Algorithms
and
Data Structures Using Java

- Soumya

Recursion

- In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion.
- A physical world example would be to place two parallel mirrors facing each other.
- Any object in between them would be reflected recursively.

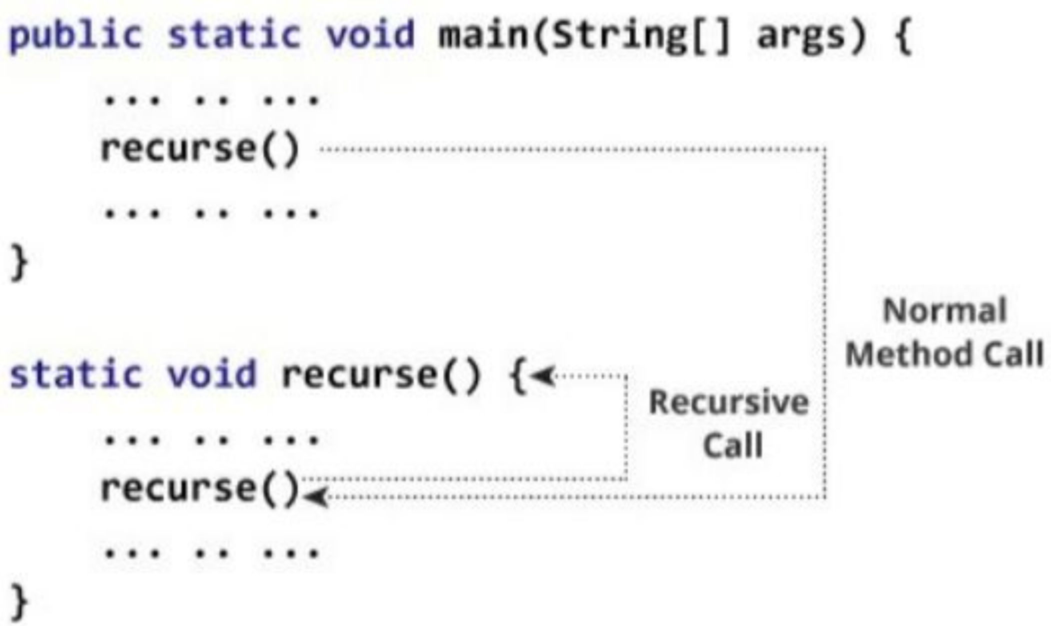
Recursion

Any function which calls itself directly or indirectly is called Recursion and the corresponding function is call as recursive function.

- **A recursive method solves a problem by calling a copy of itself to work on a smaller problem.**
- **It is important to ensure that the recursion terminates.**
- **Each time the function call itself with a slightly simple version of the original problem.**
- **Using recursion, certain problems can be solved quite easily.**

E.g: Tower of Hanoi (TOH), Tree traversals, DFS of Graph etc.,

How it works?


public static void main(String[] args) {
... .
recurse()
... .
}
static void recurse() {
... .
recurse()
... .
}
Normal Method Call
Recursive Call

What is base condition in recursion?

- In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
  if (n < = 1) // base case
    return 1;
  else
    return n*fact(n-1);}
```

What is base condition in recursion?

- In the above example, base case for $n = 1$ is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Why base condition ?

```
class A {

    abc() {

        abc(); // Recursive call to itself

    }


    main() {

        abc(); // First call to abc() from main

    }

}
```

Direct Recursion

• Direct and indirect recursion in Java are two types of recursion where a function calls itself.

• Direct recursion occurs when a function directly calls itself.

Example: calculate the factorial of a number uses direct recursion:

```
public static int factorial(int n) {

if (n == 0) {

return 1;

} else {

return n * factorial(n - 1);

}

}
```

Indirect Recursion

- Indirect recursion occurs when a function calls another function, which then calls the original function directly or indirectly.
- For example, the following two functions use indirect recursion to reverse a string:

Indirect Recursion

```
public static String reverse(String str)

{

if (str.length() == 0) {

return "";

} else {

return reverse(str.substring(1))+str.charAt(0);

}

}

public static String reverseHelper(String str) {

return reverse(str);

}
```

Memory Allocation

- When a function is called in Java, a stack frame is allocated on the stack.
- The stack frame is a region of memory that stores the local variables and parameters of the function. When the function returns, the stack frame is deallocated.
- In recursive functions, a new stack frame is allocated for each recursive call.
- This means that the memory usage of a recursive function can grow exponentially with the number of recursive calls.

Memory Allocation

To avoid this, it is important to design recursive functions carefully.

- One way to do this is to use a base case to stop the recursion as soon as possible. Another way to reduce memory usage is to use tail recursion.
- Tail recursion is a type of recursion where the recursive call is the last thing the function does.
- This means that the stack frame for the current recursive call can be deallocated before the stack frame for the previous recursive call is returned.

Memory Allocation

Here is an example of a recursive function that is not tail recursive:

```
public static int factorial(int n) {
if (n == 0) {
return 1;
} else {
return n * factorial(n - 1);
}
}
```

Memory Allocation

- This function will allocate a new stack frame for each recursive call.
- Here is an example of a tail recursive function to calculate the factorial of a number:

```
public static int factorialTailRecursive(int n, int accumulator) {
if (n == 0) {
return accumulator;
} else {
return factorialTailRecursive(n - 1, n * accumulator);
}
}
```

Memory Allocation

➢ Here are some tips for reducing memory usage in recursive functions:

– Use a base case to stop the recursion as soon as possible.

– Use tail recursion whenever possible.

– Avoid using global variables in recursive functions.

– Pass as few arguments to recursive functions as possible.

– Use tail call optimization (TCO), if available on your compiler.

Pros: Recursion

• Elegance: Recursive functions can be very concise and elegant, especially for problems that can be naturally divided into subproblems.

• Expressiveness: Recursion can be used to express complex algorithms in a clear and concise way.

• Modularity: Recursive functions can be used to implement complex algorithms in a modular way, making them easier to understand and maintain.

• Efficiency: Tail recursive functions can be very efficient, and some compilers can optimize them to use the same stack frame for all recursive calls.

Cons: Recursion

- **Memory usage:** Recursive functions can use a lot of memory, especially if they are not tail recursive.
- **Complexity:** Recursive functions can be difficult to understand and debug, especially for complex problems.
- **Stack overflows:** Recursive functions can cause stack overflows if the recursion depth is too large.

Function complexity

- The function complexity during recursion depends on the following factors:
  - The number of recursive calls: The more times the function calls itself, the more complex the function will be.
  - The complexity of the recursive calls: The complexity of the recursive calls also contributes to the overall complexity of the function.
  - The complexity of the base case: The complexity of the base case is the complexity of the simplest form of the problem that can be solved directly.