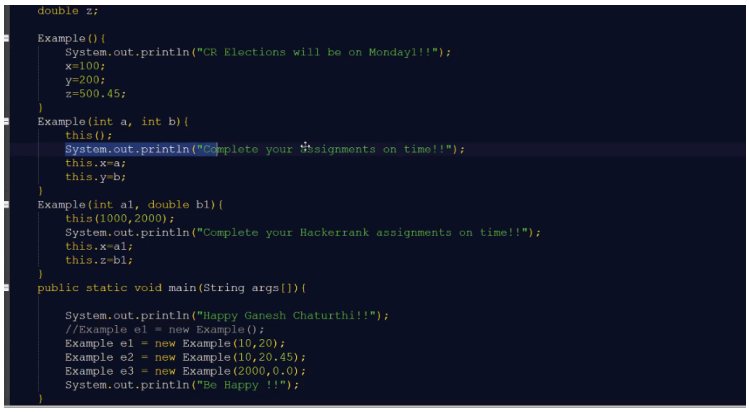
Day_5_OOPJ_Sanket_Shalukar

Saturday, August 30, 2025 9:42 AM

Constructor!



```
double z;

Example(){
    System.out.println("CR Elections will be on Monday!!");
    x=100;
    y=200;
    z=500.45;
}
Example(int a, int b){
    this();
    System.out.println("Complete your assignments on time!!");
    this.x=a;
    this.y=b;
}
Example(int a1, double b1){
    this(1000,2000);
    System.out.println("Complete your Hackerrank assignments on time!!");
    this.x=a1;
    this.z=b1;
}
public static void main(String args[]){

    System.out.println("Happy Ganesh Chaturthi!!");
    //Example e1 = new Example();
    Example e1 = new Example(10,20);
    Example e2 = new Example(10,20.45);
    Example e3 = new Example(2000,0.0);
    System.out.println("Be Happy !!");
}
```

Below is a code explanation for the above snippet!

1. Instance Variables

```
int x, y;
double z;
```

- These are fields of the class `Example`.

2. Default Constructor

```
Example(){
    System.out.println("CR Elections will be on Monday!!");
    x = 100;
    y = 200;
    z = 500.45;
}
```

- This constructor initializes default values.
- Prints `"CR Elections will be on Monday!!"` whenever called.

3. Constructor with Two Integers

```
Example(int a, int b){
    this();  // Calls default constructor
    System.out.println("Complete your assignments on time!!");
    this.x = a;
    this.y = b;
}
```

- Calls the **default constructor** first (so its message will print).
- Prints `"Complete your assignments on time!!"`.
- Then assigns values to x and y.

4. Constructor with int and double

```
Example(int a1, double b1){
    this(1000,2000); // Calls constructor with two ints
    System.out.println("Complete your Hackerrank assignments on time!!");
    this.x = a1;
    this.z = b1;
}
```

- Calls the constructor `(int, int)` first (which itself calls default constructor).

- Prints "Complete your Hackerrank assignments on time!!".
- Assigns values to x and z.

5. Main Method

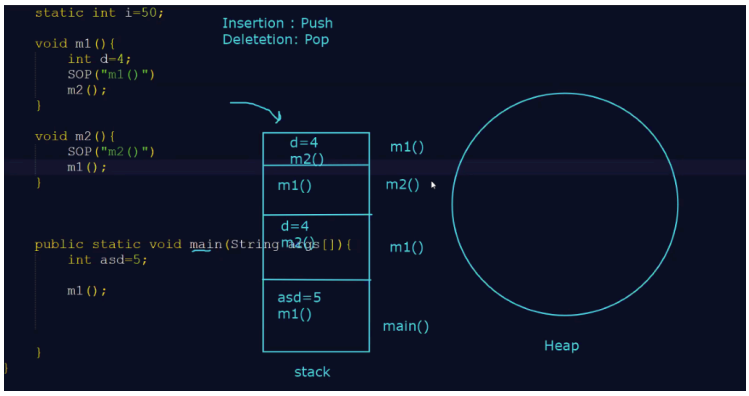
```
public static void main(String args[]){
    System.out.println("Happy Ganesh Chaturthi!!");
    //Example e1 = new Example();
    Example e1 = new Example(10,20);
    Example e2 = new Example(10,20.45);
    Example e3 = new Example(2000,0.0);
    System.out.println("Be Happy !!");
}
```

