CIS129
Advanced Computer Programming

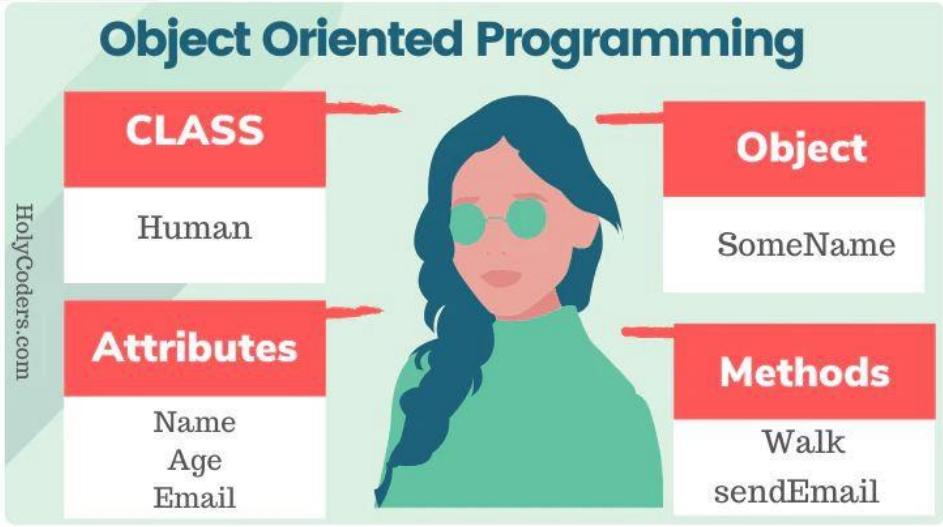
Chapter 9: Introduction to Object-Oriented Programming (OOP)

Mr. Horence Chan

The Basic Ideas of OOP

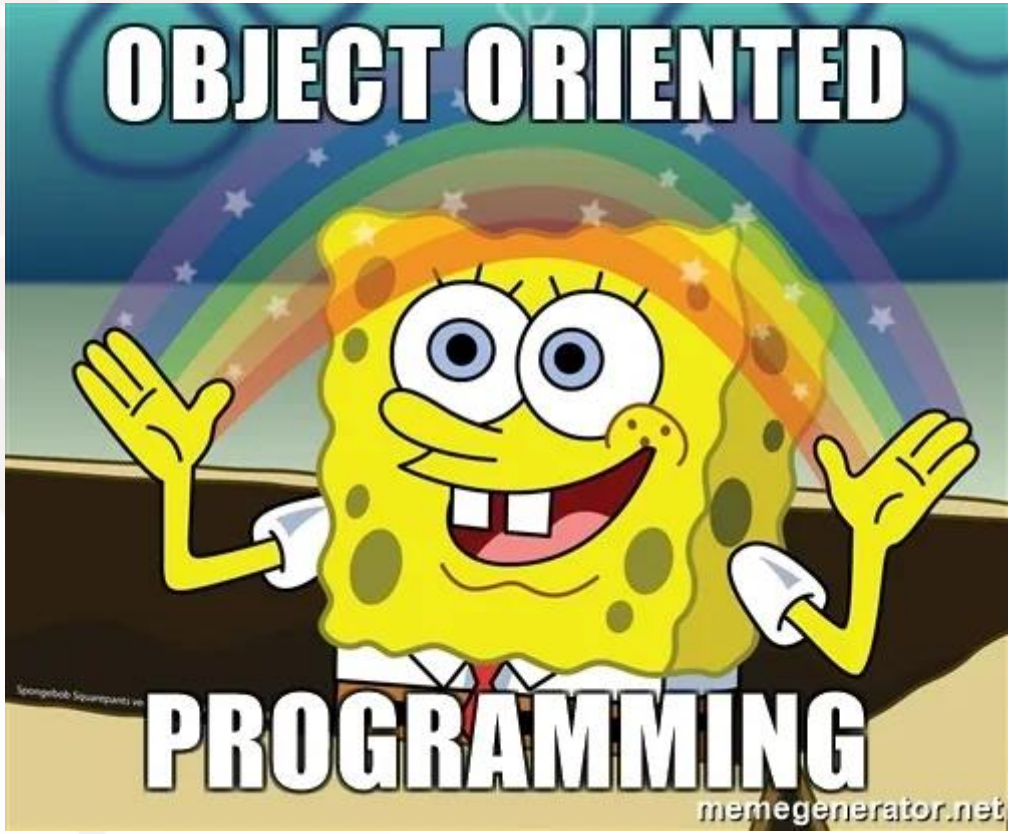
- Object-Oriented Programming (OOP) is about creating objects that contain both data and functions.
- For example:

