Encapsulation

• What if we need to be able to change one or more of the fields of a class instance?
  – The (first) solution: *mutator* methods.
  – These provide an interface through which outside code may *safely* change the object’s state.
A First Object

```c++
void Person::setName(string name)
{
    this->name = name;
}

void Person::setAge(int age)
{
    this->age = age;
}
```
Encapsulation

• Is this necessarily the correct solution, though?
  – It depends on the purpose for our class.
• Note that we allow both the “name” of our “Person” and his/her “age” to change, freely.
Encapsulation

• Should we allow a “Person” to change his/her name?
  – It does happen in the real world, but for simplicity, let us suppose that we do not wish to allow people to change names.
  – In such a case, we should remove the setName() method.
Encapsulation

```cpp
void Person::setName(string name)
{
    this->name = name;
}

void Person::setAge(int age)
{
    this->age = age;
}
```
Encapsulation

• However, we shouldn’t stop here. If we wish to make sure that a person may *never* have their name changed, can we make sure that even code from within the class may not change it?
  – Yes: use the `const` keyword.
  – In Java: “`final`”.
class Person
{
    private:
    const string name;
    int age;
}

• When a field is marked as const, it can only be initialized in a special part of the constructor.
Encapsulation

Person::Person(string name, int age):
    :name(name) {
        //this->name = name;
        /* This line would be
         * a compiler error! */
        this->age = age;
    }
Encapsulation

Person::Person(string name, int age) :
  :name(name)
{
  //this->name = name;
  /* This line would be
   * This would be
   * a compiler error!  
   */
  this->age = age;
}

This is the only valid way to initialize a const variable.
Encapsulation

• Should we allow a “Person” to change his/her age?
  – Last time I checked, everyone ages.
  – However, note that a person’s age cannot change freely.
  – Nobody ages in reverse.
  – A person can only add one year to their age, well, every year.
Encapsulation

```cpp
void Person::setName(String name)
{
    this->name = name;
}

void Person::setAge(int age)
{
    this->age = age;
}
```
Encapsulation

```cpp
void Person::haveABirthday()
{
    this->age++;  
}
```
Encapsulation

• At first, encapsulation may seem to be unnecessary.
  – It does add extra effort to using values that you *could* just directly access instead.
  – However, someone else might not know how to properly treat your object and may mess it up if you don’t encapsulate.
Encapsulation

• There are other benefits to encapsulation.
  – What if you later realize there’s an even better way to implement your class?
  – You can provide the same methods for accessing object data while changing its internals as needed.
Encapsulation

• Is our current implementation of age “the best”?
  – A possible alternative: track birthdays instead!
  – Birthdays only come once a year, after all, and at known, preset times.
Encapsulation

•Disclaimer – C++ does not provide a simple way to calculate differences in dates.
  –As a result, know that the code coming up is representative of what *could* be done, *if* the appropriate class existed.
class Person
{
    private:
    const string name;
    const MyDate birthday;

    //...
}
public class Person
{
    //...

    public:
    Person(string name, MyDate bday)
    {
        int getAge();
        string getName();
    }
}
A First Object

int Person::getAge()
{
    return MyDate.differenceInYears(
        MyDate.now(), birthday);
}

Person::Person(string name, MyDate bday)
:name(name), birthday(bday)
{
}
Analysis

• Note that the “inputs” to an object are managed through its constructors and mutator methods.
• The “outputs” are managed through its accessor methods in such a way that the “constraints” are still enforced.