IMPLIMENTING NHERITANCE IN JAVA

Introduction

- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.
- With the use of inheritance the information is made manageable in a hierarchical order.
- The class which inherits the properties of other classes is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
- Remember the three types of inheritance (hierarchical, multi-class, and hybrid)
- In Java **extends** is the keyword used to inherit the properties of a class.

Syntax

```
class Super{
..... .....
}
class Sub extends Super{
..... .....
 }
```

- In this syntax super class is the base class while sub is the derived class.

Sample code

```
class Vehicle{
public String sound (){
return("Voom!"); }
}
public class Car extends Vehicle {
    public static void main (String [] arg){
    Car my_car= new Car();
    System.out.println(my_car.sound());
    }
}
```

Cont.

- The code in the previous slide has two classes namely vehicle (base class) and Car derived class. Class Car inherits from class vehicle
- In class Vehicle we have a method known as sound which returns (“Voom”!). Thus when class Car inherits from Vehicle it as well inherits this method.
- Class Car has control of the main method thus the code should be saved as Car.java
- we have created an object of class Car known as my_Car.
- Then this object has implemented the method sound which was created in class Vehicle.
- It is evident that with inheritance we don’t need to create another method sound for class car we just inherit what is defined in the base class.

Sample Code 2: Multi-Level Inheritance

```
class Animals {
public String sound(String x){
return (x);
} }
class WildAnimals extends Animals {
}
```

```
public class Lion extends
WildAnimals {
    public static void main (String []
args){
    Lion big_lion= new Lion();
    String x="roar";

System.out.println(big_lion.sound(x))
;
    }
}
```

Sample Code 2: Cont.

- In sample code 2 Animals is the base class, Wild animals inherits from Animals, and the Lion inherits from WildAnimals.
- Thus the object big_lion is able to implement method sound which is defined in the bae class.
- This is a multi-level inheritance scenario

Inheriting States

- A derived class is able to inherit all the states of the base class.
- This makes it possible for derived classes to avoid defining their own states and thus their objects can implement the sates defined in the base class.
- Take an example below.
- We have class vehicles with states color and brand then class Car inherits from vehicle then its object which is my_car is able to implement the two states defined in the base class as follows.
  - My_car.color=“Blue”
  - My_car.Brand=“Isuzu Dmax”

Example 3: Inheriting States

```
class Vehicle{
    String color;
    String brand ;

public class Car extends Vehicle {
    public static void main (String [] arg){
    Car my_car= new Car();
  my_car.color="Blue";
  my_car.brand="Isuzu Dmax";
  System.out.println(my_car.color);   }
}
```

The output will be “Blue”

Practice Exercise

- Create a base class known as SCIT, two derived classes known as CS and IT.
  - in class SCIT create a method known as display
  - Let the output of that method be “ Welcome to SCIT”
  - Create one object for each of the departments
  - Let the objects implement method display

Assignment

- Discuss at least six Access and non-Access modifiers in Java (30 marks).
  - The assignment should be handwritten in A4 papers
  - To be submitted on or before Thursday 17th February 2021 at 8:00 am.

Control Structures in Java

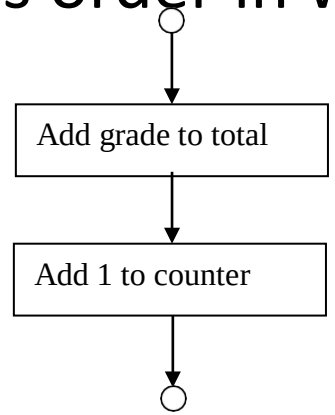
What are a control structures?

- These are programming statements that control the flow of execution in a program. They are organized into three kinds of control structures.
  - Sequence
  - Selection/ Decision
  - Repetition/Iteration/Looping

Sequence control structure

- The sequence structure is essentially built into Unless directed otherwise, the computer automatically executes statements one after the other in the order in which they are written.
- NB: Flowlines indicates order in which actions are performed.

total=total +grade;



Selection/ Decision Control Structure

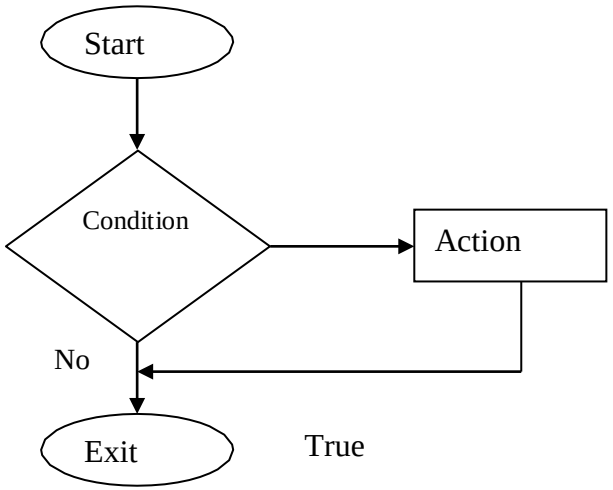
- Decision making structures have one or more conditions to be evaluated or tested by the program, along with a statement or statements that are to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.
- They bring logic to a program by making it possible for programs to evaluate conditions and react to those conditions.