- Prints "Happy Ganesh Chaturthi!!".
- Creates three objects:
- e1 → calls (int,int) constructor → also calls default constructor.
- e2 → calls (int,double) constructor → also calls (int,int) → also calls default constructor.
- e3 → same as above.
- Finally prints "Be Happy !!".

Execution Flow (Output)

When running this program, the output will be:

```
Happy Ganesh Chaturthi!!
CR Elections will be on Monday!!
Complete your assignments on time!!
CR Elections will be on Monday!!
Complete your assignments on time!!
Complete your Hackerrank assignments on time!!
CR Elections will be on Monday!!
Complete your assignments on time!!
Complete your Hackerrank assignments on time!!
Be Happy !!
```



What is **Stack Overflow**?

- In Java, **method calls** are managed using the **stack memory**.
- Each time a method is called, a **stack frame** is created (containing local variables, return address, etc.).
- When the method finishes, its frame is removed (popped).

👉 If methods keep calling each other (or themselves) **without termination**, new stack frames keep getting created, and at some point, the **stack memory is exhausted** → This throws **`StackOverflowError`**.

🔹 Code in the Image

```
static int i=50;

void m1(){
    int d=4;
    SOP("m1()");
    m2();
```

```
}

void m2(){
    SOP("m2()");
    m1();
}

public static void main(String args[]){
    int asd=5;
    m1();
}
```

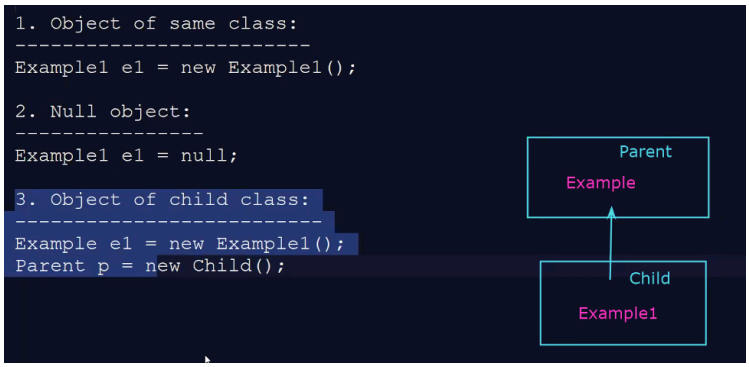
🔹 Execution Flow

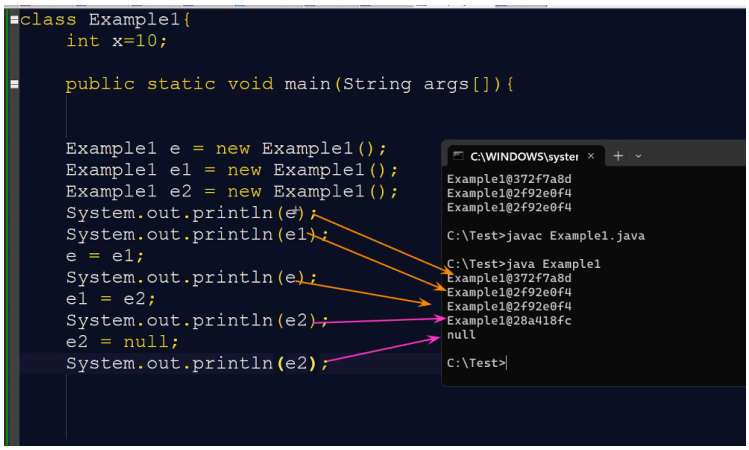
1. `main()` starts → local var `asd=5` is stored in stack.
2. `main()` calls `m1()` → new stack frame for `m1`.
3. Inside `m1`, local var `d=4`, prints `"m1()"` → then calls `m2()`.
4. Now `m2()` stack frame created → prints `"m2()"` → calls `m1()`.
5. Again `m1()` calls `m2()`, and so on… **infinite recursion**.

