Flow Control

Date / /
Page

Flow Control — describes the order in which the statements will be executed

① Selection statements
1.) if-else
2.) switch()

② Iterative statements
1.) while()
2.) do-while()
3.) for()
(1.5V) 4.) for-each loop

③ Transfer statement
1.) break
2.) continue
3.) return
4.) try, catch, finally
(1.4V) 5.) assert

⊛ if-else statement

→ if argument always should be boolean otherwise we will get compile-time error.

```
int x=0;
if(x)
{
  s.o.p("H");
}
else
{
  s.o.p("C");
}
```

C.E. Incompatible types F: int R: boolean

```
int x=10;
if(x=20);
{
  s.o.p("H");
}
else
{
  s.o.p("C");
}
```

Switch Statement

→ When no. of cases are too much better to go for switch instead of if-else

→ Allowed Argument types for the switch statements:

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

(new Allowed types) (new Allowed types)

→ curly braces are mandatory in switch (unlike) all other flow control statements (if-else, loops)

→ Both case: & default: are optional.

↳ Empty switch

```
switch(n)
{
}
```

→ Inside switch, every statements should be case or default, Independent statements not allowed.

```
int n=10;
if (n==20)
{
  s.o.p("H");
}
else
{
  s.o.p("C");
}
```
o/p = C

```
boolean b=true;
if (b==false)
{
  s.o.p("H");
}
else
{
  s.o.p("C");
}
```
o/p = C

```
boolean b=true;
if (b==true)
{
  s.o.p("H");
}
else
{
  s.o.p("C");
}
```
o/p = H

→ else part & curly braces {} are optional

★ without curly braces {} only 1 statement is allowed, which should not be declarative statement

```
if (true)
  s.o.p("H");
```
✓ o/p = H

```
if (true);
```
✓

```
if (true)
  int n=10;
```
✗ C.E.

```
if (true)
{
  int n=10;
}
```
✓

→ semicolon is a valid java statement ★ which is known as Empty statement

jjj

→ Every case label should be in the range of switch argument type.

```
byte b = 10;
switch(b)
{
  case 10: ≈
  case 100: ≈
  case 1000: ≈
}
```

(1000 is marked ✗)

C.E.
PLP
F: int
P: byte

```
byte b = 10
switch(b+1)
{
  case 10: ≈
  case 100: ≈
  case 1000: ≈
}
```

(switch(b+1) → int; ✓)

→ Duplicate case labels are not allowed.

```
int x = 10;
switch(x)
{
  case 97: ≈
  case 98: ≈
  case 99: ≈
  case 101: ≈
}
```

C.E.
Duplicate case label

```
int x = 10;
switch(x)
{ s.o.p("Hi"); }
```

C.E. case, default or } expected

→ case label should be compile time constant (constant expression)

```
int x = 10;
int y = 20;
switch(x)
{
  case 10: ≈
  case y: ≈
}
```

C.E.
constant expression required

If we declare y as final, there is no C.E.

→ Both switch Arguments & case label can be expression, but label should be constant one

```
int x = 10;
switch(x+1)
{
  case 10: ≈
  case 10+20: ≈
}
```

(switch(x+1) → int ✓; case 10+20 ✓)

```
switch(n)
{
  case 0:
      s.o.p(0);
  case 1:
      s.o.p(1);
      break;
  case 2:
      s.o.p(2);
  default:
      s.o.p("def");
}
```

n=0	n=1	n=2	n=3
0 1	1	2 def	def

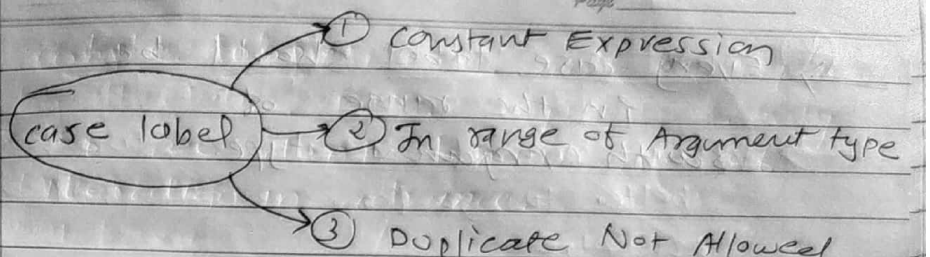
→ within switch, we can take "default case" most once

