More on Methods and Encapsulation

Hsuan-Tien Lin

Deptartment of CSIE, NTU

OOP Class, March 31, 2009

Local Variables (7/7, courtesy of Prof. Chuen-Liang Chen)

Category of Java Variables

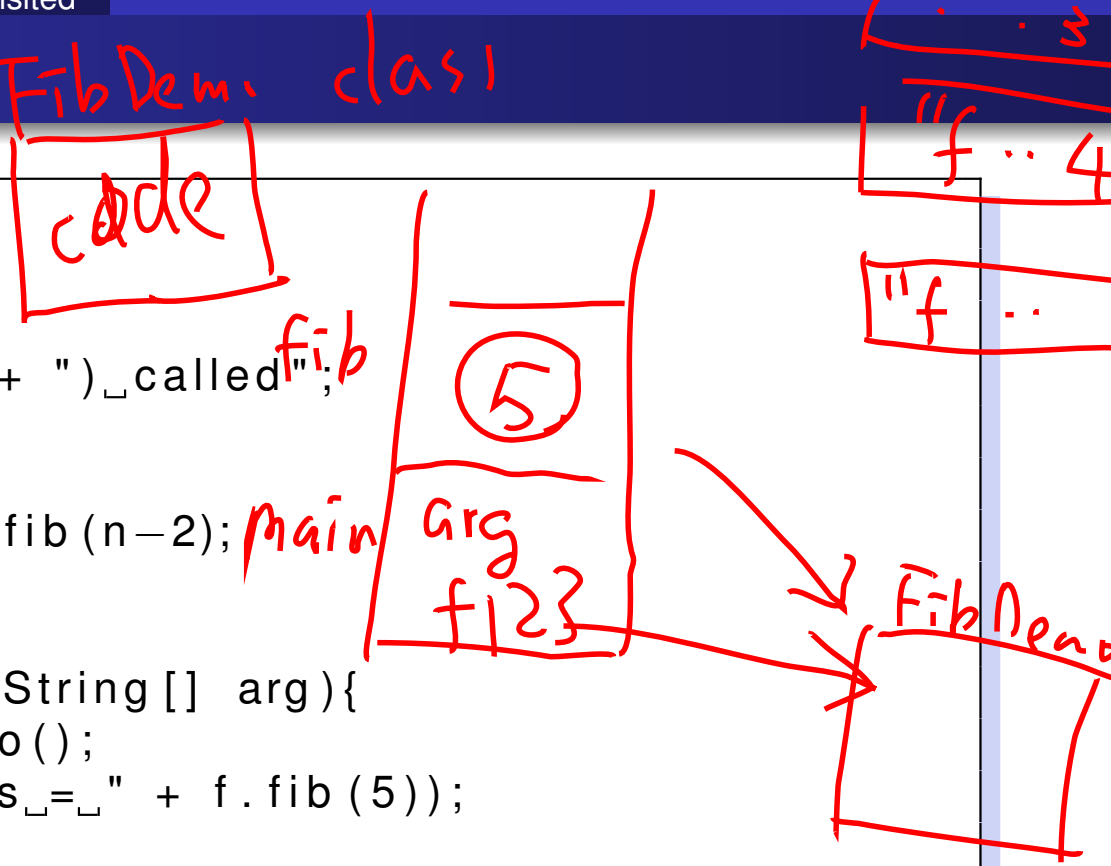
	local variable	instance variable	class (static) variable
belong to	method invocation	instance	class
declaration	within method	within class	within class
modifier `static`	NO	NO	YES
allocation (when)	method invocation	instance creation	class loading
allocation (where)	stack memory	heap memory	heap memory
initial to 0	NO	YES	YES
de-allocation	method return	automatic garbage collection	NO
scope	usage range	direct access range	
	from declaration to end of block	whole class	whole class

Local Variables: Key Point

local variables: the “status” of the current frame
—by spec **not** necessarily initialized