🔹 Stack Visualization (like your diagram)

```
main()
└─ m1()
     └─ m2()
          └─ m1()
               └─ m2()
                    └─ m1()
                         ...
```

This keeps growing until **stack memory is full**, then Java throws:





Stack Heap and Method Area :

Stack (JVM Stack)

- Created **per thread**.
- Stores **method calls, local variables, and references**.
- When a method ends, its stack frame is removed.

Heap

- Shared memory for the whole program.
- Stores **objects and arrays**.
- Garbage Collector cleans unused objects.

Method Area

- Part of JVM memory for **class-level data**.
- Stores **class info, static variables, constants, and methods' bytecode**.

Access Modifiers in Java :

- **public** → Accessible from anywhere (inside the class, outside the class, other packages).
- **private** → Accessible only inside the same class. No one else can access it.
- **protected** → Accessible inside the same package and also in child classes (subclasses).
- **default (no modifier)** → Accessible only inside the same package.

Date:

- Represent some specifict instancet in time.
- SimpleFormated: 'SimpleDateFormat' for formatting
- Constructor:
  - Date () : curretn date and time
  - Date (long ms) : specific time
- Common Methods () :
- long getTime () :milliseconds
- boolean before (Date d) / after (Date d) : compare dates
- int compareTo (Date d) : compare two dates
- toString () : human-readable string/ normal string

Date Class (java.util.Date)

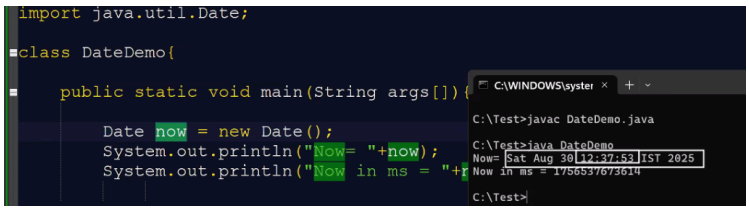
- The Date class represents a specific moment in time.
- By default, if you create a Date object without arguments, it gives the current date and time.
- You can also create a Date object using milliseconds since 1 Jan 1970 (Epoch time).

Common Methods in Date

- **getTime()** → returns the time in milliseconds since 1 Jan 1970.
- **before(Date d)** → returns true if the calling date is before the given date.
- **after(Date d)** → returns true if the calling date is after the given date.
- **compareTo(Date d)** → compares two dates and returns:
- 0 if both are equal,
- positive value if the calling date is after the given date,
- negative value if the calling date is before the given date.

SimpleDateFormat

- Used to format or display dates in different styles.
- Example formats:
- "dd/MM/yyyy" → 30/08/2025
- "MM-dd-yyyy" → 08-30-2025
- "E, MMM dd yyyy" → Sat, Aug 30 2025
- "hh:mm:ss a" → 10:25:45 AM



Calender :
Abstract class for date manipulation (add, sub, day, diff)
Filel based access: Year, Month, Day_of_Month,…
Local time zone.

