**Flow Control**

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

Flow Control

Selection Statements Iterative Statements Transfer Statements

If else while() break

Switch do-while() continue

for() return

for each loop() try – catch – finally

assert

**Selection Statements**

* **if else**

SYNTAX:

if(boolean argument)

{

Action if b is true

}

else

{

Action if b is false

}

The argument to the if statement should be of boolean type. Any other type gives compile type error

**EG 1: ..invalid o/p = compile time error**

int x = 10;

if(x)

{

SOP(“Hello”);

}

Else

{

SOP(“Hii”);

}

**EG 2: ..invalid o/p: Compile time error**

int x = 10;

if(x = 20)

{

SOP(“Hello”);

}

Else

{

SOP(“Hii”);

}

**EG 3: valid o/p = Hii**

int x = 10;

if(x == 20)

{

SOP(“Hello”);

}

Else

{

SOP(“Hii”);

}

**EG 4 : ..valid o/p = Hii**

boolean b = true;

if(b = false)

{

SOP(“Hello”);

}

Else

{

SOP(“Hii”);

}

**EG 5 : ..valid o/p = Hello**

boolean b = false;

if(b == false)

{

SOP(“Hello”);

}

Else

{

SOP(“Hii”);

}

Else part conclusion 2:

Else part and curl braces are optional .Without curly braces only one statement is allowed under if ,which should not be declarative statement.

EG 1: valid EG 2: valid EG 3: valid

if (true) if (true); if (true)

SOP(“Hello”) {

int x = 10;

}

EG 4: invalid

int (true)

int x = 10;

NOTE:

Semi-colon is a valid java statement which is also known as empty statement.

There is no dangling else problem in java. Every else is mapped to the nearest if statement.

**Switch Statement**

If several options are available then it is not recommended to use nested if-else because it reduces readability.

To handle this requirement we should go for switch statement

**SYNTAX:**

switch(x)

{

case 1:

Action 1

break;

case 2:

Action 2

break;

.

.

case n:

Action n

break;

default:

default Action

}

The allowed argument types for the switch statement are byte ,short, char, int until 1.4 version.

But from 1.5 version onwards corresponding wrapper classes and enum type is also allowed. From 1.7 version onwards String type is also allowed

|  |  |  |
| --- | --- | --- |
| 1.4 version | 1.5 version | 1.7 version |
| byte | Byte | 1.4 + 1.5 +  String |
| Short | Short |
| Char | Character |
| int | Integer |
|  | enum |

Curly braces are mandatory. Except switch everywhere curly braces are optional.

Both case and default are optional. i.e an empty switch statement is a valid java syntax.

EG:

int x = 10;

switch(x)

{

SOP(“Hello”);

}

Inside a switch every statement should be under some case or default that is independent statements are not allowed inside switch.

EG:

int x = 10;

switch(x)

{

SOP(“Hello”);

}

O/P: Compile time error: case ,default or } expected

EG:

**int** x = 10, y = 20;

**switch**(x) {

**case** 10:

System.***out***.print(10);

**break**;

**case** y:

System.***out***.print(20);

**break**;

}

o/p: Unresolved compilation problem:

case expressions must be constant expressions

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

If we declare y as final variable then we will not get any compile time error

Both switch argument and case label can be expressions. But case label should be constant expression

**EG:**

**int** x = 10;

**final** **int** y = 20;

**switch**(x+1-1) {

**case** 10:

System.***out***.print(10);

**break**;

**case** 10+20:

System.***out***.print(20);

**break**;

}

o/p: 10

**EG 1 : invalid**

byte b = 10;

switch(b)

{

case 10:

SOP(10);

Break;

case 100:

SOP(100);

Break;

case 1000:

SOP(1000);

}

o/p: Unresolved compilation problem:

Type mismatch: cannot convert from int to byte

**EG 2:valid**

byte b = 10;

switch(b+1)

{

case 10:

SOP(10);

Break;

case 100:

SOP(100);

Break;

case 1000:

SOP(1000);

}

Every case label should be in the range of switch argument type. Otherwise we will get compile time error.

EG 3:

**byte** b = 97;

**switch**(b) {

**case** 97:

System.***out***.print(10);

**break**;

**case** 98:

System.***out***.print(20);

**break**;

**case** 99:

System.***out***.print(30);

**break**;

**case** 'a':

System.***out***.print(20);

**break**;

}

o/p: Unresolved compilation problems:

Duplicate case

Duplicate case

Duplicate case labels are not allowed. Otherwise we will get compile time error.

SUMMARY:

1. Case Label
2. It should be constant expression.
3. The value should be in the range of switch argument type
4. Duplicate case labels are not allowed

* **Fall through inside switch**

Within the switch if any case 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).

