

Basic Java OOP

10/12/2015

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

- history/definition of Java
- basic OO needs
- a little bit “from C to Java”, with a hint on the usage of `import`

Java towards Basic OOP Needs

- 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 —**[3] object lifecycle**

[0] Types

Types (1/2)

- **primitive** (`int`, `double`, ...):
 - one single piece of data with literal support (e.g. `5.2`, `true`)
 - no extended actions except basic operations
- **extended (classes)**:
 - one or many pieces of data (instance variables)
 - all instances with the same pieces, but (possibly) with different values
 - extended actions (instance methods)
 - all instances with the same capability, but (possibly) with different behavior depending on status
 - declared (designed) by you, or used (`new`, action call) by your code
- Java is **strongly typed** with static type checking like C
 - every (instance/local/...) variable must have a type in the first place, and compiler checks type compatibility

Types (2/2)

```
class java.lang.String
```

- one extended type **with native support**
- the same as any extended type you see
- literals "abc" recognized by the language
- native operation support (e.g. +)
(roughly) "abc" + "def" is changed to

```
(new StringBuffer()).append("abc").append("def").toString()
```


by compiler
- some other special handling
—can even be used in `switch` statements since Java 7

Type: Key Point

type: confining variable to **(internal data + external actions)**, where extended type corresponds to classes, the core of Java programming

[1] Methods

Method (1/2, Callee's View)

```
1 public class OOPStudent{
2     private int score;
3     private String name;
4     public void set_score(int new_score){ score = new_score; }
5     public String get_name(){ return name; }
6     public double get_adjusted(){ return (60 + score * 0.4); }
7 }
```

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

Method (2/2, Caller's View)

```
1 public class OOPStudent{
2     private int score;
3     private String name;
4     public void set_score(int new_score){ score = new_score; }
5     public String get_name(){ return name; }
6     public double get_adjusted(){ return (60 + score * 0.4); }
7 }
```

```
1 public class OOPStudentDemo{
2     public static void main(String[] arg){
3         OOPStudent CharlieL = new OOPStudent();
4         CharlieL.set_score(80);
5         System.out.println(CharlieL.get_adjusted());
6         System.out.println(CharlieL.get_name());
7     }
8 }
```

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

Method: Key Point

method: meaning **action**, where information (message) is passed through argument/parameter and return values

Return Values (1/1)

```
1 public class OOPStudent{
2     private int score;
3     private String name;
4     public void set_score(int new_score){ score = new_score; }
5     public String get_name(){ return name; }
6     public double get_adjusted(){ return (60 + score * 0.4); }
7 }
```

```
1 public class OOPStudentDemo{
2     public static void main(String [] arg){
3         OOPStudent CharlieL = new OOPStudent();
4         CharlieL.set_score(80);
5         System.out.println(CharlieL.get_adjusted());
6         System.out.println(CharlieL.get_name());
7     }
8 }
```

- void: must-have to mean no return value
- primitive/extended return types possible

Return Values: Key Point

`void` for no return value, like C

Primitive Argument/Parameter (1/1)

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

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

Primitive Argument/Parameter: Key Point

argument \Rightarrow parameter: call by value (same as C)
—change in parameter does not change argument

Extended Argument/Parameter

to be discussed later

this (1/1)

```
1 public class OOPStudent{
2     private String name;
3     public String get_name(){ return name; }
4     public String get_the_name(OOPStudent WHO)
5         { return WHO.name; }
6 }
```

- `CharlieL.get_name()` **returns** `CharlieL.name`
- `Ptt.get_name()` **returns** `Ptt.name`
- **how?** a hidden parameter `WHO` can do

```
1     public String get_name(){ return this.name; }
2     public void set_score(int score){ this.score = score; }
```

- when accessing instance variables or calling instance methods within, an implicit `this` is assumed by compiler

this: Key Point

`this`: a hidden parameter in the method to keep in touch with the instance

Method Overloading (1/2)

```
1 public class Printer{  
2     public void printInteger(int a);  
3     public void printDouble(double a);  
4     public void printString(String a);  
5 }
```

- “Integer” and “int” are basically saying the same thing
- lazy people don't want to type so many words

Method Overloading (2/2)

```
1 public class Printer{
2     public void print(int a);
3     public void print(double a);
4     public int print(String a);
5     public void print(int a, int b);
6 }
7 public class PrinterThatCompilerSees{
8     public void print_int(int a);
9     public void print_double(double a);
10    public int print_String(String a);
11    public void print_int_int(int a, int b);
12 }
```

- Java's (and many modern language's) solution: one method name, many possible argument types
- called **method overloading**
- Java "signature" of a method: include name and parameters (but **NO** return types)

Method Overloading (2/2)

```
1  public class Printer{
2      public void print(int a);
3      public int print(int a);
4      public double print(int a);
5  }
6  public class PrinterDemo{
7      public static void main(String [] argv){
8          Printer p = new Printer();
9          p.print(P.print(3) + 5);
10         //which one do you want to call?
11     }
12 }
```

- determine programmer's intention from arguments: easy
- determine programmer's intention from return value: hard —can cast, can discard, etc.
- Java “signature” of a method: include name and parameters only
- compiler's job: from arguments (type), determine which method (name+parameters) to call
- cannot have two methods with the same signature