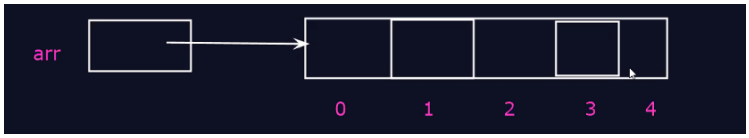
Calander cal =

```java
import java.util.*;
import java.text.*;

class DateDemo {
    Run | Debug
    public static void main(String args[]) {
        Calendar cal = Calendar.getInstance();
        Date d1 = new Date ();
        SimpleDateFormat s1 = new SimpleDateFormat (pattern:"MM-dd-yyyy HH:mm:ss");
        System.out.println("Formatted date : " + s1.format (d1));
            System.out.println("Year = " + cal.get(Calendar.YEAR));
            System.out.println("Month = " + (cal.get(Calendar.MONTH) + 1)); // Months are 0-based
            System.out.println("Day = " + cal.get(Calendar.DAY_OF_MONTH));
            System.out.println("Now in ms = " + cal.getTimeInMillis());
    }
}
```

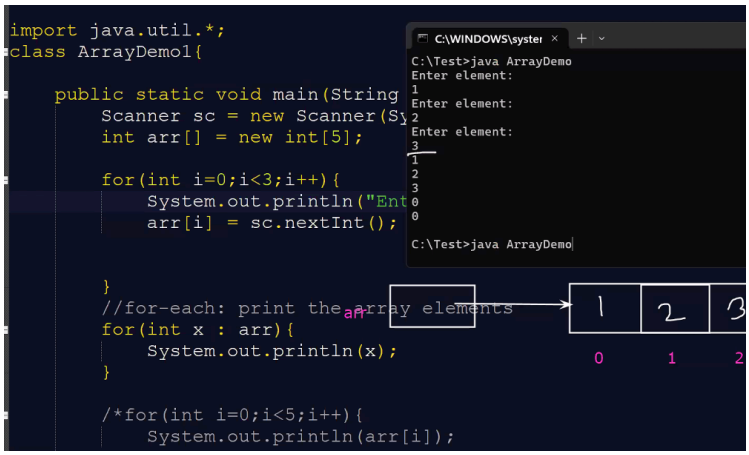
Arrays :

It is a collection of homogeneous elements stored in sequential memory locations and accessed using indexes.
Homogenous elements, Indexed collection, sequential




```java
int[][] a; //2D
int []a[]; //allowed
int [][][] b; //3D
int []a, b;
int[] a[], b;
int[][] a, b;
int[][a,b];

int arr[] = new int[5];
```



6. **Array creation**

`int arr[] = new int[5];`

- Creates an integer array of size **5**.
- All elements are initialized to **0** by default.

7. **Input loop**
   `for(int i=0; i<3; i++) { System.out.println("Enter element:"); arr[i] = sc.nextInt(); }`
- Runs **3 times** (`i = 0,1,2`).
- Takes 3 inputs from the user and stores them in the first 3 positions of the array.
- The remaining 2 positions stay **0** (default value).

8. **For-each loop**
   `for(int x : arr) { System.out.println(x); }`
- Prints all 5 elements of the array.
- Even though only 3 values were entered, it still prints all 5 (last two are `0`).

Execution Flow (Example)

User enters: `1, 2, 3`

- Array looks like: `[1, 2, 3, 0, 0]`
- Output:

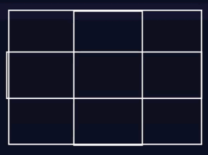
```
1
2
3
0
0
```

```java
import java.util.*;
class ArrayDemo1{

    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int arr[] = new int[5];
        for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            System.out.println("Enter element:");
            arr[i][ = sc.nextInt();

        }
        }
        //for-each: print the array elements
        for(int x : arr){
            System.out.println(x);
        }

        /*for(int i=0;i<arr.length ;i++){
            System.out.println(arr[i]);
```



```java
int arr[] = new int[-5];//Error: Exception : NegativeArraySizeException

Array Initialize:
---------------
int arr[] = new int[5];

int[] num={1,2,3,4,5,6};

Traversing:
-----------
for(int i=0;i<num.length;i++){
    SOP(num[i]);
}

1. Read matrix of 3X3
2. Read 2 matrix 3X3
3. Addition , print resultant matrix
4. Substraction , print resultant matrix
5. Multiplication , print resultant matrix
6. Transpose , print resultant matrix
```