JVM Memory

```
1  public class FibDemo{
2    int fib(int n){
3      int res;
4      String s = "fib(" + n + ")_called";
5      System.out.println(s);
6      if (n <= 1) res = 1;
7      else res = fib(n-1) + fib(n-2);
8      return res;
9    }
10   public static void main(String[] arg){
11     FibDemo f = new FibDemo();
12     System.out.println("res_=_" + f.fib(5));
13   }
14 }
```



- a stack that holds frames (n, res, s, arg, f)
- a **heap** that holds objects (where arg, f, s point to)
- a constant pool that holds constant objects ("fib(")
- somewhere that holds the class (variables and codes)
- HEAP: 2nd one, or the last three (automatic GC on **heap**)

JVM Memory: Key Point

method stack: for method calls
HEAP: for objects/classes (with GC on **heap**)

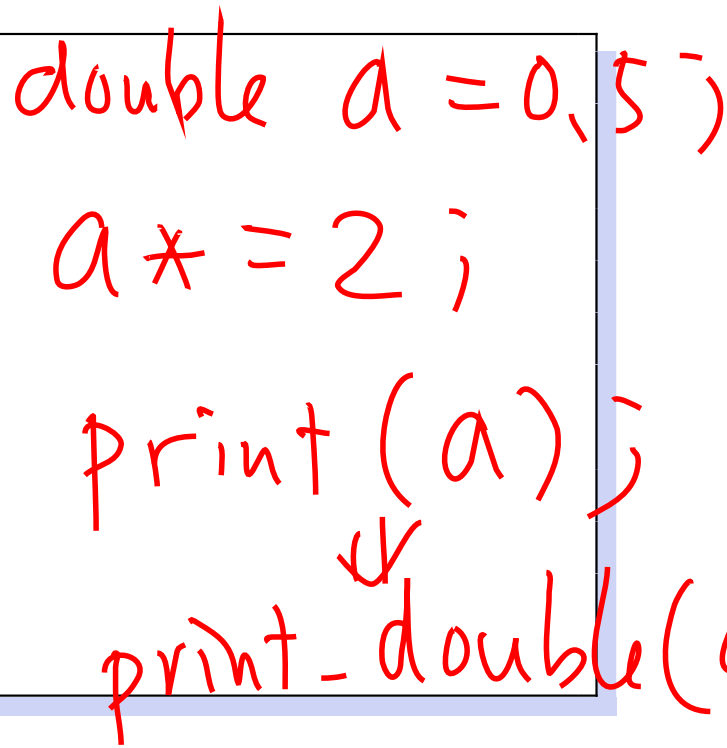
Method Overloading (1/2)

```
1  class Printer{
2    void printInteger(int a);
3    void printDouble(double a);
4    void printString(String a);
5  }
```

- “Integer” and “int” are basically saying the same thing
- lazy people don’t want to type so many words

Method Overloading (2/2)

```
1  class Printer{
2    void print(int a);
3    void print(double a);
4    int print(String a);
5    void print(int a, int b);
6  }
7  class PrinterThatCompilerSees{
8    void print_int(int a);
9    void print_double(double a);
10   int print_String(String a);
11   void print_int_int(int a, int b);
12 }
```



- Java's (and many modern language's) solution: one method name, many possible argument types
- have seen it in constructors
- called **method overloading**
- Java "signature" of a method: include name and parameters (but **NO** return types)

Method Overloading (2/2)

```
1  class Printer{
2    static void print(int a);
3    static int print(int a);
4    static double print(int a);
5
6    static void main(String[] argv){
7      Printer.print(Printer.print(3) + 5);
8      //which one do you want to call?
9    }
10 }
```

- determine programmer's intention from arguments: easy
- determine programmer's intention from return value: hard —can cast, can discard, etc.
- Java "signature" of a method: include name and parameters only
- compiler's job: from arguments (type), determine which method (name+parameters) to call
- cannot have two methods with the same signature

Method Overloading: Key Point

method overloading: a compiler's help by looking at "signature" rather than "name" in calling

Operator Overloading

```
1  class Demo{
2    static void main(String[] argv){
3      int a = 5;
4      System.out.println(3 + a); //plus(3, a)
5      System.out.println("" + a); //plus("", a)
6    }
7  }
```

