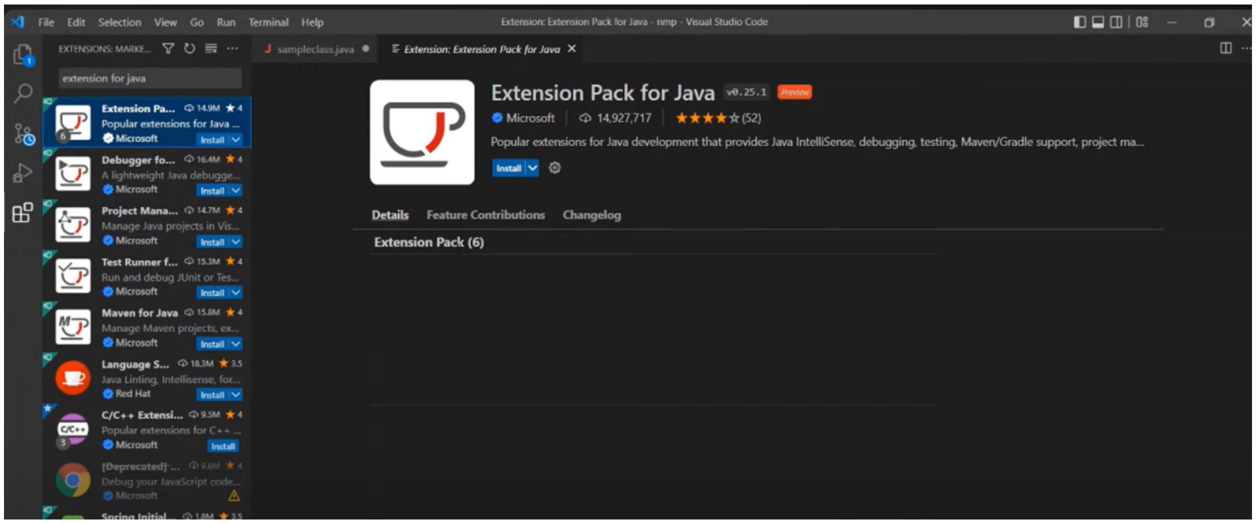
Professional Development Skills Session-1