Cont.

- They include:
  i. The if selection statement(Single selection statement)
  ii. The if……else selection statement(Double-selection statement)
  iii. The if……else selection statement(Multiple-selection statement)
  iv. Switch…case statement(Multiple-selection statement)
  v. Nested if…elseif statement

The If…Selection

- It performs (selects) an action if a condition is true or skips the action if the condition is false.



Syntax

```
if(Condition/ Boolean expression)

{

//Statements will execute if the Boolean expression is true
}
```

- If the Boolean expression evaluates to true then the block of code inside the if statement will be executed.
- If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

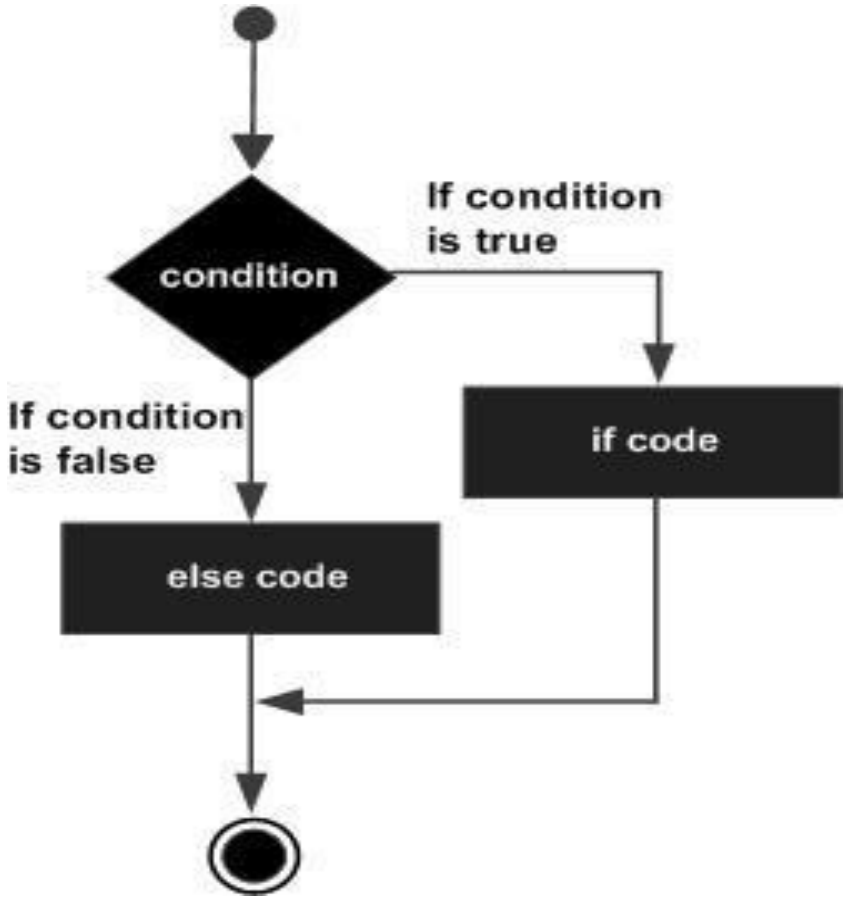
Example 1: Write a program to classify a student as pass if they get 50 marks and above.

```
import java.util.Scanner;

public class Java101 {
    public static void main(String[] args) {
      Scanner sc= new Scanner(System.in);
      System.out.println("Enter Student Marks");
      double marks= sc.nextDouble();
      if (marks>=50){
      System.out.println("Pass");
      }
    }}
```

The if…else selection statement

- It performs an indicated action only when the condition is true; otherwise it executes the else block.



