- In earlier days the computer's memory was occupied with only one program. After completion of that program it was possible to execute another program it is known as **UNI programming**

- ***Drawbacks of UNI programming***
  - Most of the times memory will be wasted
  - CPU utilization will be reduced because only one program is allowed to execute at a time

To overcome above problems Multiprogramming was introduced

- Executing several programs simultaneously is called **multiprogramming**
- All these programs are controlled by the **CPU scheduler**
- **CPU scheduler** will allocate a particular time period for each and every program so that CPU resources will not be wasted

```
1 struct Student{
2     int a;
3     float f;
4     char name[10];
5 }
6
7 union Student{
8     int a;
9     float f;
10    char name[10];
11 }
```

4+4+10=18

4+4+10=10

Thread

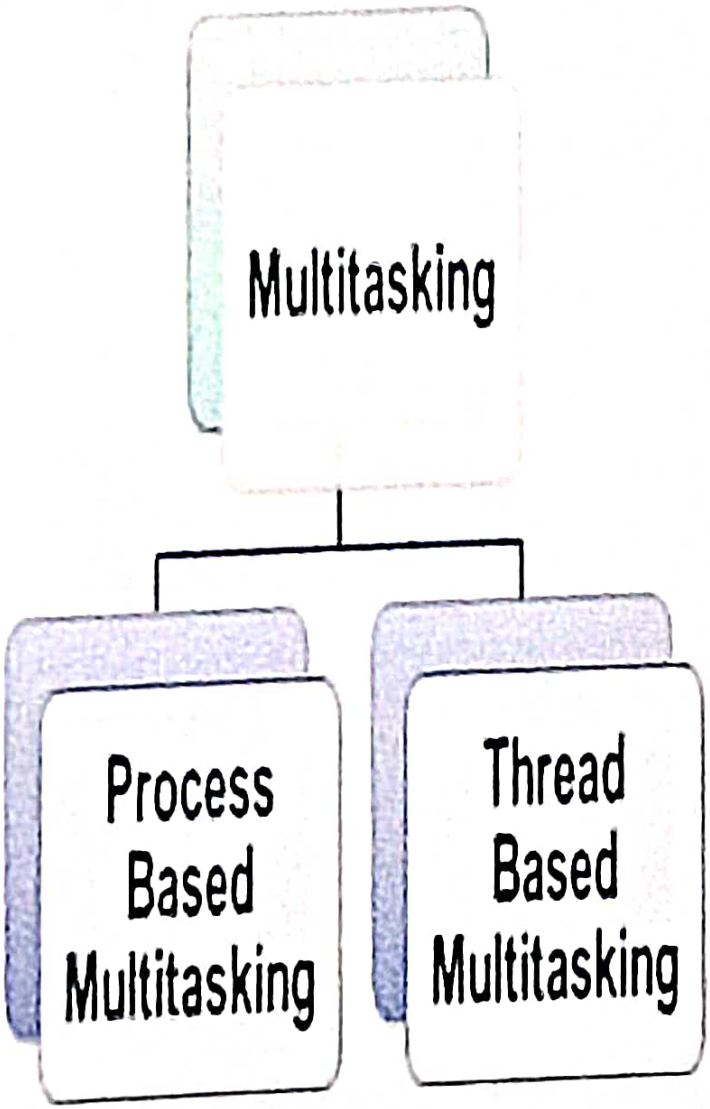
- The independent execution is called as **thread**
- **Thread** is the separate path of sequential execution
- The **thread** is light weighted process because whenever we are creating **thread** it is not occupying the separate memory, it uses the same shared memory, sharing memory means it is not consuming more memory

Multithreading

- Executing more than one thread a time is called **multithreading**
- **Multithreading** is process of executing more than one thread at a time simultaneously

