* Advanced switch statement.

```
switch (exp) {
case "    " -> Sout ("        ");
case "    " -> Sout ("        ");
default -> Sout ("        ");
```

07 - Functions / Methods in Java

① Methods

→ A method is a block of code which only runs when it is called.

→ we can pass data, known as parameters into a method.

→ Syntax

```
public class main {
    static void myMethod () {
}}
```

static ← means the method belong to Main class and not an object of Main class

void ↓ this method does not have return value

myMethod ↓ name of method

```
access - modifiers return-type method () {
        //code
     return Statement;
}
```

② Return type

→ A return stmt causes the program control to transfer back to the caller of a method

→ Return type may be primitive type like int, char, or void (returns nothing)

Few important things to understand about returning values.

(i) The type of data returned by a method must be compatible with the return type specified by the method.

17:50

eg: if return type of some method is boolean, we cannot return integer.

(ii) The variable receiving the value returned by a method must also be compatible with the return type specified for method.

✓

```
{int ans = sum(4,5)}
 sum(a,b){
   int c=a+b;
   return c;
 }
```

X

```
{String ans = sum(4,5)}
 sum(a,b){
   int c= a+b;
   return c;
 }
```

* code: Returing int -

```
public class Sum {
    psvm() {
        int ans = sum2();        ①
        S.out.print (ans);
    }}
  Static int sum2() {
        int num1 = sc.nesctInt();        ②
        int num2 = sc.nesctInt();
        sum = num1 + num2;
        return sum;
  }
```

* Returing string

```
public class String {
    psvm() {
        String message = greet();
        S.out.print (message);
    }
  static String greet() {
        String greeting = "        ";
        return greeting;
    }
}
```

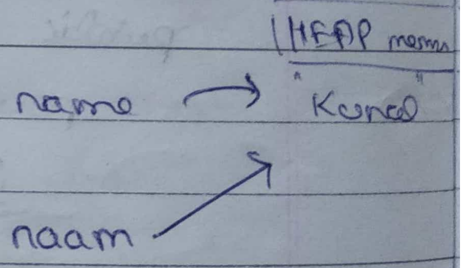
```
//using parameters
psum() {
    int ans = sum3(20,30);
    S.out.print(ans);
  }
Static int sum3(int a, int b){
    int sum= a+b;
    return sum;
  }
```

29:00

In Java, there is only pass by value and not pass by reference

Here the value is passed ①

```
main(){          ↗ object
   name = " Kunal"
   greet(name);
 }
greet (naam){
   Sout(naam);
 }
```



① A copy of the value of ref variable is passed

```
main() {                                  ① name → "Ujwal"
① String name = "Ujwal";
   changename(name);                      ② naam ↗
   Sout (name);                           ③ naam → Rahul
}
② static void changename(String naam){
③      naam = "Rahul"
}
output:  Ujwal
```

③ ③ Here we are not changing, we are creating new object.

Primitives: int, short, char, byte → just pass value
Objects & stuff : pass value of the reference
eg: name → Ujwal
naam ↗

* not changing value.

```
psvm() {
*  a=10;  ①                 ① a → 10
   b=20;                    ② num1 ↗
   swap(a,b)                ① b → 20
}                           ② num2 ↗

swap(num1, num2){ ②
    temp = num1;  ③         ③ temp = 10
    num1 = num2;  ④         ④ num~~2~~1 = 20
    num2 = temp   ⑤         ⑤ num2 = 10
}
```

→ Here the no will not be swapped because num1 and num2 is swapped and not a&b

* Changing original value

```
psvm {
  main() {
*   int[] arr = {1, 3, 2};  ①
     change (arr);
     Sout (Arrays.toString(arr));
  }
  static void change (int[] nums) {  ②
    nums[0] = 99;     ③
  }}
```

① arr → [1, 3, 2]

② nums ↗

③ nums → [99, 3, 2]

if you make a change to the object via this ref variable, <u>same obj will be changed</u>.

45.54 In last example, we didn't changed the object (string value). But here we changed

* Scope

(1) Function scope

Variables declared inside a method / function scope (inside method) <u>cannot</u> be accessed outside the method.

```
eg: psvm () {

    }
    all() {
      int x;
    }
```

cant be accessed outside

(2) Block scope

```
psvm() {
int a = 10;
int b = 20;
 { int a = 5  ×
    a = 5  ✓
   int c = 20  ✓
 }
   c = 10;  ×
 int c = 15; ✓
   a = 50;  ✓
 }
```

→ variables initialized outside the block can be updated inside the box (block)

→ variables initialized inside the block cannot be updated outside the block but can be reinitialized outside

→ variables like 'a' here, is declared outside the box can be updated inside and outside the block.

(3) Loop scope.
Variables declared inside the loop are having loop scope.

(4) Shadowing

→ Shadowing in Java is the practice of using variables in overlapping scopes with the same name where the variable in low level scope overrides the variable of high level scope.

→ Here the variable at high level scope is shadowed by low-level scope variable.

→ eg:

```
~~psvm(){~~
  ~~static int x=90;~~
  public class Shadowing {
      static int x = 90;
      psvm(){
      sout(x);   //90
      int x;        →        here high-level scope
      x=40;                    is shadowed by
      S.out(x);   //40         low level scope.
      fun();
   }
     static void fun(){
      S.out(x);         //90
   }
```

```
OUTPUT :   90
           40
           90
```

* Variable Arguments (VarArgs)

⇒ Variable argument is used to take a variable number of arguments. A method that takes a variable number of arguments is a varargs method.

→ Syntax:

```
Static void fun (int ..... a){
        // method body
    }
```

→ Here parameters would be array of type int[]

→ eg:

```
psvm(){
  fun (2,3,4,56,7);
}
Static void fun (int ...v){
   Sout ( Arrays.toString(v));
}}
```

* Function overloading:

→ It happens when 2 functions have same name

eg:
```
psvm(){
     sum (3,4)
     sum (3,4,5)
}
Static int sum (int a, int b){
     return a+b;
 }
Static int sum (int a, int b, int c){
       return a+b+c;
 }
```

* Prime number

→ eg: n = 36

1×36 = 36
2×18 = 36
3×12 = 36
4×9 = 36
6×6 = 36

9×4 = 36
12×3 = 36
18×2 = 36
36×1 = 36

repetition

```
for (i=2; i<=n; i++){
    if( n%i==0)
        { print (not prime)}
```

Here it will check from 1 to 36

more optimal

```
for (i=2; i<=√n ; i++)
    {  // code  }
```

Here it will check from 1 to 6

→ if I am checking the 2×18 = 36 then there is no need to check whether 18×2=36. If we do so, we are repeating our steps.

→ The time complexity is reduced to half.

→
```
Start
input n
if (n<=1)
        print( neither prime nor composite)
    c=2
while c*c <=n;            // or   c <= √n
        if n%c==0;
            output (not prime)
            exit
        c=c+1;
    end while
    output 'prime'
exit
```