Class	Object	Attribute	Methods
Fruit	Apple, Orange, Mango	Size, Color	How to plant How to eat
Student	Peter Parker, Mary Jane	Age, Program, Grade	Do assignment, Attend class, Sleep



The Basic Ideas of OOP

- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY (Don't Repeat Yourself), and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

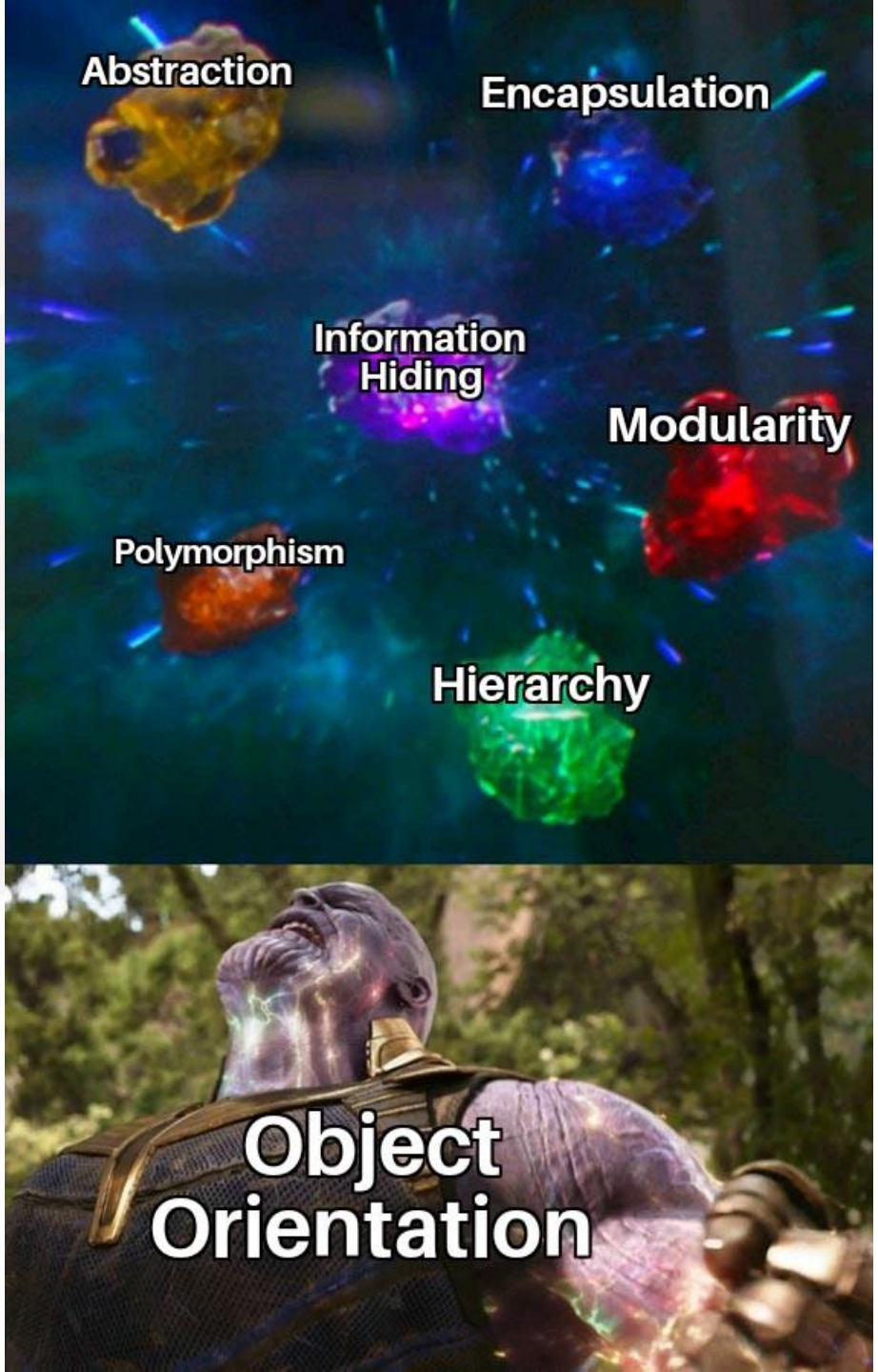


Features of OOP

- **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects

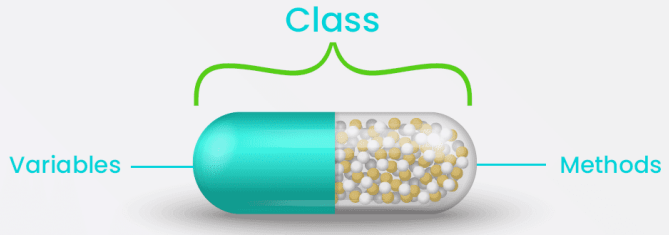
- **Inheritance**: allowing code to be reused between related types

- **Polymorphism**: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type



Encapsulation

- The purpose of encapsulation is to make sure that "**sensitive**" data is **hidden** from users.
- To achieve this, we must declare class variables/attributes as _____ (cannot be accessed from outside the class).
- If we want others to read or modify the value of a private member, we can provide _____ get and set methods.