--Debugging
--Complexity Analysis
--Practice programs

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

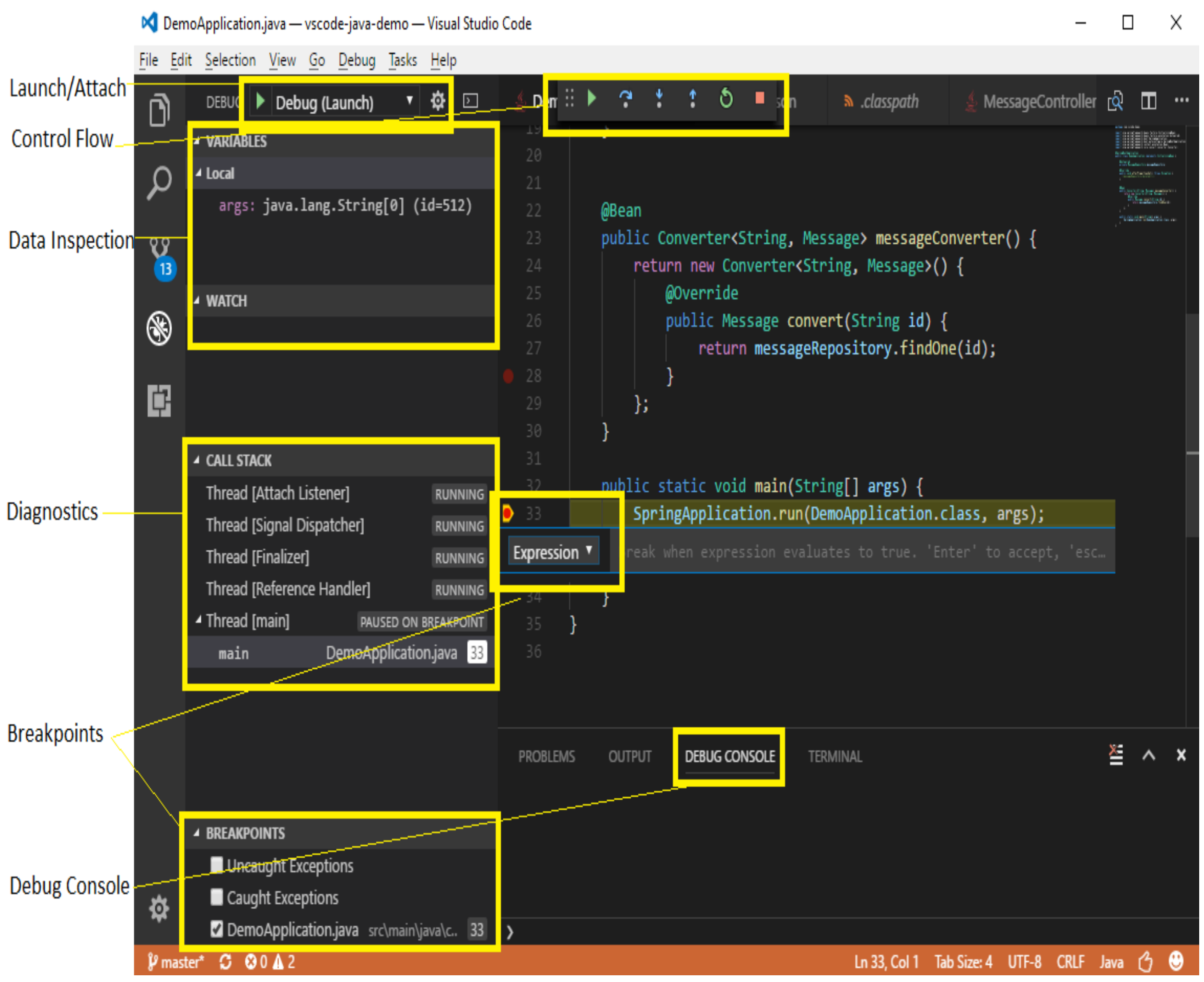

File Edit Selection View Go Run Terminal Help
Extension: Extension Pack for Java - nmp - Visual Studio Code
EXTENSIONS: MARKE...
sampleclass.java
Extension: Extension Pack for Java
extension for java
Extension Pa... 14.9M 4
Popular extensions for Java ...
Microsoft Install
Debugger fo... 16.4M 4
A lightweight Java debugge...
Microsoft Install
Project Mana... 14.7M 4
Manage Java projects in Vis...
Microsoft Install
Test Runner f... 15.3M 4
Run and debug JUnit or Tes...
Microsoft Install
Maven for Java 15.8M 4
Manage Maven projects, ex...
Microsoft Install
Language S... 18.3M 3.5
Java Linting, Intellisense, for...
Red Hat Install
C/C++ Extensi... 9.5M 4
Popular extensions for C++ ...
Microsoft Install
[Deprecated] ... 9.6M 4
Debug your JavaScript code...
Microsoft
Spring Initial... 1.8M 3.5
Extension Pack for Java v0.25.1 Preview
Microsoft 14,927,717 (52)
Install
Details Feature Contributions Changelog
Extension Pack (6)

Popular extensions for Java development that provides Java IntelliSense, debugging, testing, Maven/Gradle support, project ma...

#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.



```
args: java.lang.String[0] (id=512)
```

```
Thread [Attach Listener]      RUNNING
Thread [Signal Dispatcher]    RUNNING
Thread [Finalizer]            RUNNING
Thread [Reference Handler]    RUNNING
Thread [main]                 PAUSED ON BREAKPOINT
  main        DemoApplication.java 33
```

```
Uncaught Exceptions
Caught Exceptions
DemoApplication.java  src\main\java\c..  33
```

```java
    @Bean
    public Converter<String, Message> messageConverter() {
        return new Converter<String, Message>() {
            @Override
            public Message convert(String id) {
                return messageRepository.findOne(id);
            }
        };
    }

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
```

```
reak when expression evaluates to true. 'Enter' to accept, 'esc…
```

```java
    }
}
```

Introduction- Problem Solving, Debugging

Problem Statement:- Swapping of Two Numbers

Different methods of solving the problem

1. Swapping Two Numbers Using a Temporary Variable
2. Swapping Two Numbers Using Arithmetic Operators + and –
3. Swapping Two Numbers Using Arithmetic Operators x and /
4. Swapping Two Numbers Using Bitwise XOR operator ^

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 Solution

Time 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 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.

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 $c > 0$, n_0 such that $0 \leq f(n) \leq c*g(n)$ for $n \geq n_0$. **Ex**:- Searching in an unsorted list: O(n).	express the best-case scenario $f(n) = \Omega(g(n))$ if ∃ constants $c > 0$, n_0 such that $0 \leq c*g(n) \leq f(n)$ for $n \geq n_0$. **Ex**:- Inserting an element in a sorted array: Ω(1).	Express both worst and best cases $f(n) = \Theta(g(n))$ if ∃ constants $c_1, c_2 > 0$, n_0 such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$. **Ex**:- Linear search in a sorted array, where the element is always in the middle: Θ(n).

<u>General Rules for calculating Time Complesity:-</u>

<u>Rule1:-</u> The running time of for loop is the running time of the statements in the for loop n^2

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

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

<u>Rule2:-</u> 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++){                  ------n+1
       for(j=0;j<n;j++){           ------n*(n+1)
           c[i][j]=a[i][j]+b[i][j];  ------n*n
        }
}                                  Total is 2n^2+2n+1

                                   Ignore the lower exponents & constants
                                   So the time complexity is O(n^2)
```

<u>Rule3:-</u> consecutive statements, the running time is maximum value.

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

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

```
Ex:-
if(n<0)              ----1
   return n;         ----1
else{                ----1
for(i=0;i<n;i++)     -----n+1
     s=s+i;          -----n
return s;            -----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)

Type	Time Complexity (Big-O)	Space Complexity (Big-O)	Description
Constant	$O(1)$	$O(1)$	The algorithm/operation takes the same amount of time/space, regardless of input size.
Logarithmic	$O(\log n)$	$O(\log n)$	The time/space grows logarithmically with the input size (e.g., binary search).
Linear	$O(n)$	$O(n)$	The time/space grows directly proportional to the input size (e.g., iterating over an array).
Linearithmic	$O(n \log n)$	$O(n \log n)$	The time/space grows as the product of input size and its logarithm (e.g., merge sort).
Quadratic	$O(n^2)$	$O(n^2)$	The time/space grows proportional to the square of the input size (e.g., nested loops).
Cubic	$O(n^3)$	$O(n^3)$	The time/space grows proportional to the cube of the input size (e.g., matrix multiplication).
Exponential	$O(2^n)$	$O(2^n)$	The time/space doubles with every additional unit of input (e.g., recursive problems without optimization).
Factorial	$O(n!)$	$O(n!)$	The time/space grows factorially with the input size (e.g., permutations generation).
Polynomial	$O(n^k)$ $(k > 3)$	$O(n^k)$	The time/space grows as a polynomial function of input size (e.g., certain brute-force algorithms).
Log-Linear	$O((\log n)^n)$	$O((\log n)^n)$	A less common complexity, appearing in specialized problems.

Practice Programs

Addition of two numbers

Program to accept an integer, a floating-point number, and a character