Ví dụ thêm: Kiểu Phân Số - Cách tiếp cận dùng hàm thành phần

```
struct Fraction
{
    int numerator, denominator;
    void reduce();
    void print();
    void input();
    Fraction add(Fraction b);
    Fraction sub(Fraction b);
    Fraction mul(Fraction b);
    Fraction div(Fraction b);
};

int gcd(int a, int b)
{
    if (b == 0) return abs(a);
    return gcd(b, a%b);
}
```

Listing cont…

```
void Fraction::reduce()
{
   int u = gcd(numerator, denominator);
   numerator /= u;
   denominator /= u;
   if (denominator < 0)
   {
      numerator = -numerator;
      denominator = -denominator;
   }
}
```

Listing cont…

```
void Fraction::print()
{
   printf("%d", numerator);
   if (numerator || denominator != 1)
   printf("/%d", denominator);
}

void Fraction::input()
{
   scanf_s("%d%d", &numerator,
   &denominator);
}
```

Listing cont...

```
Fraction Fraction::add(Fraction b)
{
    Fraction c;
    c.numerator = numerator*b.denominator +
    denominator*b.numerator;
    c.denominator = denominator*b.denominator;
    c.reduce();
    return c;
}

Fraction Fraction::sub(Fraction b)
{
    Fraction c;
    c.numerator = numerator*b.denominator -
    denominator*b.numerator;
    c.denominator = denominator*b.denominator;
    c.reduce();
    return c;
}
```

Listing cont...

```
Fraction Fraction::mul(Fraction b)
{
    Fraction c;
    c.numerator = numerator*b.numerator;
    c.denominator = denominator*b.denominator;
    c.reduce();
    return c;
}

Fraction Fraction::div(Fraction b)
{
    Fraction c;
    c.numerator = numerator*b.denominator;
    c.denominator = denominator*b.numerator;
    c.reduce();
    return c;
}
```

Listing cont…

```
int main()
{
    Fraction a = { 1,4 }, b = { 1,2 }, c;
    c = a.add(b);
    c.print();
    puts("");
    c = a.sub(b);
    c.print();
    puts("");
    c = a.mul(b);
    c.print();
    puts("");
    c = a.div(b);
    c.print();
    puts("");
    return 0;
}
```

Cài đặt lớp - Ví dụ so sánh

- *Nhận xét: (Nhắc lại)*
- Giải quyết được vấn đề.
- Dữ liệu và các hàm xử lý dữ liệu được gom vào một chỗ bên trong cấu trúc. Do đó dễ theo dõi quản lý, dễ bảo trì nâng cấp.
- Các thao tác đều bớt đi một tham số so với cách tiếp cận cổ điển. Vì vậy việc lập trình gọn hơn. Tư tưởng thể hiện ở đây là *đối tượng đóng vai trò trọng tâm*. Đối tượng thực hiện thao tác trên chính nó.
- Trình tự sử dụng qua các bước: Khởi động, sử dụng thực sự, (dọn dẹp).

Lớp

- Trong cách tiếp cận dùng struct và hàm thành phần, người sử dụng có toàn quyền truy xuất, thay đổi các thành phần dữ liệu của đối tượng thuộc cấu trúc.

```
Stack s;
s.Init(10);
s.size = 100; // Nguy hiem
for (int i = 0; i < 20; i++) s.Push(i);

Fraction a;
a.numerator = 2;
a.denominator = 0; // Oh dear! 2/0!!!
```

Lớp

- Vì vậy, ta không có sự an toàn dữ liệu. Lớp là một phương tiện để khắc phục nhược điểm trên.
- Lớp có được bằng cách thay từ khoá struct bằng từ khoá class (C++, Java, C#).
- Trong lớp mọi thành phần mặc nhiên đều là *riêng tư* (private) nghĩa là từ bên ngoài không được phép truy xuất. Do đó có sự an toàn dữ liệu.

