# More About Methods

Hsuan-Tien Lin

Deptartment of CSIE, NTU

OOP Class, March 8-9, 2010

# Method (1/2, Callee's View)

```
1 public class OOPStudent{
2     public int score;
3     public 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 object) do
- **parameter**: what I get from the caller
- return value: what I want to tell the caller

# Method (2/2, Caller's View)

```java
public class OOPStudent{
    public int score;
    public String name;
    public void set_score(int new_score){ score = new_score; }
    public String get_name(){ return name; }
    public double get_adjusted(){ return (60 + score * 0.4); }
}
```

```java
public class OOPStudentDemo{
    public static void main(String[] arg){
        OOPStudent CharlieL = new OOPStudent();
        CharlieL.set_score(80);
        System.out.println(CharlieL.get_adjusted());
        System.out.println(CharlieL.get_name());
    }
}
```

92

null

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

# Method: Key Point

method: an abstraction of **action**, where information is passed through argument/parameter and return values

# Return Values (1/1)

```
1 public class OOPStudent{
2     public int score;
3     public 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 }
```

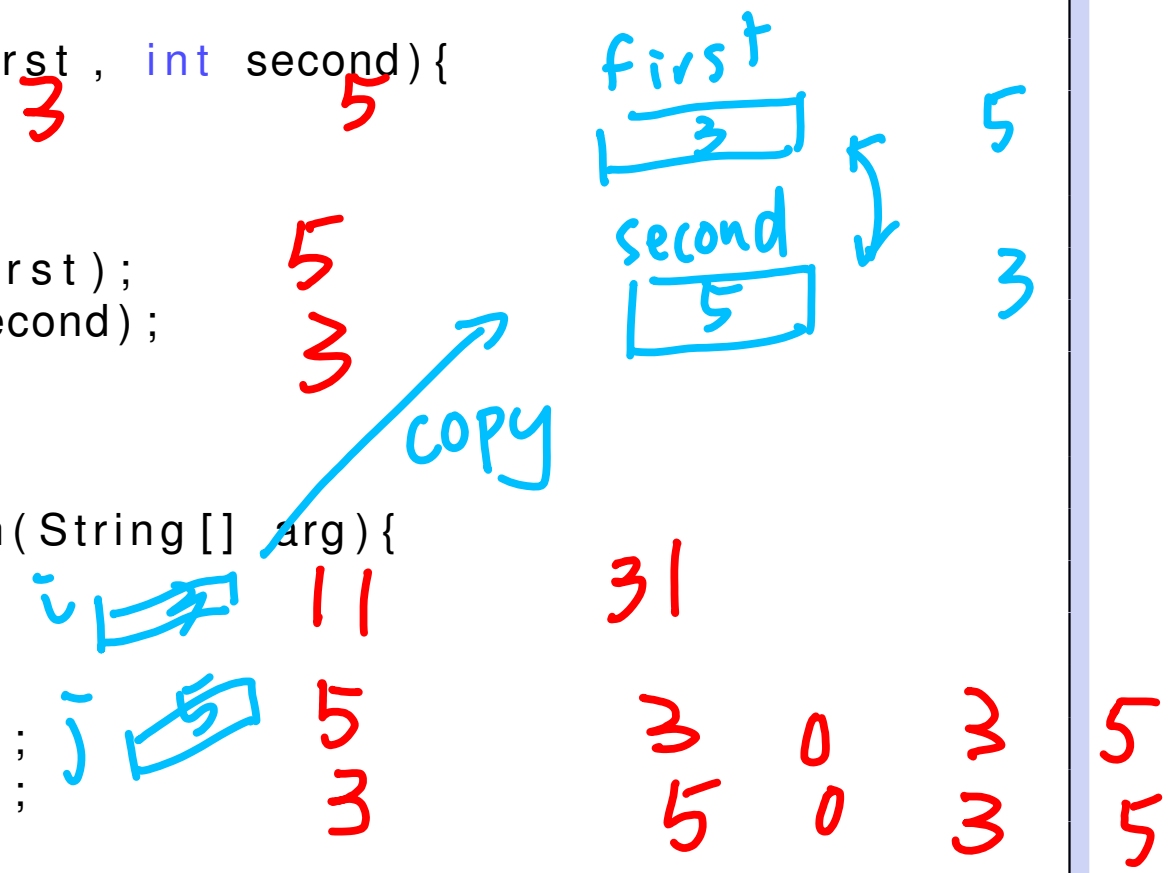
- `void`: must-have to mean no return value
- primitive/extended return types possible

# Return Values: Key Point

`void` for no return value

# 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

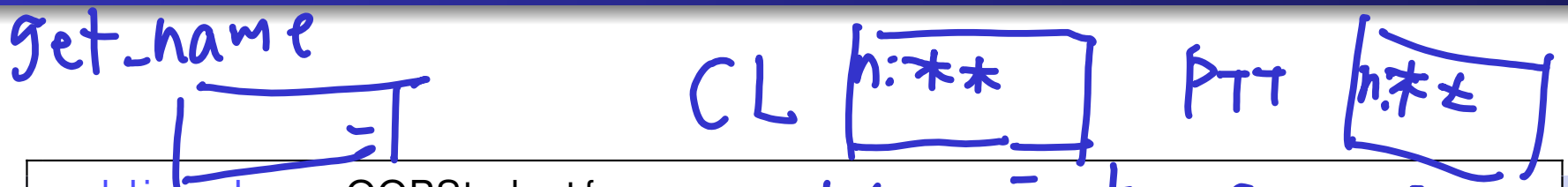
# Primitive Argument/Parameter: Key Point

argument $\Rightarrow$ parameter: call by value (copying)
–change in parameter does not change argument

# Extended Argument/Parameter

to be discussed in Chapter 5

# this (1/1)




