Object Oriented Programming

Object-oriented programming… why?

Why did you write functions? What did it accomplish?

Object-oriented programming… why?

Why did you write functions? What did it accomplish?

- You could reuse functions anywhere with different arguments
- It reduced the size of our main function
- You could compute a value from the function and return it

Sometimes, there are too many functions all over the place! And it gets unreadable, they all take may parameters and it’s super confusing.

Object-oriented programming… why?

- Variables too! They are so many all over the place and it’s difficult to keep track of it.
- Simple solution to group related variables together?... Any suggestions?

We could use global variables, but remember any function can access them and change their values - **no single point of responsibility**

Note: Use global variables only when you have global constants to define. For example, `const double PI = 3.14;`

Pokemon

Design and develop an interactive Pokemon game that allows you to engage in battles, capture wild pokemons, train, evolve and earn badges.

Pokemon

- A scenario: Think of a battle between 2 pokemons. How do you model that?
- Pikachu:
  - `pikachu_type = “electric”;`
  - `pikachu_hp = 60;`
  - `pikachu_level = 20;`
  - `pikachu_attack_points = 95;`
  - `pikachu_defense_points = 75;`
- Meowth
  - `meowth_type = “normal”;`
  - `meowth_hp = 50;`
  - `meowth_level = 25;`
  - `meowth_attack_points = 75;`
  - `meowth_defense_points = 80;`

```
battle(pikachu_type, pikachu_hp, pikachu_level, pikachu_attack_points,
pikachu_defense_points, meowth_type, meowth_hp, meowth_level,
meowth_attack_points, meowth_defense_points);
```

Object-oriented programming

- This function keeps track of attributes of two pokemons, and the logic for battle

```
battle(pikachu_type, pikachu_hp, pikachu_level, pikachu_attack_points,
pikachu_defense_points, meowth_type, meowth_hp, meowth_level,
meowth_attack_points, meowth_defense_points);
```

How do we make this simpler?

Object-oriented programming

- This function keeps track of attributes of two pokemons, and the logic for battle

```
battle(pikachu_type, pikachu_hp, pikachu_level, pikachu_attack_points,
pikachu_defense_points, meowth_type, meowth_hp, meowth_level,
meowth_attack_points, meowth_defense_points);


battle(pikachu, meowth); // much simpler
```

Object-oriented programming (OOP)

"A programming **style** where we model objects (real world entities) into self-contained units."

You can model cars, games, classrooms, furnitures, bank accounts, cash registers, buildings etc.

We use a keyword named `class` to create the *blueprint* for objects

Classes and Objects

- **A fundamental concept:** Most objects in real world has attributes of data, and methods of interacting with them

Classes and Objects

- Let's consider a car
- What are some attributes of a car?

- And how do you interact (or interface with) a car?

Classes and Objects

- Let’s consider a car
- What are some attributes of a car?
  - Make
  - Model
  - Price
  - Year
  - Top Speed
  - Horse power
- And how do you interact (or interface with) a car?
  - start the engine
  - drive the car
  - stop the car
  - play radio
  - reverse the car

Classes and Objects

- Coming back to Pokemon example,
    - Both pikachu and meowth have the same kinds of data and methods associated with them!
    - It would be nice if there was a type of variable with all that info built into it

We call that data type as a ***class***, and the variable (or variables) as ***objects***

Classes and Objects

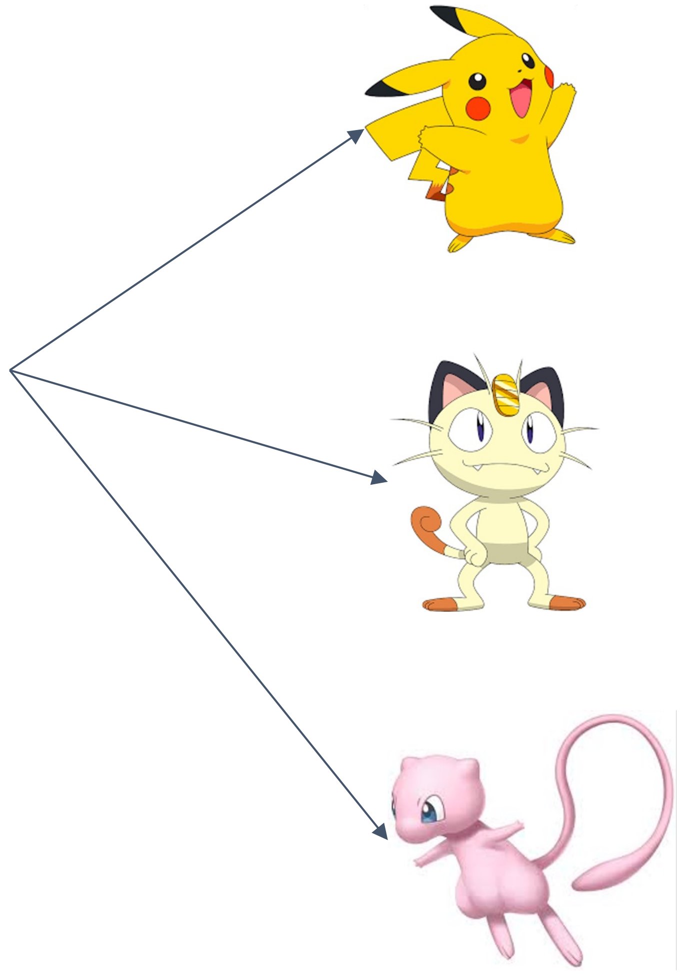
1. Creating a class (the blueprint for objects - *what they have*, and *what can they do*)

```
class Pokemon {
   private:
         string name;
         string type;
         int hp;
   public:
         void battle();
         void evolve();
};
```

