NO: _____ DATE: _____

week4 : **Methods** [modularisation = process of building large system by decomposing them into smaller modules]

< Access modifier > <Return Type > < MethodName > (< Optional parameter >)

```
{
    // Statement block
}
```

method signature → method name
→ parameter

eg:
```
public void someMethod(int someParameter)   method header
{                                            → access modifier
    // code1                                 → return type
    // code2                                 → method name
}                                            → parameter.
```
method body

method overloading

→ 2 or more method in same class with same method name but diff method signature

eg:
```
public int returnTotal (int valOne)
public double returnTotal (int valOne, double valTwo)
```

return type

① void
→ no need to use "return"

② return a value
→ eg: int, String, etc.
→ use return statement
→ eg:
```
public int returnInt(int num1){
    return num1;
}
```

formal parameter

eg:
```
public int calculateTotal(int numOne){
    int total = numOne;
    return total;
}
```
local variable
actual argument

local variable - defined within method body, only accessible within that body

formal parameter - variable defined with specific data type that act as placeholder in method header.

actual argument - value that is passed when method executed, then substitute formal parameter

NO: _____ DATE: _____

Variable Scope & lifetime

	scope	lifetime
formal parameter & local variables	within their method	execution time of the method
class level variables eg: instance "	their class, including all method in the class	from object instantiation to obj destruction

* {} is used to (delimit) mark the start & end of the body of a method

* input & output
=> parameter & return value

method parameter = formal parameter

method argument = actual parameter

Week 5:

Memory, stack, heap

- primitive type data & Address data → store in Stack
- reference type data
  - → address store in stack
  - → object data store in heap

Arrays [reference variable]

elemType[] arrayName; eg: int[] array;

arrayName = new elemType[length]; array = new int[5];

OR

elemType[] arrayName = new elemType[length]; eg: int nums1[] = {1,2,3,4,5};

eg: int[] arr = new int[5];

① int array ② variable name ③ "new" keyword ④ int array's size.

→ array can't be resized once sized

→ all variable in array must be of same data type.

→ reference data type, storing address of the value in the variable.

	Placeholder	"\n" new line character
	%od Decimal integer	
	%f floating point	
	%s String	
	%15s right justified String	
	%-15s left justified String.	
	%.2f 2dp floating point	

print from last element of an array

```
int[] array= {1,2,3,4,5};
for (int i = array.length-1; i>=0 ; i--){
        System.out.println (array[i]);
}
```

To demonstrate side effect

```
public void calling(){
        int[] arr = new int[5];
Before  System.out.println(Arrays.toString(arr)); //reference variable
side    System.out.println( arr[0]); //value type variable
effect
        called(arr, arr[0]);
After   System.out.println( Arrays.toString (arr));
side            "       (arr[0]);
effect
}

public void called(int[] array, num){
        array[1] ++;
        num++;
}
```

=> value type variable hold data value of stack, when passed to a method, another copy is made & therefore they're pointing to diff obj, changes in one ~~x~~ affect the other.

=> reference variable hold the address in stack, when passed to a passed, the address is copied & both variable are now pointing to same obj, changes in one impact the other in array

=> arr[1] will change value, arr[0] will not