ROBOTICS CORNER

botics Corner

botics Corner

OOP relations

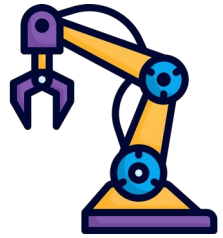
01 Is-a Relation

02 Has-a Relation

03 Private inheritance

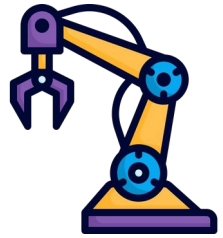
ROBOTICS CORNER

Object relationships

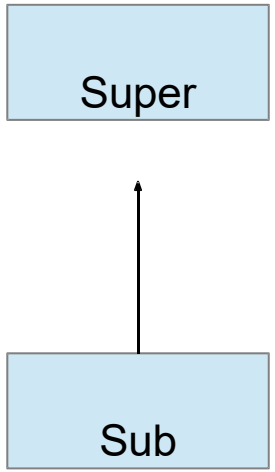


- The ***is-a*** relationship
  - Private inheritance
  - Multiple inheritance

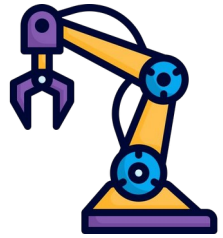
- The ***has-a*** relationship
  - Association
    - Composition (strong containment)
    - Aggregation (weak containment)

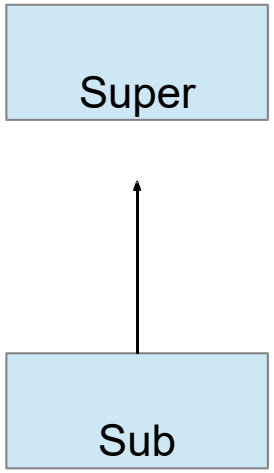


- The *is-a* relationship – *Client's view (1)*
  - works in only *one direction:*
    - every **Sub** object **is** also **a** **Super** one
    - but **Super** object **is not a** **Sub**




```
void foo1( const Super& s );
void foo2( const Sub& s);
Super super;
Sub sub;

foo1(super); //OK
foo1(sub);   //OK
foo2(super); //NOT OK
foo2(sub);   //OK
```

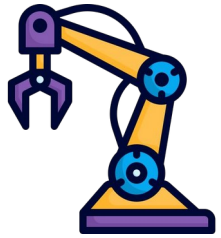


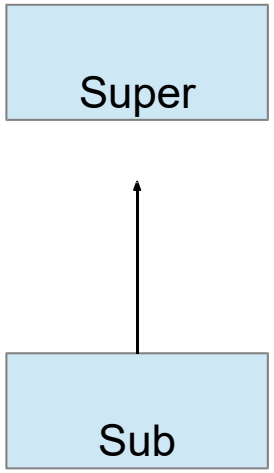


- The *is-a* relationship – *Client's view (2)*

```
class
Super
{  public:
   virtual void method1();
};
class Sub : public
Super{  public:
   virtual void method2();
};
```

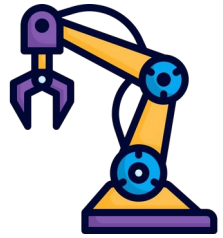
```
Super * p= new Super();
p->method1(); //OK

p = new Sub();
p->method1(); //OK
p->method2(); //NOT OK
((Sub *)p)->method2();//OK
```





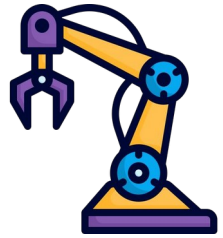
- The *is-a* relationship – *Sub-class's view*

  - the `Sub` class augments the `Super` class by **adding additional methods**
  - the `Sub` class **may override** the `Super` class **methods**
  - the subclass can use all the public and protected members of a superclass.



- The *is-a* relationship: *preventing inheritance* ***C++11***
  - `final` classes – cannot be extended

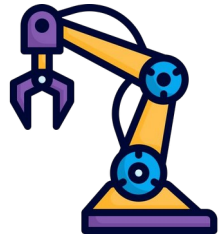
```
class Super final
{

};
```



- The *is-a* relationship: *a client's view of overridden methods(1)*
  - *polymorphism*

```
class
Super
{  public:
    virtual void method1();
};
class Sub : public
Super{  public:
    virtual void method1();
};
```