Syntax

```
if(Condition/ Boolean expression){
//Executes when the Boolean expression is true
}else{
//Executes when the Boolean expression is false
}
```

- If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

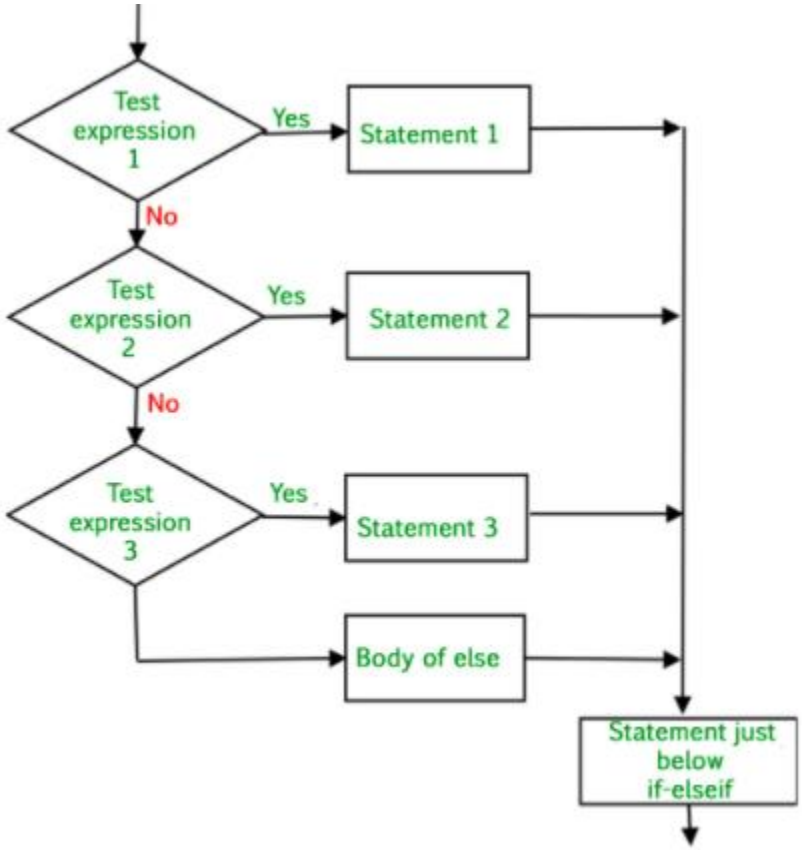
Example: Write a program to classify a student as pass if they get 50 marks and above and fail if they get anything else.

```
import java.util.Scanner;

public class Java101 {
    public static void main(String[] args) {
      Scanner sc= new Scanner(System.in);
      System.out.println("Enter Student Marks");
      double marks= sc.nextDouble();
      if (marks>=50){
      System.out.println("Pass");
      } else {
      System.out.println("fail"); }  }  }
```

Multiple if…else Statement

- It tests for multiple cases by having multiple if…else statements. It improves program logic
- By making it possible for one
- To evaluate multiple Conditions



Syntax

```
if(Boolean_expression 1){
        Statement 1
}else if(Boolean_expression 2){
        Statement 2
}else if(Boolean_expression 3){
        Statement 3
}else {
        Statement 4
}
```

Example

Write a program to grade students based on the following criteria.

Below 40- fail

40-49- D

50-59- C

60-69- B

70-100 A

Above 100- Invalid

Example

```
import java.util.Scanner;

public class Java101 {
    public static void main(String[] args) {
       Scanner sc= new Scanner(System.in);
       System.out.println("Enter Student Marks");
       double marks= sc.nextDouble();
       if (marks<0){
       System.out.println("invalid");
       }
else if(marks<40) {
       System.out.println("fail");
```

Cont.

```
} else if(marks<=49 && marks >=40){
      System.out.println("D");
      } else if(marks<=59 && marks >=50){
      System.out.println("C");
      } else if(marks<=69 && marks >=60){
      System.out.println("B");
      }else if(marks<=100 && marks >=70){
      System.out.println("A");
      } else {System.out.println("Invald");}
   }
}
```

Nested If…else Structure

- This is where an if…else statement is put inside another if…else statement.
- This can be useful when you want to test a condition inside another condition.
- Think of a program that is supposed to classify student as pass if the student has 60% in Cats and 50% in exam marks.
- Then it means that a student must satisfy both conditions. This can be implemented using nested if…else
- The nested section is highlighted in bold in the following example

Cont.

```
import java.util.Scanner;

public class Java101 {


    public static void main(String[] args) {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter Student CatMarks (%)");
        double cat_marks= sc.nextDouble();
        System.out.println("Enter Student ExamMarks (%)");
        Scanner sc2= new Scanner(System.in);
        double exam_marks=sc2.nextDouble();
        if (cat_marks>= 60){
            if (exam_marks>=50){
                System.out.println("pass");
            }
        }
    }}
```

Decision Structures with Strings

- Unlike C programming, Java makes it possible for variables with string datatypes to be evaluated using decision structure.
- This is very useful when creating real world applications.
- Think of a login page: it has a username and a password. In most cases a user name is usually alphanumeric meaning that it has letters, special characters and even numbers combined. Thus it can only be handled as a String datatype.
- To handle decision making in string datatype java uses the **.equals** method rather than = sign.
- See example below

Cont.

