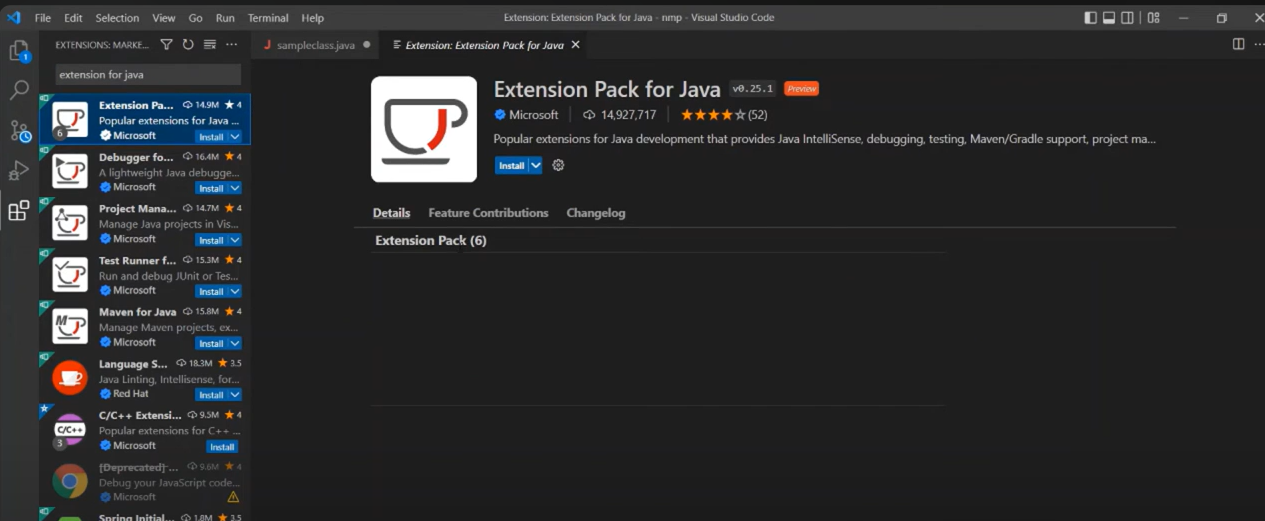
# VS Code Debugging

**#Enable autocomplete or intellisense for Java in visual studio code**

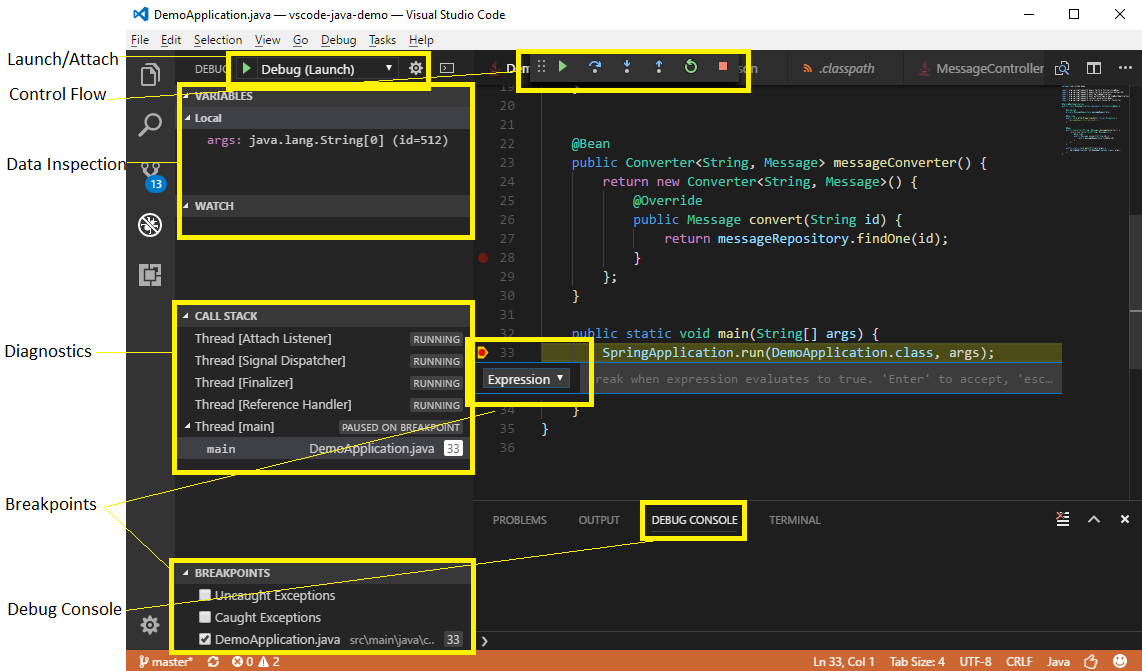
Click on Extentions icon -🡪 type “extension for java” 🡪 Install “Extension pack for java”🡪 then restart



## #[Supported features](https://code.visualstudio.com/blogs/2017/09/28/java-debug#_supported-features)

Following are supported features:

* **Launch/Attach** - You can either launch the Java project within VS Code or attach to any running JVM process in debug mode, locally or remotely.
* **Breakpoints** - Conditional breakpoints by Hit Count is supported and can easily be set using the inline breakpoint settings window. This allows you to conveniently add conditional breakpoints to your code, directly in the source viewer, without requiring a modal window. Break on exceptions is also supported.
* **Control flow** - Including **Pause**, **Continue** F5, **Step over** F10, **Step into** F11, **Step out** Shift+F11
* **Data inspection** - When you're stopped at a breakpoint, the debugger has access to the variable names and values that are currently stored in memory. Inspect/Watch/Set Variables are supported.
* **Diagnostics** - The **CALL STACK** panel shows the call stack of your program and allows you to navigate through the call path of each captured allocation. Multi-threaded debugging is supported by parallel stacks.
* **Debug Console** - The Debug Console lets you see information from both stdout and stderr.



|  |
| --- |
| **Introduction**- Problem Solving, Debugging |

# Problem Statement:- Swapping of Two Numbers

Different methods of solving the problem

1. [Swapping Two Numbers Using a Temporary Variable](https://www.shiksha.com/online-courses/articles/swapping-of-two-numbers-in-java/#1)
2. [Swapping Two Numbers Using Arithmetic Operators + and –](https://www.shiksha.com/online-courses/articles/swapping-of-two-numbers-in-java/#2)
3. [Swapping Two Numbers Using Arithmetic Operators x and /](https://www.shiksha.com/online-courses/articles/swapping-of-two-numbers-in-java/#3)
4. [Swapping Two Numbers Using Bitwise XOR operator ^](https://www.shiksha.com/online-courses/articles/swapping-of-two-numbers-in-java/#4)

|  |
| --- |
| **Method1:- using Temporary variable**  public class Main {  public static void main(String[] args) {  int x = 1;  int y = 2;  System.out.println("Before swapping: x = " + x + " y = " + y);  // Use a temporary variable to store the value of x  int temp = x;  // Assign the value of y to x  x = y;  // Assign the value of the temporary variable (the original value of x) to y  y = temp;  System.out.println("After swapping: x = " + x + " y = " + y);  }  } |
| **Time Complexity: O(1)** as it takes constant time complexity to execute regardless of the size of the input. The algorithm only executes a fixed number of operations (three assignments) regardless of the values of a and b.  **Space Complexity : O(1)** as the algorithm uses only a fixed amount of extra space, regardless of the input size. But this approach does use an extra temporary space to preserve the values, this space can be optimised in the later approaches. |
| Method2:- **Using Arithmetic Operators + and –** public class Main {  public static void main(String[] args) {  int x = 1;  int y = 2;  System.out.println("Before swapping: x = " + x + " y = " + y);  // Swap the numbers using addition and subtraction  x = x + y; // x is now 3  y = x - y; // y is now 1  x = x - y; // x is now 2  System.out.println("After swapping: x = " + x + " y = " + y);  }  } |
| Time Complexity: O(1)Space Complexity: O(1) |
| Method3:- **Using Arithmetic Operators x and /** public class Main {  public static void main(String[] args) {  int x = 1;  int y = 2;  System.out.println("Before swapping: x = " + x + " y = " + y);  // Swap the numbers using division and multiplication  x = x \* y; // x is now 2  y = x / y; // y is now 1  x = x / y; // x is now 2  System.out.println("After swapping: x = " + x + " y = " + y);  }  } |
| Time Complexity: O(1)Space Complexity: O(1) |
| Method4:- **Using Bitwise XOR operator ^** public class Main {  public static void main(String[] args) {  int x = 1;  int y = 2;  System.out.println("Before swapping: x = " + x + " y = " + y);  // Swap the numbers using bitwise XOR  x = x ^ y; // x is now 3 (011 ^ 010 = 011)  y = x ^ y; // y is now 1 (011 ^ 010 = 001)  x = x ^ y; // x is now 2 (011 ^ 001 = 010)  System.out.println("After swapping: x = " + x + " y = " + y);  }  } |
| Optimal SolutionTime Complexity: O(log2(log2N + 1)) as it takes constant time complexity to execute regardless of the size of the input. The algorithm only executes a fixed number of operations (three assignments) regardless of the values of a and bSpace Complexity : O(1) as the algorithm uses only a fixed amount of extra space, regardless of the input size. But this approach does use an extra temporary space to preserve the values, this space can be optimised in the later approaches. |

|  |
| --- |
| **Algorithms -** Complexity-calculation rules, complexity classes, Estimatiing complexity |

**Algorithm** is a method of representing the step-by-step logical procedure for solving a problem.

**Properties of algorithm**

1. **Finiteness**, algorithm must terminate in finite number of steps.
2. **Definiteness**, each step must be precise & unambigious
3. **Effectiveness**, each step must be primitive i.e, easily converted to program
4. **Generality**, algorithm must be complete in itself.
5. **Input/Output**

**Complexity of an algorithm**, refers to the amount of resources (such as time or memory) required to solve a problem or perform a task.

