# Abstraction: More Object Thinking

Hsuan-Tien Lin

Deptartment of CSIE, NTU

OOP Class, March 17, 2009

# More on `null` (1/3)

```
1  class Record{
2    String name;
3    String ID;
4    int score;
5  }
6
7  public class RecordDemo{
8    public static void main(String[] arg){
9      Record r1 = new Record();
10     System.out.println(r1.score); // 0
11     System.out.println(r1.name);  // null
12   }
13 }
```

- `null`: Java's reserved word of saying "no reference"
- default initial value for reference types (if initialized automatically)
- `0`, `NULL`, anything equivalent to integer 0: C's way of saying "no reference"

# More on `null` (2/3)

```
1  class Record{
2    String name;
3    String ID;
4    int score;
5  }
6
7  public class RecordDemo{
8    public static void main(String[] arg){
9      Record r1 = new Record();
10     r1.name = 0; //?
11     r1.name = (String)0; //?
12   }
13 }
```

- `null` is special, `null` is not 0
- any reference type: `null`, non-null (actual object)
- boolean type: `false`, `true`
- integer type: 0, non-zero (any other numbers)
- Java: separate the three; C: mix them as integers

# More on `null` (3/3)

```
1  class Record{
2    String name;
3    String ID;
4    int score;
5  }
6
7  public class RecordDemo{
8    public static void main(String[] arg){
9      Record r1 = null;
10     System.out.println(r1.score);
11     System.out.println(r1.name);
12   }
13 }
```

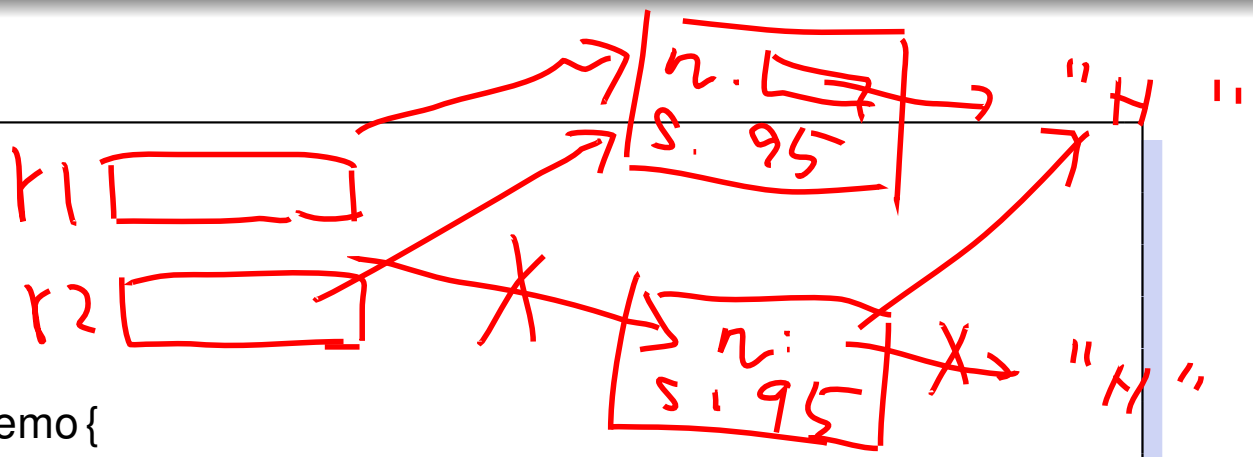
- null pointer exception (run time error): accessing the component of “no reference”

# More on `null`: Key Point

`null`: Java's special way of saying "no reference"

# Reference Equal (1/2)

```
1  class Record{
2    String name;
3    int score;
4  }
5
6  public class RecordDemo{
7    public static void main(String[] arg){
8      Record r1, r2;
9      r1 = new Record(); r2 = new Record();
10     r1.name = "HTLin"; r1.score = 95;
11     r2.name = "HTLin"; r2.score = 95;
12     System.out.println(r1 == r2);
13     r2 = r1;
14     System.out.println(r1 == r2);
15   }
16 }
```




- reference equal: comparison by “reference value”