- In general, both equals() and “==” operators in Java are used to compare objects to check equality, but here are some of the differences between the two:
- The main difference between the .equals() method and == operator is that one is a method, and the other is the operator.
- We can use == operators for reference comparison (**address comparison**) and .equals() method for **content comparison**.
- In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

Example: Implementing decision structure in string

```
public class Java201 {
    public static void main(String [] arg){
    String user_name="Kamiri";
    if(user_name.equals("Kamiri")){
    System.out.println("Login Successfull");
    }
    }
}
```

Switch...Case Selection Statement

- A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

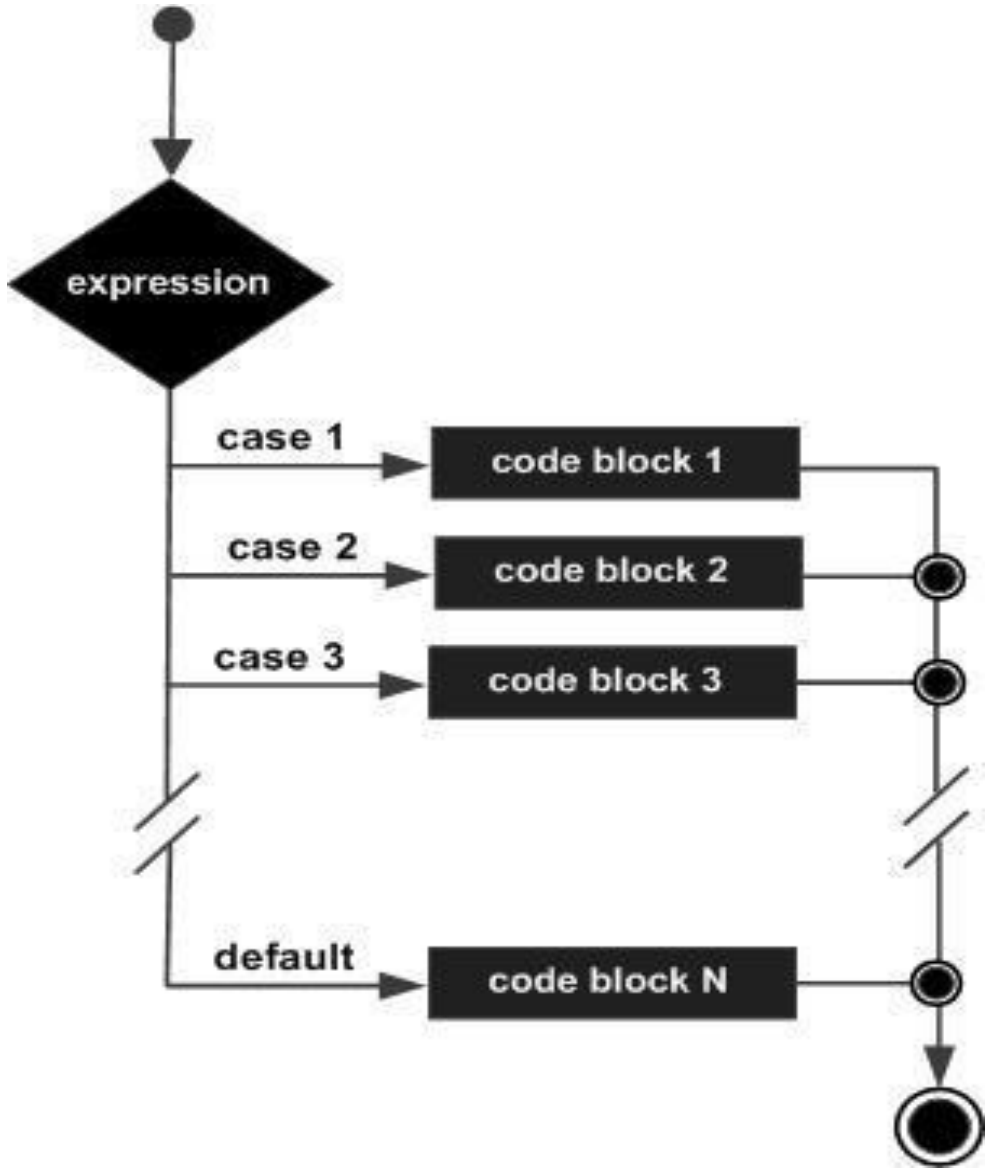
Syntax

```
switch(expression){
case value :
      //Statements
break; //optional
case value :
      //Statements
break; //optional
      //You can have any number of case statements.
default : //Optional
      //Statements
}
```