```
1 public class OOPStudent{
2     public 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

OOPstu this

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

instance parameter

# `this`: Key Point

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

# 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 ⇒ 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

# Method Overloading (1/2)

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

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

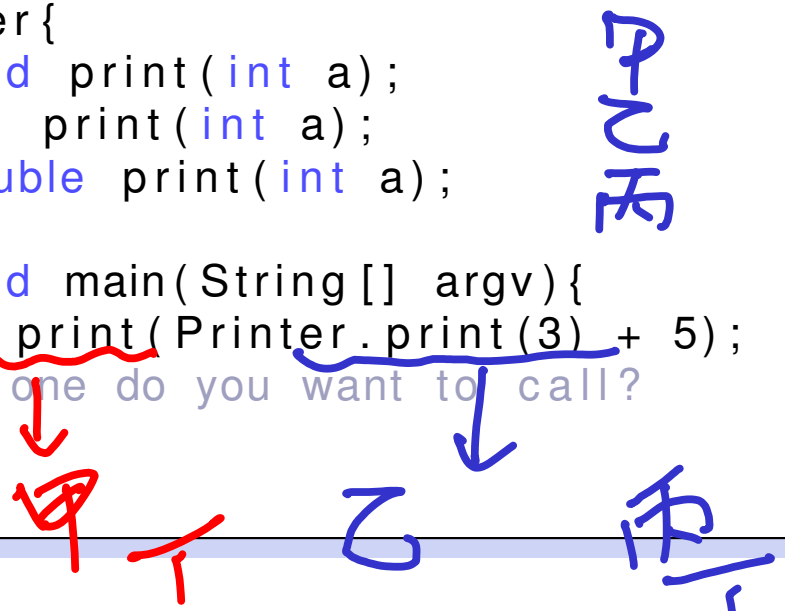
# Method Overloading (2/2)

```
1  class Printer{
2    void print(int a);
3    void print(double a);
4    int print(String a);
5    void print(int a, int b);
6  }
7  class PrinterThatCompilerSees{
8    void print_int(int a);
9    void print_double(double a);
10   int print_String(String a);
11   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  class Printer{
2    static void print(int a);
3    static int print(int a);
4    static double print(int a);
5
6    static void main(String[] argv){
7      Printer.print(Printer.print(3) + 5);
8      //which one do you want to call?
9    }
10 }
```



- 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  class Demo{
2    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  }
```

plus_int_int

plus_str_int

- 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)