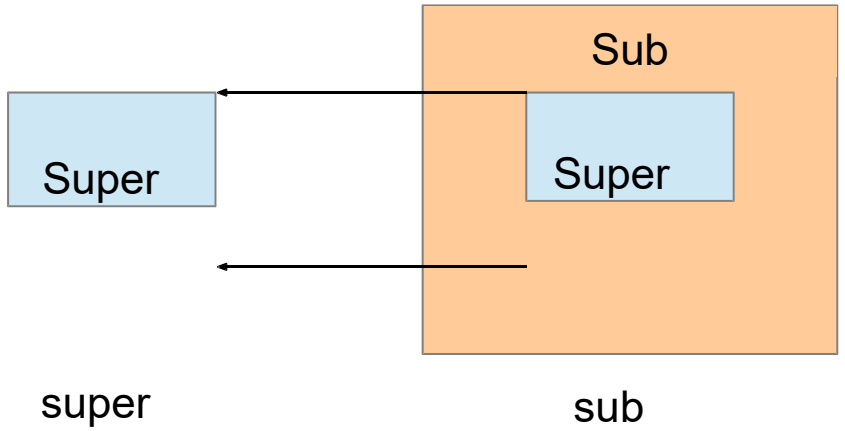
```
Super super;
super.method1(); //Super::method1()

Sub sub;
sub.method1();   //Sub::method1()

Super& ref =super;
ref.method1();   // Super::method1();

ref = sub;
ref.method1();   // Sub::method1();

Super* ptr =&super;
ptr->method1(); // Super::method1();

ptr = &sub;
ptr->method1();   // Sub::method1();
```

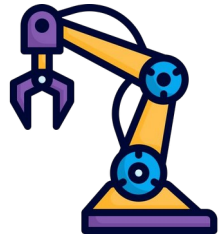


- The *is-a* relationship: *a client's view of overridden methods* (2)
  - object slicing

```
class
Super
{  public:
    virtual void method1();
};
class Sub : public
Super{  public:
    virtual void method1();
};
```

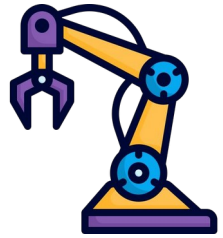
```
Sub sub;
Super super = sub;
super.method1(); // Super::method1();
```



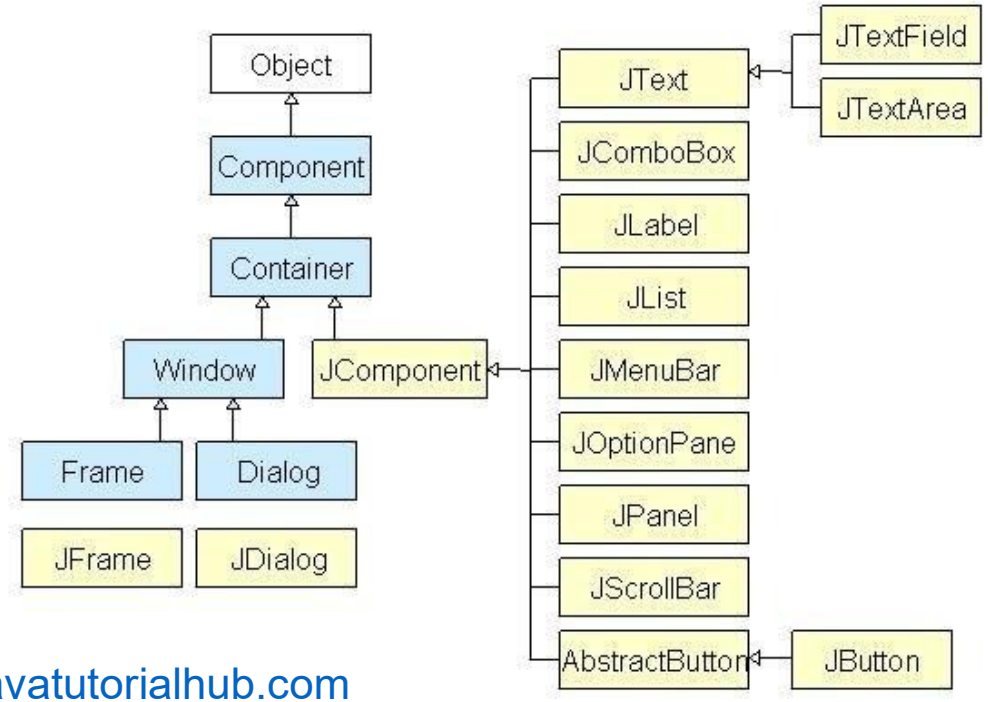


- The *is-a* relationship: *preventing method overriding* ***C++11***

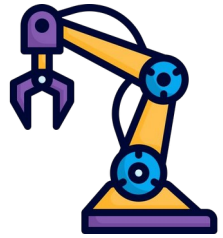
```
class
Super
{  public:
    virtual void method1() final;
};
class Sub : public
Super{  public:
    virtual void method1(); //ERROR
};
```



