3. Flow Control

→ Flow control describes the order in which statements will be executed at runtime.

Flow-control

1. Selection Statements
   1. if-else
   2. switch
2. Iterative Statements
   1. while
   2. do-while
   3. for loop
   4. for-each loop (1.5v) *
3. Transfer Statements
   1. break
   2. continue
   3. return
   4. try-catch-finally
   5. assert

1. Selection Statements :-

1. if-else :-

Syntax :-

```
if (b)        → boolean data type.
{
   Action if b is true
}
else
{
   Action if b is false
}
```

→ The argument into if statement should be boolean type. If we are providing any other type we will get compile time error.

Ex:

```
int x=0;
if (x)
{
  S.o.p("Hello");
}
else
{
  S.o.p("Hi");
}
```

CE: incompatible types
found : int
required : boolean

```
int x=10
if (x=20)
{
  S.o.p("Hello");
}
else
{
  S.o.p("Hi");
}
```

```
int x=10
if (x==20)
{
  S.o.p("Hello");
}
else
{
  S.o.p("Hi");
}
```

o/p : Hi

```
boolean b = true;
if (b = false)
{
  S.o.p ("Hello");
}
else
{
  S.o.p ("Hi");
}
```

o/p : Hi ✓

```
boolean b = false;
if (b == false)
{
  S.o.p ("Hello");
}else{
  S.o.p ("Hi");
}
```

o/p : Hello. ✓

→ Both else part and curly braces are optional. Without curly braces we can take only one statement under if, which should not be declarative statement.

Ex:

```
if (true)
  S.o.p("Hello");
```
✓

```
if (true);
```
✓

```
if (true)
  int x=10;
```
✗

```
if (true)
{ int x=10;
}
```
✓

*** Note:- ; (semicolon) is a valid java statement, which is also known as Empty statement.

2. switch statement :-

→ If several options are available then it is never recommended to use if-else, we should go for switch statement.

→ The advantage of this approach is readability will be improved.

Syntax:-

```
switch (x)
{
  case 1:
       Action1;
       break;
  case 2:
       Action2;
       break;
    :
    :
  case n:
       Action n;
       break;
  default:
       default Action;
}
```

→ The valid argument types for switch statement are byte, short, char, int. But, this rule is applicable until 1.4 version.

→ But, from 1.5 v onwards corresponding wrapper classes and enum types also allowed.

1.4 v	1.5 V	1.7 V
byte	Byte	
short	Short	String
char	Character	
int	Integer	
	+	
	enum	

→ Curly braces are mandatory.

→ switch is the only place where curly braces are mandatory.

→ Within the switch both case and default are optional.

Ex:
```
int x=10;
switch(x)
{
}
```
✓

→ Every statement inside switch should be under some case or default i.e., we can't write independent statements inside switch.

Ex:-
```
int x=10;
switch(x)
{
    S.o.p ("Hello");
}
```
✗

CE: case, default, or '}' expected.

→ Every case label should be compile time constant. If we are taking variable as case label then we will get compile time error.

Ex:
```
int x=10;
int y=20;
switch(x)
{
  case 10:
       S.o.p("10");
       break;
  case y:
       S.o.p("20");
}
```
→ CE: constant expression required.

→ If we declare y as final then we won't get any compile time error.

Ex:-
```
int x=10;
final int y=20;
switch(x)
{
  case 10: S.o.p("10");
  case y: S.o.p("20");
}
```
→ Both switch argument and case label can be expressions. But, case label should be constant expression.

Ex:
```
int x=10;
switch(x+1)
{
  case 10: S.o.p(10);
           break;
  case 10+1: S.o.p(11);
}
```
→ Every case label should be within the range of switch argument type. O.W, we will get compile time error.

Ex:

```
byte b=10;
switch(b)          // byte (-128 to 127) range is
{
  case 10: S.o.p("10");
  case 100: S.o.p("100");
  case 1000: S.o.p("1000");
}
```