Switch Statement Rules

- The variable used in a switch statement can only be integers, convertible integers (byte, short, char), strings and enums.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram



Switch Case Example

```
public class Test {
public static void main(String args[]){
char grade = 'C';
switch(grade)
{
case 'A' :
System.out.println("Excellent!");
break;
case 'B' :
case 'C' :
System.out.println("Well done");
break;
case 'D' :
System.out.println("You passed");
case 'F' :
System.out.println("Better try again");
break;
default :
System.out.println("Invalid grade");
}
System.out.println("Your grade is " + grade);
}
}
```

Cont.

- In the switch…Case example in the previous page we have declared variable grade as Char C. then we have used the use case structure to determine what the output will be if the case is C.
- The output of the program is

  Well done

  Your grade is C

Loop Control Structures

Loop Control Structures

- There may be a situation when you need to execute a block of code several number of times.
- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A **loop** statement allows us to execute a statement or group of statements multiple times
- In Java we have the following categories of loop structures
  - For loop
  - While loop
  - Do.. While loop

While.. Loop Structures

- A **while** loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

while(Boolean_expression)

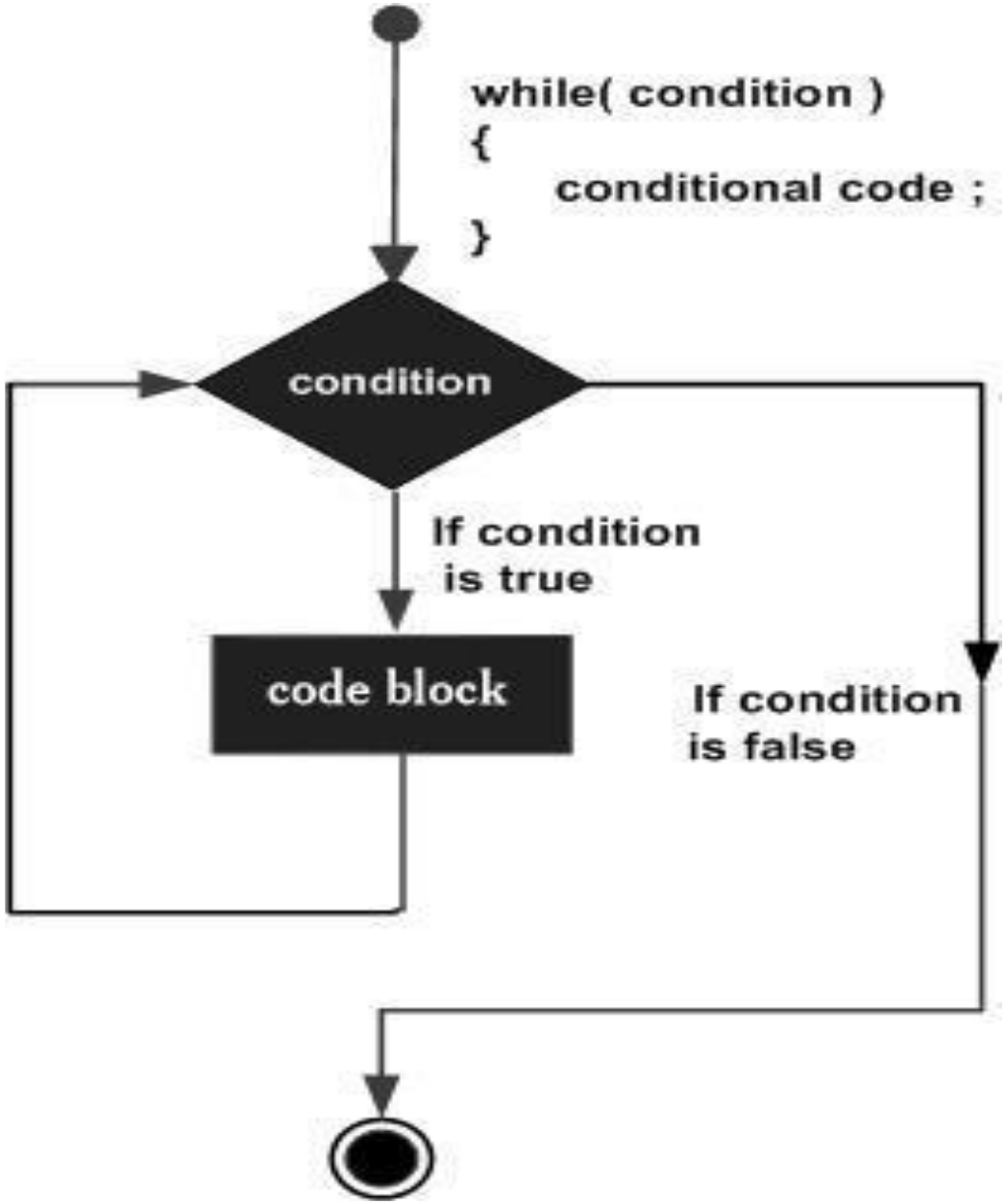
{

//Statements

}

- Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression.
- When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.
- When the condition becomes false, program control passes to the line immediately following the loop.

Flow Graph


while(condition)
{
conditional code ;
}
condition
If condition
is true
code block
If condition
is false

While loop: to print numbers from 10-19

```
public class Test {
public static void main(String args[]) {
int x = 10;
while( x < 20 ) {
System.out.print(x );
x++;
System.out.print("\n");
} }
}
```

For Loop

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.
- A **for** loop is useful when you know how many times a task is to be repeated.