Method Overloading: Key Point

method overloading: a compiler's help by looking at "signature" rather than "name" in calling

Operator Overloading

```
1 public class Demo{
2     public static void main(String [] argv){
3         int a = 5;
4         System.out.println(3 + a); //plus(3, a)
5         System.out.println("" + a); //plus("", a)
6     }
7 }
```

- Java: a limited special case for String (actually, StringBuffer); the usual cases for primitive types; but not for other extended types
- C++: can overload almost “any” operator for any class
- double-sided sword: powerful, but easily misused

Operator Overloading: Key Point

operator overloading: very limited support in Java
(up to now, and possibly will be)

Local Variables (1/7)

```
1 public class FibCompute{
2     public int fib(int n){
3         int res;
4         String s = "fib(" + n + ")_called";    System.out.println(s);
5         if (n <= 1) res = 1;
6         else{    res = fib(n-1);    res = res + fib(n-2); }
7         return res;
8     }
9 }
```

```
1 public class FibDemo{
2     public static void main(String [] arg){
3         FibCompute f = new FibCompute();
4         System.out.println("res_=_ " + f.fib(5));
5     }
6 }
```

- a so-called **recursive** method
- local primitive `n`: allocated, and assigned by argument \Rightarrow parameter

Local Variables (2/7)

```
1 public class FibCompute{
2     public int fib(int n){
3         int res;
4         String s = "fib (" + n + ")_called";    System.out.println(s);
5         if (n <= 1) res = 1;
6         else{    res = fib(n-1);    res = res + fib(n-2); }
7         return res;
8     }
9 }
```

```
1 public class FibDemo{
2     public static void main(String[] arg){
3         FibCompute f = new FibCompute();
4         System.out.println("res_=_ " + f.fib(5));
5     }
6 }
```

- local extended `this`: allocated, and assigned by argument (f) \Rightarrow parameter (this)

Local Variables (3/7)

```
1 public class FibCompute{
2     public int fib(int n){
3         int res;
4         String s = "fib(" + n + ")_called";    System.out.println(s);
5         if (n <= 1) res = 1;
6         else{    res = fib(n-1);    res = res + fib(n-2); }
7         return res;
8     }
9 }
```

```
1 public class FibDemo{
2     public static void main(String [] arg){
3         FibCompute f = new FibCompute();
4         System.out.println("res_=_ " + f.fib(5));
5     }
6 }
```

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

Local Variables (4/7)

```
1 public class FibCompute{
2     public int fib(int n){
3         int res;
4         String s = "fib (" + n + ")_called";    System.out.println(s);
5         if (n <= 1) res = 1;
6         else{    res = fib(n-1);    res = res + fib(n-2); }
7         return res;
8     }
9 }
```

```
1 public class FibDemo{
2     public static void main(String[] arg){
3         FibCompute f = new FibCompute();
4         System.out.println("res_=_ " + f.fib(5));
5     }
6 }
```

- local extended `s`: allocated, **not** initialized, links to a valid instance by ourselves

Local Variables (5/7)

```
1 public class FibCompute{
2     public int fib(int n){
3         int res;
4         String s = "fib(" + n + ")_called";    System.out.println(s);
5         if (n <= 1) res = 1;
6         else{    res = fib(n-1);    res = res + fib(n-2); }
7         return res;
8     }
9 }
```

```
1 public class FibDemo{
2     public static void main(String[] arg){
3         FibCompute f = new FibCompute();
4         System.out.println("res_=_ " + f.fib(5));
5     }
6 }
```

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

Local Variables (6/7)

```
1 public class FibCompute{
2     public int fib(int n){
3         int res;
4         String s = "fib(" + n + ")_called";    System.out.println(s);
5         if (n <= 1) res = 1;
6         else{    res = fib(n-1);    res = res + fib(n-2); }
7         return res;
8     }
9 }
```

```
1 public class FibDemo{
2     public static void main(String [] arg){
3         FibCompute f = new FibCompute ();
4         System.out.println("res_=_ " + f.fib(5));
5     }
6 }
```

- when call returns: the result is “copied” to the caller’s space-of-interest somewhere
- local variables: discarded

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

Local Variables: Key Point

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

[2] new and Beyond

Constructor (1/3)

```
1 public class Record{
2     private String name;
3     private int score;
4 }
5 // ...
6 r = new Record();
```

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

```
1 r = new Record();
2 r.set_name("HTLin");
3 r.set_score(99);
```

- out of laziness, you want to do this:

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

How?

Constructor (2/3)

```
1  public class Record{
2      private String name;
3      private int score;
4      public Record(String init_name ,
5                     int init_score){
6          name = init_name;
7          score = init_score;
8      }
9  }
10 // ...
11 r = new Record("HTLin", 90);
```

Constructor (3/3)

- constructor: called by `new` to **initialize**
- name: same as class name
- default constructor (if you didn't write any code): same as

```
1 public Record(){  
2 }
```

- constructor without argument (“replace” the default one):

```
1 public Record(){  
2     score = 60;  
3 }
```

Constructor: Key Point

a special method to initialize the instance during
new