Từ khoá class thay cho struct

```
class Stack
{
   Item *st, *top;
   int size;
   void init(int sz){ top = st = new
   Item[size = sz];}
   void cleanUp(){ delete[] st; }
   bool empty() { return top <= st; }
   bool full() { return top - st >= size; }
   bool push(Item x);
   bool pop(Item *px);
};
```

Lớp

- Mọi cố gắng truy xuất các thành phần của lớp sẽ bị báo lỗi biên dịch.

```
s.size = 100; // Bao sai
a.denominator = 0; // Bao sai
```

- Nghĩa là lỗi luận lý được trình biên dịch hỗ trợ phát hiện, giúp giảm rất nhiều bug khó tìm.

Thuộc tính truy xuất

- Tuy nhiên lớp như trên trở thành vô dụng vì các hàm thành phần cũng trở thành riêng tư và không thể được.

```
Stack s;
s.init(8);     // Bao sai
s.push(14);    // Bao sai
Fraction a;
a.setNumerator(2);          // Bao sai
a.setDenominator(14);       // Bao sai
```

- Thuộc tính truy xuất có thể giải quyết vấn đề trên.
- Thuộc tính *truy xuất* của một thành phần nào của lớp là phần chương trình có thể nhìn thấy, truy xuất thành phần đó.

Thuộc tính truy xuất

- Thuộc tính *private*: Chỉ có các phương thức của lớp được phép truy xuất.
- Thuộc tính *public*: Có thể được truy xuất từ bất cứ nơi nào trong chương trình nguồn từ sau khai báo lớp.
- Phần public tạo nên phần *giao diện* của lớp.
- Nguyên tắc chung: Các thuộc tính dữ liệu của lớp có thuộc tính private, các phương thức để người sử dụng dùng có thuộc tính public.

Thuộc tính truy xuất

- Trong C++, thuộc tính mặc nhiên của lớp *private,* thuộc tính của struct là *public.*
- Ta có thể thay đổi thuộc tính truy xuất bằng nhãn *private* hoặc *public.* Sự thay đổi có tác dụng từ sau nhãn.

Thuộc tính truy xuất: Ví dụ

```
class Stack
{
   Item *st, *top;
   int size;
public:
   void Init(int sz) { st = top = new
   Item[size = sz]; }
   void CleanUp() { if (st) delete[] st; }
   bool Full()const { return (top - st >=
   size); }
   bool Empty() const { return (top <= st); }
   bool Push(Item x);
   bool Pop(Item *px);
};
```

Các thành phần private không được truy xuất từ bên ngoài

Thuộc tính truy xuất: Ví dụ

```
class Fraction
{
   int numerator, denominator;
   void reduce();
public:
   void print();
   void input();
   Fraction add(Fraction b);
   Fraction sub(Fraction b);
   Fraction mul(Fraction b);
   Fraction div(Fraction b);
   void setNumerator(int n) { numerator = n; }
   void setDenominator(int d) { denominator = d;}
};
```

Các thành phần private không được truy xuất từ bên ngoài

Thuộc tính truy xuất

- Các thành phần private *không thể* được truy xuất từ bên ngoài nhưng các thao tác public thì được.

```
Stack s;
s.size = 100;              // Error
s.init(8);                 // Ok
s.push(14);                // Ok
Fraction a;
a.denominator = 0;         // Error
a.setNumerator(2);         // Ok
a.setDenominator(14);      // Ok
```

Thuộc tính truy xuất

- Phạm vi truy xuất được sử dụng đúng sẽ cho phép ta kết luận: Nhìn vào lớp *thấy được mọi thao tác* trên lớp.
- Người dùng bình thường có thể khai thác hết các chức năng của lớp thông qua phần giao diện (public) của lớp.
- Người dùng cao cấp có thể thay đổi chi tiết cài đặt, cải tiến giải thuật cho các phương thức.

Cài đặt lớp trong Objective C

- Cú pháp cài đặt lớp trong Objective C khác với C++, gần giống với Smalltalk.
- Lớp trong Objective C có hai phần tách biệt, phần giao diện (interface) và phần cài đặt (implementation).
- Trong Objective C, thuộc tính dữ liệu định nghĩa trong phần cài đặt mặc nhiên là private, các thao tác trong phần giao diện mặc nhiên là public.

