Data Abstraction: Status of an Object

Hsuan-Tien Lin

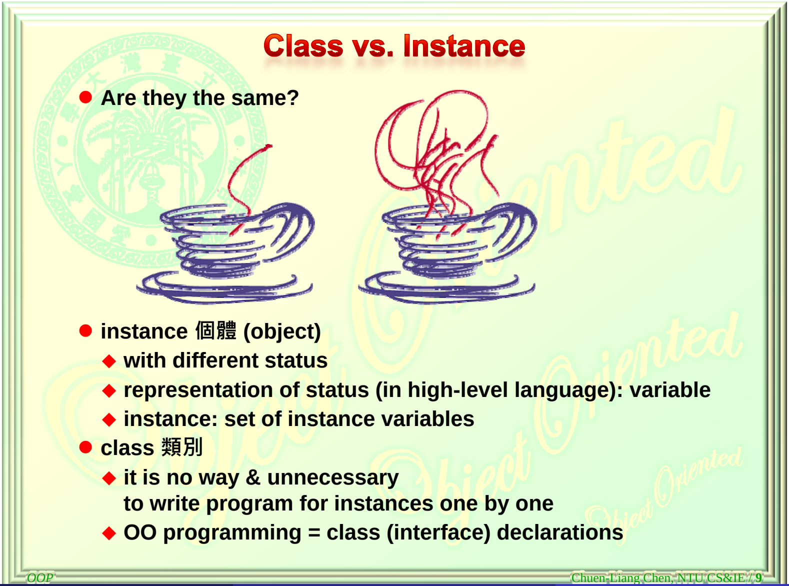
Deptartment of CSIE, NTU

OOP Class, March 10, 2009

What We Should Have Known

- very serious consequences when violating **the principle**
- **write-once, use forever**; not use-once, dump forever
- Java is not only a type of coffee, but also a **language**
- Java and C share many similarities, and **some differences**
- noodle-oriented programming is **not** (always) the best solution
- POP: organized CODE (procedure) + data
- **OOP**: organized DATA + organized code (ACTION)
- **one class, many instances**

Class versus Instances (courtesy of Prof. Chuen-Liang Chen)



Class vs. Instance

- Are they the same?
- instance 個體 (object)
  - with different status
  - representation of status (in high-level language): variable
  - instance: set of instance variables
- class 類別
  - it is no way & unnecessary
    to write program for instances one by one
  - OO programming = class (interface) declarations

OOP Chuen-Liang Chen, NTU CS&IE / 9

Class versus Instances

```
class Record{ //class
  String name; //variable declaration
  String ID; //variable declaration
  public bool isB86(){ //action
    return ID.startsWith("B86");
    //here ID is an instance of the class String
    //and performs an action (method) startsWith()
  }
}

Record r1 = new Record(); //r1 is an instance
                          //with r1.name and r1.ID
                          //as its data (variables)
Record r2 = new Record(); //r2 is another instance
Record[] rarray = new Record[3];

if (r2.isB86()) {} //r2 performs an action (method)
```

An OO Design of the RandomIndex class

- DATA: a randomly permuted index array of size N
- ACTION: setSize, initializeIndex, permuteIndex, getNext
- see `RandomIndex.java`
- now you can use it for name calling in class, distributing cards in POO games, etc.

You Have Seen Some Classes/Instances

- the `java.lang.String` class and its instances `"abc", "def"`
- the `java.lang.System` class, but no instances
- the `java.io.PrintStream` class, and one instance `System.out`

Read the API, **Guess**, and Write the progam you want

What is Data?

資料

準備要被處理的東西

一堆文字和數字

用來表現狀態

處理完的東西

已知物

程式需要的東西

分析處理後可得結果的東西

What is Data?

