Inheritance

Hsuan-Tien Lin

Deptartment of CSIE, NTU

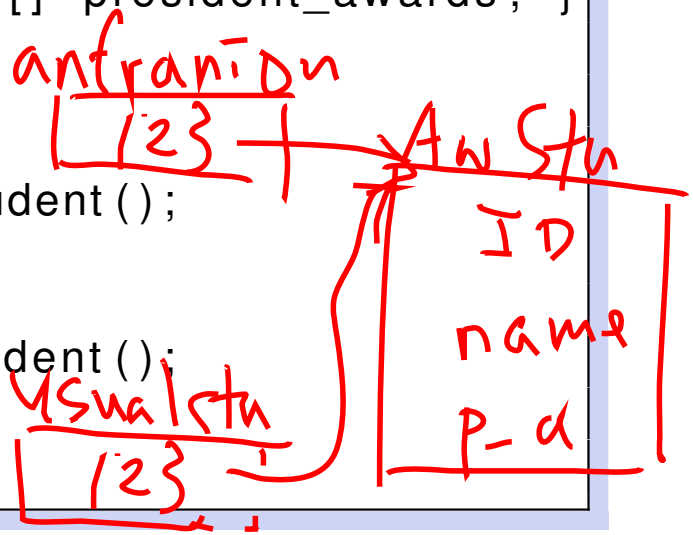
OOP Class, April 14, 2009

Java Type Hierarchy

See slides from last time

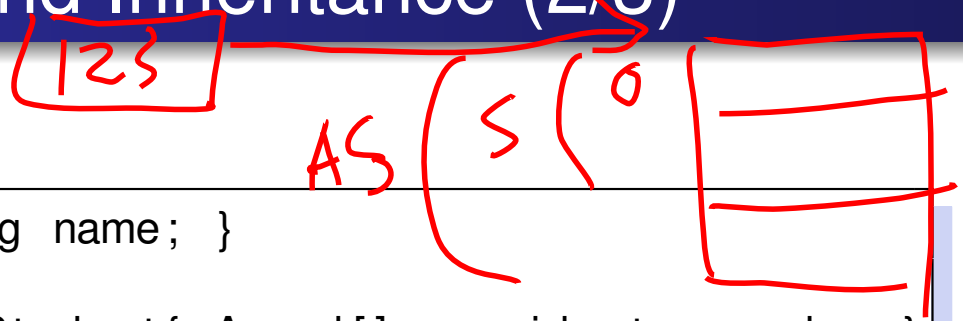
Reference Assignment and Inheritance (1/3)

```
1  class Student{ int ID; String name; }
2  class AwardStudent extends Student{ Award[] president_awards; }
3
4  public class StudentDemo{
5    public static void main(String[] ){
6      AwardStudent anfranion = new AwardStudent();
7      //I want to be a usual student
8      Student usualstudent = anfranion;
9      Student anotherusualstudent = new Student();
10   }
11 }
```

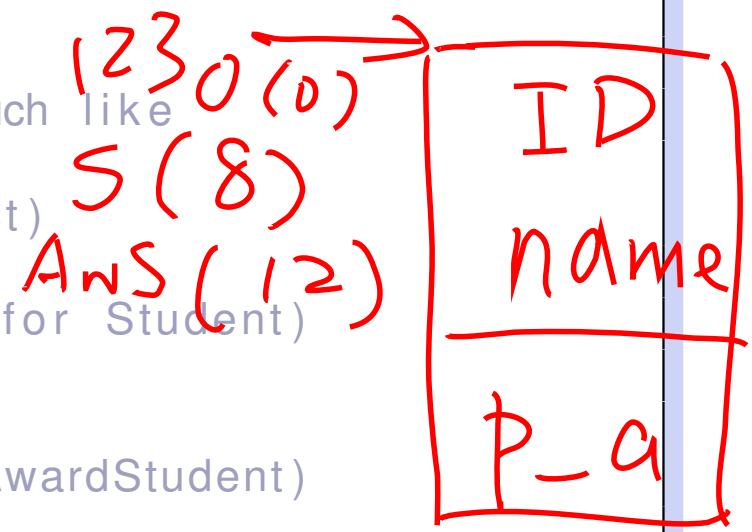


- if “copying assignment”, a copy of anfranion can be a usual student
- but “reference assignment” in Java, how can anfranion be usual?