"default case" will executed if & only if there is no other case matched.

within the switch we can write default case anywhere (Recommanded → last case)

```
switch(n)
{
  default:
      s.o.p("def");
  case 0:
      s.o.p(0);
      break;
  case 1:
      s.o.p(1);
  case 2:
      s.o.p(2);
}
```

n=0	n=1	n=2	n=3
0	1 2	2	def 0



→ Within the switch if any case is matched from that case onwards all statements will be executed Untill "break" or End of switch

↳ Fall through inside switch

Main Advantage of fall through inside switch is, we can define common Action for Multiple actions (Code Reusability)

```
switch(n)
{
  case 1:
  case 2:
  case 3: s.o.p("Quater-1");
          break;
  case 4:
  case 5:
  case 6: s.o.p("Quater-2");
          break;
  :
  :
}
```

Library

Date / /
Page

```
while (true)
{
  s.o.p("H");
}
s.o.p("C");
```

C.E: unreachable statement

```
while (false)
{
  s.o.p("H");
}
s.o.p("C");
```

C.E: unreachable statement

```
int a=10, b=20;
while (a<b)
{
  s.o.p("H");
}
s.o.p("C");
```

O/P: H H ...

```
int a=10, b=20;
while (a>b)
{
  s.o.p("H");
}
s.o.p("C");
```

O/P: C

```
final int a=10, b=20;
while (a<b)
{ s.o.p("H");
}
s.o.p("C");
```

C.E: unreachable statement

```
final int a=10, b=20;
while (a>b)
{
  s.o.p("H");
}
s.o.p("C");
```

C.E: unreachable statement

Date / /
Page

while loop

→ If we don't know no. of iteration in advance.

```
while(rs.next())
{
}
```

```
while(iter.hasNext())
{
}
```

→ Argument should be boolean or we will get compile time error.

```
while (b)   // b → boolean
{
  ACTION
}
```

```
while (1)   // 1 → incompatible type
{           // F: int
}           // R: boolean
```

→ curly braces {} are optional, without them we can take only 1 statement under while, which should not be declarative

```
while (true)
  s.o.p("H");
```

```
while (true);
```

```
while (true)
  int i=10;
```

```
while (true)
{
  int i=10;
}
```

→ Curly braces are optional, if not taken, one stmt should not be declarative

```
do                    do;                  do
 s.o.p("Hello");      while(true);         int x=10;
while(true);          ✓                    while(true);
✓                                          ✗
```

★ (Semicolon itself is a statement)

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

[So, one statement is mandatory in do-while loop]

→
```
do while(true)
s.o.p("Hello");
while(false);
```
⟹
```
do
  while(true)
    s.o.p("Hello");
  while(false);
```
✓

o/p = Hello Hello ...

Here, compiler does not reach to while(false), but as it is a combination syntax do-while, no unreachability error we going to get, bcz while(false); is a part of that do-while statement

→ Every final variable will be replaced by the value at the compile time only.

final
```
int a=10;
int b=20;

s.o.p(a);          After          s.o.p(10);
s.o.p(b);   ⟹ compilation of      s.o.p(b);
                these statements
```

→ If every argument is final variable (compile time constant) then that operation should be performed compile time only

```
final int a=10, b=20;
    int c=30;

s.o.p(a+b);        After          s.o.p(30);
s.o.p(a+c);   ⟹ compilation of    s.o.p(10+c);
s.o.p(a>b);        these          s.o.p(false);
s.o.p(a<c);        statements     s.o.p(10<c);
```

✱ do-while loop

→ If we want to execute loop body at least once...

```
do
{
 //body          → boolean
}while(b);       → mandatory
```

(*) for loop

→ If we know no. of iteration in advance.

① ②,⑤,⑧ ④,⑦

```
for(initialization; condition; inc/dec)
{
    body ③,⑥,⑨
}
```

→ Only braces are optional, without them only 1 stmt can be taken, which can't be declarative.

```
for(int i=0; true; i++)
    S.O.P("Hello");
```
✓

```
for(int i=0; i<10; i++);
```
✓

```
for(int i=0; i<10; i++)
    int n=10;
```
✗

→ Initialization part execute only once in loop cycle, here we can declare variable for for-loop, but they should be of (same type)

```
int i=0,j=0;          ✓
int i=0, String s="ab"; ✗
int i=0; int j=0;     ✗
```

In this section, we can take any valid java statement.