(Wikipedia) Data refer to a collection of facts usually collected as the result of experience, observation or experiment, or processes within a computer system, or a set of premises. This may consist of numbers, words, or images, particularly as measurements or observations of a set of variables. Data are often viewed as a lowest level of abstraction from which information and knowledge are derived.

data (in execution): memory interpretations
data (in language): variables

Data in Execution: Memory Interpretation (1)

- content before interpretation:
  e.g. bits 01010000010011110100111100000000
- type of interpretation:
  e.g. a little-endian integer (that occupies 32 bits)
- value after interpretation:
  e.g. 1347374848

Data in Execution: Memory Interpretation (2)

- content before interpretation:
  e.g. bits 01010000010011110100111100000000
- type of interpretation:
  e.g. a big-endian integer (that occupies 32 bits)
- value after interpretation:
  e.g. 5197648

Data in Execution: Memory Interpretation (3)

- content before interpretation:
  e.g. bits 01010000010011110100111100000000
- type of interpretation:
  e.g. a 0-terminated character array
- value after interpretation:
  e.g. "POO" (’P’, ’O’, ’O’, 0)

Data in Language: Variables

- variables: a (named) representation of data in language
- variable declaration: set type (and name)
  e.g. `int a; double b; String s;`
- variable assignment: alter content
  e.g. `a = 3;`
- variable evaluation: obtain value
  e.g. `if (4 == a + 2) ...;`

Type: Defining Memory Interpretation

- primitive type: what the language supports as a basic building block
- extended type:
    - e.g. String (as character array in C)
    - e.g. structures in C, classes in Java

Data Abstraction:
don't care (much) about what the bytes contain,
care about what the type **means**

Extended Type: What It Means?

```
class Record{
  String name;
  int score;
}
```

we intend to extract a `String` type memory space, and a `int` type memory space from a `Record` type variable

Eight Java Primitive Types

primitive type: defining direct memory interpretations

- `byte, short, int, long`: 8/16/32/64 bit (big-endian) integers
- `float, double`: 32/64 bit floating point numbers
- `boolean`: true or false
- `char`: 16 bit unicode

all (except boolean) very similar to C

Many Java Extended Types

`class WhatEverYouWant`

- `class 2DPoint`
- `class Record`
- `class java.io.PrintStream`

One Java Extended Type with Native Support

`class String`

- the same as any extended type you see
- native operation support (e.g. `+`)
- literals `"abc"` recognized by the language (much like `3.14`)
- some other special handling

But Wait!

we intend to extract a `String` type memory space, and a `int` type memory space from a `Record` type variable

What REALLY happens in the memory of JVM?

Declaration/Allocation/Assignment

primitive types:

```
int i; //declaration and allocation
i = 3; //assignment
```

Declaration/Allocation/Assignment

extended types:

```
Record r; //declaration
r = new Record(); //allocation and
                  //assignment (of instance)
r.score = 10; //assignment (of member)
```

Java has no pointer?!

Declaration/Allocation/Assignment

String type:

```
String s; //declaration
s = "123"; //assignment
           //(of instance and member)
```

Extended Type Revisited

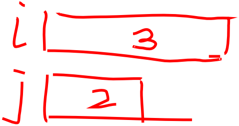
- each extended-type variable holds a **reference** (more restricted type of pointer) to the actual memory space at **declaration time**
- the extended-type variable won't point to a legitimate memory space unless we do **allocation** and **reference assignment**
- the extended-type variable does not contain meaningful data unless we do **member assignment**

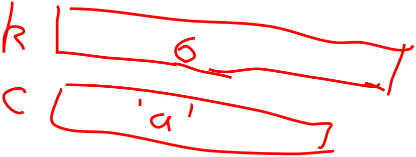
> Java: no pointer arithmetic, but yes pointer!
> (with nickname **reference**)!

Primitive Type Revisited

- each primitive-type variable holds a **allocated memory slot** at **declaration time**
- thus, the primitive-type variable always associates with a legitimate memory space
- the primitive-type variable does not contain meaningful data unless we do **value assignment**

Fun Time (1)

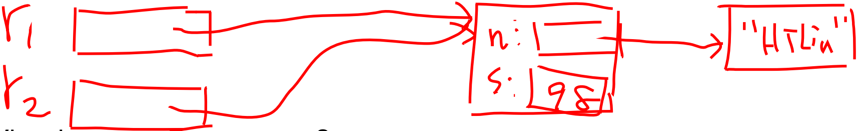




What happens in memory?

```
int i;
short j;
double k;
char c = 'a';
i = 3; j = 2;
k = i * j;
```

Fun Time (2)



What happens in memory?