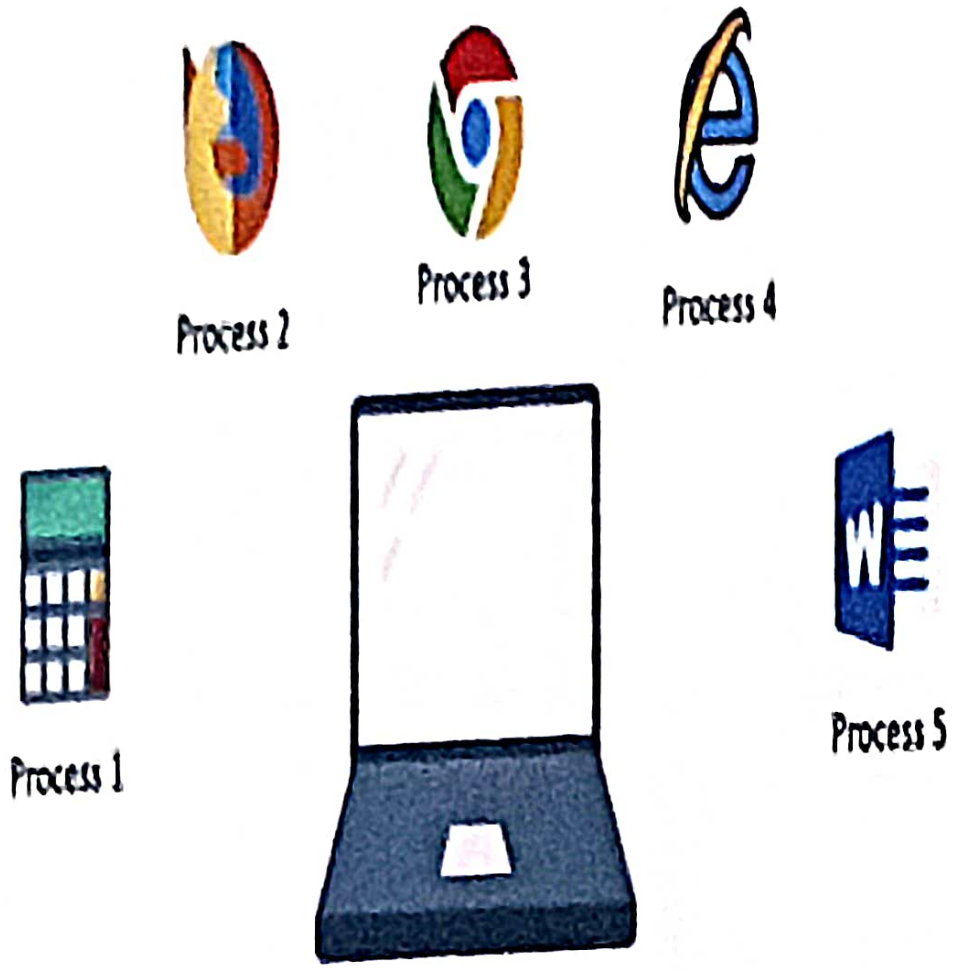
Multithreading this is the way to achieve multitasking

