Be Concrete about Abstractions

Hsuan-Tien Lin

Deptartment of CSIE, NTU

OOP Class, March 24, 2009

Object Lifecycle (1/1)

```
class Record{
  int score;
  Record(int init_score){ score = init_score; }
  protected void finalize() throws Throwable{ }
}
public class RecordDemo{
  public static void main(String[] arg){
    Record r; //reference declared
    Record r2;//reference declared
    r = new Record(60); //memory allocated (RHS)
                        //and constructor called
                        //reference assigned (LHS)
    r2 = r;             //reference copied
    r.score = 3;        //object content accessed
    r.show_score();     //object action performed
    r2 = null; r = null;//memory slot isolated
                        // ....
                        //finalizer called
                        //or JVM terminated
  }
}
```

Object Lifecycle: Key Point

we control birth, life, death, funeral design, but not the exact funeral time (nor whether it would happen)

Class Lifecycle (1/1)

```java
class Record{
  static{//class initializer
    count = 0; //actually, done by default
  }
  static int count;
  int score;
  Record(int init_score){ score = init_score; count++; }
  protected void finalize() throws Throwable{ }
  //classFinalize? has been removed
}
//the class RecordDemo should be loaded after 'java RecordDemo'
public class RecordDemo{
  public static void main(String[] arg){
    //the class Record should be loaded (initialized) before
    //the following lines
    Record r, r2;
    r = new Record(60);
    r2 = r;
    //....
    //by default, no unload (and hence count would remain)
    //if unloaded, no guarantees
  }
}
```

Class Lifecycle: Key Point

automatic and dynamic loading/linking in JVM:
almost no need to care about load/unload

Method (1/2, Callee's View)

```
class Record{
  int score;
  void add_to(int inc){ score += inc; }
  double get_adjusted(){ return (double)(score + 30); }
}
```

- method: what I (the object) do
- **parameter**: what I get from the caller
- return value: what I want to tell the caller

Method (2/2, Caller's View)

```
class Record{
  int score;
  void add_to(int inc){ score += inc; }
  double get_adjusted(){ return (double)(score + 30); }
}
public class RecordDemo{
  public static void main(String[] arg){
    Record r = new Record();
    r.addto(50);
    System.out.println(r.get_adjusted());
  }
}
```

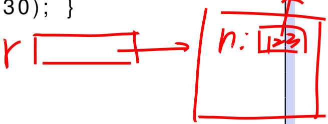
- method: what I (caller) want the object to do
- **argument**: what I tell the callee
- return value: what I want to hear from the callee

Method: Key Point

method: an abstraction of **action**, where information is passed through argument/parameter and return values

Return Values (1/1)

```java
class Record{
  String name;
  int score;
  void add_to(int inc){ score += inc; }
  double get_adjusted(){ return (score + 30); }
  String get_name(){ return name; }
}
public class RecordDemo{
  public static void main(String[] arg){
    Record r = new Record();
    r.name = "HTLin";
    System.out.println(r.get_name() == r.name);
  }
}
```



- void: no return
- primitive type: its value
- reference type: its value (a.k.a. reference, not content)

Return Values: Key Point

Java: return by primitive/reference values

Primitive Argument/Parameter (1/1)

```
class Tool{
  void swap(int first, int second){
    int tmp = first;
    first = second;
    second = tmp;
    System.out.println(first);
    System.out.println(second);
  }
}
public class Demo{
  public static void main(String[] arg){
    Tool t = new Tool();
    int i = 3; int j = 5;
    t.swap(i, j);
    System.out.println(i);
    System.out.println(j);
  }
}
```

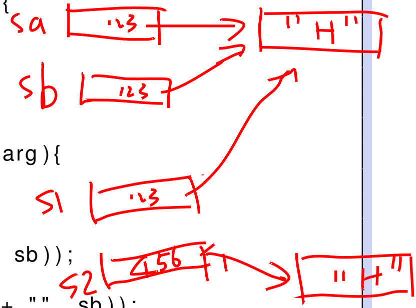
- first, second: swapped
- i, j: didn't

Primitive Argument/Parameter: Key Point

argument $\Rightarrow$ parameter: by value copying
–change in parameter does not change argument

Reference Argument/Parameter (1/3)