# Reference Equal (2/2)

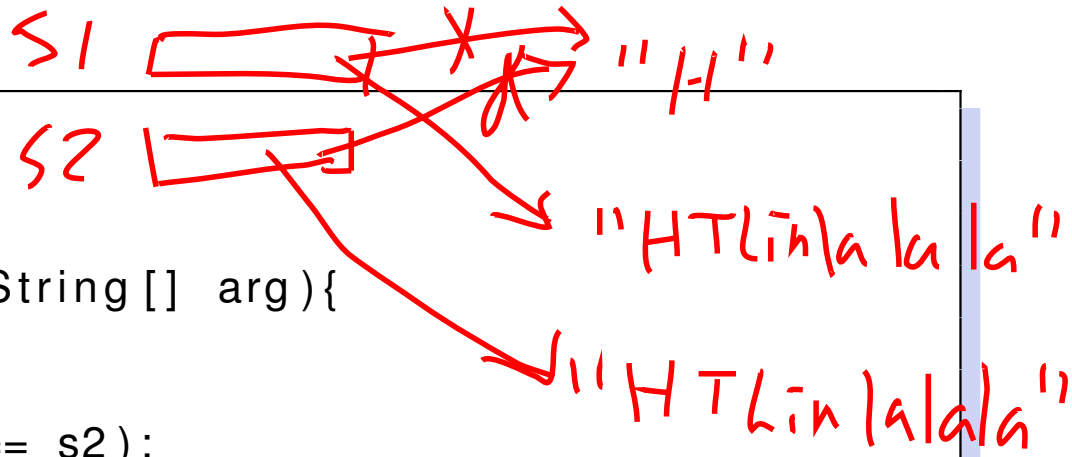
```
1  class Record{
2    String name;
3    int score;
4  }
5
6  public class RecordDemo{
7    public static void main(String[] arg){
8      Record r1, r2;
9      r1 = null; r2 = new Record();
10     System.out.println(r1 == r2);
11     r2 = r1;
12     System.out.println(r1 == r2);
13   }
14 }
```

- `null` does not equal non-null ..... o_O
- `null` equals `null` ..... O_o

# Reference Equal: Key Point

==: reference equal rather than
content equal for extended types

# String Equal (1/1)




```
1  public class StringDemo{
2    static String s1;
3    static String s2;
4    public static void main(String[] arg){
5      s1 = "HTLin";
6      s2 = "HTLin";
7      System.out.println(s1 == s2);
8      s1 = s1 + "lalala";
9      s2 = s2 + "lalala";
10     System.out.println(s1 == s2);
11     System.out.println(s1.equals(s2));
12   }
13 }
```

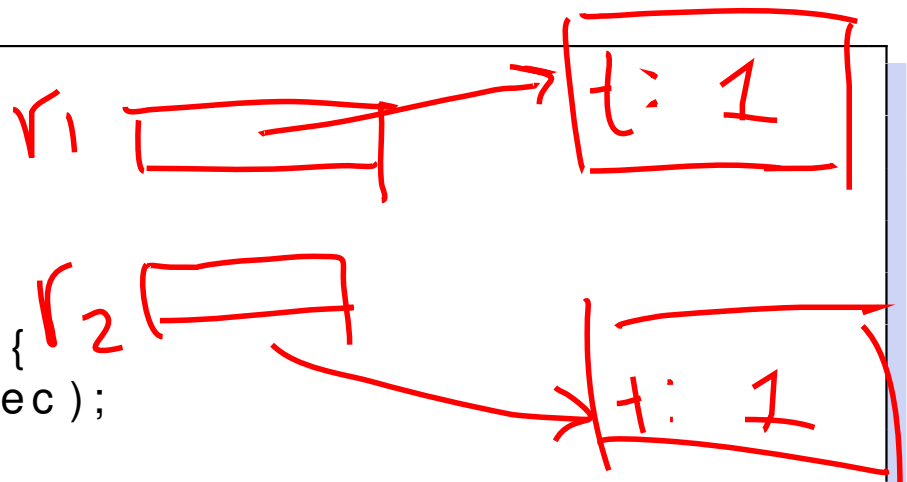
- first `true`: compiler allocates one constant string only
- second `false`: two different string references
- third `true`: an action (method) for content comparison