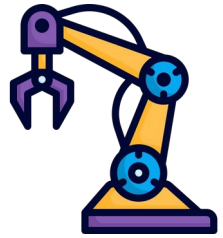
- Inheritance for polymorphism



www.javatutorialhub.com



- The *has-a* relationship

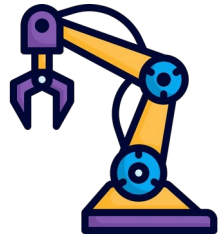


- Implementing the *has-a* relationship
  - An object **A** has an object **B**

```
class B;

class
A
{
privat
e:
    B b;
};
```

```
class B;

class
A
{
privat
e:
    B* b;
};
```

```
class B;

class
A
{
privat
e:
    B& b;
};
```

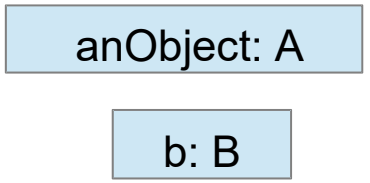


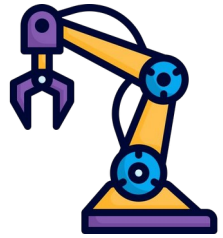
- Implementing the *has-a* relationship

  - An object **A** has an object **B**
    - **strong containment (composition)**

```
class B;

class
A
{
private:
    B b;
};
```

```
A anObject;
```



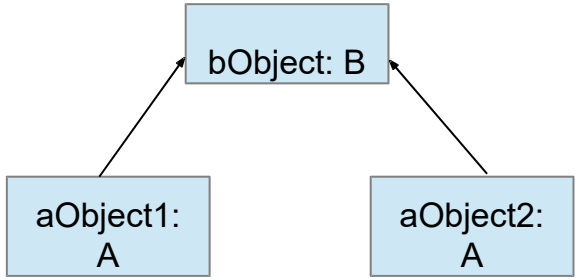


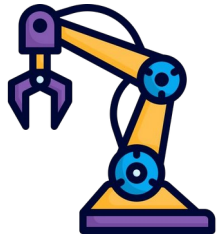
- Implementing the *has-a* relationship

  - An object **A** has an object **B**
    - **weak containment (aggregation)**

```
class B;

class
A
{
private:
    B& b;
public:
    A( const B& pb):b(pb){}
};
```

```
B bObject;
A aObject1(bObject);
A aObject2(bObject);
```





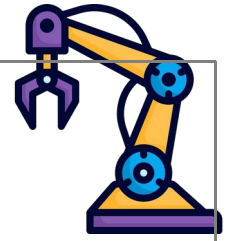
- Implementing the *has-a* relationship
  - An object **A** has an object **B**

weak containment

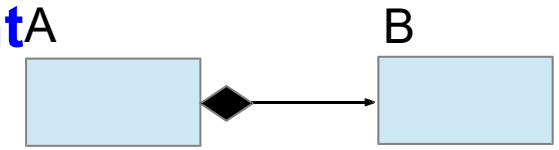
```
class B;

class A{
private:
     B* b;
public:
     A( B* pb):b( pb ){}
};
```

strong containment

```
class B;

class A{
private:
     B* b;
public:
     A(){
          b = new B();
     }
     ~A(){
          delete b;
     }
};
```



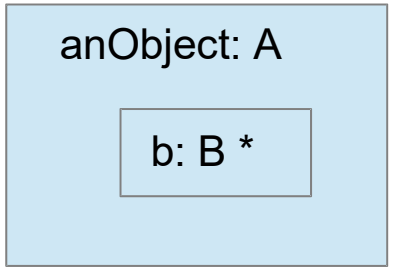
- Implementing the *has-a* relationship
  - An object **A** has an object **B** **strong containment**

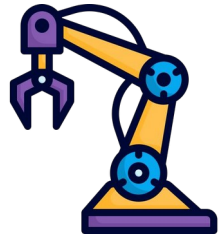



```
class B;

class A{  private:
     B* b;  public:
     A(){
          b = new B();
     }
     ~A(){
          delete b;
     }
};
```