Reference Assignment and Inheritance (2/3)



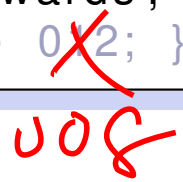

```
1  class Student{ int ID; String name; }
2  //takes 4 + (4) bytes
3  class AwardStudent extends Student{ Award[] president_awards; }
4  //takes 4 + (4) + (4) bytes
5
6  //excluding some other information, much like
7  class AwardStudent{
8    //first 0 bytes (for java.lang.Object)
9
10   //first 4 + (4) bytes (specifically for Student)
11   int ID;
12   String name;
13   //last (4) bytes (specifically for AwardStudent)
14   Award[] president_awards;
15 }//takes 4 + (4) + (4) bytes
```



- shared prefix mechanism for single inheritance (Java!!)

Reference Assignment and Inheritance (3/3)

```
1  class Student{ int ID; String name; }
2  //class Student{ int 000; reference 004; }
3  class AwardStudent extends Student{ Award[] president_awards; }
4  //class AwardStudent{ int 000; reference 004; reference 012; }
```



- one possible mechanism: variable name $\Rightarrow$ a single offset
- (after instance type check) objects can just safely access memory contents by constant offsets

Reference Assignment and Inheritance: Key Point

a simple run-time mechanism (shared prefix):
ancestor first, descendant last

Constructor and Inheritance (1/3)

```
1  class Student{ int ID; String name;
2    Student(int ID, String name){this.ID = ID; this.name = name;}
3  }
4  class AwardStudent extends Student{
5    Award[] awards;
6    AwardStudent(int ID, String name, int nAward){
7      super(ID, name); //means invoke Student(ID, name);
8      awards = new Award[nAward];
9    }
10 }
```

- initialize the ancestor part **first**
- construct AwardStudent $\Rightarrow$ (i.e. calls) construct Student $\Rightarrow$ construct Object
- thus, the “Object” parts of the memory initialized first, then the “Student” part, then the “AwardStudent” part

Constructor and Inheritance (2/3)

```
1  class Student{ int ID; String name;
2    Student(int ID, String name){this.ID = ID; this.name = name;}
3  }
4  class AwardStudent extends Student{
5    Award[] awards;
6    AwardStudent(int ID, String name, int nAward){
7      super(ID, name); //means invoke Student(ID, name);
8      //any utility function in Student can be used at this stage
9      awards = new Award[nAward];
10   }
11 }
```

- `super` goes first such that there is a valid “Student” object in the very beginning

Constructor and Inheritance (3/3)

```
1  class Student /* extends Object */ {
2    int ID; String name;
3    Student(int ID, String name){this.ID = ID; this.name = name;}
4    /*
5    Student(int ID, String name){
6      super(); //like Object();
7      this.ID = ID; this.name = name;
8    }
9    */
10 }
11 class AwardStudent extends Student{
12   Award[] awards;
13   AwardStudent(int ID, String name, int nAward){
14     //?      super();
15   }
16 }
```

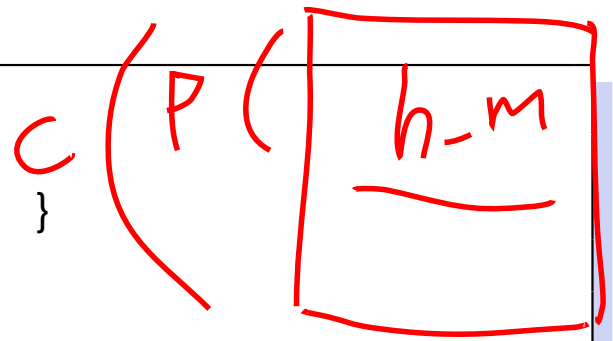
- `super()` automatically added if no explicit call in the beginning

