Java Fundamentals

Siddharth Sharma

Iteration Statements:

- **While:**

while(*condition*) {
// body of loop }

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop.

```
class While
{
   public static void main(String args[])
   {
      int tick=10;
      while(tick>0)
      {
         System.out.println("Tick is "+tick);
         tick--;
      }
   }
}
```

OUTPUT:

```
Tick is 10
Tick is 9
Tick is 8
Tick is 7
Tick is 6
Tick is 5
Tick is 4
Tick is 3
Tick is 2
Tick is 1
```

- Even putting a semi-colon is syntactically correct. To demonstrate this, see the below example:

class example

```
{
   public static void main(String args[])
   {
      int i,j;
      i=100;
      j=200;
      while(++i<--j);
      System.out.println("Midpoint is "+i);
   }
}
```

OUTPUT:

```
Midpoint is 150
```

• do-while loop:

Sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.

do {
// body of loop

} while (*condition*);

```
class example
{
   public static void main(String args[])
   {
      int tick=10;
      do{
         System.out.println(“Tick is “+tick);
         tick--;
      }
      while(tick>0);
   }
}
```

OUTPUT:

```
Tick is 10
Tick is 9
Tick is 8
Tick is 7
Tick is 6
Tick is 5
Tick is 4
Tick is 3
```

```
Tick is 2
Tick is 1
```

- Even this will do:

```
do
    {
       System.out.println("Tick is "+tick);
    }
    while(--tick>0);
```

First the decrementing is done, followed by comparison.

- **The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.**

```
class example
{
   public static void main(String args[])
   throws java.io.IOException
   {
      char choice;
      do{
         System.out.println("1. Addition");
         System.out.println("2. Subtraction");
         System.out.println("3. Multiplication");
         System.out.println("4. Division");
         System.out.println("Enter choice");
         choice=(char)System.in.read();
      }
      while(choice<'1'||choice>'4');
      switch(choice)
      {
         case '1':
            {
               System.out.println("Addition is a+b");
               break;
            }
         case '2':
            {
               System.out.println("Subtraction is a-b");
               break;
            }
         case '3':
            {
               System.out.println("Multiplication is a*b");
```

```
            break;
          }
        case '4':
          {
            System.out.println("Division is a/b");
            break;
          }
        default:
          {
            System.out.println("Please enter a valid choice.");
          }
      }
    }
}
```

OUTPUT:

```
1. Addition
2. Subtraction
3. Multiplication
4. Division
Enter choice
Multiplication is a*b
```

- **System.in.read( )** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program.

- Because **System.in.read( )** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors. It is part of Java's exception handling features.

•for:

Traditional for statement:

for(*initialization*; *condition*; *iteration*) {

// body

}

```
class example
{
```

```
    public static void main(String args[])
    {
        int i,num;
        num=20;
        boolean isprime= true;
        for(i=2;i<=num/2;i++)
        {
            if(num%i==0)
            {
                isprime=false;
                break;
            }
        }
        if(isprime==true)
        System.out.println(""+num+" is prime");
        else
        System.out.println(""+num+" is not prime");
    }
}
```

OUTPUT:

```
20 is not prime
```

- **Here we can even declare i directly within the for loop statement. But remember this is the case only when we do not wish to use the counter variable again. Because we cannot use i again because it will be out of scope.**

```
class example
{
    public static void main(String args[])
    {
        int num;
        num=23;
        boolean isprime= true;
        for(int i=2;i<=num/2;i++)
        {
            if(num%i==0)
            {
                isprime=false;
                break;
            }
        }
        if(isprime==true)
        System.out.println(""+num+" is prime");
        else
```

```
        System.out.println(""+num+" is not prime");
    }
}
```

- **There will be times when you will want to include more than one statement in the initialisation and iteration portions of the for loop.**

```
class example
{
    public static void main(String args[])
    {
        int a,b;
        for(a=1,b=4 ; a<b ; a++,b--)
        {
            System.out.println("a= "+a);
            System.out.println("b= "+b);
        }
    }
}
```

Above code can also be coded without using the coma.

```
class example
{
    public static void main(String args[])
    {
        int a,b;
        b=4;
        for(a=1;a<b;a++)
        {
            System.out.println("a= "+a);
            System.out.println("b= "+b);
            b--;
        }
    }
}
```

Both snippets produce the same output.

OUTPUT:

```
a= 1
b= 4
a= 2
b= 3
```

NOTE: If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. In Java, the comma is a separator.

- **Some for loop variations:**

• Condition controlling the **for** can be any Boolean expression.

```
boolean done = false;
for(int i=1; !done; i++) {
  // ...
  if(interrupted()) done = true;
}
```

The program runs until the value of ‘done’ is set to false. As soon as ‘done’ is set to true, condition becomes not true i.e. false.

• Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for** empty.

```
for(     ;       ;       )
{
  // ...
}
```

•<u>for-each:</u>

For-each style of **for** is also referred to as the *enhanced* **for** loop.
A for- each style loop is designed to cycle through a collection of objects, such as an array, in strictly **sequential fashion, from start to finish**.

for(*type itr-var : collection*) *statement-block*

- **‘Type’** specifies the type and **‘itr-var'** specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The **collection** being cycled through is specified by collection.
- With each iteration of the loop, the **next element in the collection is retrieved and stored in itr-var.** The loop repeats until all elements in the collection have been obtained.
- The iteration variable receives values from the collection, **type** must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array.

```
class example
{
   public static void main(String args[])
   {
      int sum=0;
      int array[]={1,2,3,4,5,6,7,8,9,10};
      for(int i:array)
      {
         sum=sum+i;
```

```
        }
        System.out.println("Sum is "+sum);
    }
}
```

OUTPUT:

```
Sum is 55
```

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement.

```
class example
{
    public static void main(String args[])
    {
        int sum=0;
        int array[]={1,2,3,4,5,6,7,8,9,10};
        for(int i:array)
        {
            sum=sum+i;
            if(i==5)
            break;
        }
        System.out.println("Sum is "+sum);
    }
}
```

OUTPUT:

```
Sum is 15
```

Not only is the syntax streamlined, but it also prevents boundary errors.

A WORD OF CAUTION:

The iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can’t change the contents of the array by assigning the iteration variable a new value.

```
class example
{
    public static void main(String args[])
    {
        int array[]=new int[10];
        int j=1;
        for(int i=0;i<10;i++)
        {
            array[i]=j;
            j++;
        }
```

```
        for(int x:array)
        {
            System.out.print(" "+x);
        }
        for(int x:array)
        {
            x=x*10;
        }
        System.out.println();
        for(int x:array)
        {
            System.out.print(" "+x);
        }

    }
}
```

OUTPUT:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

ITERATING OVER MULTI-DIMENSIONAL ARRAYS USING FOR-EACH STATEMENT:

```
class example
{
    public static void main(String args[])
    {
        int sum=0;
        int array[][]= new int[3][5];
        int i,j;
        for(i=0;i<3;i++)
        {
            for(j=0;j<5;j++)
            {
                array[i][j]=(i+1)*(j+1);
            }
        }
        for(int x[]:array)
        {
            for(int y:x)
            {
                System.out.println("Value is "+y);
                sum=sum+y;

            }
        }
        System.out.println();
        System.out.println("Sum is "+sum);

    }
}
```

OUTPUT:

```
Value is 1
Value is 2
Value is 3
Value is 4
Value is 5
Value is 2
Value is 4
Value is 6
Value is 8
Value is 10
Value is 3
Value is 6
Value is 9
Value is 12
Value is 15

Sum is 90
```