# String Equal: Key Point

String `==`: still reference equal, use `.equals` if want content equal

# Static Variables (1/3)

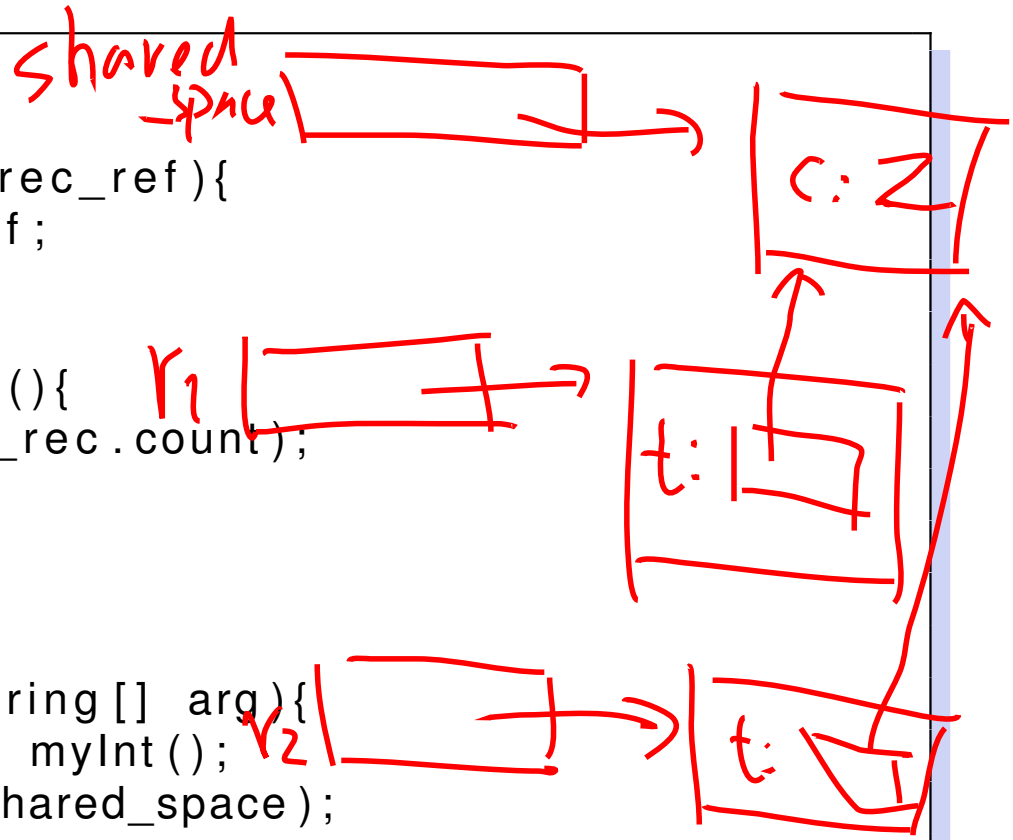
```
1  class Record{
2    int total_rec;
3    public Record(){
4      total_rec += 1;
5    }
6    public void show_total_rec(){
7      System.out.println(total_rec);
8    }
9  }
10 public class RecordDemo{
11   public static void main(String[] arg){
12     Record r1 = new Record();
13     r1.show_total_rec();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




- no shared space to store the total records

# Static Variables (2/3)

```
1  class Record{
2    myInt total_rec;
3    public Record(myInt total_rec_ref){
4      total_rec = total_rec_ref;
5      total_rec.count ++;
6    }
7    public void show_total_rec(){
8      System.out.println(total_rec.count);
9    }
10 }
11 class myInt{ int count; }
12 public class RecordDemo{
13   public static void main(String[] arg){
14     myInt shared_space = new myInt();
15     Record r1 = new Record(shared_space);
16     r1.show_total_rec();
17     Record r2 = new Record(shared_space);
18     r2.show_total_rec();
19   }
20 }
```




- do-able, but complicated, and requires many explicit steps in `main`

# Static Variables (3/3)

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

- `static`: shared between all X-type objects
- like a global variable within the scope of the class
- use scarcely

# Static Variables: Key Point

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

# Static Final Variables (1/3)

```
1  class Circle{
2    double r;
3    public Circle(double radius){
4      r = radius;
5    }
6    public void show_area(){
7      System.out.println(3.141592654 * r * r);
8    }
9    public void show_cir(){
10     System.out.println(2.0 * 3.141592654 * r);
11   }
12 }
13 public class CircleDemo{
14   public static void main(String[] arg){
15     Circle c = new Circle(3);
16     c.show_area();
17   }
18 }
```

- typing many 3.141592654 looks silly
- 3.141592654 does not need to be per-object

# Static Final Variables (2/3)