There are two ways to achieve multitasking



Process Based Multitasking

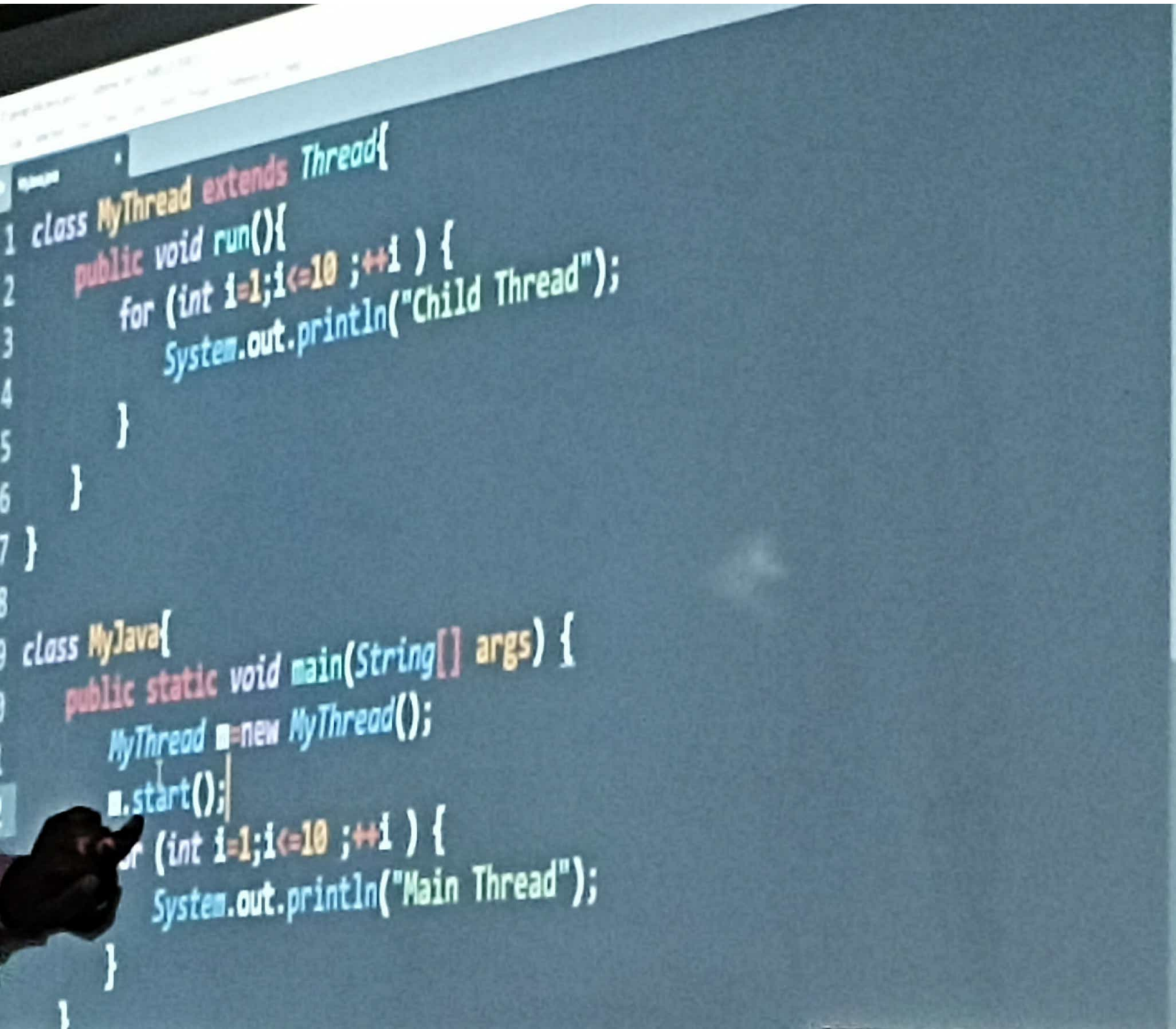
We knows that executing multiple task is the multitasking and when each independent task is a separate independent process then it is called as process based multitasking



Communic ation	Communication between two processes is expensive and limited	Communication between two threads is less expensive as compared to process
Switching	Context switching from one process to another process is very expensive	Context switching from one thread to another thread is less expensive as compared to process
Control	It is not under control of Java	It is the under control of Java
Substitute	Process is heavyweight task	Thread is lightweight task.

Application Areas of Multithreading

- Development of *video games*
- Development of *animation*
- Implementing *multimedia graphics*
- Development of *client-server type application*



```java
class MyThread extends Thread{
    public void run(){
        for (int i=1;i<=10 ;++i ) {
            System.out.println("Child Thread");

        }
    }
}

class MyJava{
    public static void main(String[] args) {
        MyThread m=new MyThread();
        m.start();
        for (int i=1;i<=10 ;++i ) {
            System.out.println("Main Thread");
        }
    }
}
```

```
class MyThread extends Thread{
    public void run(){
        for (int i=1;i<=10 ;++i ) {
            System.out.println("Child Thread");
        }
    }
}

class MyJava{
    public static void main(String[] args) {
        MyThread m=new MyThread();
        m.start();
        for (int i=1;i<=10 ;++i ) {
            System.out.println("Main Thread");
        }
    }
}
```

```java
class MyThread implements Runnable{
    public void run(){
        for (int i=1;i<=10 ;++i ) {
            System.out.println("Child Thread");
        }
    }
}

class MyJava{
    public static void main(String[] args) {
        MyThread m=new MyThread();
        Thread t=new Thread(m);
        t.start();
        for (int i=1;i<=10 ;++i ) {
            System.out.println("Main Thread");
        }
    }
}
```

Difference between t.start() and t.run()

- In the case of t.start(), Thread class start() is executed and a new thread will be created that is responsible for the execution of run() method.
- But in the case of t.run() method, no new thread will be created and the run() is executed like a normal method call by the main thread.