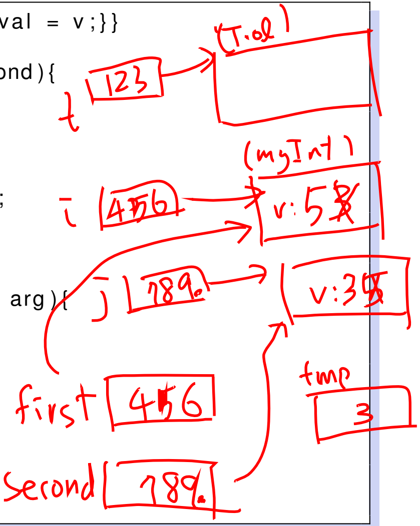
```
class Tool{
  bool tricky(String s1, String s2){
    s2 = s2 + "";
    return (s1 == s2);
  }
}
public class Demo{
  public static void main(String[] arg){
    Tool t = new Tool();
    String sa = "HTLin";
    String sb = sa;
    System.out.println(t.tricky(sa, sb));
    System.out.println(sa == sb);
    System.out.println(t.tricky(sa + "", sb));
  }
}
```



- reference parameter passing: again, value copying
- sa, sb copied to s1, s2
- s2 (reference) changed, sb didn't

Reference Argument/Parameter (2/3)

```
class myInt{int val; myInt(int v){val = v;}}
class Tool{
  void swap(myInt first, myInt second){
    int tmp = first.val;
    first.val = second.val;
    second.val = tmp;
    System.out.println(first.val);
    System.out.println(second.val);
  }
}
public class Demo{
  public static void main(String[] arg){
    Tool t = new Tool();
    myInt i = new myInt(3);
    myInt j = new myInt(5);
    t.swap(i, j);
    System.out.println(i.val);
    System.out.println(j.val);
  }
}
```



- swapped as requested

Reference Argument/Parameter (3/3)

```
class myInt{int val; myInt(int v){val = v;}}
class Tool{
  void swap(myInt first, myInt second){
    myInt tmp = first;
    first = second;
    second = tmp;
    System.out.println(first.val);
    System.out.println(second.val);
  }
}
public class Demo{
  public static void main(String[] arg){
    Tool t = new Tool();
    myInt i = new myInt(3);
    myInt j = new myInt(5);
    t.swap(i, j);
    System.out.println(i.val);
    System.out.println(j.val);
  }
}
```

- what happens?

Reference Argument/Parameter: Key Point

argument $\Rightarrow$ parameter: by reference copying
–value change: does not change argument
–content change: change the “object”

Array Argument/Parameter (1/1)

```
class Tool{
  void swap(int[] both){
    int tmp = both[0];
    both[0] = both[1];
    both[1] = tmp;
  }
}
public class Demo{
  public static void main(String[] arg){
    Tool t = new Tool();
    int[] arr = new int[2];
    arr[0] = 3; arr[1] = 5;
    t.swap(arr);
    System.out.println(arr[0]);
    System.out.println(arr[1]);
  }
}
```

- array is just special reference, same calling mechanism

Array Argument/Parameter: Key Point

argument $\Rightarrow$ parameter: by reference value copying
–same as reference argument/parameter

Static Variables Revisited (1/1)

```
1  class Record{
2    static int total_rec = 0;
3    int id;
4    public Record(){ id = total_rec++;}
5  }
6  public class RecordDemo{
7    public static void main(String[] arg){
8      Record r1 = new Record();
9      Record r2 = null;
10     Record r3 = new Record();
11     System.out.println(r1.total_rec);
12     System.out.println(r2.total_rec);
13     System.out.println(Record.total_rec);
14     System.out.println(r1.id);
15     System.out.println(r2.id);
16     System.out.println(Record.id);
17   }
18 }
```

11 yes
12 no, null
yes. class
13 yes
14 yes
15 no: null ptr ex.
16 no: compile error

- r2.total_rec ⇒ Record.total_rec in **compile time**

Static Variables Revisited: Key Point

`static` variable:
of the **class** (shared), not of an object

Static Methods (1/2)

```
class myMath{
  double mean(double a, double b){
    return (a + b) * 0.5;
  }
}
public class MathDemo{
  public static void main(String[] arg){
    double i = 3.5;
    double j = 2.4;
    myMath m = new MyMath();
    System.out.println(m.mean(i, j));
  }
}
```

- new a `myMath` object just for computing `mean`
  –lazy people don't want to do so

Static Methods (2/2)

```java
class myMath{
  static double mean(double a, double b){
    return (a + b) * 0.5;
  }
}
public class MathDemo{
  public static void main(String[] arg){
    double i = 3.5;
    double j = 2.4;
    System.out.println(myMath.mean(i, j));
    System.out.println((new myMath()).mean(i, j));
  }
}
```

- make the method a `static` (class) one
  –no need to new an object
- similar to static variable usage

Static Methods: Key Point

`static` method:
associated with the **class**,
no need to create an object

Use of Static Methods (1/3)

```
public class UtilDemo{
  public static void main(String[] arg){
    System.out.println(Math.PI);
    System.out.println(Math.sqrt(2.0));
    System.out.println(Math.max(3.0, 5.0));
    System.out.println(Integer.toBinaryString(15));
  }
}
```

- commonly used as utility functions
  (so don't need to create object)