```
1  class Circle{
2    static double p = 3.141592654;
3    double r;
4    public Circle(double radius){ r = radius; }
5    public void show_area(){
6      System.out.println(p * r * r);
7    }
8    public void show_cir(){
9      System.out.println(2.0 * p * r);
10   }
11 }
12 public class CircleDemo{
13   public static void main(String[] arg){
14     Circle c = new Circle(3); c.show_area();
15     c.p = 10; c.show_area();
16   }
17 }
```

- prevention: don't use names `r`, `p`
- prevention: don't allow modify `p`

# Static Final Variables (3/3)

```
1  class Circle{
2    static final double p = 3.141592654;
3    double r;
4    public Circle(double radius){ r = radius; }
5    public void show_area(){
6      System.out.println(p * r * r);
7    }
8    public void show_cir(){
9      System.out.println(2.0 * p * r);
10   }
11 }
12 public class CircleDemo{
13   public static void main(String[] arg){
14     Circle c = new Circle(3);
15     c.show_area();
16     c.p = 10; //a typo here
17   }
18 }
```

- static final: Java's way of saying constant (over the class)

# Static Final Variables: Key Point

`static final` variable: **constant**

# More on Static Final Variables (1/1)

```
1  class MyDouble{
2    double val;
3    public MyDouble(double v){ val = v; }
4  }
5  class Circle{
6    static final MyDouble PI = new MyDouble(3.141592654);
7    double r;
8    public Circle(double radius){ r = radius; }
9    public void show_area(){
10     System.out.println(PI.val * r * r);
11   }
12 }
13 public class CircleDemo{
14   public static void main(String[] arg){
15     Circle c = new Circle(3); c.show_area();
16     c.PI.val = 10;
17     c.PI = new MyDouble(10);
18     c.show_area();
19   }
20 }
```

- final on “reference”, not final on “content”

# More on Static Final Variables: Key Point

`static final` reference variable:
constant on **refernce**, not on content

# Primitive Array (1/2)

```
1  public class ArrayDemo{
2    public static void main(String[] arg){
3      int[] arr = new int[3];
4      //think: intArray arr = new intArray(3);
5      arr[0] = 1; //think: arr.setElement(0, 1);
6      arr[1] = 3;
7      arr[2] = 5;
8      arr[3] = 9;
9      System.out.println(arr.length);
10     arr.length = 5;
11     arr = null;
12   }
13 }
```

- array is a reference by itself
- `new`, `null` like usual reference objects
- primitive array: `new` initialize element to default
- length: read-only
- index out of bound: run time error

# Primitive Array (2/2)

```
1  public class ArrayDemo{
2    public static void main(String[] arg){
3      int[] arr = {1, 3, 5};
4      //compare String s = "HTLin";
5      System.out.println(arr.length);
6    }
7  }
```

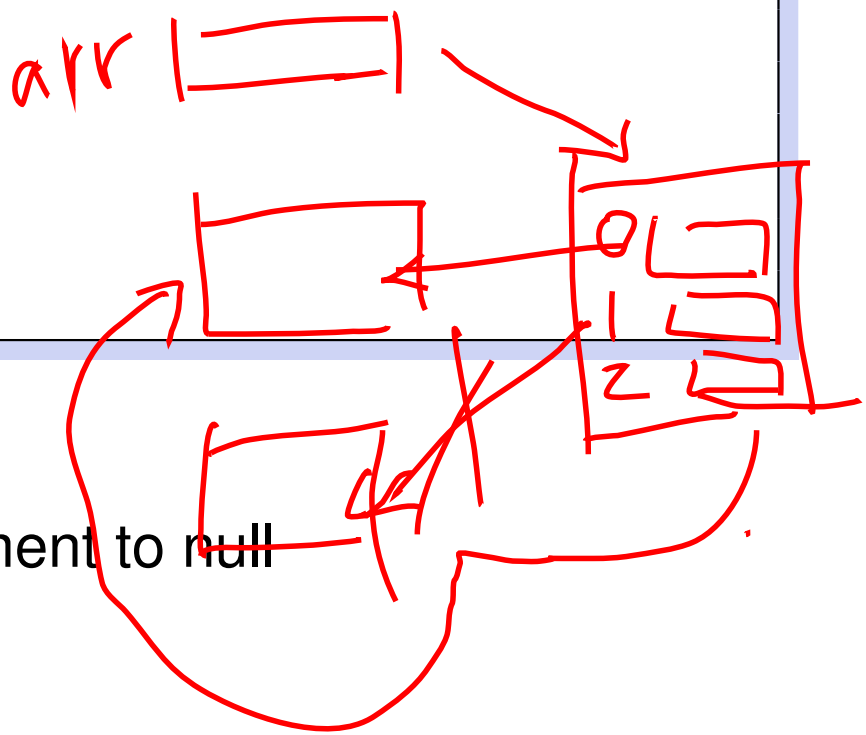
- construct an array object (with automatic length calculation), then assign its address to reference

# Primitive Array: Key Point

primitive array: reference to “a batch of values”

# Reference Array (1/1)