Classes and Objects

```
class Pokemon {
   private:
      string name;
      string type;
      int hp;
   public:
      void battle();
      void evolve();
};
```

Blueprint



They all have the same pieces of information (attributes) and they all can do the same thing (functions).

The access modifiers: public and private

Designing a class is tricky. When you think of an object, consider the two points

- How do the objects of the class interface with outside world?
  public interface
- What are the attributes that those objects maintain?
  private data

Encapsulation and Interface

- public:
    - accessible outside the class definition
    - member functions
- private:
    - not accessible outside the class definition
    - data members

Encapsulation - Objects provide a public interface, while hiding the implementation details internally.

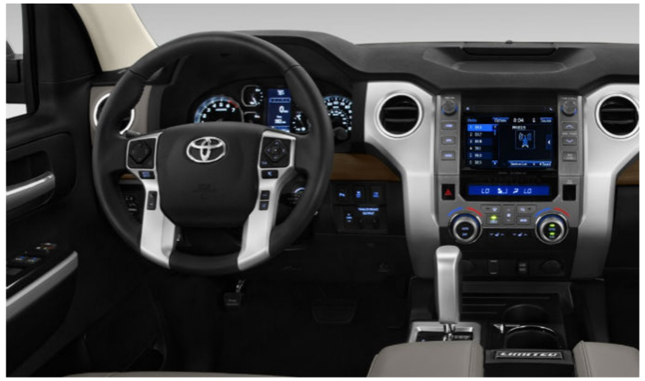
Encapsulation and Interface

The private data members are said to be encapsulated because:

- they are hidden from other parts of the program
- accessible only through the class’s member functions.
  - hides all the nitty-gritty details so people using the class don’t have to worry about it. Ex: Think about `string` class

Encapsulation and Interface

- Think about an interface of a car
- You can drive, change gears, change volume, tune radio etc. without requiring to know what happens behind the scenes.
- So when you develop a class, you are keeping in mind what needs to be available outside versus what you need to track off internally! **Encapsulation**



A generic class interface

```
class NameOfClass
{
  public:
     // the public interface

  private:
     // the data members
};
```

A generic class interface

```
class NameOfClass
{
  public:
    // the public interface

  private:
    // the data members
};
```

Use CamelCase for the names of classes

A generic class interface

```
class NameOfClass
{
  public:
    // the public interface

  private:
    // the data members
};
```

Use CamelCase for the names of classes

Any part of our program should be able to call the member functions.
→ they go in the public interface

A generic class interface

```
class NameOfClass
{
   public:
      // the public interface


   private:
      // the data members
};
```

Use CamelCase for the names of classes

Any part of our program should be able to call the member functions.
→ they go in the public interface

Data members are defined in the *private section* of the class. Only member functions (within our class) can access the data members. They're hidden from
the rest of the program
→ they go in the private section of the class

Member functions

1. **Mutators** are member functions that modify the data members

- Set a data member / attribute to a given value
- Clear out a data member value

1. **Accessors** are member functions that query a data member(s) of the object, and returns the value(s) to the user

- Get the value of a data member / attribute

Designing a class: Pokemon

```
class Pokemon
{
public:
   int getHP() const;
   void setHP(int hp);
   double getType() const;
   void evolve();
private:
   // data members will go here
};
```

Question: Which member functions are getters (accessors) and which are setters (mutators)?

Designing a class: Pokemon

```
class Pokemon
{
public:
    int getHP() const;
    void setHP(int hp);
    double getType() const;
    void setLevel(int level);
private:
    // data members will go here
};
```

setters because they change the value of data members

Question: Which member functions are getters (accessors) and which are setters (mutators)?

Designing a class: Pokemon

```
class Pokemon
{
public:
   int getHP() const;
   void setHP(int hp);
   double getType() const;
   void setLevel(int level);
private:
   // data members will go here
};
```

getters because they just return the value to the caller

getters only report the values of data members, and never alter them, we declare these functions to be **const** so they can't mess our stuff up

Dot Notation

You call the member functions by first creating a variable of type **Pokemon** and then using the dot notation:

```
Pokemon pikachu;
...
pikachu.setHP();
pikachu.setLevel(20);
...
string type = pikachu.getType();
cout << "Type of my pokemon: " << type << endl;
```

Representing objects in memory

- Every `Pokemon` object has its own copy of these data members

```
Pokemon pikachu;
Pokemon meowth;
```

Quiz

Which of the following will work?

