Java Learning Center

No.1 In Java Training & placement

DSA

Module 1

Introduction to DSA

Author

Srinivas Dande

Java Learning Center

1. Data types

- Data type represents:
  - Type of data you want to use
  - Amount of memory allocation required for your data.

There are two types of data types

1) Primitive Data Types
2) User Defined Data Types

Primitive Data Types

- There are 8 primitive types available in Java.
- 8 keywords are defined to represent 8 primitive data types.

- **Following are primitive data types:**

boolean	byte	char	short
int	long	float	double

User-Defined Data Types

- There are four types of User Defined Data types:
  - Class type
  - Interface type
  - Enum type (From JAVA 5)
  - Annotation type (From JAVA 5)

2. Variables

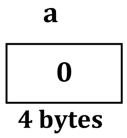
- Variable is the container which holds user data.
- Memory will be allocated for the variable while executing the program.
- Value of the variable can be changed any number of times during the program execution.

Syntax:

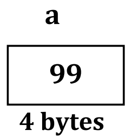
<Data type> <varName>;
<Data type> <varName> = <value>;

Ex:

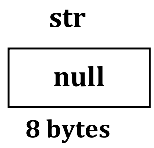
int a ;



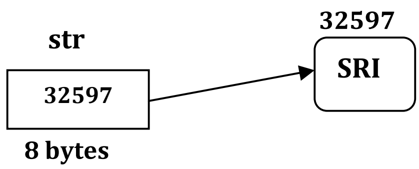
int a = 99;



String str;



String str="SRI";



Types of Variables

There are two types of variables based on data type used to declare the variable.

1) Primitive Variables
2) Reference Variables

2.1 Primitive Variables

- Variables declared with primitive data types are called as primitive variables.

  Ex:

  ```
  int a;
  int b = 99;
  ```

2.2 Reference Variables

- Variables declared with user defined data types are called as reference variables.

  Ex:

  ```
  String str1;
  String str2 = "JLC";
  ```

3. Data Structures

- Data Structure is a special way for storing and organizing data so that it can be used efficiently.
- Following are Types of Data Structures
    - Arrays
    - Linked Lists
    - Stacks
    - Queues
    - Trees
    - Graphs

  etc

- We can divide the Data Structures into two Types depending on the organization of the elements
    1. Linear Data Structures
    2. Non – Linear Data Structures

3.1 Linear Data Structures

- Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially.

  Ex:

  Linked Lists
  Stacks
  Queues

3.2 Non – Linear Data Structures

- Elements of this data structure are stored/accessed in a non-linear order.

  Ex:

  Trees
  Graphs

4. Abstract Data Types

- Define Data Type by Combining the data structures with their operations which is called as Abstract Data Type (ADT).
- An Abstract Data Type(ADT) is an abstraction of a Data Structure that provides only the interface to which the Data Structure must adhere.
- The interface does not give any specific details about how something should be implemented or in what programming language.
- ADT consists of two parts:
    1. Declaration of data
    2. Declaration of operations
- Following are Commonly used ADTs:
    - Linked Lists
    - Stacks
    - Queues
    - Priority Queues
    - Binary Trees
    - Dictionaries
    - Disjoint Sets
    - Hash Tables
    - Graphs
    - etc
- Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks.
- By the end of this Course, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

5. Exploring Algorithms

- An algorithm is the step-by-step unambiguous instructions to solve a given problem.

- There are two main criteria for judging the algorithms:
  - a) Correctness
  - b) Efficiency

a) Correctness

- Does the algorithm give solution to the problem

b) Efficiency

- How much resources (in terms of memory and time) does it take to execute

Examples:

Ex1: Algorithm for Adding Two Numbers

```
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */

Step 1: Start

Step 2: Declare 3 Variables a , b and sum.

Step 3: Read values for a and b

Step 4: Add a and b and assign the result to a variable sum.

Step 5: Display sum

Step 6: Stop
```

Ex2: Algorithm for Adding Two Numbers

```
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
```

Step 1: Start

Step 2: Declare 3 Variables a , b and sum.

```
int a;
int b;
int sum;
```

Step 3: Read values for a and b

```
a = 10;
b = 20;
```

Step 4: Add a and b and assign the result to a variable sum.

```
sum = a+b;
```

Step 5: Display sum

```
print(sum);
```

Step 6: Stop

Ex3: Algorithm for Printing Numbers from 1 to 10

```
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */

Step 1: Start

Step 2: Declare and Initialize the Variable i to 1
        int i=1;

Step 3: Display i
        print(i);

Step 4: Increment i by 1
        i = i + 1;

Step 5: Check if the value of i is less than or equal to 10.
        IF  i<=10 THEN
              GOTO Step 3
        otherwise
              GOTO Step 6

Step 6: Stop.
```

6. Algorithm Alalysis

- **Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.**

- **Goal of the analysis of algorithms is to compare algorithms (or solutions).**

How to Compare Algorithms:

1) Execution times?