```
1  class Record{ String name; int score; }
2  public class ArrayDemo{
3    public static void main(String[] arg){
4      Record[] arr = new Record[3];
5      System.out.println(arr[0]);
6      arr[0] = new Record();
7      arr[1] = new Record();
8      arr[2] = arr[0];
9      arr[1] = null;
10     arr = null;
11   }
12 }
```




- array is a reference
- reference array: `new` initialize element to null

# Reference Array: Key Point

reference array: reference to “a batch of references”

# Multidimensional Array (1/3)

```
1  public class ArrayDemo{
2    public static void main(String[] arg){
3      int[][] arr = new int[3][];
4      //think: intArray[] arr = new intArray[3];
5      arr[0] = new int[5];//think arr[0] = new intArray(5);
6      arr[1] = arr[0];
7      arr[2] = null;
8      System.out.println(arr.length);
9      System.out.println(arr[1].length);
10   }
11 }
```

- multidimensional: array of “array references”
- can be irregular

# Multidimensional Array (2/3)

```
1  public class ArrayDemo{
2    public static void main(String[] arg){
3      int[][] arr = new int[3][5];
4      System.out.println(arr.length);
5      System.out.println(arr[1].length);
6    }
7  }
```

- still array of “array references”
- regular, automatic construction

# Multidimensional Array (3/3)

```
1  public class ArrayDemo{
2    public static void main(String[] arg){
3      int[][] arr = {null, {0, 1}, {2, 3, 4}};
4      System.out.println(arr.length);
5      System.out.println(arr[1].length);
6    }
7  }
```

- construct an array, and assign its address to reference

# Multidimensional Array: Key Point

multidimensional array: a special reference array, reference to “a batch of (multidimensional) arrays”

# More on Constructors (1/3)

```
1  class Record{
2    String name; int score;
3  }
4  public class RecordDemo{
5    public static void main(String[] arg){
6      Record r1 = new Record();
7      Record r2 = new Record;
8      System.out.println(r1.score);
9    }
10 }
```

- default value (when `new`): 0/false/null
- default constructor when no self-defined constructor

# More on Constructors (2/3)

```
1  class Record{
2    String name; int score;
3    Record(int init_score){score = init_score;}
4  }
5  public class RecordDemo{
6    public static void main(String[] arg){
7      Record r1 = new Record(60);
8      Record r2 = new Record();
9      System.out.println(r1.score);
10     System.out.println(r2.score);
11   }
12 }
```

- if there is self-defined constructor, no default one
- self-defined constructor: same-name, no return value

# More on Constructors (3/3)