Use of Static Methods (2/3)

```java
class Record{
  static int total_rec = 0;
  Record(){ total_rec++; }
  static void show_total_rec(){
    System.out.println(total_rec);
  }
}
public class RecordDemo{
  public static void main(String[] arg){
    Record r1 = new Record();
    Record.show_total_rec();
  }
}
```

- class related actions rather than object related actions

Use of Static Methods (3/3)

```
public class MainDemo{
  static void printStr(String s){
    System.out.print(s);
  }
  public static void main(String[] arg){
    printStr("I am main; I am also static");
  }
}
```

- main is static (called by classname during '`java className`')
- as tools for other static methods

Use of Static Methods: Key Point

`static` method:

- compile time determined
- per class
- sometimes useful

Instance versus Class (1/2)

```java
class Record{
  static int count;
  int id;
  Record(){ id = count++; }
  static void show_int(int num){
    System.out.println(num);
  }
  static void show_count(){
    show_int(count);
  }
  void show_id(){
    show_int(id);
  }
  void show_id_and_count(){
    show_id();
    show_count();
  }
}
```

- instance methods: can access all members/methods
- static methods: can access static members/methods

Instance versus Class (2/2)

```
//this: means my
class Record{
  static int count; int id;
  Record(){ this.id = Record.count++; }
  static void show_int(int num){
    java.lang.System.out.println(num);
  }
  static void show_count(){
    Record.show_int(Record.count);
  }
  void show_id(){
    Record.show_int(this.id);
  }
  void show_id_and_count(){
    this.show_id(); Record.show_count();
  }
}
```

- what compiler sees: a fully qualified intention
- `this`: the "object" reference (known in run-time)
- `Record`: the "class" name (known in compile-time)

Instance versus Class: Key Point

- null/non-null instance access class method/variable: as if **YES**
- class access instance method/variable method: **NO**
- instance method access class method/variable: **YES**
- class method access instance method/variable: **NO**

this (1/3)

```java
class Record{
  static int count;    int id;
  Record(){ this.id = Record.count++; }
  static void show_int(int num){
    java.lang.System.out.println(num);
  }
  static void show_id(Record r){
    Record.show_int(r.id);
  }
  void orig_show_id(){
    Record.show_int(this.id);
  }
}
public class RecordDemo{
  public static void main(String[] arg){
    Record r = new Record();
    r.orig_show_id();    Record.show_id(r);
  }
}
```

- a static implementation of show_id
- “almost” what the compiler/JVM does

this (2/3)

```
class Record{
  static int count;    int id;
  Record(){ this.id = Record.count++; }
  static void show_int(int num){
    java.lang.System.out.println(num);
  }
  static void show_id(Record THIS){
    Record.show_int(THIS.id);
  }
  void orig_show_id(){
    Record.show_int(this.id);
  }
}
public class RecordDemo{
  public static void main(String[] arg){
    Record r = new Record();
    r.orig_show_id();    Record.show_id(r);
  }
}
```

- implicitly, instance method can access an additional reference-type parameter `this` passed from the caller

this (3/3)

```
class Record{
  int score;
  static void show_score;
  void set_to(int score){ this.score = score; }
}
```

- this: the reference to specifically say “my” variable/method

this: Key Point

`this`: the reference used to specifically say “mine”
—implicit in every instance method

Recursive Calls (1/5)

```java
class Dir{ File[] files; Dir[] dirs;}
class File{    String name; }
public class DirDemo{
  static void listAll(Dir current){
    int i, j;
    for(i=0;i<files.length;i++)
      System.out.println(files[i].name);
    for(j=0;j<dirs.length;j++)
      listAll(dirs[j]);
  }
  public static void main(String[] arg){
    // ...
    Dir d = new Dir();
    listAll(d);
  }
}
```

- recursive: when a method calls itself

Recursive Calls (2/5)

```java
public class FibDemo{
  static int fib(int n){
    int res;
    System.out.println("fib(" + n + ") called");
    if (n <= 1)
      res = 1;
    else
      res = fib(n-1) + fib(n-2);
    System.out.println("fib(" + n + ") returning");
    return res;
  }
  public static void main(String[] arg){
    System.out.println(fib(5));
  }
}
```

- method call: do last task first (recursive or non-recursive)
- method call stack: implement “last task first” (last-in-first-out)

Recursive Calls (3/5)

```
public class FibDemo{
  static int fib(int n){
    int res;
    System.out.println("fib(" + n + ") called");
    if (n <= 1)
      res = 1;
    else{
      int m = n-1; res = fib(m); res = res + fib(n-2);
    }
    System.out.println("fib(" + n + ") returning");
    return res;
  }
  public static void main(String[] arg){ int a = fib(5); }
}
```

