Recursion

The term “recursion” means a method calling itself.

Recursion in Java is a basic programming technique in which a method calls itself repeatedly. A method that calls (or invokes) itself is called recursive method.

In other words, a process in which a method invokes itself directly or indirectly is called recursion, and the related method is called a recursive method.

A recursive method is a chain of method calls to the same method. It is like the loop with or without a terminating condition. It calls itself, either directly or indirectly, via another method.

A famous example often used to describe the recursion in Java is the calculation of factorial (the factorial for a value of 5 is 5 * 4 * 3 * 2 * 1).

```
package recursion;
public class Recursion1 {
static void display()
{
  System.out.println("Hello world");
  display(); // recursive call
}
public static void main(String[] args)
{
 display(); // normal method call
  }
}
```

we have called the display() method from inside the main() method. This is a normal method call.

When we are again calling the same display() method inside the display() method, then it is a recursive call.

The display() is a recursive method that is invoking itself inside the method. It is also possible that recursive method invokes itself indirectly.

We can stop the recursive call by providing some conditions inside the method. While defining recursive method, it is necessary to specify a termination condition. If we do not specify such a condition, then the method may call infinite times.

```
package recursion;
public class Recursion2 {
static int count = 0;
static void display()
{
  count++;
  if(count <= 5) // base case
  {
    System.out.println("Hello world");
    display(); // recursive call
  }
}
public static void main(String[] args)
{
 display(); // normal method call
  }
}
```

(n) * (n – 1) * (n – 2) * (n – 3) * . . . . . * 1.

```
package recursion;
public class Factorial {
  static int fact_itr(int num) {
    if (num < 0)
      return num;
    int total = 1;
    for (int n = num; n > 1; n--) {
      total = total * n;
    }
    return total;
  }

  static int fact_rec(int num) {
    if (num == 1 || num == 0) // base case.
    {
      return 1;
    }
    return num * fact_rec(num - 1); // recursive call.
  }

  public static void main(String[] args) {
    System.out.println("The factorial of 5 is " + fact_itr(5));
    System.out.println("The factorial of 6 is " + fact_rec(6));
  }
}
```

```
static int fact_itr(int num) {
    int total = 1;
    for (int n = num; n > 1; n--) {
      total = total * n;
    }
    return total;
  }

//n * (n-1) * (n-2) * (n-3)...
```

we can see a recurrence relation of:

```
//n * (n-1) * (n-2) * (n-3)...
n!:   1 for n = 0,
      (n-1)! * n for n > 0
```

Looking at this from a programming point of view, this means:
`factorial(0); = 1`, and `factorial(n); = factorial(n-1) * n;`

Base Case:

Since `factorial(0);` is the simplest form we can achieve, it is our base case. This means we want to keep calling our `factorial();` method until the input is equal to `0`.

Recursive Case:

Given that `factorial(0);` is our ***Base Case*** we can conclude that `factorial(n) = factorial(n - 1) * n;` is our ***Recursive Case***.

Now that we have our ***Base Case*** and ***Recursive Case*** we can construct our recursive method:

```
private int factorial(int n)
{
    // Our Base Case
    if(n == 0)
    {
        return 1;
    }

    // Our Recursive Case
    return factorial(n - 1) * n;
}
```

- fact(6) returns 6 * fact(5)
- fact(5) returns 6 * 5 * fact(4)
- fact(4) returns 6 * 5 * 4 * fact(3)
- fact(3) returns 6 * 5 * 4 * 3 * fact(2)
- fact(2) returns 6 * 5 * 4 * 3 * 2 * fact(1)
- fact(1) returns 6 * 5 * 4 * 3 * 2 * 1.
- fact(1) or fact(0) returns 1. 1! is equal 1 and 0! is also 1.

1, 1, 2, 3, 5, 8, 13, 21, ...

```
package recursion;
public class Fibonacci {
  static int getFibo_itr(int num) {
    if (num <= 1)
      return num;
    int sum = 0;
    int last = 1;
    int last_last = 0;
    for (int i = 1; i < num; i++) {
      sum = last_last + last;
      last_last = last;
      last = sum;
    }
    return sum;
  }
  static int getFibo_rec(int num) {
    if (num <= 1) // base case.
    {
      return num;
    } else {
      return getFibo_rec(num - 1) + getFibo_rec(num - 2);
    }
  }

  public static void main(String[] args) {
    System.out.println("Sum of Fibonacci numbers: " + getFibo_itr(10));
    System.out.println("Sum of Fibonacci numbers: " + getFibo_rec(10));
  }
}
```

```
package recursion;
public class Sum_of_digits {
  // Create a static method to check sum of its digits using recursion.
  static int getSum(int n) {
    if (n == 0)
      return 0;
    return (n % 10 + getSum(n / 10));
  }

  public static void main(String[] args) {
    int num = 12345;
    int result = getSum(num);
    System.out.println("Sum of digits of a number " + num + " = " + result);
  }
}
```