- Java: a limited special case for String (actually, StringBuffer); the usual cases for primitive types; but not for other extended types
- C++: can overload almost “any” operator for any class
- double-sided sword: powerful, but easily misused

Operator Overloading: Key Point

operator overloading: very limited support in Java (up to now, and possibly will be)

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_ENODING(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_ENODING(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. member 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

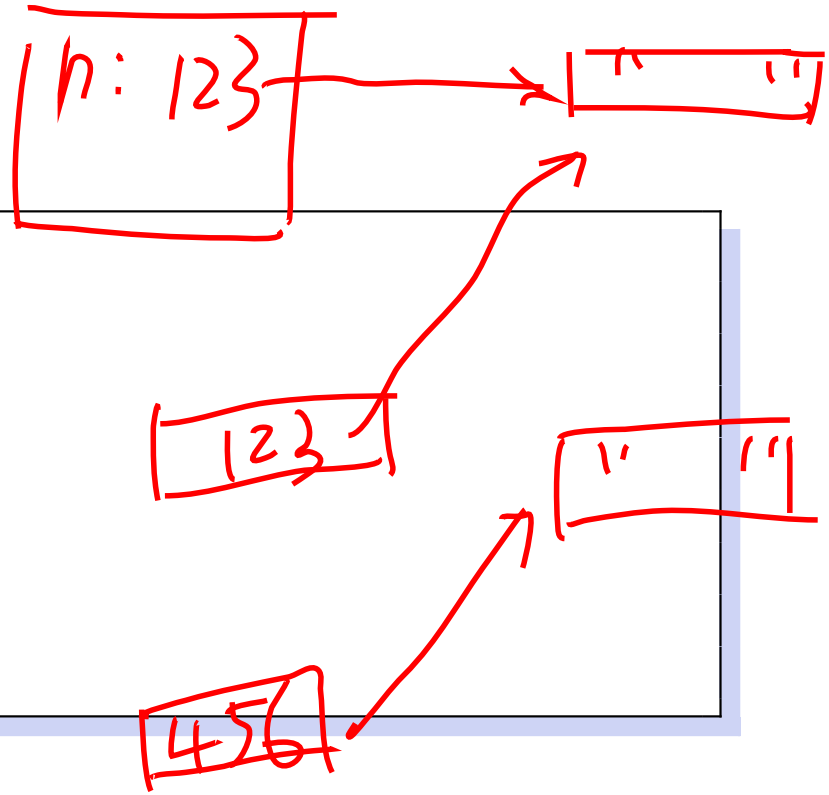
Hiding Variables from All Classes (1/3)

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

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

Hiding Variables from All Classes (2/3)

