

Encapsulation

10/26/2015

Hsuan-Tien Lin (林軒田)

htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)



Recall: Basic OOP Needs

Decompose computation into interactions of “computing parts” called *objects*, each containing its own data (to be manipulated by itself) and own code (to be called by other objects)

- designing object (what variables? what methods?)
- creating “first” object and calling its first action
- creating other objects
- calling other objects
- manipulating object status
- deleting objects

Encapsulation

Decompose computation into interactions of “computing parts” called *objects*, each containing its own data **(to be manipulated by itself)** and own code **(to be called by other objects)**

- preventing others from manipulating the object
- letting others call the allowed code

Encapsulation (1/5)

```
1  class Record{
2      String name;
3      String password;
4  }
5
6  public class RecordDemo{
7      public static void main(String [] argv){
8          Record r;
9          String s, p;
10         s = getLoginNameFromUser();
11         r = getRecordFromFile(s);
12         System.out.println(r.password);
13         p = getPasswordFromUser();
14         if (p.equals(r.password)){
15             // ...
16         }
17     }
18 }
```

- if password not encoded, the SYSOP might easily get your password by `getRecordFromFile`

Encapsulation (2/5)

```
1  class Record{
2      String name;
3      String encoded_password;
4  }
5
6  public class RecordDemo{
7      public static void main(String [] argv){
8          Record r;
9          String s, p;
10         s = getLoginNameFromUser();
11         r = getRecordFromFile(s);
12         p = getPasswordFromUser();
13         if (YOUR_ENCODING(p).equals(r.encoded_password)){
14             // ...
15         }
16         //A new and careless programmer adds this line
17         r.encoded_password = null;
18     }
19 }
```

- even when password encoded, a careless programmer may make stupid bugs

Encapsulation (3/5)

```
1  class Record{
2      private String encoded_password;
3      public String get_encoded_password(){
4          return encoded_password;
5      }
6  }
7  public class RecordDemo{
8      public static void main(String [] argv){
9          Record r;
10         String s, p;
11         s = getLoginNameFromUser();
12         r = getRecordFromFile(s);
13         p = getPasswordFromUser();
14         if (YOUR_ENCODING(p).equals(r.get_encoded_password())){
15             // ...
16         }
17         //A new and careless programmer adds this line
18         r.encoded_password = null; //won't work
19     }
20 }
```

- what if you want to set a new password?

Encapsulation (4/5)

```
1  class Record{
2      String name;
3      private String encoded_password;
4      public String get_encoded_password(){
5          return encoded_password;
6      }
7      public void set_encoded_password(String raw_password){
8          if (blahblah)
9              encoded_password = YOUR_ENCODING(raw_password);
10     }
11 }
```

- **separate implementation and use:** you implement the `Record` class, and other people (possibly you after two years) use it
- **don't trust other people:** silly mistakes can happen
- **hide unnecessary details** (a.k.a. instance variables)
- **think about possible correct/incorrect use of your class:** check them in the methods

Encapsulation (5/5)

```
1  public class RecordDemo{
2      public static void main(String [] argv){
3          Record r;
4          String s, p;
5          s = getLoginNameFromUser();
6          r = getRecordFromFile(s);
7          p = getPasswordFromUser();
8          if (r.match_password(p)){
9              //no need to show encoding to the outside
10             }
11             r.set_encoded_password(old_password, new_password);
12             //don't want this to happen
13             r.encoded_password = null;
14         }
15     }
```

- freedom on making assignments: a potential hole to do bad and/or make bugs

Encapsulation: Key Point

as a designer, you should avoid giving the users of your code too much freedom to do bad and/or make bugs

Hiding Variables from All Classes (1/3)

```
1  class Record{  
2      private String encoded_password;  
3  }
```

- `private`: hiding from all classes (except myself, of course)

```
1  class Record{  
2      private String encoded_password;  
3      public boolean compare_password(Record another_record){  
4          return (  
5              this.encoded_password ==  
6              another_record.encoded_password  
7          ); //okay?  
8      }  
9  }
```

Hiding Variables from All Classes (2/3)

```
1  class Record{
2      private String name;
3      public String get_name(){
4          return name;
5      }
6      public String get_copied_name(){
7          return new String(name);
8      }
9  }
```

- `public`: accessible by all classes
- **accessor**: get the content of the instance

Hiding Variables from All Classes (3/3)

```
1  class Record{
2      private String name;
3      public void set_name(String name){
4          if (name != null)
5              this.name = name;
6      }
7  }
```

- **mutator**: check and set the instance variable to a value

Hiding Variables from All Classes: Key Point

private instance variables
public accessor/mutator instance methods

More on Hiding Details (1/3)

```
1  class Date{ //implemented for your desktop
2      private int month, day;
3      public int get_month(){ return month; }
4      public int get_day(){ return day; }
5  }
6  class Date_TWO{ //implemented on a small-memory machine
7      private short encoded_month_and_day;
8      public int get_month(){
9          return encoded_month_and_day / 100;
10     }
11     public int get_day(){
12         return encoded_month_and_day % 100;
13     }
14 }
```

- two implementations, same behavior—easy for users to switch on different machines
- trade-offs: memory usage, computation, etc.

More on Hiding Details (2/3)

```
1  class Distance{
2      private double mile;
3      public double get_mile(){
4          return mile;
5      }
6      public double get_km(){
7          return mile * 1.6;
8      }
9      public void set_by_km(double km){
10         this.mile = km / 1.6;
11     }
12     public void set_by_mile(double mile){
13         this.mile = mile;
14     }
15 }
```

- one storage, different information from different mutator/accessor

More on Hiding Details (3/3)

Some rules of thumb:

- make all instance variables private
- use mutators/accessors for safely manipulate the variables

Cons:

- accessing requires method calls, slower

Pros:

- less chance of misuse by other users
- flexibility

More on Hiding Details (Yet Another Case)

```
1  class Solver{  
2      public void read_in_case (){}  
3      public void compute_solution (){}  
4      public void output_solution (){}  
5      public void solve (){  
6          read_in_case ();  
7          compute_solution ();  
8          output_solution ();  
9      }  
10 }
```

- should the three utility functions be public?

More on Hiding Details: Key Point

hiding details: don't directly access internal stuff to gain flexibility and avoid misuse

Java Member Encapsulation (1/2)

```
1  class Distance{
2      private double mile;
3      public double get_mile(){
4          return mile;
5      }
6      private double get_ratio(){
7          return 1.6;
8      }
9      public double get_km(){
10         return mile * ratio;
11     }
12 }
```

- private: hidden, on variables and methods that you do not want anyone to see
- public: on variables and methods that you want everyone to see

Java Member Encapsulation (2/2)

```
1  class Demo{
2      private double mile;
3      default int a; //imagine, but not correct grammar
4      int b;
5      public double get_mile(){
6          return mile;
7      }
8      private double get_ratio(){
9          return 1.6;
10     }
11     double get_km(){
12         return mile * ratio;
13     }
14 }
15 class Another{
16     void lalala(){ int lulu = new Demo().b + 1; }
17 }
```

- default: classes in the same source file (et al.) can access it
- a “gray-area” usage

Java Member Encapsulation: Key Point

- public/private:
the more common pair for OO
programmers
- default:
for laziness of beginners, or real-advanced
use (later)