- what needs to be stored in each "frame"?
  - my local version of `n`
  - my local version of `res`
  - any temporary value generated by compiler (e.g. like `m`)
  - (my return value and where I am returning to)

Recursive Calls (4/5)

```
public class Fib{
  int N;
  Fib(int N){ this.N = N; }
  int get(){
    if (N <= 1)
      return 1;
    else
      return (new Fib(N-1)).get() + (new Fib(N-2)).get();
  }
  public static void main(String[] arg){
    Fib f = new Fib(20);
    System.out.println("res = " + f.get(5) );
    System.out.println("res = " + f.get(10) );
  }
}
```

- one advanced use: one class, two behaviors
  - utility behavior: static main
  - object behavior: with a state
- is `get` recursive? seems No (different objects) but actually Yes (same method with different "this")

Recursive Calls (5/5)

```java
public class Fib{
  int MAXN; int[] computed;
  Fib(int MAXN){
    this.MAXN = MAXN; computed = new int[MAXN+1]
  }
  int get(int n){
    if (computed[n] == 0){
      if (n <= 1) computed[n] = 1;
      else computed[n] = get(n-1) + get(n-2);
    }
    return computed[n];
  }
  public static void main(String[] arg){
    Fib f = new Fib(20);
    System.out.println("res = " + f.get(5) );
    System.out.println("res = " + f.get(10) );
  }
}
```

- recursive, but no repeated computation
- no need to create so many objects
- think: any better implementations?

Recursive Calls: Key Point

method call: not just “goto”
—comes with a frame of status passing, storing, manipulating, and returning

Local Variables (1/7)

```
public class FibDemo{
  int fib(int n){
    int res;
    String s = "fib(" + n + ")␣called";
    System.out.println(s);
    if (n <= 1)
      res = 1;
    else{
      res = fib(n-1);
      res = res + fib(n-2);
    }
    return res;
  }
  public static void main(String[] arg){
    FibDemo f = new FibDemo();
    System.out.println("res␣=␣" + f.fib(5));
  }
}
```

- local primitive `n`: allocated, and assigned by argument ⇒ parameter

Local Variables (2/7)

```java
public class FibDemo{
  int fib(int n){
    int res;
    String s = "fib(" + n + ") called";
    System.out.println(s);
    if (n <= 1)
      res = 1;
    else{
      res = fib(n-1);
      res = res + fib(n-2);
    }
    return res;
  }
  public static void main(String[] arg){
    FibDemo f = new FibDemo();
    System.out.println("res = " + f.fib(5));
  }
}
```

- local referece `this`: allocated, and assigned by argument (f) ⇒ parameter (this)

Local Variables (3/7)

```
public class FibDemo{
  int fib(int n){
    int res;
    String s = "fib(" + n + ") called";
    System.out.println(s);
    if (n <= 1)
      res = 1;
    else{
      res = fib(n-1);
      res = res + fib(n-2);
    }
    return res;
  }
  public static void main(String[] arg){
    FibDemo f = new FibDemo();
    System.out.println("res = " + f.fib(5));
  }
}
```

- local primitive `res`: allocated, **not** initialized assigned by ourselves

Local Variables (4/7)

```
public class FibDemo{
  int fib(int n){
    int res;
    String s = "fib(" + n + ") called";
    System.out.println(s);
    if (n <= 1)
      res = 1;
    else{
      res = fib(n-1);
      res = res + fib(n-2);
    }
    return res;
  }
  public static void main(String[] arg){
    FibDemo f = new FibDemo();
    System.out.println("res = " + f.fib(5));
  }
}
```

- local reference `s`: reference allocated, **not** initialized, point to a valid object by ourselves

Local Variables (5/7)

```java
public class FibDemo{
  int fib(int n){
    int res;
    String s = "fib(" + n + ") called";
    System.out.println(s);
    if (n <= 1)
      res = 1;
    else{
      res = fib(n-1);
      res = res + fib(n-2);
    }
    return res;
  }
  public static void main(String[] arg){
    FibDemo f = new FibDemo();
    System.out.println("res = " + f.fib(5));
  }
}
```

- some other local variables generated by compiler: allocated, **not** initialized, used internally

Local Variables (6/7)

```java
public class FibDemo{
  int fib(int n){
    int res;
    String s = "fib(" + n + ") called";
    System.out.println(s);
    if (n <= 1) res = 1;
    else res = fib(n-1) + fib(n-2);
    return res;
  }
  public static void main(String[] arg){
    FibDemo f = new FibDemo();
    System.out.println("res = " + f.fib(5));
  }
}
```

- when call returns: the result is "copied" to the previous frame somewhere
- local primitive: simply discarded
- local reference: discarded (and then the "object" GC'ed some time if no longer used)

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

OOP Chuen-Liang Chen, NTU CS&IE / 5

Local Variables: Key Point

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