Encapsulation

- Analogy:
- A lecturer from the school of computer science wants to know his/her student grade in the English class.

- The lecturer is not allowed to directly access student grade in English class.

- Instead, the lecturer need to contact colleagues in the school of humanity and language and then request them to give the student grade records.

Encapsulation

```
#include <iostream>
using namespace std;
class Employee {
  public:
    int salary;
};
int main() {
  Employee Emp1;
  Emp1.salary = 25000;
  cout << "Employee 1 monthly salary: $"
     << Emp1.salary;
  return 0;
}
```

- For example, salary of an employee is a **sensitive** data in a company.
- It is not recommended to modified the salary of an employee by simply `Emp1.salary = 25000;` in the main function.
- We should not set the **attributes** (salary) to _____.

Output:

Employee 1 monthly salary: $25000

Encapsulation

```
#include <iostream>
using namespace std;
class Employee {
  private:
     int salary;
  public:
      void setSalary(int s) {
        salary = s;
      }
      int getSalary() {
      return salary;
      }
};
```

- We should create a class to set the **attributes** (salary) to _____, which have restricted access.
- Then create some public **functions/methods** that can get and set some attributes.
- The public `setSalary()` method takes a parameter (`s`) and assigns it to the `salary` attribute.
- The public `getSalary()` method returns the value of the private `salary` attribute.

Encapsulation

```
int main() {
   Employee Emp1;
   Emp1._____(25000);
   cout << "Employee 1 monthly salary: $"
      << Emp1._____();
   return 0;
}
```

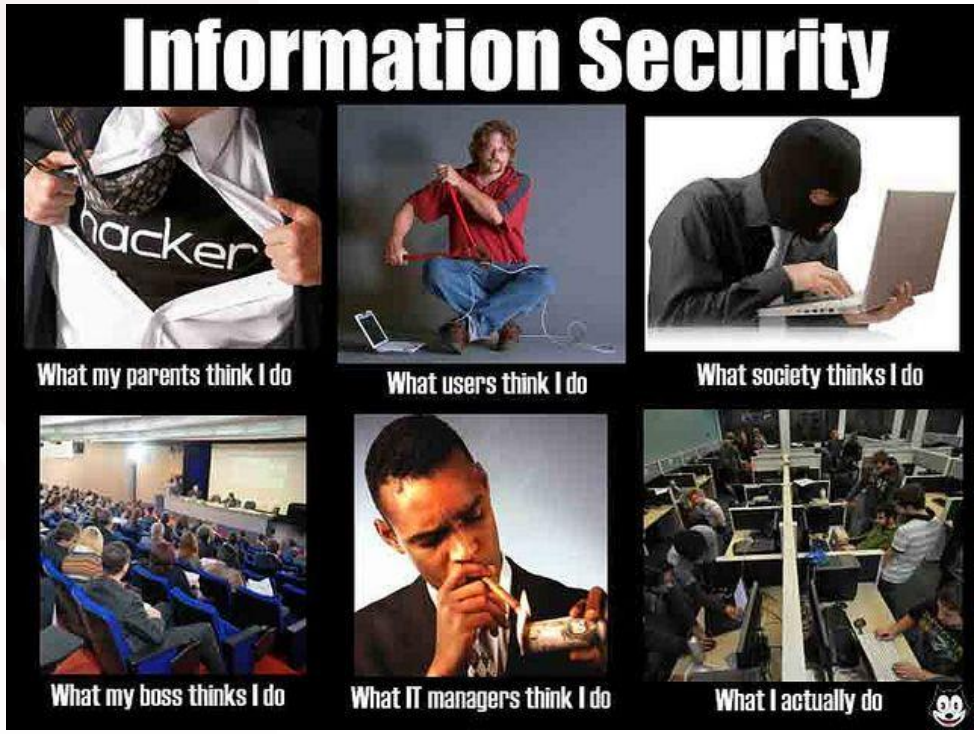
- Inside `main()`, we create an **object** of the `Employee` class. (`Emp1`)
- Then we can use the `setSalary()` method to set the value of the private attribute to `25000`.
- Next we call the `getSalary()` method on the object to return the value.