**Claculating Complexity:-**

There are 2 ways for calculating complexity of algorithm

1. Frequency count / step count method
2. Asymptotic notations

**1.Frequency count / step count method**

It specifies the number of times a statement is to be executed

**Rule1**:-For comments & declarations the step count is 0

**Rule2**:-For return & assignment statemnets the step count is 1

**Rule3**:-Ignore lower component exponents when higer order exponents are present

**Rule4**:-Ignore constant multipliers

Ex:-

|  |  |  |
| --- | --- | --- |
| Code | Time Complexity | SpaceComplexity |
| int sum( int a[], int n){  int s=0;  for(i=0;i<n;i++)           s=s+a[i];  return s;  } | ----1  ----n+1  ----n  -----1  Total is 2n+3  Constants are ignored so Time complexity is O(n) | a[]---- n  n-------1  s------- 1  i--------1  Total is n+3  Constants are ignored so space complexity is O(n) |

**2.Asymptotic Notations:-**

Asymptotic notation describe the efficiency and scalability of algorithms. It provides a high-level understanding of an algorithm's behavior, especially as the input size grows.

Asymptotic notation categorizes algorithms based on their performance as the input size grows. They helps predict how an algorithm will perform under different conditions.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

|  |  |  |
| --- | --- | --- |
| **Big - Oh (O)** | **Big - Omega (Ω)** | **Big - Theta (Θ)** |
| express the worst-case scenario  f(n) = O(g(n)) if ∃ constants c > 0, n₀ such that  0 ≤ f(n) ≤ c\*g(n) for  n ≥ n₀.    **Ex**:- Searching in an unsorted list: O(n). | express the best-case scenario  f(n) = Ω(g(n)) if ∃ constants c > 0, n₀ such that   0 ≤ c\*g(n) ≤ f(n) for n ≥ n₀.    **Ex**:- Inserting an element in a sorted array: Ω(1). | Express both worst and best cases  f(n) = Θ(g(n)) if ∃ constants c₁, c₂ > 0, n₀ such that  0 ≤ c₁g(n) ≤ f(n) ≤ c₂g(n) for all n ≥ n₀.  **Ex**:- Linear search in a sorted array, where the element is always in the middle: Θ(n). |

**General Rules for calculating Time Complesity:-**

**Rule1:-** The running time of for loop is the running time of the statements in the for loop n2

|  |  |
| --- | --- |
| Ex:-for(i=0;i<n;i++)        s=s+i; | -----n+1  -----n  Total 2n+1 , as constants are ignored  complexity is O(n) |

|  |  |
| --- | --- |
| Ex:- for (int i = 1; i <= n; i++) {  for (int j = 1; j <= m; j++) {  // code  }  } | ---- n  ----m  Total O(nm) |

**Rule2:-** Nested loops, the total running time of statements in nested loops is the sum of product of all sizes of the loops.

|  |  |
| --- | --- |
| Ex:-  for(i=0;i<n;i++){          for(j=0;j<n;j++){              c[i][j]=a[i][j]+b[i][j];           }  } | ------n+1  ------n\*(n+1)  ------n\*n  Total is 2n2+2n+1  Ignore the lower exponents & constants  So the time complexity is O(n2) |

**Rule3:-** consecutive statements, the running time is maximum value.

|  |  |
| --- | --- |
| Ex:-  for(i=0;i<n;i++)        s=s+i;  for(i=0;i<n;i++){          for(j=0;j<n;j++){              c[i][j]=a[i][j]+b[i][j];           }  } | ------n+1  ------n  ------n+1  ------n\*(n+1)  ------n\*n  Total is 2n2+4n+2  Ignore the lower exponents & constants  So the time complexity is O(n2) |

**Rule4:-** if else, the running timeis maximum of statements of if & else bolcks.

|  |  |
| --- | --- |
| Ex:-  if(n<0)      return n;  else{  for(i=0;i<n;i++)        s=s+i;  return s;  } | ----1  ----1  ----1  -----n+1  -----n  -----1  Total is 2n+5  Ignore the constants  So the time complexity is O(n) |

**Rule5:-**  Recursion

The time complexity of a recursive function depends on the number of times the function is called and the time complexity of a single call. The total time complexity is the product of these values.

|  |  |
| --- | --- |
| Ex:- void f(int n) {  if (n == 1)  return;  f(n-1);  } | The call f(n) causes n function calls, and the time complexity of each call is O(1). Thus, the total time complexity is O(n). |
| Ex:- void g(int n) {  if (n == 1)  return;  g(n-1);  g(n-1);  } | In this case each function call generates two other calls, except for n = 1. Let us see what happens when g is called with parameter n. The following table shows the function calls produced by this single call:  function call number of calls  g(n) 1  g(n−1) 2  g(n−2) 4  ··· ···  g(1) 2n−1  Based on this, the time complexity is 1+2+4+··· +2 n−1 = 2 n −1 = O(2n ) |