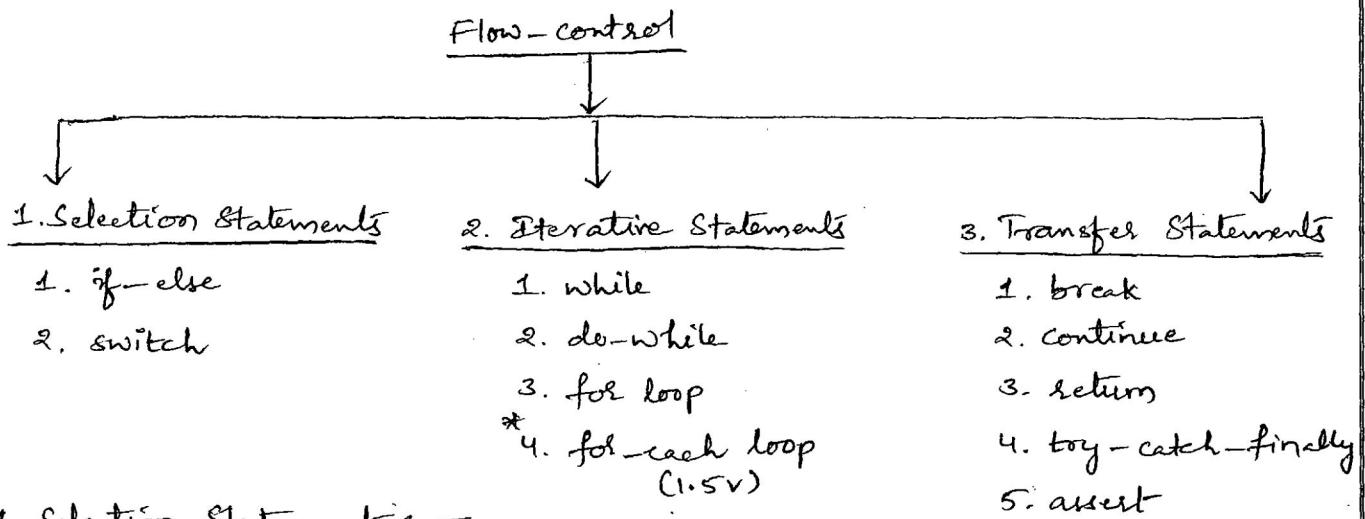


### 3. Flow Control

DURGA SOFTWARE SOLUTIONS

SCJP MATERIAL

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



#### 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 a=0;

```

if (a)
{
    S.o.p("Hello");
}
else
{
    S.o.p("Hi");
}

```

CE: incompatible types  
 found : int  
 required : boolean

int a=10

```

if (a = 20)
{
    S.o.p("Hello");
}
else
{
    S.o.p("Hi");
}

```

int a=10

```

if (a == 20)
{
    S.o.p("Hello");
}
else
{
    S.o.p("Hi");
}

```

Output : 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.

<u>Ex:</u>	if(true)	if(true);	if(true)	if(true)
**	S.o.p("Hello");	✓	int x=10;	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	
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)  
{  
}  
y`

- 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");  
}`

X

*(Ex: 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:
        System.out.println("10");
        break;
    case y:
        System.out.println("20");
}

```

cc: 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: System.out.println("10");
    case y: System.out.println("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: System.out.println(10);
    break;
    case 10+1: System.out.println(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; <sup>range is</sup>  
 switch(b) → byte [-128 to 127]  
 {  
 case 10: S.o.p ("10");  
 case 100: S.o.p ("100");  
 case 1000: S.o.p ("1000");  
 }  
 CE: PLP  
 found : int  
 required : byte

→ 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

- 1. It should be compile time constant
- 2. It should be constant expression
- 3. Value should be within the range of switch argument type
- 4. Duplicate case labels are not allowed

### 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;
}
```

default case :-

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");
}
```

<u>x=0</u>	<u>x=1</u>
0	1

<u>x=2</u>	<u>x=3</u>
2	def

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");
}
```

<u>x=0</u>	<u>x=1</u>
0	1

<u>x=2</u>	<u>x=3</u>
2	def

}

## 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(`or. next()`) {  
 }                    while(`itr. hasNext()`) {  
 }                    while(`c. 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");`  
}

cc: incompatible types  
found : int  
required : boolean

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

<u>Ex:</u>	while (true) S.o.p ("Hello");	while (true);	while (true) int x=10; X	while (true) { int x=10; }
------------	----------------------------------	---------------	--------------------------------	-------------------------------------

<u>Ex:</u> while(true)	while(false)	int a=10, b=20; while(a < b)	int a=10, b=20; while(a > b)
{ S.o.p("Hello"); } S.o.p("Hi");	{ S.o.p("Hello"); } S.o.p("Hello");	S.o.p("Hello"); S.o.p("Hi");	S.o.p("Hello"); S.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 'l'

