# Basic Java OOP

10/19/2015

Hsuan-Tien Lin (林軒田)
htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

# What We Have Done

- designing object (what variables? what methods?)
  —**[0] extended types**, **[1] method declaration/implementation**
- creating "first" object and calling its first action
  (done: `java ClassName` will call `main`)
- creating other objects —**[2] `new` and beyond**
- calling other objects (partially done: method invocation, will talk more in `[1]`)
- manipulating object status (partially done: instance variable assignments)
- deleting objects —**[TODO 3] object lifecycle**

# More on Constructors: Key Point

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

# [3] Object Lifecycle

# Garbage Collection (1/2)

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

# Garbage Collection (1/2)

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

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

# Garbage Collection (2/2)

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

# Garbage Collection (2/2)

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

- 100 instances 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  public class Record{
2    private int score;
3    public Record(){ sys.mem_usage += 10; }
4    public void when_truck_comes(){ sys.mem_usage -= 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     }
12   }
13 }
```

# Finalizer (1/2)

```
1  public class Record{
2    private int score;
3    public Record(){ sys.mem_usage += 10; }
4    public void when_truck_comes(){ sys.mem_usage -= 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     }
12   }
13 }
```

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

# Finalizer (2/2)

```
1  public class Record{
2    private int score;
3    public Record(){ sys.mem_usage += 10; }
4    protected void finalize() throws Throwable{
5      sys.mem_usage -= 10;
6      System.out.println("Good Bye!");
7    }
8  }
```

# Finalizer (2/2)

```
1  public class Record{
2    private int score;
3    public Record(){ sys.mem_usage += 10; }
4    protected void finalize() throws Throwable{
5      sys.mem_usage -= 10;
6      System.out.println("Good Bye!");
7    }
8  }
```

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

# Finalizer: Key Point

finalizer:
a mechanism to let the instance say goodbye

# Object Lifecycle (1/1)

```
1  public class Record{
2    private int score;
3    public Record(int init_score){ score = init_score; }
4    protected void finalize() throws Throwable{ }
5  }
6  public class RecordDemo{
7    public static void main(String[] arg){
8      Record r; //variable declared
9      r = new Record(60); //memory allocated (RHS)
10                         //and constructor called
11                         //variable linked (LHS)
12     r.show_score();     //instance action performed
13     r = null;//memory slot orphaned
14     //....
15                         //finalizer called
16                         //or JVM terminated
17   }
18 }
```

# Object Lifecycle: Key Point

we control birth, life, death, funeral design, but not the exact funeral time

# [4] Back to Class

# Static Variables (1/3)

```
1  public class Record{
2    private 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 }
```

# Static Variables (1/3)

```
1  public class Record{
2    private 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  public class RecordShared{
2    private int count;
3    public void increase_count(){ count++; }
4    public int get_count(){ return count; }
5  }
6  class Record{
7    RecordShared shared;
8    public Record(RecordShared s){
9      share = s; shared.increase_count();
10   }
11   public void show_total_rec(){
12     System.out.println(shared.get_count());
13   }
14 }
15 public class RecordDemo{
16   public static void main(String[] arg){
17     RecordShared shared_space = new RecordShared();
18     Record r1 = new Record(shared_space);
19     r1.show_total_rec();
20     Record r2 = new Record(shared_space);
21     r2.show_total_rec();
22   }
23 }
```

# Static Variables (2/3)

```
1  public class RecordShared{
2    private int count;
3    public void increase_count(){ count++; }
4    public int get_count(){ return count; }
5  }
6  class Record{
7    RecordShared shared;
8    public Record(RecordShared s){
9      share = s; shared.increase_count();
10   }
11   public void show_total_rec(){
12     System.out.println(shared.get_count());
13   }
14 }
15 public class RecordDemo{
16   public static void main(String[] arg){
17     RecordShared shared_space = new RecordShared();
18     Record r1 = new Record(shared_space);
19     r1.show_total_rec();
20     Record r2 = new Record(shared_space);
21     r2.show_total_rec();
22   }
23 }
```

- do-able, but complicated, and requires many explicit steps

# Static Variables (3/3)

```
1   public class Record{
2     private 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 Variables (3/3)

```
1  public class Record{
2    private 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 instances
- 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 instance

# Static Variables Revisited (1/1)

```
1  public class Record{
2    private static int total_rec = 0;
3    private 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 }
```

# Static Variables Revisited (1/1)

```
1  public class Record{
2    private static int total_rec = 0;
3    private 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 }
```

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

# Static Variables Revisited: Key Point

`static` variable:
of the **class** (shared), not of an instance; compile-time binding (i.e. static binding)

# Static Methods (1/2)