- getSum(12345) returns 5 + getSum(12345 / 10)
- getSum(1234) returns 5 + 4 + getSum(1234 / 10)
- getSum(123) returns 5 + 4 + 3 + getSum(123 / 10)
- getSum(12) returns 5 + 4 + 3 + 2 + getSum(12 / 10)
- getSum(1) return 5 + 4 + 3 + 2+ 1 + getSum(1 / 10)
- 0 algorithm stops.

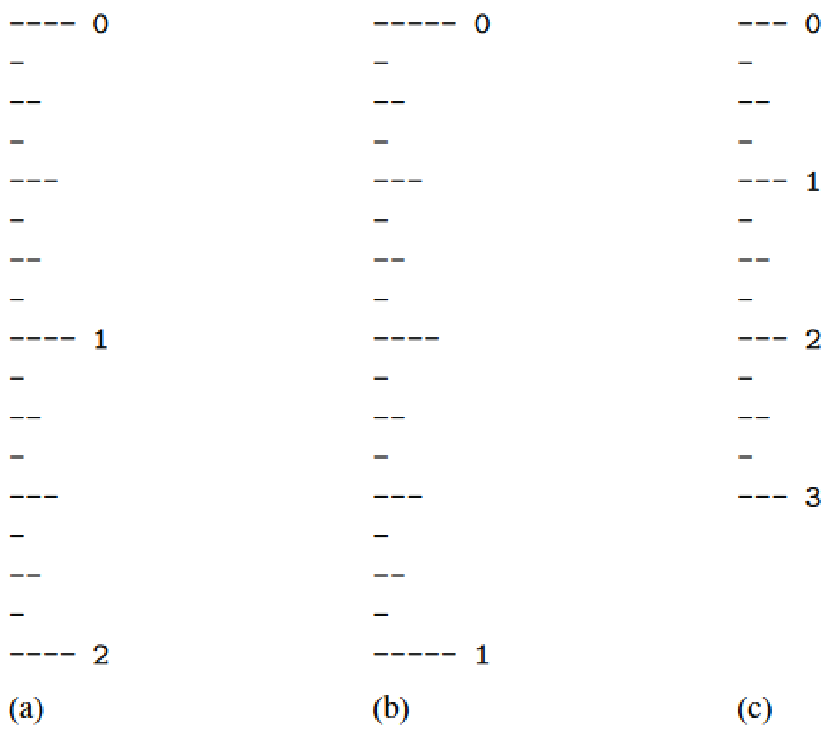
```
package recursion;
public class SpellDigit {
  static void spell_num(int number) {
    if (number < 10) // base case.
    {
      System.out.println(number);
    } else {
      spell_num((int) Math.floor(number / 10)); // recursive call.
      System.out.println(number % 10);
    }
  }

  public static void main(String[] args) {
    spell_num(789);
  }
}
```

- static void spell_num(789) // method called with input 789.
- static void spell_num(78) // method called with input 78.
- static void spell_num(7) // method called with input 7.

```
package recursion;
public class Palindrome {
  static boolean checkPalindrome(String s, int left, int right) {
    if (left >= right) {
      return true;
    }
    if (s.charAt(left) != s.charAt(right)) {
      return false;
    }
    return checkPalindrome(s, left + 1, right - 1);
  }

  static void palindrome(int a) {
    String s = Integer.toString(a);
    int n = s.length();
    if (checkPalindrome(s, 0, n - 1)) {
      System.out.println("The number " + a + " is palindrome");
    } else {
      System.out.println("The number " + a + " is not palindrome");
    }
  }

  public static void main(String[] args) {
    palindrome(1235321);
    palindrome(12101201);
  }

}
```

Drawing an English Ruler

how to draw the markings of a typical English ruler. For each inch, we place a tick with a numeric label. We denote the length of the tick designating a whole inch as the major tick length. Between the marks for whole inches, the ruler contains a series of minor ticks, placed at intervals of
1/2 inch, 1/4 inch, and so on. As the size of the interval decreases by half, the tick
length decreases by one. demonstrates several such rulers with varying
major tick lengths (although not drawn to scale).



```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

(a)

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

(b)

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

(c)

Three sample outputs of an English ruler drawing: (a) a 2-inch ruler
with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch
ruler with major tick length 3.

```
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|                   |                   |                   |                   |
0                   1                   2                   3                   4
```