Syntax

```
for(initialization; Boolean_expression; update)
{
//Statements
}
```

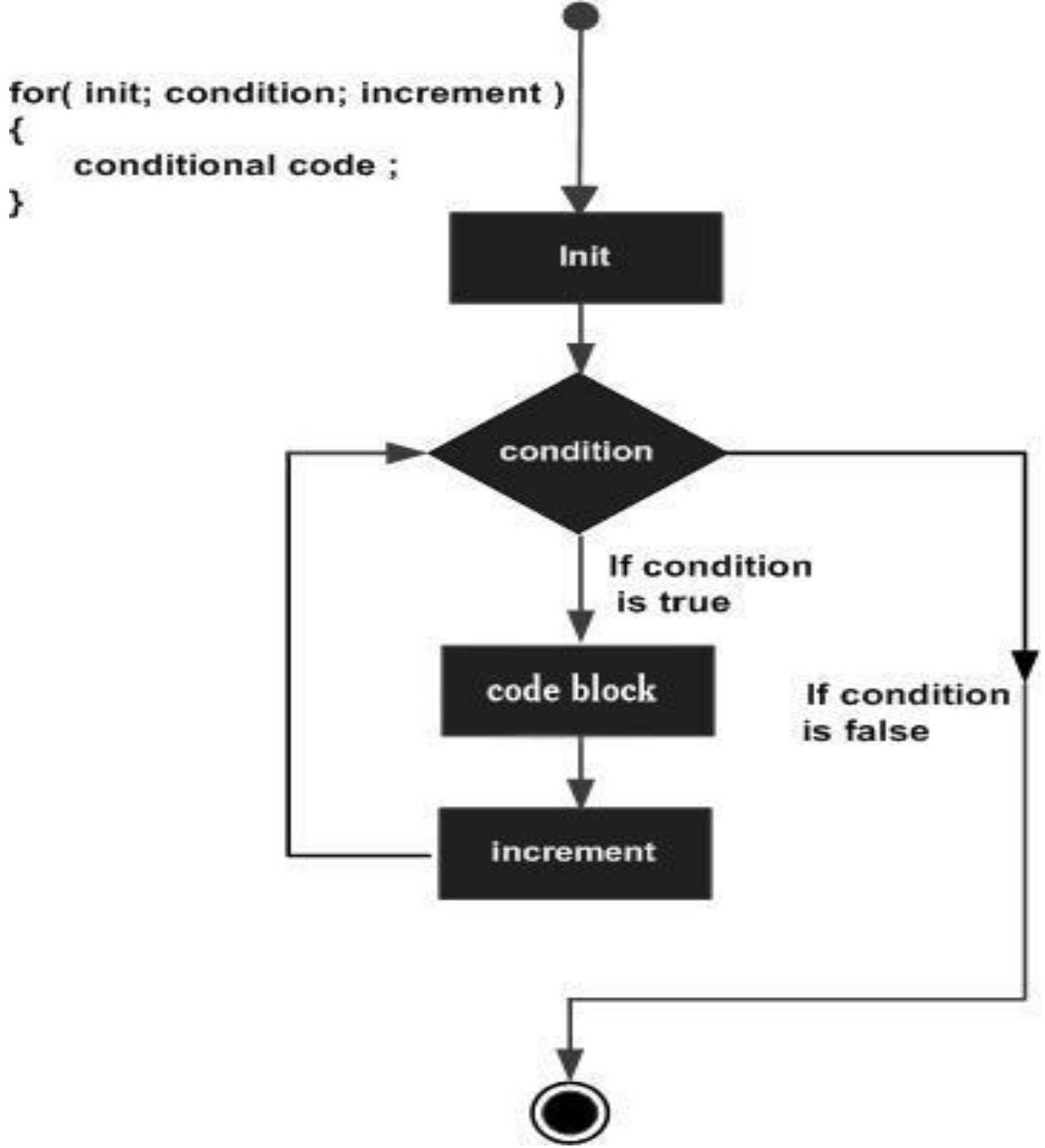
Flow of control in for… loop

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.
- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end. Update can be incrementing of decrementing

Cont.

- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Loop Flow Graph