Cài đặt lớp trong Objective C

```
@interface ClassName: ParentClassName
   propertyDeclarations;
   methodDeclarations;
@end

@implementation ClassName
{
   memberDeclarations;
}
methodDefinitions;
@end
```

Lớp Phân Số

```
//  Fraction.h

#import <Foundation/Foundation.h>

@interface Fraction : NSObject
+(int)gcd : (int)a and : (int)b;
-(void)setNumerator: (int)n;
-(void)setDenominator: (int)d;
-(void)set: (int)n over : (int)d;
-(void)print;
-(Fraction *)add: (Fraction *)b;
-(Fraction *)sub: (Fraction *)b;
-(Fraction *)mul: (Fraction *)b;
-(Fraction *)div: (Fraction *)b;
@end
```

```
//  Fraction.m

#import "Fraction.h"

@implementation Fraction
{
   int numerator, denominator;
}

+(int)gcd: (int)a and : (int)b {
   if (b == 0)
      return a;
   return[Fraction gcd : b and : a % b];
}
```

```
-(void)setNumerator: (int)n {
    numerator = n;
}

-(void)setDenominator: (int)d {
    denominator = d;
}

-(void)set: (int)n over: (int)d {
    int u = [Fraction gcd : n and : d];
    numerator = n / u;
    denominator = d / u;
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
}
```

```
-(int)numerator {
   return numerator;
}

-(int)denominator {
   return denominator;
}

-(void)print {
   if (denominator)
      NSLog(@"%i / %i", numerator, denominator);
   else
      NSLog(@"%i", numerator);
}
```

```
-(Fraction *)add: (Fraction *)b {
    Fraction *c = [Fraction alloc];
    c = [c init];
    [c set: numerator * b.denominator + denominator *
    b.numerator over: denominator * b.denominator];
    return c;
}

-(Fraction *)sub: (Fraction *)b {
    Fraction *c = [Fraction alloc];
    c = [c init];
    [c set: numerator * b.denominator - denominator *
    b.numerator over: denominator * b.denominator];
    return c;
}
```

```
-(Fraction *)mul: (Fraction *)b {
    Fraction *c = [[Fraction alloc] init];
    [c set: numerator * b.numerator over:
    denominator * b.denominator];
    return c;
}

-(Fraction *)div: (Fraction *)b {
    Fraction *c = [[Fraction alloc] init];
    [c set: numerator * b.denominator over:
    denominator * b.numerator];
    return c;
}

@end
```

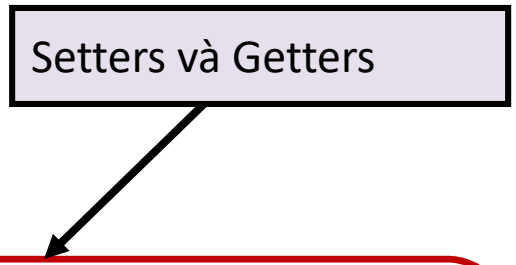
```
//  main.m
#import <Foundation/Foundation.h>
#import "Fraction.h"

int main(int argc, const char * argv[]) {
@autoreleasepool {
        Fraction *a = [Fraction alloc];
        a = [a init];
        [a set: 1 over: 3];
        Fraction *b = [Fraction alloc];
        [b set: 5 over: 6];
        Fraction *c = [a add: b];
        [c print];
        [[a sub: b] print];
        [[a mul: b] print];
        [[a div: b] print];
    }
    return 0;
}
```

setter và getter

- Thiết kế lớp chuẩn thì các thành phần dữ liệu sẽ có thuộc tính truy xuất là private nên không thể truy xuất trực tiếp.
- Để có thể thao tác trên trên các dữ liệu ta dùng các phương thức đưa vào và lấy ra, gọi là *setter* và *getter*.
- Trong lớp Stack, phương thức đưa vào là push và phương thức lấy ra là pop.