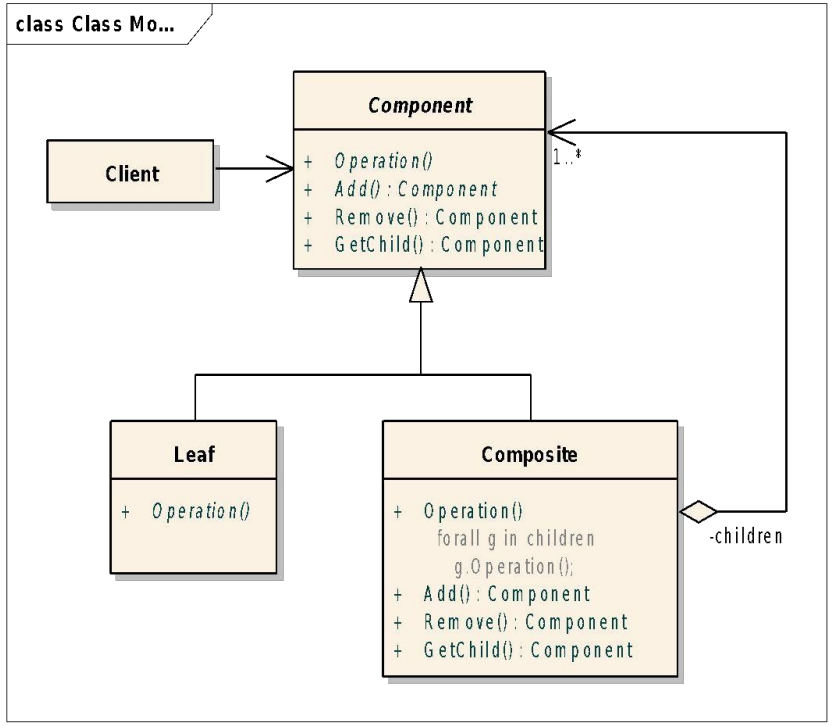
Usage:

```
A aObject;
```

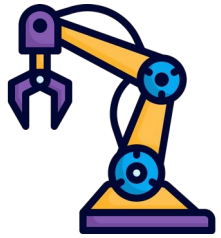


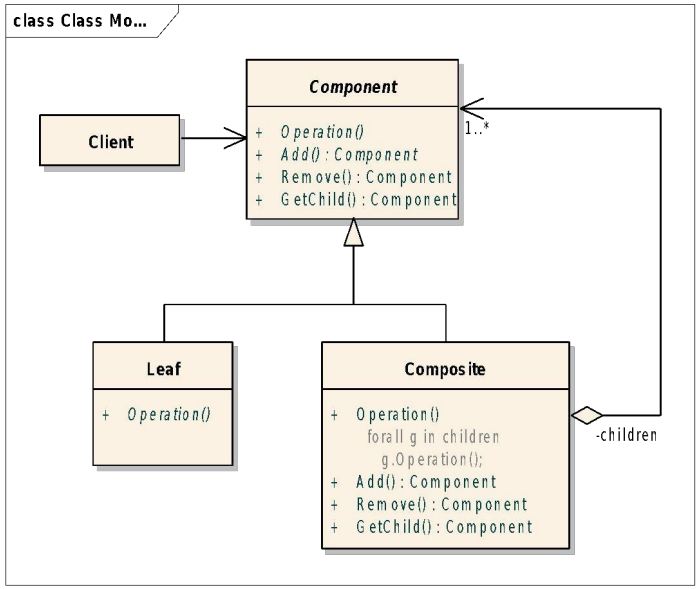


- Combining the *is-a* and the *has-a* relationships



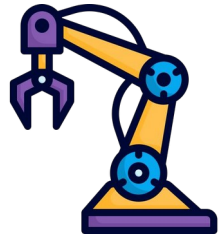
02 Has-a Relation





- Compose objects into tree structures to represent **part-whole hierarchies.**
- Let clients treat **individual objects** and the **composition of objects uniformly.**

ROBOTICS CORNER

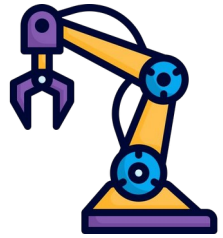


Composite Design Pattern

Examples:

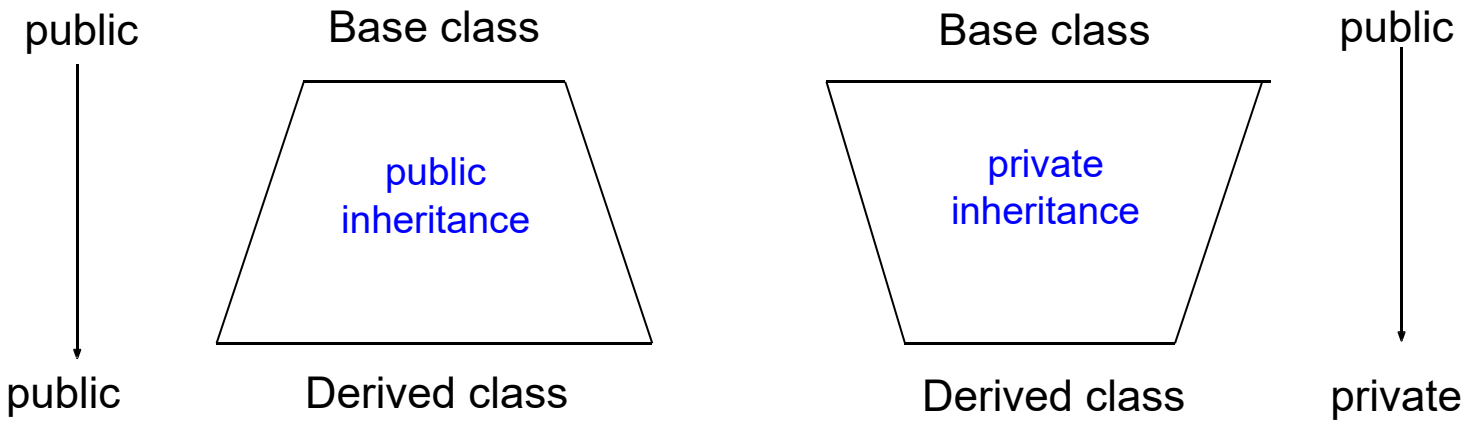
- Menu – MenuItem: Menus that contain menu items, each of which could be a menu.
- Container – Element: Containers that contain Elements, each of which could be a Container.
- GUI Container – GUI component: GUI containers that contain GUI components, each of which could be a container

Source: http://www.oodesign.com/composite-pattern.html



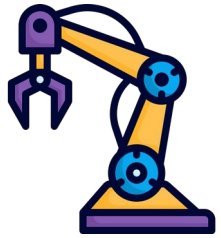
Private Inheritance

- another possibility for *has-a* relationship



Derived class **inherits** the base class behavior

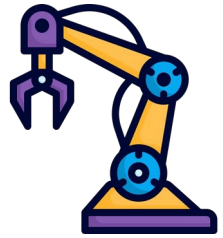
Derived class **hides** the base class behavior



Private Inheritance

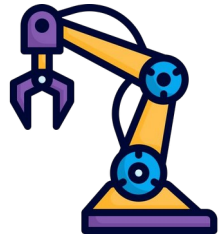
```
template <typename T>
class MyStack : private vector<T>
{  public:
    void push(T elem) {
        this->push_back(elem);
    }
    bool isEmpty() {
        return this->empty();
    }
    void pop() {
        if (!this->empty())this->pop_back();
    }
    T top() {
        if (this->empty()) throw out_of_range("Stack is empty");
        else return this->back();
    }
};
```

Why is **public inheritance** in this case dangerous???



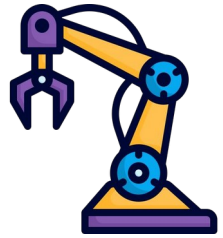
Non-public Inheritance

- it is very rare;
- use it cautiously;
- most programmers are not familiar with it;



What does it print?

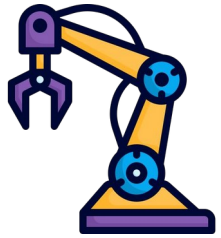
```
class
Super
{  public:
    Super(){}
    virtual void someMethod(double d)
            const{  cout<<"Super"<<endl;
    }
};
class Sub : public
Super{  public:
    Sub(){}
    virtual void someMethod(double
            d){  cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```



What does it print?

```
class
Super
{  public:
    Super(){}
    virtual void someMethod(double d)
    }       const{  cout<<"Super"<<endl;
};
class Sub : public Super{
public:
    Sub(){}
    virtual void someMethod(double
            d){  cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

creates a new method, instead of overriding the method



The override keyword C++11

```
class
Super
{  public:
    Super(){}
    virtual void someMethod(double d)
            const{  cout<<"Super"<<endl;
    }
};
class Sub : public
Super{  public:
    Sub(){}
    virtual void someMethod(double d) const
            override{  cout<<"Sub"<<endl;
    }
};

Sub sub; Super super;
Super& ref = sub;ref.someMethod(1);
ref = super; ref.someMethod(1);
```

Thank You

Do you have any questions?

ROBOTICS CORNER

01211626904

www.roboticscorner.tech

Robotics Corner

Robotics Corner

Robotics Corner

Robotics Corner