**EG 1:**

Switch(x)

{

case 1:

case 2:

case 3:

SOP(“Q-4”);

break;

case 4:

case 5:

case 6:

SOP(“Q-1”);

break;

}

**EG 2:**

Switch(x)

{

case 0:

SOP(0);

case 1:

SOP(1);

break;

case 2:

SOP(2);

default:

SOP(“default”);

}

Output:

X = 0 🡪0 1 X = 1 🡪1

X = 2 🡪2 default X = 3 🡪default

**CONCLUSIONS:**

1. Within the switch we can take default case at most once.
2. Default case will be executed iff there is no other case matched
3. Within the switch we can write default case anywhere but it is recommended to write as last case.

EG:

Switch(x)

{

default:

SOP(“default”);

case 0:

SOP(0);

break;

case 1:

SOP(1);

case 2:

SOP(2);

}

Output:

X = 0: 0 x = 1 : 1 2

X = 2: 2 x = 3 : default 0

**Iterative Statements**

**WHILE LOOP**

If we don’t know number of iterations in advance then we should go for while loop.

EG:

while(e.hasmoreElements()){}

while(itr.hasnext()){}

**SYNTAX:**

while(boolean value)

{

Action;

}

The argument should be boolean type . if any other type argument is provided then compile time error is encountered.

EG:

**while**(1)

{

System.***out***.print(**false**);

}

Output: Unresolved compilation problem:

Type mismatch: cannot convert from int to Boolean

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

while(true) while(true); while(true)

SOP(“hello”) {

int x = 10;

}

Valid valid valid

while(true)

int x = 10;

Invalid

**EG 1:**

**while**(**true**)

{

System.***out***.print("hello");

}

System.***out***.print("hii");

Output: Unresolved compilation problem:

Unreachable code

**EG 2:**

**while**(**false**)

{

System.***out***.print("hello");

}

System.***out***.print("hii");

Output: Unresolved compilation problem:

Unreachable code

**EG 3:**

**public** **static** **void** main(String[] args)

{

**int** a = 10, b = 20;

**while**(a<b)

{

System.***out***.print("hello");

}

System.***out***.print("hii");

Output: hello (infinite loop)

**EG 4:**

**public** **static** **void** main(String[] args)

{

**int** a = 10, b = 20;

**while**(a>b)

{

System.***out***.print("hello");

}

System.***out***.print("hii");

Output: hii

**EG 5:**

**public** **static** **void** main(String[] args)

{

**final** **int** a = 10, b = 20;

**while**(a>b)

{

System.***out***.print("hello");

}

System.***out***.print("hii");

}

Output: Unresolved compilation problem:

Unreachable code

**EG 6:**

**public** **static** **void** main(String[] args)

{

**final** **int** a = 10, b = 20;

**while**(a<b)

{

System.***out***.print("hello");

}

System.***out***.print("hii");

}

Output: Unresolved compilation problem:

Unreachable code

**NOTE:**

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

EG 1:

final int a = 10;

int b = 10;

SOP(a); After compilation SOP(10);

SOP(b); SOP(b);

1. If every argument is final variable (compile time constants) then that operation should be performed at compile time only.

int a = 10, b = 20;

int c = 20;

SOP(a + b); SOP(30);

SOP(a + c); After compilation SOP(10 + c);

SOP(a < b); SOP(true);

SOP(a < c); SOP(10 < c);

**DO - WHILE LOOP**

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

**SYNTAX:**

do

{

Body;

}while(b);

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

**EG 1: valid**

do

SOP(“Hello”);

while(true);

**EG 2: valid**

do;

while(true);

**EG 3: invalid**

do

int x = 10;

while(true);

**EG 4: valid**

do

{

int x = 10;

}while(true);

**EG 5: invalid**

do

while(true);

***EG :***

do while(true)

SOP(“hello”);

while(false);

do

while(true)

SOP(“hello”);

while(false);

Output: hello(infinite times)

**EG 1:**

**public** **static** **void** main(String[] args)

{

**do**

{

System.***out***.print("hello");

}**while**(**true**);

System.***out***.print("hii");

}

Output: Unresolved compilation problem:

Unreachable code

**EG 2:**

**public** **static** **void** main(String[] args)

{

**do**

{

System.***out***.print("hello");

}**while**(**false**);

System.***out***.print("hii");

}

Output: hellohi

**EG 3:**

**public** **static** **void** main(String[] args)

{

**int** a = 10, b = 20;

**do**

{

System.***out***.print("hello");

}**while**(a<b);

System.***out***.print("hii");

}

Output: hello(infinite times)