```
1  public class myMath{
2    public double mean(double a, double b){
3      return (a + b) * 0.5;
4    }
5  }
6  public class MathDemo{
7    public static void main(String[] arg){
8      double i = 3.5;
9      double j = 2.4;
10     myMath m = new MyMath();
11     System.out.println(m.mean(i, j));
12   }
13 }
```

# Static Methods (1/2)

```
1  public class myMath{
2    public double mean(double a, double b){
3      return (a + b) * 0.5;
4    }
5  }
6  public class MathDemo{
7    public static void main(String[] arg){
8      double i = 3.5;
9      double j = 2.4;
10     myMath m = new MyMath();
11     System.out.println(m.mean(i, j));
12   }
13 }
```

- new a `myMath` instance just for computing `mean` –unnecessary

# Static Methods (2/2)

```
1  class myMath{
2    static double mean(double a, double b){
3      return (a + b) * 0.5;
4    }
5  }
6  public class MathDemo{
7    public static void main(String[] arg){
8      double i = 3.5;
9      double j = 2.4;
10     System.out.println(myMath.mean(i, j));
11     System.out.println(( new myMath() ).mean(i, j));
12   }
13 }
```

# Static Methods (2/2)

```
1  class myMath{
2    static double mean(double a, double b){
3      return (a + b) * 0.5;
4    }
5  }
6  public class MathDemo{
7    public static void main(String[] arg){
8      double i = 3.5;
9      double j = 2.4;
10     System.out.println(myMath.mean(i, j));
11     System.out.println(( new myMath() ).mean(i, j));
12   }
13 }
```

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

# Static Methods: Key Point

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

# Use of Static Methods (1/2)

```
1  public class UtilDemo{
2    public static void main(String[] arg){
3      System.out.println(Math.PI);
4      System.out.println(Math.sqrt(2.0));
5      System.out.println(Math.max(3.0, 5.0));
6      System.out.println(Integer.toBinaryString(15));
7    }
8  }
```

# Use of Static Methods (1/2)

```
1  public class UtilDemo{
2    public static void main(String[] arg){
3      System.out.println(Math.PI);
4      System.out.println(Math.sqrt(2.0));
5      System.out.println(Math.max(3.0, 5.0));
6      System.out.println(Integer.toBinaryString(15));
7    }
8  }
```

- commonly used as utility functions
  (so don't need to create instance)
- main is static (called by classname during '`java className`')
- tools for other static methods

# Use of Static Methods (2/2)

```
1  class Record{
2    private static int total_rec = 0;
3    public Record(){ total_rec++; }
4    public static 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     Record.show_total_rec();
12   }
13 }
```

# Use of Static Methods (2/2)

```
1  class Record{
2    private static int total_rec = 0;
3    public Record(){ total_rec++; }
4    public static 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     Record.show_total_rec();
12   }
13 }
```

- class related actions rather than instance related actions

# Use of Static Methods: Key Point

`static` method:

- compile time determined (bound)
- per class
- sometimes useful

# Fun Time (1)

What happens in memory?

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

# Life Cycle of a Primitive Variable (C/Java)

- declared and created

```
1   int count;
```

# Life Cycle of a Primitive Variable (C/Java)

- declared and created

```
1  int count;
```

- used and modified

```
1  count += 1;
```

# Life Cycle of a Primitive Variable (C/Java)

- declared and created

```
1  int count;
```

- used and modified

```
1  count += 1;
```

- destroyed
  –automatically (when out of scope)

# Fun Time (2)

What happens in memory?

```
1 String s = "lalala";
2 String t = "abc";
3 String a = s + t;
```

# Fun Time (3)

What happens in memory?

```
1 Record r1; //r1.name, r1.score
2 Record r2;
3 r1 = new Record();
4 r2 = r1;  //how many records are there?
5 r1.name = "HTLin";
6 r2.score = 98;
```

# Fun Time (4)

What happens in memory?

```
1  class Person{ String name; Person spouse; }
2
3  Person George;
4  Person Marry;
5  George = new Person();
6  George.name = "George";
7  Marry = new Person();
8  Marry.name = "Marry";
9  Mary.spouse = George;
10 George.spouse = Marry;
```

# Fun Time (5)

What happens in memory?