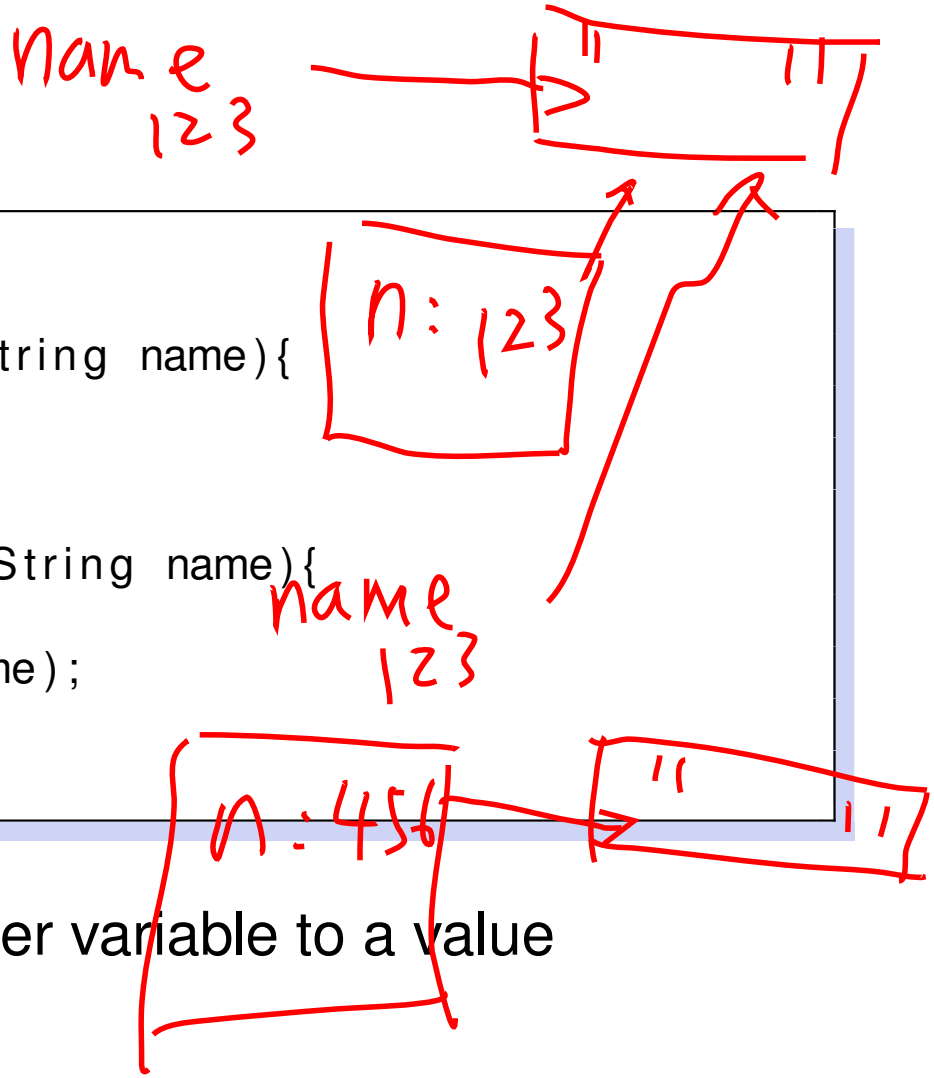
```
1  class Record{
2    private String name;
3    public String get_name(){
4      return name;
5    }
6    public String get_copied_name(){
7      return new String(name);
8    }
9  }
```



- `public`: accessible by all classes
- **getter**: get the reference/content of the class

Hiding Variables from All Classes (3/3)

```
1  class Record{
2    private String name;
3    public void set_name_by_ref(String name){
4      if (name != null)
5        this.name = name;
6    }
7    public void set_name_by_copy(String name){
8      if (name != null)
9        this.name = new String(name);
10   }
11 }
```



- **setter**: check and set the member variable to a value

Hiding Variables from All Classes: Key Point

private member variables
public getter/setter 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 getter/setter

More on Hiding Details (3/3)

Some rules of thumb:

- make all instance/class variables private
- use getters/setters for safely access 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: Trade-Off Debate

- no hiding at all

- hiding everything, showing only few allowed handles

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 }
```

get_ratio();

- 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  package tw.edu.ntu.csie;
2  class Demo{
3    private double mile;
4    default int a; //imagine, but not correct grammar
5    int b;
6    public double get_mile(){
7      return mile;
8    }
9    private double get_ratio(){
10     return 1.6;
11   }
12   double get_km(){
13     return mile * ratio;
14   }
15 }
16 class Another{
17 }
```

import java.util.*

Random

⇒ java.util.R

- default: classes in the same package can access it
  e.g. in the same source file, declared with the same package
- a “grey-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 in package making

Java Class Encapsulation (2/2)

```
1  package my.package.name;
2  public class PublicDistance{
3  }
4  private class PrivateDistance{ //NO!
5  }
6  class DefaultDistance{
7  }
```

- no private: class is meant for everyone to use
- public class should have same name as source file (thus, one public ~~code~~ class per source file)
- default class—for utility use within the same package

Java Class Encapsulation: Key Point

public: “entry point” of each source
default: for utility use

Who Can Access What?

callee class callee membership	public public	public private	public default	default public	default private	default default
callee itself	Y	Y	Y	Y	Y	Y
same source	Y	N	Y	Y	N	Y
same package	Y	N	Y	Y	N	Y
others	Y	N	N	N	N	N