→

```
do
{
    S.O.P("H");
}while(true);
S.O.P("C");
```
✗

C.E. Unreachable stmt

```
do
{
    S.O.P("H");
}while(false);
S.O.P("C");
```
✓

O/P → H C

```
int a=10,b=20;
do
{
    S.O.P("H");
}while(a<b);
S.O.P("C");
```
✓

O/P :- H H ∞

```
int a=10,b=20;
do
{
    S.O.P("H");
}while(a>b);
S.O.P("C");
```
✓

O/P :- H C

```
final int a=10,b=20;
do
{ S.O.P("H");
}while(a<b);
S.O.P("C");
```

C.E. Unreachable stmt

```
final int a=10,b=20;
do
{
    S.O.P("H");
}while(a>b);
S.O.P("C");
```

O/P :- H C

→ Unreachable stmt problem is same as while loop.

⊛ for-each loop / Enhanced for loop

(1.5 V)

→ specially designed loop to retrieve elements of Array

```
int [] x = {10, 20};
```

Normal For loop	Enhanced for loop
`for(int i=0; i<x.length; i++)`	`for(int a : x)`
`{`	`{`
`s.o.p(x[i]);`	`s.o.p(a);`
`}`	`}`

o/p : 10, 20

→ Best choice to retrive the elements of Arrays & Collection, but limitation is it is not a general purpose loop & can't write for everything

```
for(int i=0; i<2; i++)
{
    s.o.p("H");
}
```

} can't write equivalent for-each loop

```
int i=0;
for(s.o.p("H"); i<3; i++)
{
    s.o.p("C");
}
```

o/p : H C C C

→ In conditional part, we can take any valid java stmt, but should be of type (boolean), if we do not put anything, compiler by default takes (true)

```
for(int i=0; ; i++)
{
    s.o.p(i);
}
```

o/p : 0 1 2 ...

→ In Init/Dec section, we can take any valid java statement.

```
int i=0;
for(s.o.p("H"); i<3; s.o.p("C"))
{
    i++;
}
```

o/p : H C C C

→ All three parts of for loop are (independent) of each other & are (optional)

```
for(;;)
{
    s.o.p("H");
}
```

⇒ Infinite loop

```
for(;;);
```

⇒ Infinite loop

→ All Array related class & Collection implemented classes implements **Iterable** interface.

> Being a programmer, we are not required to do anything just we should aware the point

→ Differences blw Iterator & Iterable

Iterator (I)	Iterable (I)
① Related to Collection	① Related to for-each loop
② Use to retrieve elements of collection one by one	② Target element in for-each loop should be Iterable
③ java.util package	③ java.lang package
④ 3 Methods 1) hasNext() 2) next() 3) remove()	④ 1 Method 1) iterator()

→ For:each loop can print Array only in original order

```
int[] n={1,2,3};
for(int i=n.length-1;i>=0;i--)
{
    s.o.p(n[i]);
}
```

⟹ can't write for-each loop directly.

→ Iterable (I)

```
for(eachitem n : target)
{
    ≋≋
}
```

Array / Collection

Iterable object

→ The (target) element in for each should be Iterable Object

→ An object is said to be Iterable object if and only if corresponding class implements Iterable interface (j.l.Iterable)

→ Iterable interface introduced in 1.5V

It contains only one method

iterator()

```
public Iterator iterator()
```

Date ___/___/_____
Page _____

✳ break

① Inside switch - (To stop fall through)
② Inside loop - (To break loop execution)
③ Inside Labeled block

```
class Test {
    P S V main(String[] args) {
        int x=10;
        l1:
        {
            S.O.P("begin");
            if(x==10) break l1;
            S.O.P("end");
        }
        S.O.P("H");
    }
}
```

O/P :

```
begin
H
```

→ If we use break anywhere else, we will get C.E. break outside switch or loop

✳ Continue

① Inside loop only (To continue new iteration)
(Not Inside switch / labeled block as they are not iterative)

if we use continue anywhere else we will get C.E continue outside loop

⊛ Continue with do-while

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

x =
✗
2
3
4
5
6
7
8
9
10
11

O/P
1
4
6
8
10

End of Flow Control

Start of Declarations & Modifiers