- pikachu.level_ = 5;
- cout << pikachu.type_;
- pikachu.setHP = 7;
- pikachu.setHP(7);

Quiz

Which of the following will work?

- pikachu.level_ = 5; $\longrightarrow$ Private member, cannot be accessed
- cout << pikachu.type_; $\longrightarrow$ Private member, cannot be accessed
- pikachu.setHP = 7; $\longrightarrow$ setHP is a function
- pikachu.setHP(7);

Abstraction

- When you drive/ride, do you need to know how it works inside of an automobile? *All you need to know is what controls to use.*
- In similar terms, programmers using objects should only care about what interfaces to use. Much of the implementation details is ***abstracted away***
- For example, do you care how a dish is prepared at a restaurant. No right! You only want the dish at your table. *The preparation of the dish is abstracted away from you (the customer)*

Change in the way we code from now

- To achieve abstraction and encapsulation, embrace the following changes in your programming style
- We separate our program into 3 files
  - header file
  - implementation file
  - driver file (program)

Header Files

```
class Pokemon
{
public:
    void setHP(int hp);
        int getHP();
    int getLevel();
    void setLevel(int level);
private:
    string name_;
        int hp_;
        int level_;
};
```

Header files will only consist of class definition, with some header guards!

The order of variables and functions inside of class is not important

Implementation Files

- Start with the setHP() member function:

```
void Pokemon::setHP(int hp) {
     hp_ = hp; // set the value of hp_ to hp
}
```

- Note the extra `Pokemon::` in the definition of the function. It's telling the compiler that `setHP` is a function within the `Pokemon` class
- Complete other member functions (the getters and setters)

Implicit Parameters

- When we call `setHP(50)`, how does it know which `hp_` to update?

```
Pokemon pikachu, mew;
 … [stuff happens] …
pikachu.setHP(50);
```

`pikachu` → pass as an implicit parameter into the `setHP()` function

Implicit Parameters

```
pikachu.setHP(20);


void Pokemon::setHP(int hp) {
     hp_ = hp;
}
```

Exactly the same way str1.length() gives you the size of string str1 and not length of some other string.

Constructors

- A constructor is a special **member function** that initializes the data members of an object.

- The constructor function is called when the object is declared for the first time.

- Instead of calling setter functions to feed in the values to our object, we can use a constructor to set initial values.

Constructors

```
class Pokemon {
   private:
         string name_;
      int hp_;
   public:
      Pokemon() {
         name_ = “”;
         hp_ = 0;
      }
};
```

What’s the name of our constructor?

What did we do inside the constructor?

Constructors

```
class Pokemon {
   private:
            string name_;
      int hp_;
   public:
      Pokemon() {
            name_ = “”;
            hp_ = 0;
      }
};
```

What’s the name of our constructor?
It’s the same name as that of the class!

What did we do inside the constructor?
We set the data members of our object

Constructor’s Code Expanded

```
Pokemon() {
         name_ = “”;
         hp_ = 0;
         // anything more you add here
   }
```

1. No return type on the function
2. Function name is the same as that of class name
3. May/may not contain input parameters
4. Mostly used to set members of the object

Default Constructors

- When you don’t write a constructor, the compiler provides you with a default constructor. There is no guarantee of the initial values of data members. (*they are arbitrarily set*)
- `Pokemon p1; // default constructor is called`
- The data members of p1 could be anything!

- To fix this, write a constructor of your own and set the initial values as you need!

Default vs Parameterized Constructors

```
Pokemon() {
      name_ = “”;
      hp_ = 0;
      level_ = 5;
}
```

VS

```
Pokemon(string name, int hp, int level) {
      name_ = name;
      hp_ = hp;
      level_ = level;
}
```

- Constructors that don’t take in parameters
- `Pokemon p1;`

- Constructors that take in parameters
- `Pokemon p1(“pikachu”, 50, 5);`

Have you noticed you can create a string this way?

```
string car_name(“tesla”);
```

Overloaded Constructors

- You can have as many constructors for a class as you want
- But the names of those functions are same!
- `Pokemon() and Pokemon(string name, int hp, int level);`

- Your program still works because of a concept termed as **polymorphism**
- **Polymorphism -** Represent same entity in more than one way!

Polymorphism

- Just as you order a pizza and customize it as you need

`Pizza pizza;`

`Pizza pizza(“olives”, “onions”);`

`Pizza pizza(“mushrooms”, “chicken”, “cheese”);`

On the same lines, you can write multiple constructors with different parameter types or different number of parameters