Output:

Employee 1 monthly salary: $25000

Encapsulation

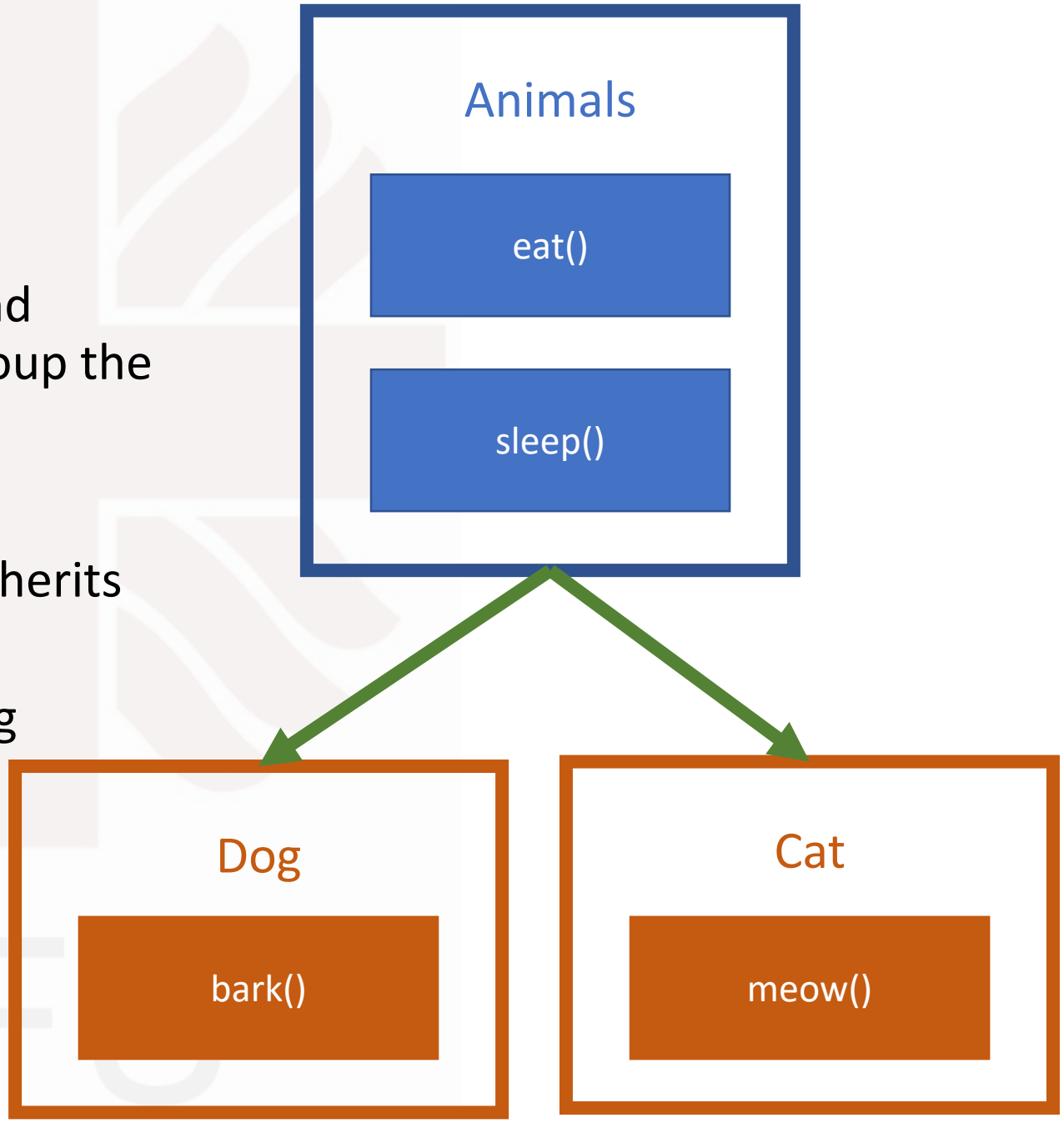
- It is considered good practice to declare your class attributes as _____ (as often as you can).
- Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased **security** of data