```
1  class Record{
2    String name; int score;
3    Record(int init_score){score = init_score;}
4    Record(){ Record(40);}
5  }
6  public class RecordDemo{
7    public static void main(String[] arg){
8      Record r1 = new Record(60);
9      Record r2 = new Record();
10     System.out.println(r1.score);
11     System.out.println(r2.score);
12   }
13 }
```

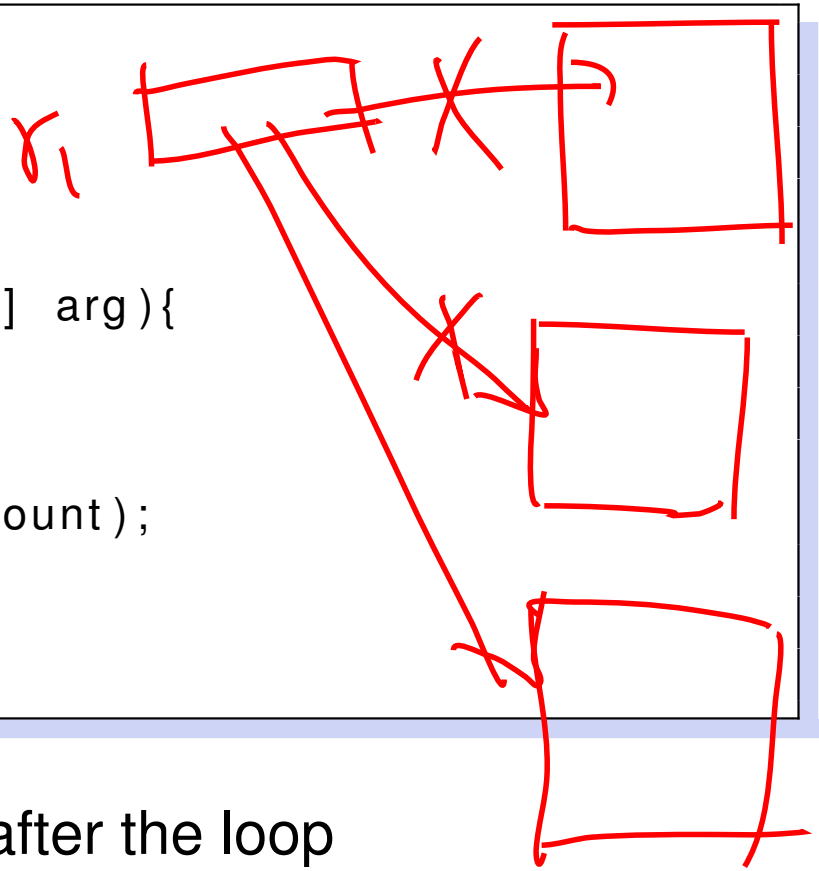
- can **overload**: same name, different parameters
- can call other constructors to help initialize

# More on Constructors: Key Point

often better to use self-defined and overloaded constructors to help initialize

# Garbage Collection (1/2)

```
1  class Record{
2    static int count = 0;
3    Record(){ count++; }
4  }
5  public class RecordDemo{
6    public static void main(String[] arg){
7      int i; Record r1;
8      for(i = 0; i < 100; i++){
9        r1 = new Record();
10       System.out.println(Record.count);
11     }
12   }
13 }
```




- 100 objects created, only 1 alive after the loop
- the other 99 memory slots: automatically recycled

# Garbage Collection (2/2)

```
1  class Record{
2    static int count = 0;
3    Record prev;
4    Record(){ count++; }
5  }
6  public class RecordDemo{
7    public static void main(String[] arg){
8      int i; Record r1 = null;
9      for(i = 0; i < 100; i++){
10       Record tmp = r1;
11       r1 = new Record();
12       r1.prev = tmp;
13       System.out.println(Record.count);
14     }
15   }
16 }
```

- 100 objects created, all of them alive

# Garbage Collection: Key Point

Garbage Collection: when a memory slot becomes an orphan (and) system in need of memory

# Finalizer (1/2)

```
1  class Record{
2    static int mem = 0;
3    Record(){ mem += 10; }
4    void when_truck_comes(){ mem -= 10; }
5  }
6  public class RecordDemo{
7    public static void main(String[] arg){
8      int i; Record r1;
9      for(i = 0; i < 100; i++){
10       r1 = new Record();
11       System.out.println(Record.mem);
12     }
13   }
14 }
```

- finalizer: something you want to do when truck comes
- calculate memory usage, write something back (say, on BBS), ...

# Finalizer (2/2)

```
1  class Record{
2    static int mem = 0; static int count = 0;
3    int id;
4    Record(){ mem += 10; count++; id = count; }
5    protected void finalize() throws Throwable{
6      System.out.print(id);
7      System.out.println(", Good Bye!");
8      mem -= 10;
9    }
10 }
11 public class RecordDemo{
12   public static void main(String[] arg){
13     int i; Record r1 = null;
14     for(i = 0; i < 100; i++){
15       Record tmp = r1; r1 = new Record();
16       System.out.println(Record.mem);
17     }
18   }
19 }
```

- GC: no guarantee on when the truck comes
- if JVM halts before truck comes, even no finalizer calls

# Finalizer: Key Point

Finalizer: a mechanism to let the object say goodbye