CE: PLP
found : int
required : byte

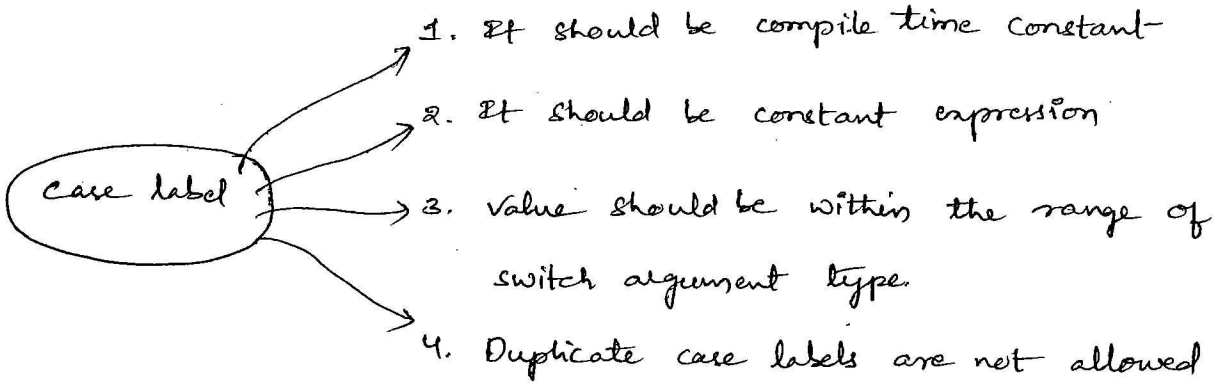
```
byte b=10;
switch(b+1)        // int
{
  case 10 : S.o.p("10");
  case 100 : S.o.p("100");
  case 1000 : S.o.p("1000");
}
```

→ Duplicate case labels are not allowed.

Ex:

```
int a=10;
switch(a)
{
  case 97 : S.o.p("97");
  case 98 : S.o.p("98");
  case 'a' : S.o.p("a");
}
```

CE: duplicate case label



Fall through inside switch :—

→ Within the switch if any case or default is matched from that case onwards all statements will be executed until break or end of the switch. This is called Fall through inside switch.

→ The main advantage of fall through inside switch is we can define common action for multiple cases. (Code reusability).

Ex①:
```
switch (x)
{
  case 1:
  case 2:
  case 3: S.o.p ("Q-1");
          break;
  case 4:
  case 5:
  case 6: S.o.p ("Q-2");
          break;
}
```

Ex②:
```
switch (x)
{
  case 0: S.o.p ("0");
  case 1: S.o.p ("1");
          break;
  case 2: S.o.p ("2");
  default: S.o.p ("def");
}
```

x=0	x=1
0	1
1	
x=2	**x=3**
2	def
def	

default case :—

→ Within the switch we can write default case atmost once.

→ If no other case matched then only default case will be executed.

→ Within the switch we can write default case anywhere but, it is convention to write as last case.

Ex:
```
switch(x)
{
  default : S.o.p("def");
  case 0 : S.o.p ("0");
           break;
  case 1 : S.o.p ("1");
  case 2 : S.o.p ("2");
}
```

x=0	x=1
0	1
	2
x=2	**x=3**
2	def
	0

2. Iterative Statements :—

1. while loop :—

→ If we don't know no. of iterations in advance then the best suitable loop is while loop.

Ex:
```
while(it.next())        | while(itr.hasNext())    | while(e.hasMoreElements())
{                       | {                       | {
  ≡                     |   ≡                     |   ≡
}                       | }                       | }
```

Syntax:—
```
while(b)   → boolean type
{
  ≡
}
```

→ The argument to the while loop should be boolean type. If we are trying to provide any other type then we will get compile time error.

Ex:
```
while(1)
{
  S.o.p("Hello");
}
```
CE: incompatible types
found : int
required : boolean

→ Curly braces are optional and without curly braces we can take only one statement under while, which should not be declarative statement.

Ex:
```
while(true)                | while(true);  | while(true)          | while(true)
  S.o.p("Hello");          |               |   int x=10;          | {
                           |               |                      |   int x=10;
```
✓ | ✓ | ✗ | ✓

Ex:
```
while(true)                | while(false)               | int a=10, b=20;          | int a=10, b=20;
{                          | {                          | while(a<b)               | while(a>b)
  S.o.p("Hello");          |   S.o.p("Hello");          | {                        | {
}                          | }                          |   S.o.p("Hello");        |   S.o.p("Hello");
S.o.p("Hi");               | S.o.p("Hi");               | }                        | }
                           |                            | S.o.p("Hi");             | S.o.p("Hi");
```
CE: unreachable statement | CE: unreachable statement '{' | O/P: Hello Hello ⋮ | O/P : Hi

```
final int a=10, b=20;
while(a<b)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```

CE: unreachable statement

```
final int a=10, b=20;
while(a>b)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```

CE: unreachable stmt '{'

Ex:

```
int a=10;
final b=20;
while(a<b)
{
  S.o.p("Hello");
}
  S.o.p("Hi");
```

O/P : Hello
Hello
⋮

```
int a=10;
while(a<20)
{
  S.o.p("Hello");
}
  S.o.p("Hi");
```

O/P : Hello
Hello
⋮

```
while(10<20)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```

CE: unreachable statement

***
Note:- If everything is compile time constant then compiler is responsible to perform that operation.

***
Note:- If atleast one variable is normal variable then compiler won't perform that operation.

2. do-while :—

→ If we want to execute loop body atleast once then we should go for do-while loop.

Syntax:—