Inheritance

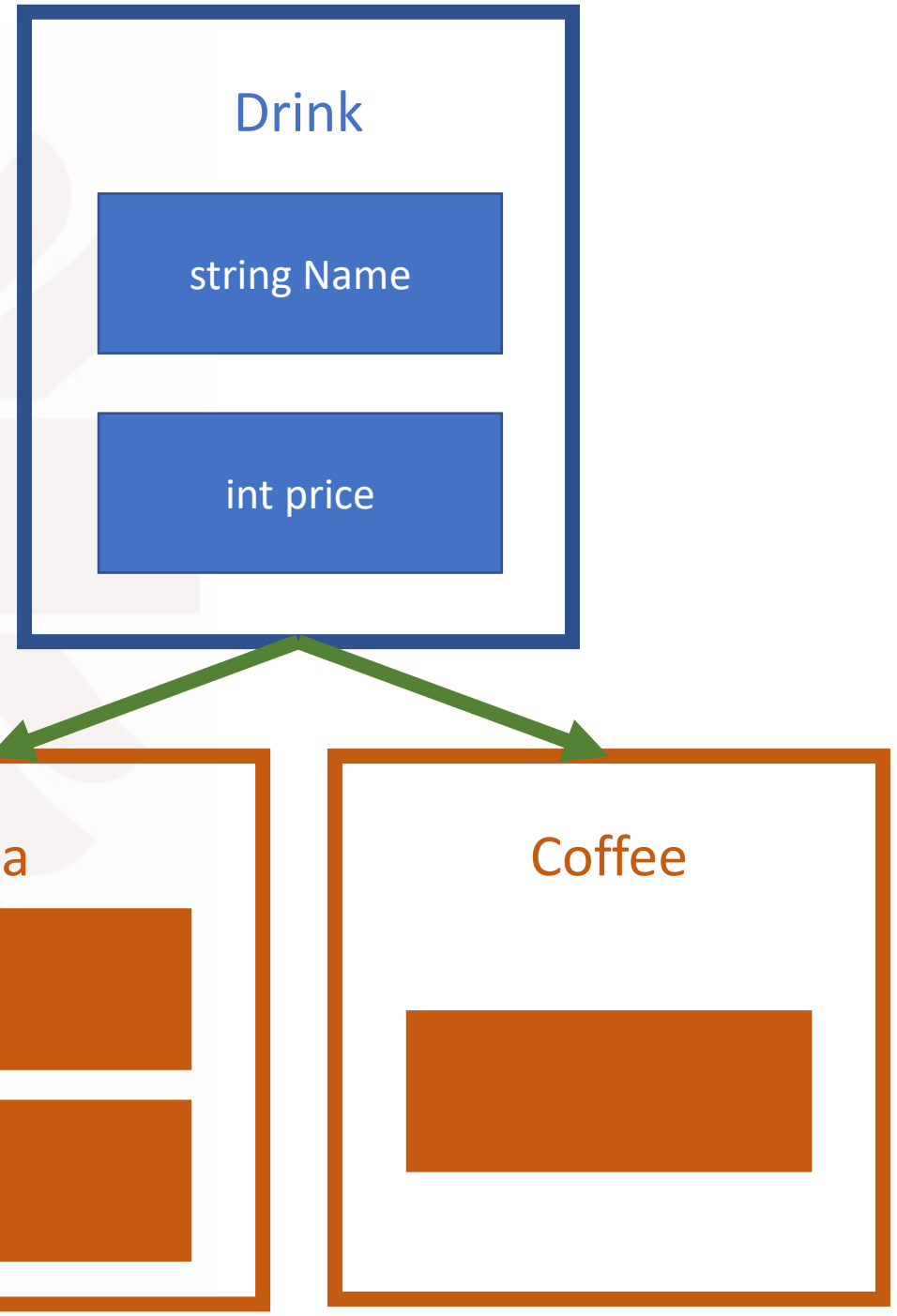
- In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- _____ class (child): the class that inherits from another class
- _____ class (parent): the class being inherited from
- To inherit from a class, use the _____ symbol.



Inheritance

```
class Drink{
     public:
         string Name;
         int price;
};
class Tea: public Drink{
    public:
         string topping;
         void tea_msg() {
        cout << "Enjoy your tea!" <<endl; }
};
class Coffee: public Drink{
    public:
         void coffee_msg() {
        cout << "Enjoy your coffee!"
                << endl; }
};
```



Inheritance

```
class Drink{
     public:
         string Name;
         int price;
};
class Tea: public Drink{
    public:
         string topping;
         void tea_msg() {
         cout << "Enjoy your tea!" <<endl; }
};
class Coffee: public Drink{
    public:
         void coffee_msg() {
         cout << "Enjoy your coffee!"
                << endl; }
};
```

- Class `Drink` is a _____ class, the code in the **based** class can be used by the **derived** class
- Class `Tea` and `Coffee` are _____ class, they can use the code from `Drink`
- ":" is used to show that class `Tea` and `coffee` are inherit from class `Drink`
- Code within a _____ class can be used by that **derived** class only, the code can't used by the **based** class or another class

Inheritance

```
int main(){
    Tea product1;
    product1.Name = "Milk Tea";
    product1.price = 25;
    product1.topping = "Bubble";
    cout << product1.topping << " "
         << product1.Name << ": $"
         << product1.price << endl;
    product1.tea_msg();
    Coffee product2;
    product2.Name = "Mocha";
    product2.price = 30;
    cout<< product2.Name<<": $"
         << product2.price << endl;
    product2.coffee_msg();
    return 0;}
```

- `product1` is an object of `Tea`
- To use the attribute of `Drink`, simply type `product1.Name`, `product1.price`, etc
- Since `product2` is not an object of `Tea`, `product2` can't use the attribute

---

Output:

Bubble Milk Tea: $25

Enjoy your tea!