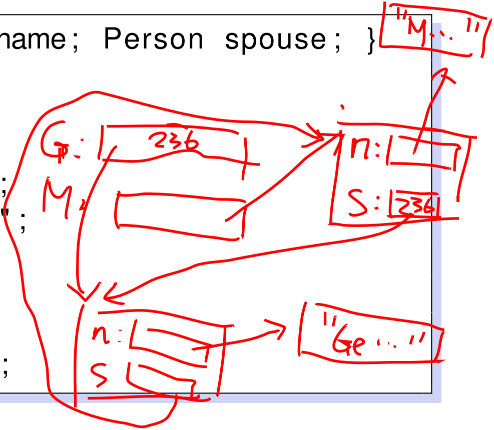
```
Record r1; //r1.name, r1.score
Record r2;
r1 = new Record();
r2 = r1;
r1.name = "HTLin";
r2.score = 98;
```

Fun Time (4)

What happens in memory?

```
class Person{ String name; Person spouse; }

Person George;
Person Marry;
George = new Person();
George.name = "George";
Marry = new Person();
Marry.name = "Marry";
Mary.spouse = George;
George.spouse = Marry;
```

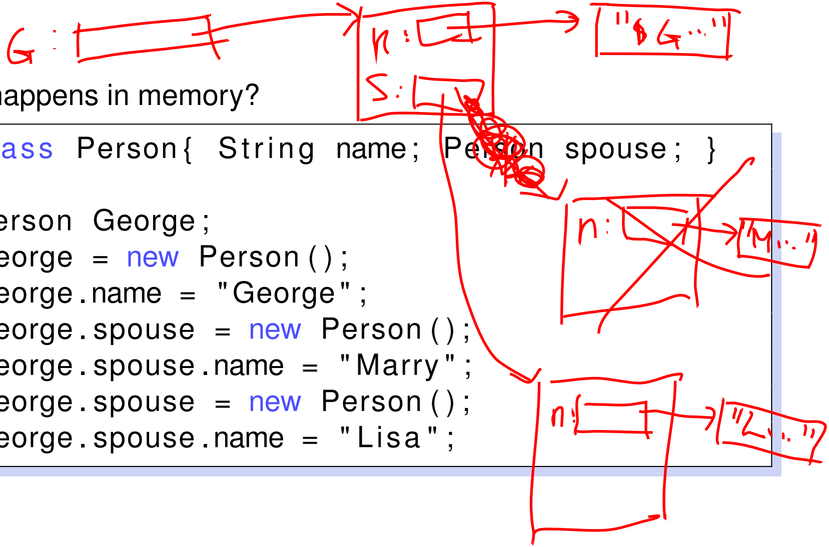


Fun Time (5)

What happens in memory?

```java
class Person{ String name; Person spouse; }

Person George;
George = new Person();
George.name = "George";
George.spouse = new Person();
George.spouse.name = "Marry";
George.spouse = new Person();
George.spouse.name = "Lisa";
```



Life Cycle of a Primitive Variable (C/Java)

- declared and created

```
int count;
```

- used and modified

```
count += 1;
```

- destroyed
  –automatically (when out of scope)

Life Cycle of an Object Instance (Java)

- reference declared

```java
Record r;
```

- instance created

```java
r = new Record();
```

- used and modified

```java
System.out.println(r.name);
```

- destroyed
  –automatically (when out of **use**)

Constructor (1/3)

```java
r = new Record();
```

- the `new` operator allocates memory for the instance
- often you will do this:

```java
r = new Record();
r.name = "HTLin";
r.score = 99;
```

- out of laziness, you want to do this:

```java
r = new Record("HTLin", 90);
```

How?

Constructor (2/3)

```
class Record{
  String name;
  int score;
  public Record(String init_name,
                int init_score){
    name = init_name;
    score = init_score;
  }
}

r = new Record("HTLin", 90);
```

Constructor (3/3)

- constructor: called by `new` to **initialize**
- name: same as class name
- remember the `public` (will come back to this later)
- default constructor (if you didn't write any code): same as

```
    public Record(){
    }
```

- constructor without argument ("replace" the default one):

```
    public Record(){
      score = 60;
    }
```

What Happens Here?

```
class Record{
  int total_rec;
  public Record(){
    total_rec += 1;
  }
  public void total_rec(){
    System.out.println(total_rec);
  }
}
...
Record r1 = new Record();
r1.total_rec();
Record r2 = new Record();
r2.total_rec();
```

r_1, r_2

total_rec 不同

1

1

What is the output?