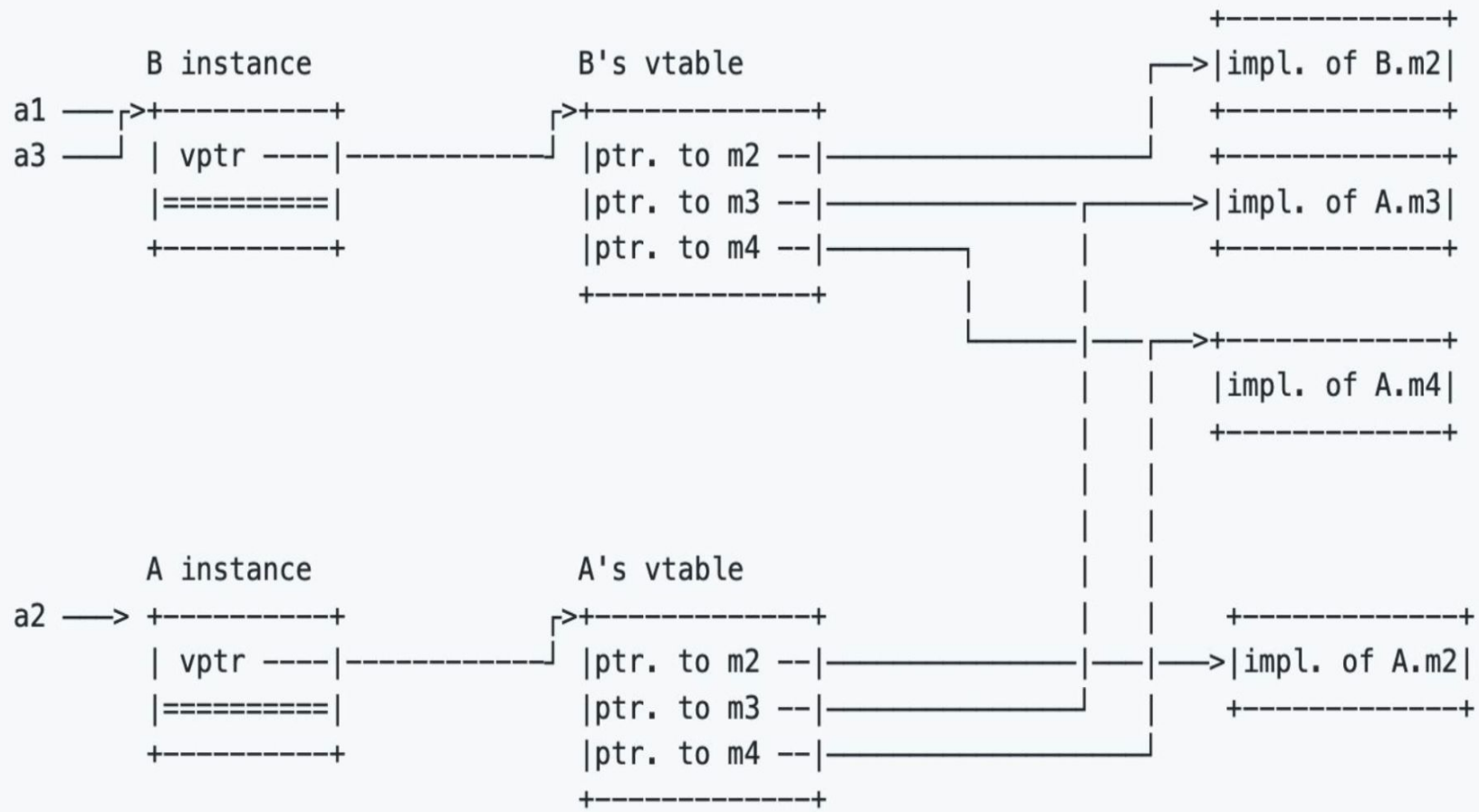
        a2.m2();

        A a3 = a1.m3();

        a3.m2();
    }
}

```

1. What are the static and dynamic types of a1, a2 and a3?
2. What methods are the call a2.m2(), a1.m3() and a3.m2() dispatched to?



Answer for Q1:

- To determine dynamic type, we check the actual instance pointed for each object.
- To determine static type, we check the type for each object in the code.

For a1: Static type: A , Dynamic Type: B

For a2: Static type: A, Dynamic Type: A

For a3: Static type: A, Dynamic type: B

Answer for Q2:

To determine dynamic dispatch, we check each object's instance, and look up the virtual table.

call for a2.m2() is dispatched to A.m2

call for a1.m3() is dispatched to A.m3

call for a3. m2() is dispatched to B. m2

# Reflection:



Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- In Java, the required classes for reflection are provided under `java.lang.reflect` package.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.

```

import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;

class Test
{
    private String s;

    public Test() { s = "test"; }

    public void method1() {
        System.out.println("Printing s: " + s);
    }

    public void method2(int n) {
        System.out.println("The number is " + n);
    }

    private void method3() {
        System.out.println("Private method");
    }
}

```

```

class Demo
{
    public static void main(String args[]) throws Exception
    {
        Test obj = new Test();

        Class cls = obj.getClass();
        System.out.println("The name of class is " +
            cls.getName());

        Constructor constructor = cls.getConstructor();
        System.out.println("The name of constructor is " +
            constructor.getName());

        System.out.println("The public methods of class are : ");
        Method[] methods = cls.getMethods();
        for (Method method:methods)
            System.out.println(method.getName());

        // creates object of desired method by providing the
        // method name and parameter class as arguments to
        // the getDeclaredMethod
        Method methodcall1 = cls.getDeclaredMethod("method2",
            int.class);
        methodcall1.invoke(obj, 19);

        Method methodcall3 = cls.getDeclaredMethod("method3");
        // allows the object to access the method irrespective
        // of the access specifier used with the method
        methodcall3.setAccessible(true);
        // invokes the method at runtime
        methodcall3.invoke(obj);
    }
}

```



# Aspect oriented programming:



- It aims to modularize cross-cutting concerns in software systems.
- A cross-cutting concern is a feature or behavior that affects multiple modules or components of a system, but cannot be cleanly encapsulated in any one of them.
- Examples of cross-cutting concerns include logging, security, caching, error handling, and transaction management.
- In AOP, the program is composed of two kinds of modules: objects and aspects.
- Objects contain the main business logic of the program, while aspects contain the cross-cutting behavior that needs to be woven into the program.
- Aspects can be applied to objects at different points in the program execution, such as method calls, field accesses, or exception handling.

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.After;

@Aspect
public class LoggingAspect {

    @Before("execution(* com.example.MyClass.myMethod(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("Entering method " + joinPoint.getSignature().getName());
    }

    @After("execution(* com.example.MyClass.myMethod(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("Exiting method " + joinPoint.getSignature().getName());
    }
}
```

In this example, we define an aspect called `LoggingAspect` that logs the entry and exit of a method called `myMethod` in a class called `MyClass`. The `@Before` annotation specifies the advice to be executed before the join point (i.e., before the method is called), and the `@After` annotation specifies the advice to be executed after the join point (i.e., after the method returns). The `JoinPoint` parameter allows access to information about the join point, such as the method signature.

Note that the `execution()` pointcut expression specifies the method signature to match, using wildcards to match any arguments.