Mocha: $30

Enjoy your coffee!

Multilevel Inheritance

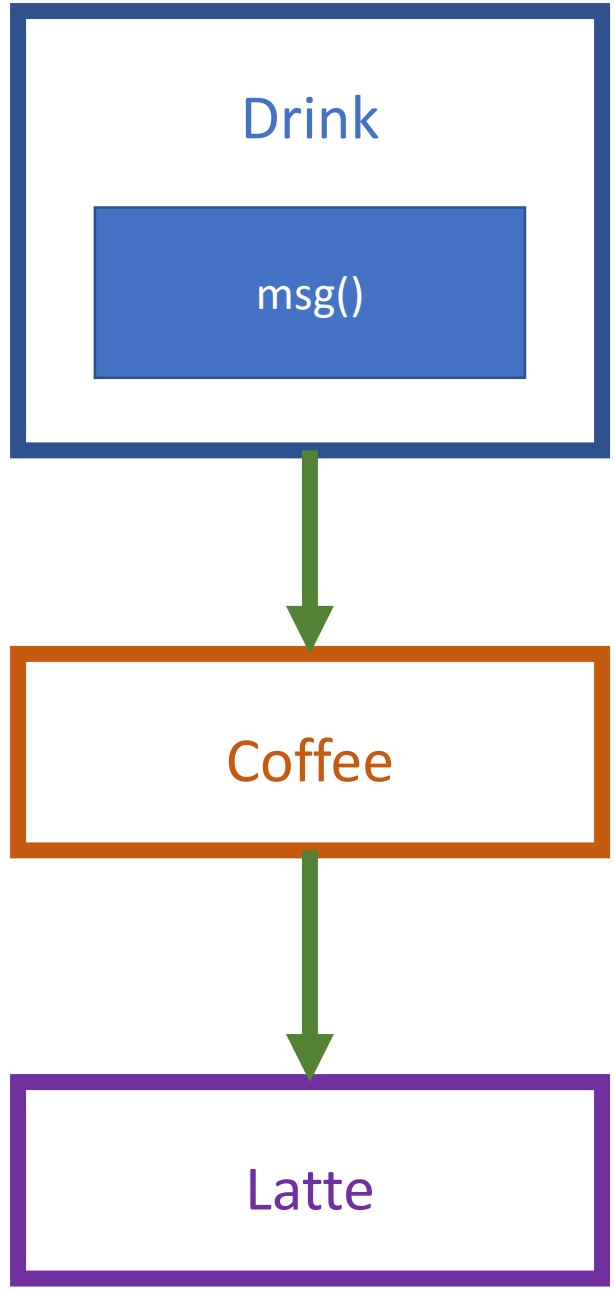
```
#include <iostream>
using namespace std;
class Drink{
      public:
         void msg() {
         cout << "Enjoy your drink!" <<endl; }};
class Coffee: public Drink{
};
class Latte: public Coffee{
};

int main(){
      Latte product1;
      product1.msg();
      return 0;
}
```