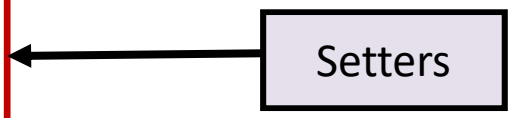
```
class Fraction
{
    int numerator, denominator;
    void reduce();
    public:
    void print();
    void input();
    Fraction add(Fraction b);
    Fraction sub(Fraction b);
    Fraction mul(Fraction b);
    Fraction div(Fraction b);
    void setNumerator(int n) { numerator = n; }
    void setDenominator(int d) { denominator = d; }
    void set(int n, int c);
    int getNumerator() { return numerator; }
    int getDenominator() { return denominator; }
};
```



Setters & Getters

```
@interface Fraction : NSObject

+(int)gcd : (int)a and : (int)b;
-(void)setNumerator: (int)n;
-(void)setDenominator: (int)d;
-(void)set: (int)n over : (int)d;
-(void)print;
-(Fraction *)add: (Fraction *)b;
-(Fraction *)sub: (Fraction *)b;
-(Fraction *)mul: (Fraction *)b;
-(Fraction *)div: (Fraction *)b;
@end
```

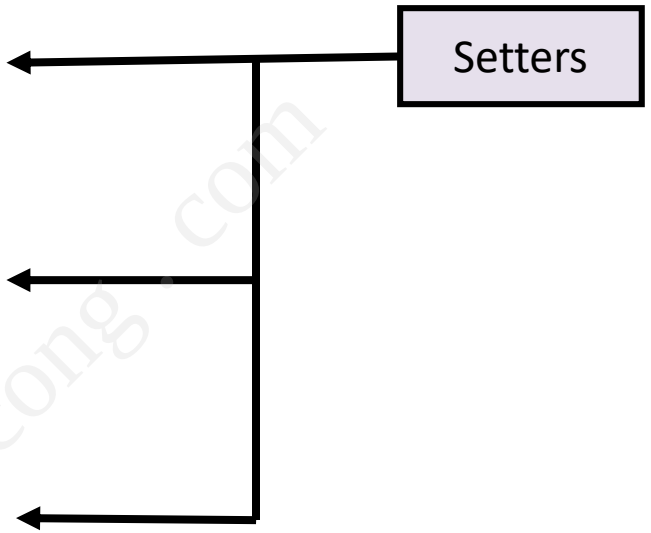


Setters & Getters

```
@implementation Fraction
//...
-(void)setNumerator: (int)n {
    numerator = n;
}

-(void)setDenominator : (int)d {
    denominator = d;
}

-(void)set: (int)n over : (int)d {
    int u = [Fraction gcd : n and : d];
    numerator = n / u;
    denominator = d / u;
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
}
```



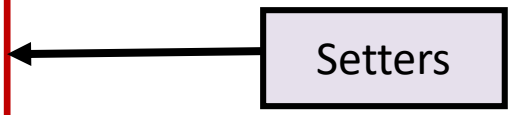
Setters & Getters

```
@implementation Fraction
{
    int numerator, denominator;
}

-(void)setNumerator: (int)n {
    numerator = n;
}

-(void)setDenominator : (int)d {
    denominator = d;
}

-(void)set: (int)n over : (int)d {
    int u = [Fraction gcd : n and : d];
    numerator = n / u;
    denominator = d / u;
    if (denominator < 0) {
        numerator = -numerator;
        denominator = -denominator;
    }
}
```



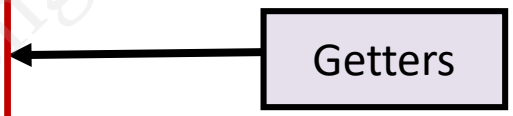
```
//....

-(int)numerator {
    return numerator;
}

-(int)denominator {
    return denominator;
}

//...

@end
```



Tự động phát sinh setters & getters trong Objective C

- Objective C có cơ chế cho phép phát sinh các phương thức setter(s) và getter(s) một cách tự động.
- Ta dùng từ khoá @property trong phần giao diện và từ khoá @synthesize trong phần cài đặt.

setters & getters

```
@interface Fraction : NSObject
@property int numerator, denominator;

+(int)gcd: (int)a and : (int)b;
// -(void) setNumerator: (int) n;
// -(void) setDenominator: (int) d;
-(void)set: (int)n over : (int)d;
-(void)print;
-(Fraction *)add: (Fraction *)b;
-(Fraction *)sub: (Fraction *)b;
-(Fraction *)mul: (Fraction *)b;
-(Fraction *)div: (Fraction *)b;
@end
```