Constructor and Inheritance: Key Point

calls ancestor construction first
(and thus ancestor forms first)

Private and Inheritance (1/1)

```
1  class Parent{
2    private int hidden_money;
3    public void show_hidden_money_amount(){ }
4  }
5  class Child extends Parent{
6    void spend_money(){
7      //can hidden_money be spent here? (a)
8      //can show_hidden_money_amount() be called here? (b)
9    }
10 }
```



- (a) by design, no (not directly); (b) yes
- does Child have a hidden_money slot in the memory?
  - yes, to make show_hidden_money_amount() work!
  - yes, to make the shared prefix mechanism work!

Private and Inheritance: Key Point

private variables are still “inherited” in memory,
but not “visible” to the subclass because of encapsulation

More on Encapsulation (1/2)

```
1  package generation.old
2  class Parent{
3    private int hidden_money;
4    public void show_hidden_money_amount(){ }
5    /* package */ void middle_age_issues();
6    /* package */ void cross_gen_issues();
7  }
8  //different file, Child.java
9  package generation.new
10 class Child extends Parent{
11 }
```

- in Child, can hidden_money be accessed? no.
- can show_hidden_money_amount be accessed? yes.
- can middle_age_issues be accessed? no.
- can cross_gen_issues be accessed? no.
  —what if we want “yes”?

More on Encapsulation (2/2)

```
1  package generation.old
2  public class Parent{
3    private int hidden_money;
4    public void show_hidden_money_amount(){ }
5    /* package */ void middle_age_issues();
6    protected void cross_gen_issues();
7  }
8  //different file, Child.java
9  package generation.new
10 class Child extends Parent{
11 }
```

public
protected
// package
private

protected
= package
+ inherited

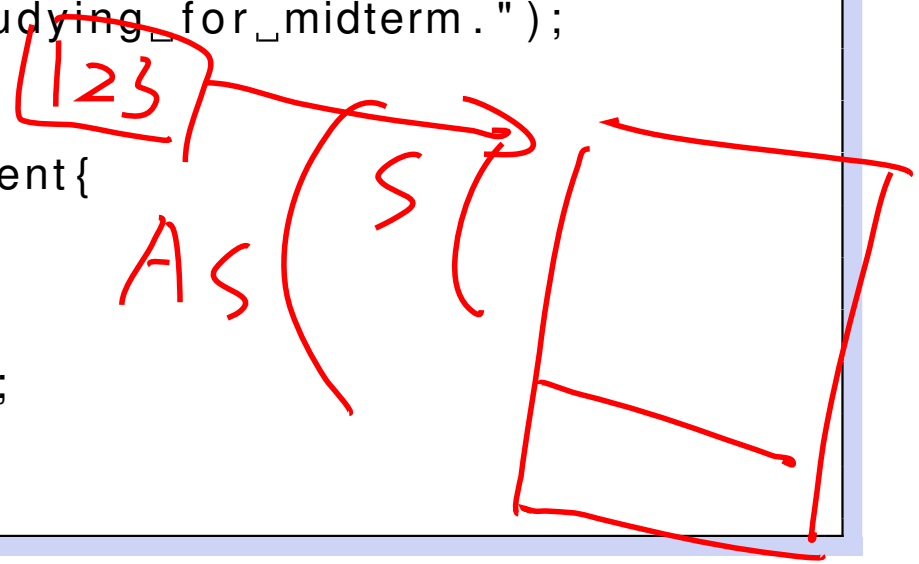
- can cross_gen_issues be accessed? no.
  —what if we want “yes”?
- `protected`: accessible to Child (sub-classes) and Friend (same-package-classes)