Example: Using for loop to print values from 10 to 19

```
public class Test {
public static void main(String args[]) {
for(int x = 10; x < 20; x = x+1) {
System.out.print(x );
System.out.print("\n");
}
}
```

Do…While Loop

- A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time before the condition is tested.

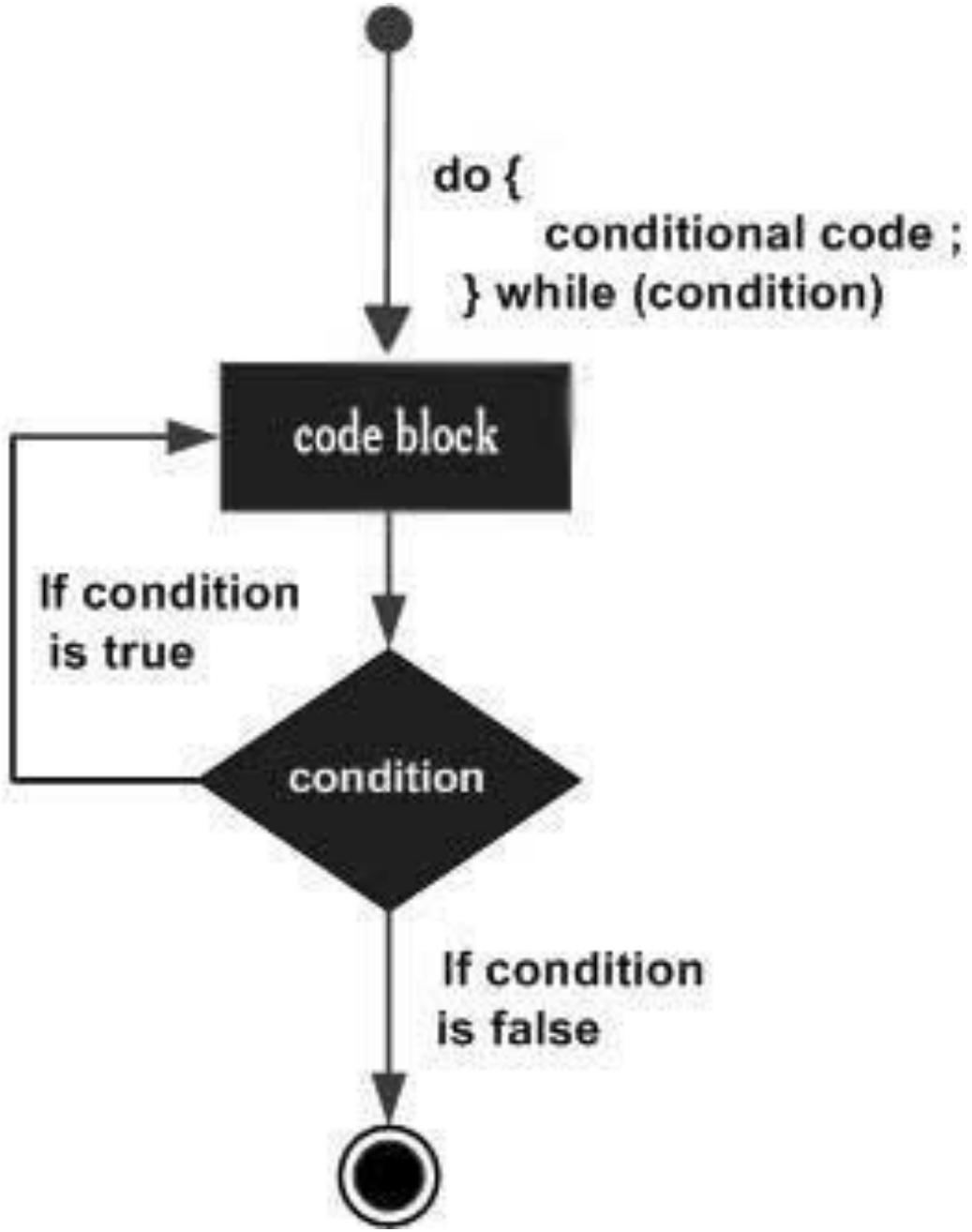
Syntax

```
do
{
//Statements
}while(Boolean_expression);
```

Cont.

- Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.
- If the Boolean expression is true, the control jumps back up to do statement, and the statements in the loop execute again.
- This process repeats until the Boolean expression is false.

Do…While flow Graph

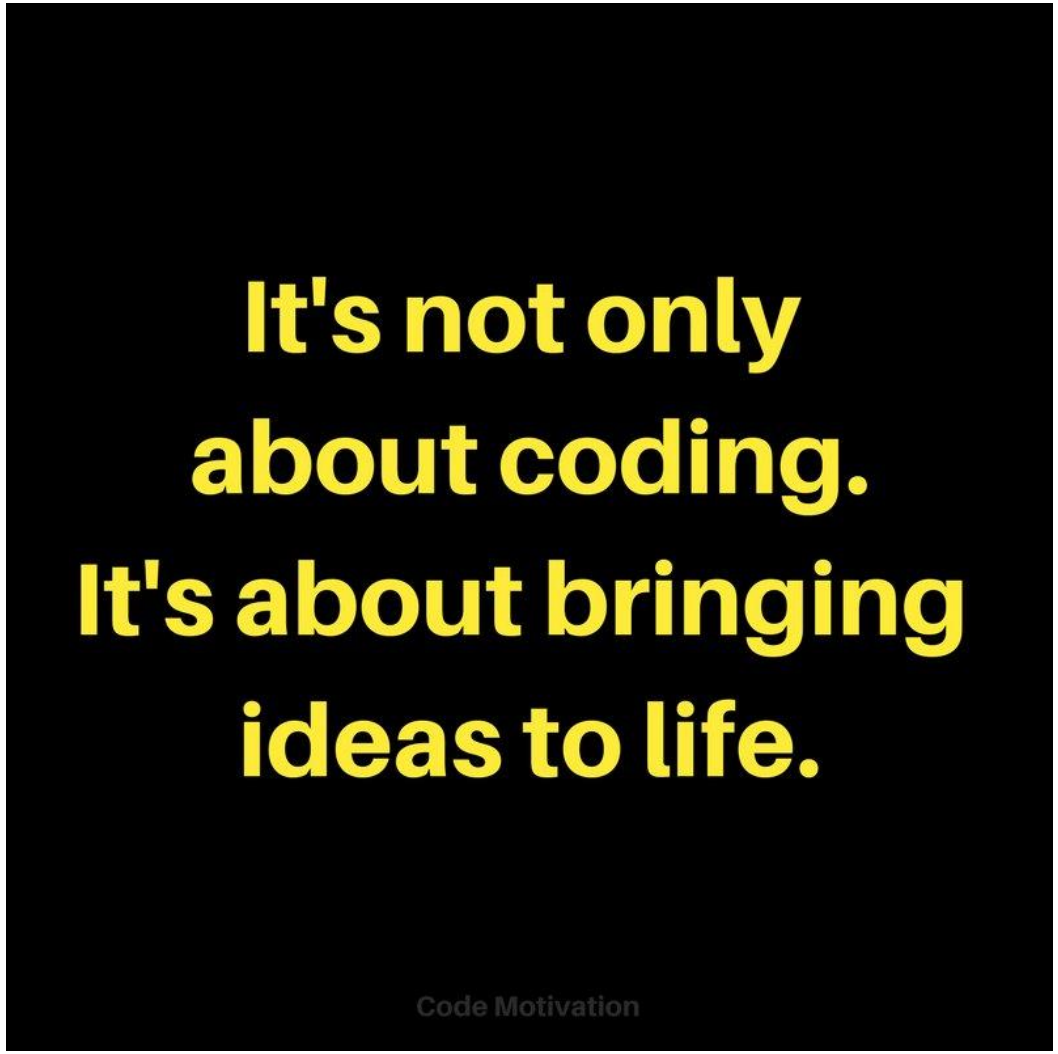


Do...While Example

```
public class Test {
public static void main(String args[]){
int x = 10;
do{
System.out.print(x );
x++;
System.out.print("\n");
}while( x < 20 );
} }
```

The output is 10, 11,12, 13, 14, 15, 16, 17, 18 ,19

The End


It's not only
about coding.
It's about bringing
ideas to life.
Code Motivation