Switch Statements:

→ Used to simplify of writing multiple if-statements inorder to do single/multiple tasks.

↳ can jump to various cases based on your expression.

```
switch (expression) {

    case one :
            // code block.
            break; // terminate the sequence.
    case two:
            // code block
            break.
    :
    default :
            // code block.
```

↳ it will execute when none of the above does.

↳ if this [default] is not at the end put a break after it - [else not required (if at the end)]

*→ if "break" statement is not written then it will continue to execute the other cases also.

*→ Here duplicate cases are not allowed.

Ex:

```
case one :
        //code
        break;
case one :
        //code
        break;
```

→ ⊗

New Syntax for switch

```
switch (expression) {
    case one → // code block ;
    case two → // code block ;
    default → // do this;
}
```

[Note: ~~not for~~

- equals() → Checks only the value.

Ex: String x = "hi";
x.equals("hi")
// returns true.

"==" → checks the value & reference of strings]

* Cases should be as same type of expressions.

Ex:-

```
public class Switch{
  psv m( ){
   Scanner sc= new Scanner(System.in);
   String fruit =in.next()
   switch(fruit){
      case "Mango":
            S.o.p("King of fruits");
            break;
      case "Apple":
            S.o.p("A sweet red fruit");
            break;
      case "Orange":
            S.o.p("Round fruit");
            break;
      default :
            S.o.p("Enter valid fruit");
   }
  }
}
```

O/P Mango
↳ King of fruits.

Switch case for weekends & weekdays.

Ex:

```
int day = in.nextInt();
switch (day) {
    case 1, 2, 3, 4, 5 → S.o.p("Week day");
    case 6, 7          → S.o.p("Weekend");
}
```

Nested Switch

↳ Switch with in another switch statements are called nested-switch statements.

Ex:

```
int emplD = in.nextInt();
String dept = in.next();
switch (emplD) {
    case 1: S.o.p("Mxxx");
            break;
    Case 2:
            S.o.p("Xavier");
            break;
    Case 3:
            S.o.p("Emp no. 3");
            Switch (dept) {
                Case "CSE":
                    S.o.p("CSE dept");
                    break;
                default : S.o.p("No dept entered")
                } break;
    default : S.o.p("Invalid");
}
```

Block of code, runs only when called.
↳ to reuse codes

Methods.

A method is a block of code which is repetitive. Which means we can use the method as many times as possible as per our requirement without writing extra lines of code everytime.

Ex: Sum of 2 numbers.

```
public class Sum{
  psvm(){
    Scanner in = new Scanner();
    int first = in.nextInt();
    int second = in.nextInt();
    int summation = sum(first, second);
    S.o.p(summation);
  }
  public static int sum(int f, int s){
      return f+s;
  }
}
```

// We can use this method to find sum of 2 numbers as many times as possible.
↳ if we use Scanner for i/p's & method.

(or)

```
public class Sum{
  psvm(){
    ~~S.o.p(sum());~~   sum();
    ~~S.o.p(sum());~~   sum();
  }
  public static void sum(){
    S.o.p("enter 1st number");
    int n1 = in.nextInt();
    S.o.p("enter 2nd number");
    int n2 = in.nextInt();
    S.o.p("sum = " + (n1+n2));
  }
}
```

Syntax

```
<access specifier> <return type> <method-name(arguments){
        // body
        return statement.
}
```

Return type: It is the type of the value that the method holds at the end (or) returning value's type.

* return statement is the last statement of the method.
  - ↳ Which indicates function is completed.
  - ↳ Anything that's written after return shall not be executed.

Ex:

```
public class StringEx{
    psvm(){
        String msg = greet();
        S.o.p(msg);
    }
    static String greet(){   // it is a method of String type, so it is returning a String
        String greeting = "Hi";
        return greeting;
    }
}
```

Pass-by-Value Ex.

name $\xrightarrow{\text{points}}$ "Xavier"

a $\xrightarrow{\text{points}}$ "Xavier"

Ex1

```
public class PassingEx{
    p s vm(){
        String name = "Xavier";
        greet(name);    // greet("Xavier")
                        ie; the copy of value of reference variable
                            is passed here.
    }
    static void greet(String a){ // Here Xavier will be
                                    replaced by a.
        S.o.p(a);                   internally.
    }
}
```

O/p: Xavier

Ex2

name $\xrightarrow{\text{points}}$ "Xavier"

n $\xrightarrow{\text{points}}$ ~~"Rahul"~~ "Alexander"

```
public class PassingEx{
    p s vm(){
        String name = "Xavier";
        changeName(name);
        S.o.p(name); // Here o/p is Xavier only
    }
    Static void ~~greet~~(changeName(String n){
        n = "Alexander";
    }
}
```

[Note: String is immutable
↓
Which cannot be changed].

O/p: Xavier.

Here name is not being changed to Alexander because; Here n is creating another object "n". (Which is not trying to change the original name/object.

for primitives

↳ It is pass-by-value.

for non-primitives

↳ It is also pass-by-value only, whereas the value of the reference variable is passed by both pointing to same object.

// Ex:- Changing values of an array through methods.

Ex. Swapping

```
PSVM()
{ a = 10;
  b = 20
  swap(a,b);
  S.o.p(a+" " +b); //Here a & b will not swap a=10, b=20 only.
}
swap(int n1, int n2) {
int temp;
  temp = n1; // temp = 10;
  n1 = n2;   // n1 = 20;
  n2 = temp; // n2 = 10;
}
```

this is not in scope, so the a & b won't change.

Explanation

a → 10
b → 20

These are swapped only in functions scope

→ To swap a & b, we can simply add print statement to print the swapped a & b's.

Note: Then also a & b were swapped in that block only.

Scope:

The fields that are going to be accessible within a particular range only.

↳ Out-side of that one cannot access them.

Scope

Block

↳ int a = 10
int b = 20;
{ int a = 30 // error
int c = 3;
a = 50;
S.o.p(a) // a = 30 }

// c is available in that block only.

↳ We cannot reinitialize same variable

↳ But we can update them.

↳ Values initialized in this block, will remain in ther only.

Loop

↳ Loop variables & fields are accesible only in that loop but not outside it.

for(int i=0; i<n; i++)
{
}

↳ i is valid in for() only.

↳ if try to access i outside of that compiler will through an error.

Ex: i is not declared

Method

↳ The fields that are present in a method those are valid & accesible in that scope only.

Ex: sum()
{ int a, b, c
c = a+b;
S.o.p(c);
}

Here a, b, c are available in scope of that method only.

(or)

sum(int a, int b)
{ int c = a+b;
}

Here a & b are scoped to that method only.

Shadowing

↳ Using the variables those having same name with in the scope that overlaps. then variable of higher level scope will be shadowed

Ex. public class Shadow {
static int x = 100;
p v s m() {
S.o.p(x); // 100
int x = 9;
S.o.p(x); // 9 } }

↳ Shadowing begins whenever the local variable is initialized.

Variable Arguments → Takes any no. of arguments
↳ Stores in form array internally

If we don't know how many no. of arguments to pass in a method then we can use variable arguments.

Ex:
```
public class VarArgs {
    psvm() {
        fun(1, 2, 3, 4, 30, 90, 100);
    }
    static void fun(int... v) {
        S.o.p(Arrays.toString(v)); //[1,2,3,4,30,
                                     90, 100].
    }
```

↳ Variable no. arguments should always be at the end in methods.

Ex:
```
fun(int a, int b, int ...v); ✓
fun(int a, int ...v, int b); ✗
```

↳ Takes 0 or 1 arguments

↳ It cannot be empty when we use ~~overriding~~ / overloading