More on Encapsulation: Key Point

`protected`: accessible to sub-classes and same-package-classes

More on `super` (1/1)

```
1  class Student{
2    int ID; String name;
3    void study_for_midterm(){
4      System.out.println("I am studying for midterm.");
5    }
6  }
7  class AwardStudent extends Student{
8    Award[] president_awards;
9    void study_for_midterm(){
10     for(int i = 0; i < 10; i++)
11       super.study_for_midterm();
12   }
13 }
```



- `super`: much like `(ParentName)this` (but NOT the same)

More on `super`: Key Point

`this`: a Sub-type reference variable pointing to the object itself
`super`: a Base-type reference variable pointing to the object itself
same reference value, different type

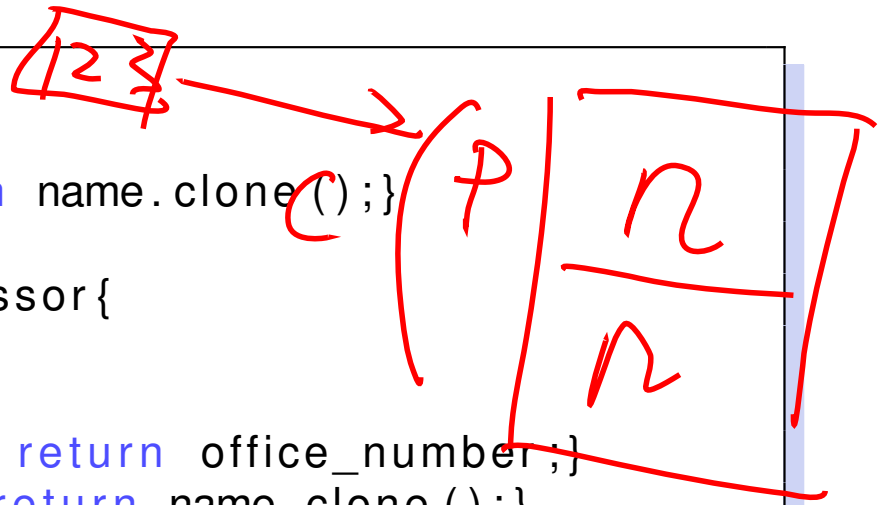
Instance Variables and Inheritance (1/3)

```
1  class Professor{
2    String name;
3
4    public String get_name(){
5      return name.clone();
6    }
7  }
8  class CSIEProfessor extends Professor{
9    int office_number;
10   public int get_office_number(){
11     return office_number;
12   }
13 }
```

- `CSIEProfessor`: two instance variables;
  `Professor`: one instance variable
- what gets changed if putting on different access modifiers?

Instance Variables and Inheritance (2/3)

```
1  class Professor{
2    String name;
3    public String get_name(){ return name.clone();}
4  }
5  class CSIEProfessor extends Professor{
6    String name; //what?
7    int office_number;
8    public int get_office_number(){ return office_number;}
9    public String get_this_name(){ return name.clone();}
10 }
11 /* CSIEProfessor HTLin = new CSIEProfessor();
12    Professor HTLin = new CSIEProfessor();
13    Professor HTLin = new Professor(); */
14 /* System.out.println(HTLin.get_name());
15    System.out.println(HTLin.get_this_name()); */
```



- `CSIEProfessor`: three instance variables;
  `Professor`: one instance variable
- which name will we get?

Instance Variables and Inheritance (3/3)

```
1  class Professor{
2    String p_name;
3
4    public String get_name(){
5      return p_name.clone();
6    }
7  }
8  class CSIEProfessor extends Professor{
9    String c_name;
10   public String get_this_name(){
11     return c_name.clone();
12   }
13 }
```

- an (almost) equivalent view of what the compiler sees

Instance Variables and Inheritance: Key Point

instance variable binding: static (i.e. determined at compile time)