Object Orientation

A Crash Course Intro

What is an Object?

- An object, in the context of object-oriented programming, is the association of a *state* with a set of *behaviors*.
  - State: its fields, or "member variables"
  - Behaviors: its associated methods, or "member functions."

A First Object

```
using namespace std;

// a very basic C++ object
class Person
{
    public:
        string name;
        int age;
}
```

A First Object

```
using namespace std;

// a very basic C++ object
class Person
{
    public:
        string name;
        int age
}
```

Note – this is **not** a properly-designed class according to object-orientation principles.

Analysis

Analysis

- Object-orientation is all about recognizing the different "actors" at work in the system being modeled.
  - First, the different data available within the system are organized appropriately.
  - Secondly, functionalities relating to the state of data and its management are bound to that data.

Analysis

- Object-orientation is all about recognizing the different "actors" at work in the system being modeled.
  - These objects ("actors") may then interact with other objects through well-formed, bounded relationships.

Analysis

- For now, let's examine how we should look at individual objects – the "actors" in a program.
  - Each object should be composed of a set of related data that represents some logical unit within the program.
  - In this case, this would be our "Person" class.

Analysis

1. Inputs: what does our object need in order to be properly formed?
   - Both from outside, and for internal representation?
2. Outputs: what parts of our object are needed by the outside world?
   - What might some other part of the program request from it?

Analysis

1. Constraints: should our object have limitations imposed on it, beyond those implied by the language we're using?
   - –Some of our internal state variables (fields) may allow values which make no sense in the context of what our object represents.

Analysis

1. Assumptions: Are we assuming something in our construction of the class which might have to change later?
   - We wish to minimize these (in the long run at least) as much as possible.

A First Object

```
// a very basic C++ object
class Person
{
    public:
        string name;
        int age;
}
```

- What is bad about the design of our current "Person" class?

Encapsulation

- *Encapsulation* refers to the idea that an object should protect and manage its own state information.
  - In a way, each object should behave like its own entity.
  - Data security is enforced by the object definition itself.
  - This allows a programmer to make ensure that the data being represented is always in a consistent form.

Encapsulation

- Generally speaking, objects should never make their fields public.
  - A public field can be accessed *and* modified at any time from code that has a reference to the object, which can invalidate its internal state.

Encapsulation

- Note that encapsulation is motivated by the desire to enforce *constraints* on our object.
  - How can we make sure our object is always in a proper, well-formed state if we can't limit how others modify it?

Encapsulation

- In object-oriented languages, objects may set their fields to be inaccessible outside of the class.
  - To do this, one may use the access modifier private.
    - This restricts access to the "private" field or method to ***only*** code in the class in which said field or method is defined.

A First Object

```
// a very basic C++ object
class Person
{
    public:
        string name;
        int age;
}
```

- So, instead of marking the fields as public…

A First Object

```
// a very basic C++ object
class Person
{
    private:
        string name;
        int age;
}
```

- We want to mark them as private.

A First Object

```
// a very basic C++ object
class Person
{
    private:
        string name;
        int age;
}
```

- This creates a new problem, though.
  - How can we initialize our object?

Initialization

- By default, when no constructor exists for a class, C++ creates a "default" constructor with no internal code.
  - Note: this "default" constructor will initalize *nothing* within the class.
  - Java's default constructor acts differently, setting values to zeros and nulls.
- Typically, we will need to create our own constructors to ensure the class is properly initialized.

A First Object

```
// a very basic C++ object
class Person
{
    public:
        Person(string name, int age);

    private:
        string name;
        int age;
}
```

A First Object

```
Person::Person(string name, int age)
{
    this->name = name;
    this->age  = age;
}
```

- Something interesting here: note the use of "->".
  - "this" is a *pointer*, and it refers to the *instance* of the class upon which the constructor/function has been called.

Initialization

- Once one constructor has been coded for a class, the default constructor no longer exists unless it is manually coded.
- A fully constructed and initialized class object can be called an *instance* of its class.