```
do
{
  body;
} while(b);
```

b → mandatory, should be boolean type.

→ Curly braces are optional and without curly braces only one statement is allowed blw do and while, which should not be declarative statement.

Ex:

```
do
  S.o.p("Hello");
while(true);
```
✓

```
do;
while(true);
```
✓

```
do
  int a=10;
while(true);
```
✗

```
do
{ int a=10;
}while(true);
```
✓

```
do
while(true);
```
✗

```
do while(true)
  S.o.p("Hello");
while(false);
```
⟹
```
do
  while(true)
    S.o.p("Hello");
  while(false);
```
⟹ O/P: Hello
Hello
:

Ex:

```
do
{ S.o.p("Hello");
}while(true);
S.o.p("Hi");
```
CE: unreachable statement

```
do
{ S.o.p("Hello");
}while(false);
S.o.p("Hi");
```
O/P: Hello
Hi

```
int a=10, b=20;
do
{ S.o.p("Hello");
}while(a<b);
S.o.p("Hi");
```
O/P: Hello
Hello
:

```
int a=10, b=20;
do
{ S.o.p("Hello");
}while(a>b);
S.o.p("Hi");
```
O/P: Hello
Hi

```
final int a=10, b=20
do
{ S.o.p("Hello");
}while(a<b);
S.o.p("Hi");
```
CE: unreachable statement

```
final int a=10, b=20;
do
{ S.o.p("Hello");
}while(a>b);
S.o.p("Hi");
```
O/P: Hello
Hi

3: for loop :—

→ This is the most commonly used loop in Java.

→ for loop is the best choice if we know the no. of iterations in advance.

Syntax:- for(initialization_section ; conditional_exp ; increment/decrement_section)

① ②,⑤,⑧ ④,⑦

{

body ③,⑥,⑨

}

→ Curly braces are optional and without curly braces we can take only one statement, which should not be declarative statement.

Initialization Section:-

→ This part will be executed only once.

→ Usually we can declare and initialize local variables of for loop in this section.

→ Here we can declare any no. of variables but, should be of the same type. i.e., we can't declare multiple variables of different data types.

Ex: int i=0; ✓

int i=0, j=0; ✓

int i=0, boolean=true; ✗

int i=0, int j=0; ✗

→ In initialization section, we can take any valid Java statement including S.O.P statement also.

Ex:
```
int i=0;
for (S.O.P ("Hello U R sleeping"); i<3 ; i++)
{
    S.O.P ("No Boss U Only Sleeping");
}
```

O/P: Hello U R Sleeping
No Boss U only Sleeping
No Boss U only Sleeping
No Boss U only Sleeping

Conditional Expression :—

→ Here we can take any valid Java expression, but should be boolean type.

→ This part is optional and if we are not taking anything then compiler will always places true.

Ex:
```
int i=0;
for(S.o.p("Hello U R sleeping");; i++)
{
    S.o.p("U only sleeping");
}
```
o/p : Hello U R sleeping
U only sleeping
U only sleeping
:
:

Increment / Decrement section :—

→ Here we can take any valid Java statement including S.o.p statement also.

Ex:
```
int i=0;
for(S.o.p("Hello"); i<3; S.o.p("Hi"))
{
    i++;
}
```
o/p : Hello
Hi
Hi
Hi

*** → All three parts of for loop are independent of each other and optional.

Ex:

`for(;;) { body }`	`for(;;);` ⟹ infinite loops.

Ex:
```
for(int i=0; true; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
CE: unreachable statement

```
for(int i=0; false; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
CE: unreachable statement '{'

```
for(int i=0;; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
CE: unreachable statement

```
int a=10, b=20;
for(int i=0; a<b; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
O/P : Hello
Hello

```
int a=10, b=20;
for(int i=0; a>b; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
O/P : Hi

```
final int a=10, b=20;
for(int i=0; a<b; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
CE: unreachable statement

```
final int a=10, b=20;
for(int i=0; a>b; i++)
{
  S.o.p("Hello");
}
S.o.p("Hi");
```
CE: unreachable statement '{'

4. for-each loop (Enhanced for loop):—

→ It has been introduced in 1.5 version.

→ It is specially designed loop to retrieve elements of arrays and collections.

Ex①: To print elements of single dimensional array by using general and enhanced for loops.