- **Not a Good Idea because execution time will be changed from Machine to Machine or Programming Language to Programming Language**

2) Number of statements executed?

- **Not a Good Idea because Number of statements will be changed from Programming Language to Programming Language**

What is Best Solution:

Need the Way to Calculate Time Complexy and Space Complexity of an Algoruthm without depending on Machine or Programming Language or Developer

7. Rate of Growth Classes

- **Rate at which the running time of algorithm grows as the input size grows is called rate of growth.**

Commonly Used Rates of Growth

Complexity	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
3^n	Exponential
n!	Factorial

Relationship between Rates of Growth Classes:

$$1 < \log n < n < n * \log n < n^2 < n^3 < 2^n < 3^n < n!$$

Ex1: Find the Time Complexity

```
void sum() {
   int a= 10;
   int b=20;
   int sum = a + b;
   System.out.println(sum);
}
```

f(n) = O(1+ 1 + 1 + 1)
= O(4)
= **O(1)** => **Constant Time Complexity**

Ex2: Find the Time Complexity

```
int  sum(int n) {
   System.out.println("Begin");
   int sum= 0;

   for(int i= 1; i<=n; i++){
         sum = sum + i;
   }

   System.out.println("End");
   return sum;
}
```

f(n) = O(1+ 1 + n + n + 1 + 1)
= O(2n + 4)
= **O(n)** => **Linear Time Complexity**

8. Types of Analysis

- **To analyze the given algorithm,We need to know**
  - **With which inputs the algorithm takes less time**
  - **With which inputs the algorithm takes a long time.**

- **We represent the algorithm with multiple expressions:**
- **One for the case where it takes less time and another for the case where it takes more time.**

- **There are three types of analysis:**
  a) **Best Case**
  b) **Average Case**
  c) **Worst Case**

a) Best Case Analysis

- **Minimum amount of time the Algorithm takes to execute for the given Input.(Faster Execution time)**

b) Average Case Analysis

- **Provides a prediction about the running time of the algorithm.**
- **Run the algorithm many times, using many different inputs and compute the Avarage Running Time**

c) Worst Case Analysis

- **Maximum amount of time the Algorithm takes to execute for the given Input.(Slower Execution Time)**

9. Asymptotic Notations

- **Asymptotic Notations are the mathematical notations used to describe the Time Complexity or Space Complexity of an algorithm**

- **There are 3 such Notations:**
  a) **Big-O Notation (Upper Bound Function)**
  b) **Omega Notation (Lower Bound Function)**
  c) **Theta Notation (Tight Bound Function)**

10. Big-O Notation

- **Big-O Notation gives tight upper bound of the given function**
- **It is Represented as f(n) = O (g(n))**
  i.e Upper bound of f(n) is g(n) at Larger Values of n.

Function f(n) = O (g(n))
if there exists +ve constants C and n0
such that f(n) <= C * g(n) for all n>= n0

Ex1:

f(n) = 10 n + 7
10 n + 7 <= 11 n + 7n for all n>= 1
10 n + 7 <= 18 n for all n>= 1

n=0	-------	7 <= 0	// Not Allowed
n=1	---------	17 <=18	// Allowed
n=2		27 <= 36	// Allowed
n=3		37 <= 54	// Allowed

SO n>= 1 is Allowed.

Here f(n) = 10n+7
C = 18
g(n) = n
n0 = 1

So f(n) = O (n) => Linear Complexity

Ex2:

$f(n) = 6n^2 + 2n + 32n$

$6n^2 + 2n + 32n \le 6n^2 + 2n^2 + 32n^2$ for all $n >= 0$

$6n^2 + 2n + 32n \le 40\, n^2$ for all $n >= 0$

Here $f(n) = 6n^2 + 2n + 33n$

$C = 40$

$g(n) = n^2$

$n0 = 0$

n=0 --------- 0 <= 0 // Allowed
n=1 --------- 40 <=40 // Allowed
n=2 --------- 101 <= 160 // Allowed
SO n>= 0 is Allowed.