```
1  class Person{ String name; Person spouse; }
2
3  Person George;
4  George = new Person();
5  George.name = "George";
6  George.spouse = new Person();
7  George.spouse.name = "Marry";
8  George.spouse = new Person();
9  George.spouse.name = "Lisa";
```

# Life Cycle of an Object Instance (Java)

- reference declared

```
1   Record r;
```

# Life Cycle of an Object Instance (Java)

- reference declared

```
1   Record r;
```

- instance created

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

# Life Cycle of an Object Instance (Java)

- reference declared

```
1  Record r;
```

- instance created

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

- used and modified

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

# Life Cycle of an Object Instance (Java)

- reference declared

```
1  Record r;
```

- instance created

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

- used and modified

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

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

# `null` Revisited (1/2)

```
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);
11     System.out.println(r1.name);
12   }
13 }
```

# `null` Revisited (1/2)

```
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);
11     System.out.println(r1.name);
12   }
13 }
```

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

# `null` Revisited (2/2)

```
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` Revisited (2/2)

```
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”

# `null` Revisited: 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 (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 }
```

# 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 }
```

# 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

# Reference Argument/Parameter (1/3)

```
1   class Tool{
2     bool tricky(String s1, String s2){
3       s2 = s2 + "";
4       return (s1 == s2);
5     }
6   }
7   public class Demo{
8     public static void main(String[] arg){
9       Tool t = new Tool();
10      String sa = "HTLin";
11      String sb = sa;
12      System.out.println(t.tricky(sa, sb));
13      System.out.println(sa == sb);
14      System.out.println(t.tricky(sa + "", sb));
15    }
16  }
```

# Reference Argument/Parameter (1/3)

```
1   class Tool{
2     bool tricky(String s1, String s2){
3       s2 = s2 + "";
4       return (s1 == s2);
5     }
6   }
7   public class Demo{
8     public static void main(String[] arg){
9       Tool t = new Tool();
10      String sa = "HTLin";
11      String sb = sa;
12      System.out.println(t.tricky(sa, sb));
13      System.out.println(sa == sb);
14      System.out.println(t.tricky(sa + "", sb));
15    }
16  }
```

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

# Reference Argument/Parameter (2/3)

```
1  class myInt{int val; myInt(int v){val = v;}}
2  class Tool{
3    void swap(myInt first, myInt second){
4      int tmp = first.val;
5      first.val = second.val;
6      second.val = tmp;
7      System.out.println(first.val);
8      System.out.println(second.val);
9    }
10 }
11 public class Demo{
12   public static void main(String[] arg){
13     Tool t = new Tool();
14     myInt i = new myInt(3);
15     myInt j = new myInt(5);
16     t.swap(i, j);
17     System.out.println(i.val);
18     System.out.println(j.val);
19   }
20 }
```

# Reference Argument/Parameter (2/3)

```
1  class myInt{int val; myInt(int v){val = v;}}
2  class Tool{
3    void swap(myInt first, myInt second){
4      int tmp = first.val;
5      first.val = second.val;
6      second.val = tmp;
7      System.out.println(first.val);
8      System.out.println(second.val);
9    }
10 }
11 public class Demo{
12   public static void main(String[] arg){
13     Tool t = new Tool();
14     myInt i = new myInt(3);
15     myInt j = new myInt(5);
16     t.swap(i, j);
17     System.out.println(i.val);
18     System.out.println(j.val);
19   }
20 }
```

- swapped as requested

# Reference Argument/Parameter (3/3)

```
1  class myInt{int val; myInt(int v){val = v;}}
2  class Tool{
3    void swap(myInt first, myInt second){
4      myInt tmp = first;
5      first = second;
6      second = tmp;
7      System.out.println(first.val);
8      System.out.println(second.val);
9    }
10 }
11 public class Demo{
12   public static void main(String[] arg){
13     Tool t = new Tool();
14     myInt i = new myInt(3);
15     myInt j = new myInt(5);
16     t.swap(i, j);
17     System.out.println(i.val);
18     System.out.println(j.val);
19   }
20 }
```

# Reference Argument/Parameter (3/3)

```
1  class myInt{int val; myInt(int v){val = v;}}
2  class Tool{
3    void swap(myInt first, myInt second){
4      myInt tmp = first;
5      first = second;
6      second = tmp;
7      System.out.println(first.val);
8      System.out.println(second.val);
9    }
10 }
11 public class Demo{
12   public static void main(String[] arg){
13     Tool t = new Tool();
14     myInt i = new myInt(3);
15     myInt j = new myInt(5);
16     t.swap(i, j);
17     System.out.println(i.val);
18     System.out.println(j.val);
19   }
20 }
```

- what happens?

# Reference Argument/Parameter: Key Point

argument $\Rightarrow$ parameter: by reference copying
**same for return value**

# this (1/1)

```
1  class Record{
2    int score;
3    void set_to(int score){ this.score = score; }
4    void adjust_score{ this.set_to(score+10); }
5  }
```

# this (1/1)

```
1  class Record{
2    int score;
3    void set_to(int score){ this.score = score; }
4    void adjust_score{ this.set_to(score+10); }
5  }
```

- which score? which set_to?
- `this`: my (the object's)

# this: Key Point

`this`: the reference variable pointing to the object itself