Tự động phát sinh setters

No need

setters & getters

```
@implementation Fraction
@synthesize numerator, denominator;
/*
{
    int numerator, denominator;
}
-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}
-(int) numerator {
    return numerator;
}

-(int) denominator {
    return denominator;
} */
```

Khởi động và dọn dẹp

Đồng bộ với @property

No need

Khởi động và dọn dẹp

Để đối tượng sẵn sàng ở trạng thái hoạt động, ta cần bảo đảm:

1. Đối tượng được khởi động.
2. Đối tượng không bị khởi động dư.
3. Tài nguyên cấp cho đối tượng được giải phóng.
4. Tài nguyên này không bị giải phóng dư.

Người sử dụng có thể *quên* khởi động hoặc dọn dẹp.

Khởi động và dọn dẹp

- Ta có thể khởi động và dọn dẹp đối tượng một cách tự động nhờ phương thức thiết lập và huỷ bỏ
- Phương thức thiết lập và huỷ bỏ được xây dựng nhằm mục đích khắc phục lỗi quên khởi động đối tượng hoặc khởi động dư. Việc quên khởi động đối tượng thường gây ra những lỗi rất khó tìm.

Khởi động và dọn dẹp

- *Phương thức thiết lập* là hàm thành phần đặc biệt được tự động gọi đến mỗi khi một đối tượng thuộc lớp được tạo ra. Người ta thường lợi dụng đặc tính trên để khởi động đối tượng.
- Trong C++, phương thức thiết lập có tên trùng với tên lớp để phân biệt nó với các hàm thành phần khác.
- Có thể có nhiều phiên bản khác nhau của phương thức thiết lập

Khởi động và dọn dẹp

- *Phương thức huỷ bỏ* là hàm thành phần đặc biệt được tự động gọi đến mỗi khi một đối tượng bị huỷ đi. Người ta thường lợi dụng đặc tính trên để dọn dẹp đối tượng.
- Trong C++, Phương thức huỷ bỏ bắt đầu bằng dấu ngã (~) theo sau bởi tên lớp để phân biệt nó với các hàm thành phần khác.
- Chỉ có thể có tối đa một phương thức huỷ bỏ.
- Với PTTL & HB, ta đưa việc sử dụng đối tượng thành qui trình 1 bước thay vì 3 bước.

```
class Stack
{
   Item *st, *top;
   int size;
   void Init(int sz){
      top = st = new Item[size = sz]; }
   void CleanUp(){ delete[] st; }
public:
   Stack(int sz = 20) { Init(sz); }
   ~Stack() { CleanUp(); }
   bool Empty() { return top <= st; }
   bool Full() { return top - st >= size; }
   bool Push(Item x);
   bool Pop(Item *px);
};
```

Tự động khởi động nếu quên khởi động

```
void XuatHe16(long n)
{
   static char hTab[] = "0123456789ABCDEF";
   Stack s(8);
   int x;
   do {
      s.Push(n % 16);
      n /= 16;
   } while (n);
   while (s.Pop(&x))
      cout << hTab[x];
}
```

Qui trình sử dụng 1 bước

Sử dụng PT Thiết Lập & Huỷ bỏ

- Với phương thức huỷ bỏ ta có thể *"dạy"* trình biên dịch:
  1. *Báo lỗi* nếu người sử dụng quên khởi động, hoặc
  2. *Khởi động tự động* nếu người sử dụng quên khởi động.
- Ta cũng có thể bảo đảm đối tượng không bị khởi động dư.
- Với phương thức huỷ bỏ, ta có cơ chế *dọn dẹp tự động*.

```
class Stack
{
   Item *st, *top;
   int size;
   void Init(int sz){
      top = st = new Item[size = sz]; }
   void CleanUp(){ delete[] st; }
public:
   Stack(int sz) { Init(sz); }
   ~Stack() { CleanUp(); }
   bool Empty() { return top <= st; }
   bool Full() { return top - st >= size; }
   bool Push(Item x);
   bool Pop(Item *px);
};
```

Báo sai nếu quên khởi động