So $f(n) = O(n^2)$ => Quadratic Complexity

Ex3:

$f(n) = 3n^2 + 5n + 9$

$3n^2 + 5n + 9 \le 4n^2$ for all $n >= 7$

$3n^2 + 5n + 9 \le 4\, n^2$ for all $n >= 7$

Here $f(n) = 3n^2 + 5n + 9$

$C = 4$

$g(n) = n^2$

$n0 = 7$

n=0 --------- 9 <= 0 // Not Allowed
n=1 --------- 17 <=4 // Not Allowed
n=2 --------- 31 <= 16 // Not Allowed
....
n=6 --------- 147 <= 144 // Not Allowed
n=7 --------- 191 <= 196 // Allowed
SO n>= 7 is Allowed.

So $f(n) = O(n^2)$ => Quadratic Complexity

11. Omega Notation

- Omega Notation gives tight lower bound of the given function
- It is Represented as f(n) = Ω (g(n))

  i.e Lower bound of f(n) is g(n) at Larger Values of n.

Function f(n) = Ω (g(n))
if there exists +ve constants C and n0
such that **f(n) >= C * g(n) for all n>= n0**

Ex1:

f(n) = 3 n + 5

3 n + 5 >= 1 n   for all n>= 0

3 n + 5 >= 2 n   for all n>= 0

Here f(n) = 3n+5

C = 2

g(n) = n

n0 = 0

n=0 ------- 5 >= 0 // Allowed
n=1 --------- 8 >=2 // Allowed
n=2 --------- 11 >= 4 // Allowed
SO n>= 0 is Allowed.

So f(n) = Ω (n) => Linear Complexity

Ex2:

f(n) = $6n^2$ + 2n+ 32

$6n^2$ + 2n+ 32 >= $5n^2$ for all n>= 0

$6n^2$ + 2n+ 32 >= 5 n^2 for all n>= 0

Here f(n) = $6n^2$ + 2n+ 32

C = 5

g(n) = n^2

n0 = 0

n=0 --------- 32 >= 0 // Allowed
n=1 --------- 40 >=5 // Allowed
SO n>= 0 is Allowed.

So f(n) = Ω (n^2) => Quadratic Complexity

Ex3:

$f(n) = 3n^2 + 5n + 9$

$3n^2 + 5n + 9 >= 2n^2$ for all $n >= 7$

$3n^2 + 5n + 9 >= 2\ n^2$ for all $n >= 7$

n=0 --------- 9 >= 0 // Allowed
n=1 --------- 17 >=2 // Allowed
n=2 --------- 31 >= 8 // Allowed
SO n>= 0 is Allowed.

Here $f(n) = 3n^2 + 5n + 9$

$C = 2$

$g(n) = n^2$

$n0 = 7$

So $f(n) = \Omega (n^2)$ => Quadratic Complexity

12. Theta Notation

- **Theta Notation gives tight bound of the given function**
- **It is Represented as $f(n) = \Theta (g(n))$**
  i.e Exact bound of f(n) is g(n) at Larger Values of n.
- **Avarage Running time of an Algorithm is always between lower bound and upper bound.**

Function $f(n) = \Theta (g(n))$
if there exists +ve constants c1, c2 and n0
such that $c1 * g(n) <= f(n) <= c2 * g(n)$ for all $n >= n0$

Ex1:

f(n) = 5 n + 7

4 n <= 5 n + 7 <= 6 n   for all n>= 7

n=0 ---- 0<= 7 <= 0 // Not Allowed
n=1 ---- 4 <= 12<=6 // Not Allowed
n=2 ---- 8 <= 17 <= 12 // Not Allowed
n=3 ---- 12 <= 22 <= 18 // Not Allowed
..
n=6 ---- 24 <= 37 <= 36 // Not Allowed
n=7 ---- 28 <= 42 <= 42 // Not Allowed
SO n>= 7 is Allowed.