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);

```

mandatory  
Should be boolean type.

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

Ex: do  
  s.o.p("Hello");  
  while(true);

do;  
  while(true);

do  
  int z=10;  
  while(true);

do  
  {  
    int z=10;  
  }while(true);

do  
  while(true);

do while(true)  
  s.o.p ("Hello");  
  while(false);

do  
  while(true)  
  s.o.p ("Hello");  $\Rightarrow$  o/p: Hello  
  while(false);

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)`

```

graph TD
    1[1] --> 2[2, 5, 8]
    1 --> 4[4, 7]
    2 --> 3[3, 6, 9]
    3 --> 9
    4 --> 7
    style 1 fill:none,stroke:none
    style 2 fill:none,stroke:none
    style 3 fill:none,stroke:none
    style 4 fill:none,stroke:none
    style 5 fill:none,stroke:none
    style 6 fill:none,stroke:none
    style 7 fill:none,stroke:none
    style 8 fill:none,stroke:none
    style 9 fill:none,stroke:none
  
```

→ 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++)
{
```

```
y 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++;
y
```

O/p : Hello

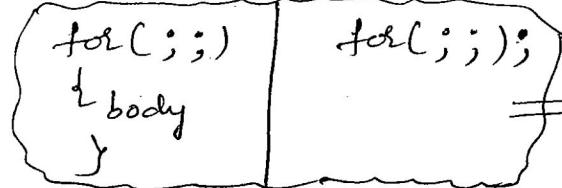
Hi

Hi

Hi

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

Ex:



infinite loops.

Ex: `for (int i=0; true; 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

`for (int i=0; false; 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.

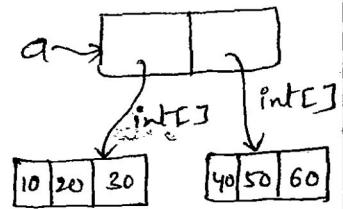
Ex①: To print elements of single dimensional array by using general and enhanced for loops.

<u>int[] a = {10, 20, 30};</u>	
Normal for loop	Enhanced for loop
<pre>for (int i=0; i&lt;a.length; i++) {     S.o.p(a[i]); }</pre>	<pre>for (int x : a) {     S.o.p(x); }</pre>

O/p :  
10  
20  
30

Ex②: To print elements of two-dimensional array by using normal and enhanced for loops.

<u>int[][] a = {{10, 20, 30}, {40, 50, 60}};</u>	
Normal for loop	Enhanced for loop
<pre>for (int i=0; i&lt;a.length; i++) {     for (int j=0; j&lt;a[i].length; j++)     {         S.o.p(a[i][j]);     } }</pre>	<pre>for (int[] x : a) {     for (int y : x)     {         S.o.p(y);     } }</pre>



<u>Ex③:</u> for (int i=0; i<10; i++)	for (int i=0; i<10; i++)
<pre>     {         S.o.p("Hello");     } </pre>	<pre>     {         S.o.p("Hello");     } </pre>

→ we can't write for-each loop directly.

- for-each loop is most convenient loop to retrieve the elements of arrays and collections.
- But, its limitation is only 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 & 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)

{  
---  
}

→ Array | Collection

→ It should be Iterable object.

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

\* \* Differences b/w Iterable & Iterator :-

#### Iterable (I)

1. It is related to for-each loop.
2. The target element in for-each loop should be Iterable object.
3. Introduced in 1.5 version.

#### Iterator (I)

1. It is related to Collections.
2. We can use Iterator object to get objects one by one from Collection.
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");    OLP: 0
                      break;
        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)    OLP: 0
  ---break;
  S.o.p(i);
}
```

- 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)    S.o.p("Hello");
      ---break l1;
      S.o.p("end");
    }
  }
}
```

- 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.r.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:  $\frac{1}{3} \frac{5}{7} \frac{9}{11}$

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

Ex: class Test

```
  {
```

```
    p.s.r.m();
```

```
    {  
        int x=10;
```

```
        if(x==10)
```

```
            continue;
```

```
        S.o.p("Hello");
```

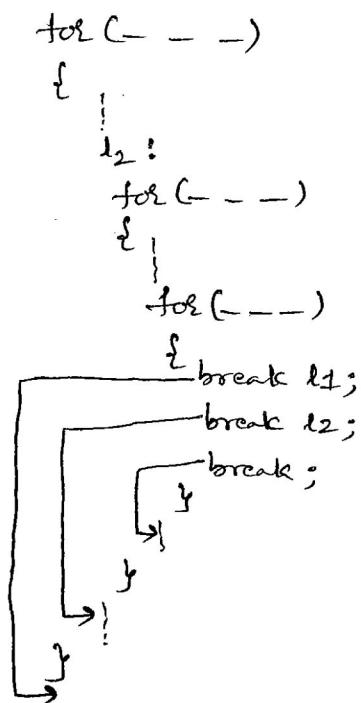
```
}  
}
```

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:  $l_1$ :



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 & continue (most dangerous combination):-

Ex: int x=0;

do

{  
    ++x;  
    X

    S.o.p(x);  
    if (++x < 5)

        continue;

    ++x;

    S.o.p(x);

} while (++x < 10);

x =  $\emptyset$   
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11

Op :

1  
4  
6  
8  
10