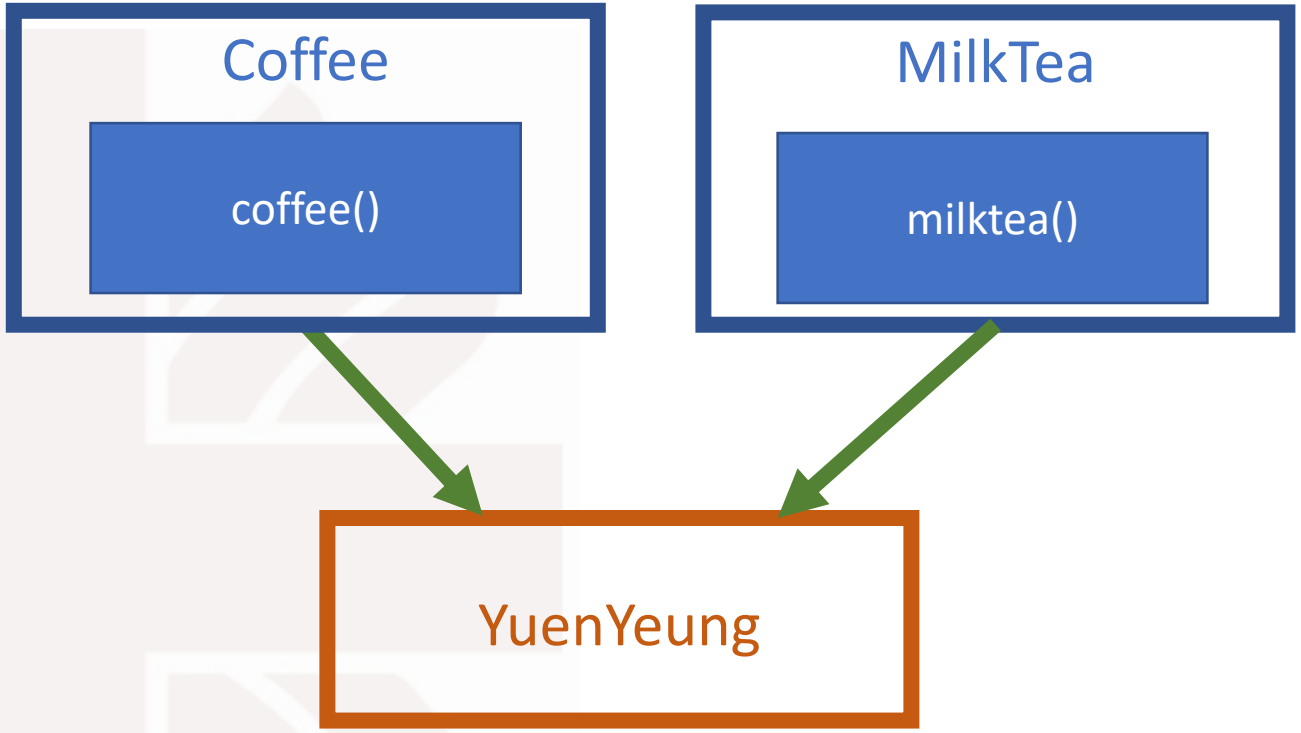
- A class can also be derived from one class, which is already _____ from another class.
- For example, `Latte` is derived from class `Coffee` (which is derived from `Drink`).

Output:

Enjoy your drink!



Multiple Inheritance




```
class MilkTea{
      public:
         void milktea() {
         cout << " Milk Tea"; }};
class Coffee{
      public:
         void coffee() {
         cout << " Coffee";}};
class YuenYeung: public MilkTea, public Coffee {
};
int main(){
      YuenYeung product1;
      cout << "YuenYeung is";
      product1.coffee();
      product1.milktea();
      return 0;
}
```

- A class can also be derived from more than one _____, using a comma (,)
- For example, class `Yuenyeung` is derived from class `MilkTea` and `Coffee`

Output:

Yuenyeung is Coffee Milk Tea

Access Specifiers: Protected

- **Protected**: Members **cannot** be accessed from outside the class, however, they can be accessed in _____ classes.
- **Private**: Members cannot be accessed from outside the class, including _____ classes.

Specifiers	Own Class	Derived Class	Main Function
Public	Yes	Yes	Yes
Private	Yes		
Protected	Yes		

Inheritance: “Protected” Access Specifiers

```
class Employee {

  _____________
    int salary;
};
class Clerk: public Employee {
  public:
    void setSalary(int s) {salary = s;}
    int getSalary() {return salary;}
};
int main() {
  Clerk clerk1;
  clerk1.setSalary(18000);
  cout << "Salary of Clerk 1: "
       << clerk1.getSalary() << endl;
  return 0;
}
```

- Class `Clerk` can access _____ attribute (`salary`) in class `Employee` through functions `setSalary()` and `getSalary()`

Output

Salary of Clerk 1: 18000

Inheritance: Overriding

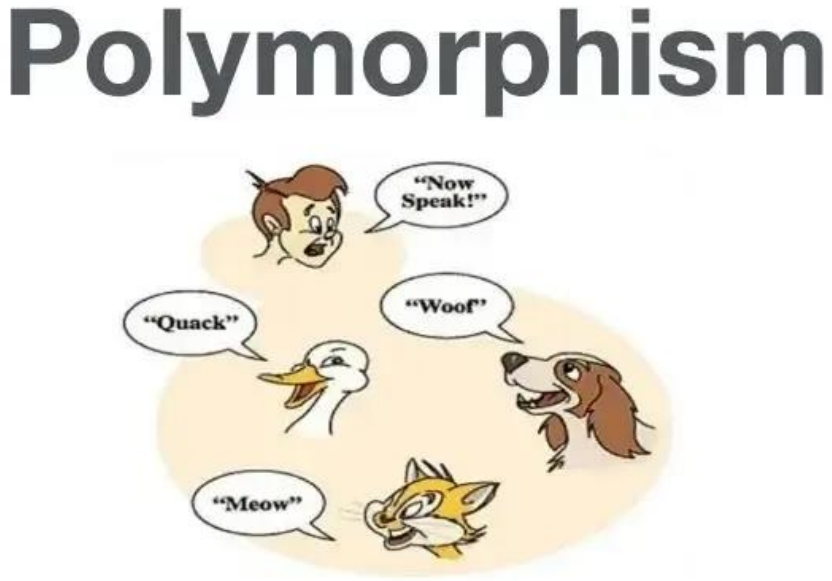
```
class Drink{
     public:
        void msg() {
        cout << "Enjoy your drink!" <<endl; }};
class Tea: public Drink{
    public:
       void msg() {
        cout << "Enjoy your tea!" <<endl; }
};
class Coffee: public Drink{
};
int main(){
     Tea product1;
     product1.msg();
     Coffee product2;
     product2.msg();
     return 0;}
```