```
int[] a={10,20,30};
```

Normal for loop

```
for(int i=0; i<a.length; i++)
{
  S.o.p(a[i]);
}
```

Enhanced for loop

```
for(int x : a)
{
  S.o.p(x);
}
```

O/p : 10
20
30

Eg②: To print elements of two-dimensional array by using normal and enhanced for loops.

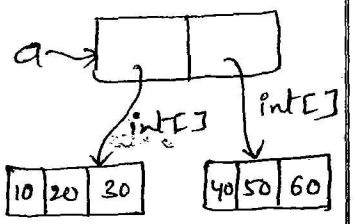
```
int[][] a={{10,20,30},{40,50,60}};
```

Normal for loop

```
for(int i=0; i<a.length; i++)
{
  for(int j=0; j<a[i].length; j++)
  {
    S.o.p(a[i][j]);
  }
}
```

Enhanced for loop

```
for(int[] x : a)
{
  for(int y : x)
  {
    S.o.p(y);
  }
}
```



Eg③:

```
for(int i=0; i<10; i++)
{
  S.o.p("Hello");
}
```

```
for(int i=0:9)
{
  S.o.p("Hello");
}
```

→ we can't write for-each loop directly.
→ for-each loop is most convenient loop to retrieve the elements of arrays and collections.
→ But, it's limitation is only applicable for arrays and collections & it is not a general purpose loop.

→ By using for-each loop we can't print array elements in reverse order.

*** Iterable Vs Iterator :—

→ The target element in for-each loop should be Iterable.

→ An object is said to be Iterable iff the corresponding class implements Iterable interface.

→ Iterable interface present in java.lang package and contains only one method iterator().

```
public Iterator iterator()
```

```
for(each item : target)
{
---
---
}
```

target → Array | Collection

→ It should be Iterable object.

→ Every Array class and Collection classes already implements Iterable interface.

*** Differences blw Iterable & Iterator :—

Iterable (I)	Iterator (I)
1. It is related to for-each loop.	1. It is related to collections.
2. The target element in for-each loop should be Iterable object.	2. We can use Iterator object to get objects one by one from Collection
3. Introduced in 1.5 version.	3. Introduced in 1.2 version

3. Transfer Statements :—

1. break statement :—

→ We can use break statement in the following cases.

① Inside switch to stop fall through.

Ex:
```
int x=0;
switch(x)
{
  case 0 : S.o.p("0");
  case 1 : S.o.p("1");        o/p : 0
           break;                   1
  default : S.o.p("def");
}
```

② Inside loops to break loop execution based on some condition.

Ex:
```
for(int i=0; i<10; i++)
{
  if(i==5)                    o/p : 0
    break;                          1
  S.o.p(i);                         2
}                                   3
                                    4
```

③ Inside labeled blocks to break block execution based on some condition.

Ex:
```
class Test
{
  p s v m(-)
  {
    int x=10;
    l1:
    {
      S.o.p("begin");
      if(x==10)
        break l1;
    } S.o.p("end");
    S.o.p("Hello");
  }
}
```

o/p : begin
Hello

→ These are the only places where we can use break statement.

→ If we are using anywhere else then we will get CE.

Ex:
```
class Test
{
  P s v m(-)
  {
    int x=10;
    if(x==10)
      break;
    S.o.p("Hello");
  }
}
```
CE: break outside switch or loop

2. continue :—

→ We can use continue statement inside loops to skip current iteration and continue for next iteration.

Ex:
```
for(int i=0; i<10; i++)
{
  if(i%2==0)
    continue;
  S.o.p(i);
}
```
O/P : 1 3 5 7 9

→ We can use continue statement only inside loops if we are trying to use any where else then we will get compile time error saying continue outside of loop.

Ex:
```
class Test
{
  P s v m(-)
  {
    int x=10;
    if(x==10)
      continue;
    S.o.p("Hello");
  }
}
```
CE: continue outside of loop

Labeled break and continue statements:—

→ We can use labeled break and continue to break or continue a particular loop in nested loops.

Ex:

```
l1:
for(---)
{
  !
  l2:
  for(---)
  {
    !
    for(---)
    {
      break l1;
      break l2;
      break;
    }
  }
}
```

Ex:

```
l1:
for(int i=0; i<3; i++)
{
  for(int j=0; j<3; j++)
  {
    if(i==j)
      break;
    S.o.p(i+"..."+j);
  }
}
```

break;

```
1...0
2...0
2...1
```

break l1;

No output

continue;

```
0...1
0...2
1...0
1...2
2...0
2...1
```

continue l1;

```
1...0
2...0
2...1
```

*** do-while Vs continue (Most dangerous combination):-

Ex:

```
int x=0;
do
{
  ++x;
  S.o.p(x);
  if(++x < 5)
    continue;
  ++x;
  S.o.p(x);
}while(++x < 10);
```

x = 0 1 2 3 4 5 6 7 8 9 10 11

o/p : 1 4 6 8 10