Here f(n) = 5n+7

C1 = 4

C2 = 6

g(n) = n

n0 = 7

So f(n) = Θ (n) => Linear Complexity

Ex2:

f(n) = $3n^2 + 5n + 9$

4 n^2 <= 3 n^2 + 5n + 9 <= 2 n^2 for all n>= (You Find)

n=0 --- 0<= 9 <= 0 // Not Allowed
n=1 ---- 4 <=17<=2 // Not Allowed
n=2 ---- 16 <= 31 <= 8 // Not Allowed
...
You Calculate n value

Here f(n) = $3n^2 + 5n + 9$

C1 = 4

C2 = 2

g(n) = n^2

n0 = (You Find)

So f(n) = Θ (n^2) => Quadratic Complexity

13. More Examples

Example 1:

```
void sum() {
   int a= 10;
   int b=20;
   int sum = a + b;
   System.out.println(sum);
}
```

f(n) = O(1+ 1 + 1 + 1)

= O(4)

= **O(1)** => **Constant Time Complexity**

Example 2:

```
int  sum(int n) {
   System.out.println("Begin");
   int sum= 0;

   for(int i= 1; i<=n; i++){
         sum = sum + i;
   }

   System.out.println("End");
   return sum;
}
```

f(n) = O(1+ 1 + n + n + 1 + 1)

= O(2n + 4)

= **O(n)** => **Linear Time Complexity**

Example 3:

```
void  show(int n) {
   System.out.println("Begin");

   for(int i= 1; i<=n; i++){
         System.out.println(i);
   }

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
         System.out.println(i);
   }

   System.out.println("End");
}
```

$f(n) = O(1 + n + 1 + n + 1)$

$= O(2n + 3)$

$= O(n)$ => Linear Time Complexity

Example 4:

```
void  show(int n) {
   System.out.println("Begin");

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
   for(int j= 1;ji<=n; j++){
         System.out.println(i+"\t"+j);
   }

   }
   System.out.println("OK");

   System.out.println("End");
}
```

f(n) = O(1+ 1 + (n * n) + 1 + 1)

= $O(n^2 + 4)$

= **$O(n^2)$ => Quadratic Time Complexity**

Example 5:

```
void  show(int n) {
   System.out.println("Begin");

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
         System.out.println(i);
   }

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
   for(int j= 1;ji<=n; j++){
         System.out.println(i+"\t"+j);
   }

   }
   System.out.println("OK");

   System.out.println("End");
}
```

$f(n) = O(1+ 1 + n+ 1 + (n * n) + 1 + 1)$

$= O(n^2 + n+ 5)$

$= \mathbf{O(n^2)}$ **=> Quadratic Time Complexity**

Example 6:

```
void  show(int n) {
   System.out.println("Begin");

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
         System.out.println(i);
   }

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
   for(int j= 1;j<=n; j++){
   for(int k= 1;k<=n; k++){

         System.out.println(i+"\t"+j+"\t"+k);
   }

    }

   }

   System.out.println("OK");

   System.out.println("End");
}
```

$f(n) = O(1+ 1 + n+ 1 + (n * n* n) + 1 + 1)$

$= O(n^3 + n+ 5)$

$= \mathbf{O(n^3)}$ **=> Cubic Time Complexity**

Example 7:

```
void  show(int m,int n) {
   System.out.println("Begin");

   System.out.println("OK");

   for(int i= 1; i<=n; i++){
         System.out.println(i);
   }

   System.out.println("OK");

   for(int i= 1; i<=m; i++){
         System.out.println(i);
   }

   System.out.println("OK");

   System.out.println("End");
}
```

f(n) = O(1+ 1 + n+ 1 m + 1 + 1)

= O(m + n+ 5)

= O(m+n) => Linear Time Complexity

14. Space Complexity

- **Space complexity refers to the total amount of memory space used by an algorithm including the space of input values.**
- **Auxiliary space is simply extra or temporary space, and it is not the same as space complexity.**

Space Complexity = Auxiliary space + space use by input values

- **The best algorithm/program should have a low level of space complexity.**
- **The less space required, the faster it executes.**

Example 1:

```
void sum(int a, int b) {
   int sum = a + b;
   System.out.println(sum);
}
```

f(n) = O(1+ 1 + 1)

= O(3)

= **O(1) => Constant Time Complexity**

Example 2:

```
int  sum(int []arr) {
   int sum= 0;

   for(int i= 0; i<arr.length; i++){
         sum = sum + arr[i];
   }
   return sum;
}
```

f(n) = O(n+ 1)

= O(n + 1)

= **O(n)** => **Linear Time Complexity**

Example 3:

```
void  show(int arr[]) {

   int myarr [] = new int[arr.length];

   for(int i= 0; i<n; i++){
    myarr[i]= arr[i] * arr[i];
   }
   }
```

f(n) = O(n+ n)

= O(2n)

= **O(n)** => **Linear Time Complexity**