- In class drink, function `msg()` contains an output statement
- If we would like to change `msg()` output statement for a certain derived class, rewrite the code in the function inside a derived class.
- Polymorphism reuse attributes and methods of an existing class when you create a new class.

Output:

Polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- **Inheritance** lets us inherit attributes and methods from another class.
- **Polymorphism** uses those methods to perform different tasks.
- This allows us to perform a single action in different ways.



Polymorphism: Virtual Function

```
class Drink {
public:
    virtual void msg() {
        cout << "Enjoy your drink!\n";}
};
class Tea : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your tea!\n";}
};
class Coffee : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your coffee!\n";}
};
class Unknown : public Drink {
};
```

- A virtual function is a member function which is declared within a _____ class
- It is re-defined (Overridden) by a derived class.
- To re-defined a virtual function in derived class, we need to write "_____"

When I override my parent's methods

Polymorphism: Virtual Function

```
int main() {
    Drink * drink;
    Tea tea;
    Coffee coffee;
    Unknown u;

    drink = & tea;
    drink -> msg();

    drink = &coffee;
    drink -> msg();

    drink = & u;
    drink -> msg();
    return 0;
}
```

- To call a virtual function for that object and execute the derived class's version of the function
- We need to refer to a derived class object using a _____ or a _____ to the base class
- Selecting the correct function at runtime is called *dynamic dispatch*

Polymorphism: Virtual Function

```
int main() {
    Drink * drink;
    Tea tea;
    Coffee coffee;
    Unknown u;

    drink = & tea;
    drink -> msg();

    drink = &coffee;
    drink -> msg();

    drink = & u;
    drink -> msg();
    return 0;
}
```

- To call the virtual function, we declared a _____ `* drink` in base class
- Then we **reference** the derived class object (e.g. `drink = & tea;`)
- `drink -> msg();` is used to **dereferences** and gets a member.

Output:

Enjoy your tea!

Enjoy your Coffee!

Enjoy your drink!

Polymorphism: Pure Virtual Function

```
class Drink {
public:
    virtual void msg() = 0;
};
class Tea : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your tea!\n";}
};
class Coffee : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your coffee!\n";}
};
/*class Unknown : public Drink {
};*/
```

- A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have an implementation we only declare it.
- A pure virtual function is declared by assigning _____ in the declaration.

Polymorphism: Pure Virtual Function

```
class Drink {
public:
    virtual void msg() = 0;
};
class Tea : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your tea!\n";}
};
class Coffee : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your coffee!\n";}
};
/*class Unknown : public Drink {
};*/
```

- Virtual function `msg()` is declared to _____.
- So it becomes a pure virtual function
- This implies that we can no longer create an instance of `Drink`
- We can only create instances of its **derived** classes which do implement the `msg()` method.

Polymorphism: Pure Virtual Function

```
class Drink {
public:
    virtual void msg() = 0;
};
class Tea : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your tea!\n";}
};
class Coffee : public Drink {
public:
    virtual void msg() override {
        cout << "Enjoy your coffee!\n";}
};
/*class Unknown : public Drink {
};*/
```

- `Drink` is then an _____ **class** : which defines only an interface, but doesn't actually implement it, and therefore cannot be instantiated.
- Note that class `Unknown` does not have re-defined function,
- We can't use this class, else compilation error occur.

Polymorphism: Pure Virtual Function

```
int main() {
    Drink * drink;
    Tea tea;
    Coffee coffee;
    //Unknown u;

    drink = & tea;
    drink -> msg();

    drink = &coffee;
    drink -> msg();

    //drink = & u;
    //drink -> msg();
    return 0;
}
```

- Note that we cannot declare variable 'u' to be of abstract type 'Unknown'

Output:

Enjoy your tea!

Enjoy your Coffee!