**EG 4:**

**public** **static** **void** main(String[] args)

{

**final int** a = 10, b = 20;

**do**

{

System.***out***.print("hello");

}**while**(a>b);

System.***out***.print("hii");

}

Output:hellohi

**EG 5:**

**public** **static** **void** main(String[] args)

{

**final int** a = 10, b = 20;

**do**

{

System.***out***.print("hello");

}**while**(a<b);

System.***out***.print("hii");

}

Output: Unresolved compilation problem:

Unreachable code

**EG 6:**

**public** **static** **void** main(String[] args)

{

**final int** a = 10, b = 20;

**do**

{

System.***out***.print("hello");

}**while**(a>b);

System.***out***.print("hii");

}

Output:hellohi

**FOR LOOP**

For loop is the most commonly used loop in java.

If we know number of iterations in advance then for loop is the best choice.

SYNTAX:

for (initialization\_section ; conditional\_check ; increment / decrement)

{

Loop body

}

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

**EG 1:** ..valid

for(int i = 0 ; true ; i++)

SOP(“hello”);

**EG 2:** .. valid

for(int i = 0 ; i < 10 ; i++);

**EG 3:** ..invalid

for(int i = 0 ; i < 10 ; i++);

int x = 10;

**Initialization Section:**

* This part will be executed only once in loop life cycle.
* Here we can declare and initialize local variables of for loop.
* Here we can declare any number of variables but they should be of same type.

By mistake if we are trying to declare different data type variables then we will get compile time error.

EG:

int i = 0 , j = 0; ..valid

int i = 0 , String s = “durga” ..invalid

int i = 0 , int j = 0; ..invalid

* In the initialization section we can take any valid java statement including SOP.

EG:

**int** i = 0;

**for**(System.***out***.println("hello You are sleeping"); i<3;i++)

{

System.***out***.println("No you are only sleeping");

}

Output: hello You are sleeping

No you are only sleeping

No you are only sleeping

No you are only sleeping

**Conditional Check:**

* Here we can take any valid java expression but should be of the type Boolean.
* This part is optional and if we are not taking anything then compiler will always place true.

**Increment / decrement Section:**

* In the increment / decrement section we can take any valid java statement including SOP.

EG:

int i = 0;

for(SOP(“hello”) ; i<3 ; SOP(“hi”))

{

i++;

}

Output:hello

hi

hi

hi

all three parts of for loop are independent of each other and optional .

EG 1:

for(;;)

{

SOP(“Hello”);

}

EG 2:

for(;;);

Both are valid infinite loops

EG 1 :

For(int i = 0 ; false ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 1 : invalid**

For(int i = 0 ; true ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 2 : invalid**

For(int i = 0 ;false; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 3 : invalid**

For(int i = 0 ; true ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 4 : invalid**

For(int i = 0 ; ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 5: valid**

int a = 10, b = 20;

For(int i = 0 ;a<b ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 6: valid**

int a = 10, b = 20;

For(int i = 0 ;a>b ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**EG 7: invalid**

final int a = 10, b = 20;

For(int i = 0 ;a>b ; i++)

{

SOP(“Hello”);

}

SOP(“Hi”);

**FOR EACH LOOP**

Introduced in 1.5 version.

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

Example 1:

To print elements of one dimensional array

int [] x = {10, 20, 30, 40};

**Normal For Loop:**

for(int i = 0 ; i < x.length ; i++)

{

SOP(“x[i]”);

}

**Enhanced For Loop:**

for(int x1 : x)

{

SOP(x1);

}

Example 2:

To print elements of two dimensional array

Int[][] x = {{10, 20, 30}, {40, 50, 60}};

**Normal For Loop:**

for(int i = 0 ; i < x.length ; i++)

{

for(int j = 0 ; j<x[i].length ; j++)

{

SOP(x[i][j]);

}

}

**Enhanced For Loop:**

for(int[] x1 : x)

{

for( int x2 : x1)

{

SOP(x2);

}

}

Example 3:

To print elements of three dimensional array

**Enhanced For Loop:**

for(int[][] x1 : x)

{

for(int[] x2 : x1)

{

for(int x3 : x2)

{

SOP(x3);

}

}

}

For each loop is the best choice to retrieve elements of arrays and collections and collections but its limitation is it is applicable only for arrays and collections and it is not a general purpose loop

EG:

for(int i = 0 ; i < 10 ; i++) We cannot write an equivalent

{ for each loop directly here

SOP(“Hello”)

}

By using normal for loop we can print array elements either in original order or in reverse order.

But by using for each loop we can print array elements only in original order but not in reverse order.

EG:

int[] x = {10, 20, 30, 40, 50};

for(int i = x.length – 1 ; i >= 0 ; i--) we cannot write an equivalent

{ for each loop directly

SOP(x[i]);

}

Output:5040302010

* SYNTAX

Iterable interface(I) array/collection

for(eachitem x : target)

{

} Iterable object

The target element in for each loop should be iterable object

An object is said to be iterable iff corresponding class implements java.lang.iterable interface.

Interable interface was introduced in 1.5 version and it contains only one method iterator.

public Iterator iterator()

All array related classes and collection implemented already implement iterable interface

Being a programmer we are not required to do anything , just we should be aware of this point.

**Difference between Iterator and Iterable**

|  |  |
| --- | --- |
| Iterator(I) | Iterable(I) |
| It is related to collections | It is related to for each loop |
| We can use to retrieve the elements of collections one by one | The target element in for-each loop should be Iterable |
| Java.util pkg | Java.lang pkg |
| Has 3 methods  1 . hasNexrt()  2 next()  3.remove() | Has 1 method  Iterator() |

**Transfer Statements**

**BREAK STATEMENT**

We can use break statement in the following places

1. **Inside switch** to stop fall through

int x = 0;

switch(x)

{

case 0:

SOP(0);

case 1:

SOP(1);

Break;

case 2:

SOP(2);

default:

SOP(“default”);

}

Output:01

1. **Inside loops** to break loop execution based on some condition

for(int i = 0 ; i<10 ; i++)

{

If (I == 5)

Break;

SOP(“ i ”);

}

Output:01234

1. **Inside labelled blocks** to break block execution based on some condition.

Class Test

{

public static void main(String[] args)

{

int x = 10;

p1:

{

SOP(“begin ”);

if(x == 0)

break;

SOP(“end”)

}

SOP(“Hello”);

}

}

Output: begin hello

These are the only places where we can sue break statement. If we are using anywhere else we will get compile time error saying break outside switch or loop.

EG:

**int** i = 10;

**if**(i==10)

**break**;

System.***out***.println("Hello");

Output: Unresolved compilation problem:

break cannot be used outside of a loop or a switch

**CONTINUE STATEMENT**

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

EG:

for(int i = 0 ; i< 10 ; i++)

{

if(I % 2 == 0)

continue;

SOP(i);

}

Output:13579

We can use continue statement only inside loops . if we are using them anywhere else thenwe will get compile time error

EG:

**int** i = 10;

**if**(i==10)

**continue**;

System.***out***.println("Hello");

Output: Unresolved compilation problem:

continue cannot be used outside of a loop

**LABELLED BREAK AND CONTINUE STATEMENT**

We can use labelled break and continue to break or continue the particular loop in nested loops.

P1:

for(----------------)

{

P2:

for(----------------)

{

P3:

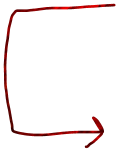
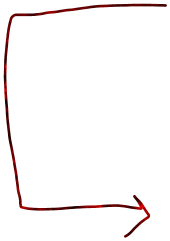
for(--------------)

{

break p1;

break p2;

break;

}

}

}

EG 1:

p1:

**for**(**int** i = 0 ; i < 3 ; i++)

{

**for**(**int** j = 0 ; j<3 ; j++)

{

**if**(i==j)

**break**;

System.***out***.println(i+"..."+j);

}

}

Output: 1...0

2...0

2...1

EG 2:

p1:

**for**(**int** i = 0 ; i < 3 ; i++)

{

**for**(**int** j = 0 ; j<3 ; j++)

{

**if**(i==j)

**break** p1;

System.***out***.println(i+"..."+j);

}

}

Output:(no output)

EG 3:

p1:

**for**(**int** i = 0 ; i < 3 ; i++)

{

**for**(**int** j = 0 ; j<3 ; j++)

{

**if**(i==j)

**continue**;

System.***out***.println(i+"..."+j);

}

}

Output: 0...1

0...2

1...0

1...2

2...0

2...1

EG 4:

p1:

**for**(**int** i = 0 ; i < 3 ; i++)

{

**for**(**int** j = 0 ; j<3 ; j++)

{

**if**(i==j)

**continue** p1;

System.***out***.println(i+"..."+j);

}

}

Output: 1...0

2...0

2...1

**DO WHILE VS CONTINUE STATEMENT**

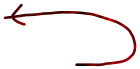
EG:

**int** x = 0;

 **do**

{

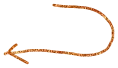
x++;

 System.***out***.println(x);

**if**(++x<5)

**continue**;

 x++;

 System.***out***.println(x);

}**while**(++x<10);

Output:

1

4

6

8

10