Thread Scheduler

- Execution of multiple threads on a single CPU in some order is called **thread scheduling**
- **Thread scheduler** is the part of JVM, if multiple Threads are waiting to get a chance of execution then in which order thread will be executed it is decided by the **thread scheduler**
- Whenever a **scheduler** comes to multithreading there is no guarantee for exact output because there is no guarantees which algorithm will be followed by JVM

Thread scheduler uses following algorithms

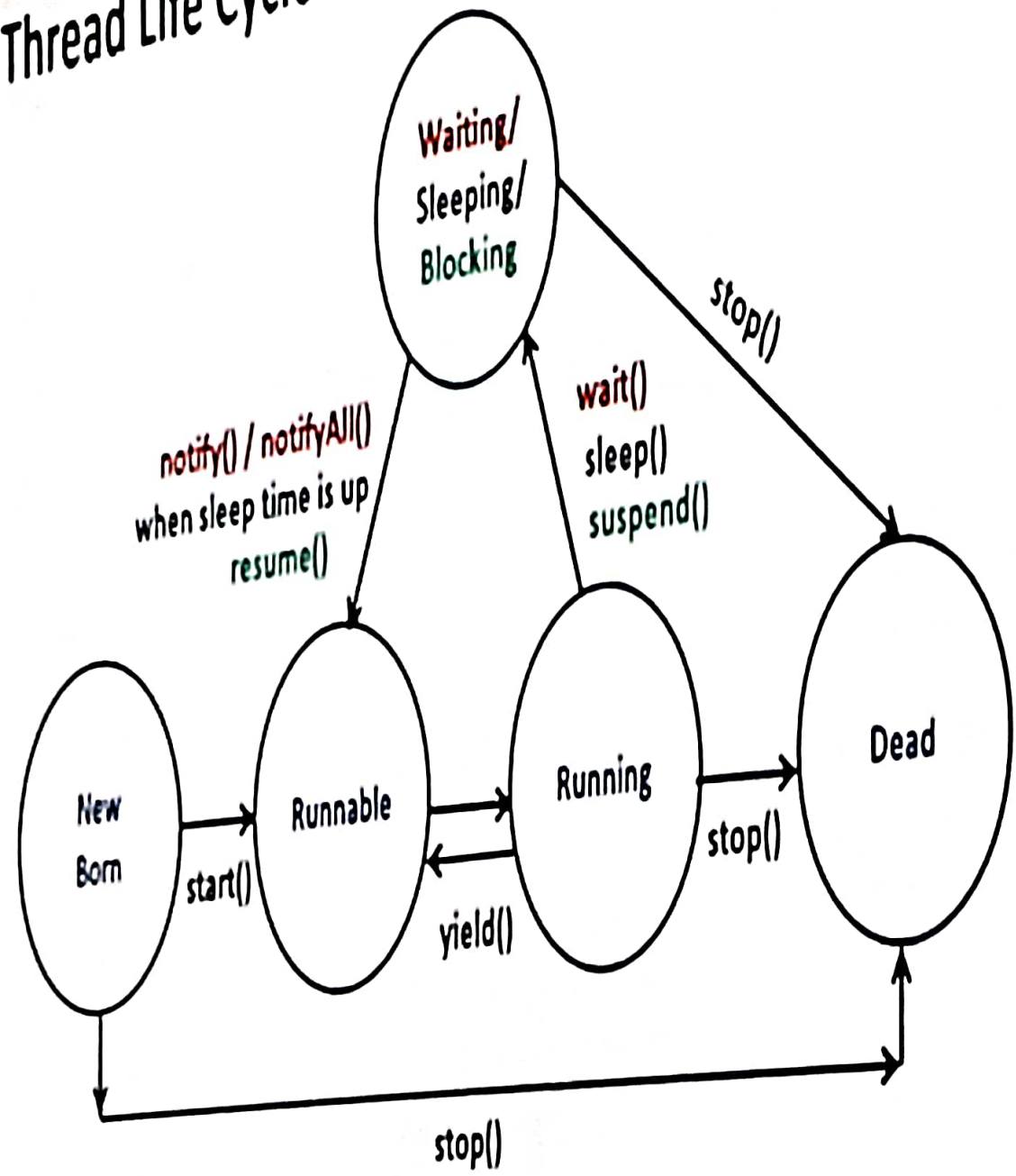
First come first serve

Round- robin

Shortest job first

By Ashish Gadpayle Sir

Thread Life Cycle



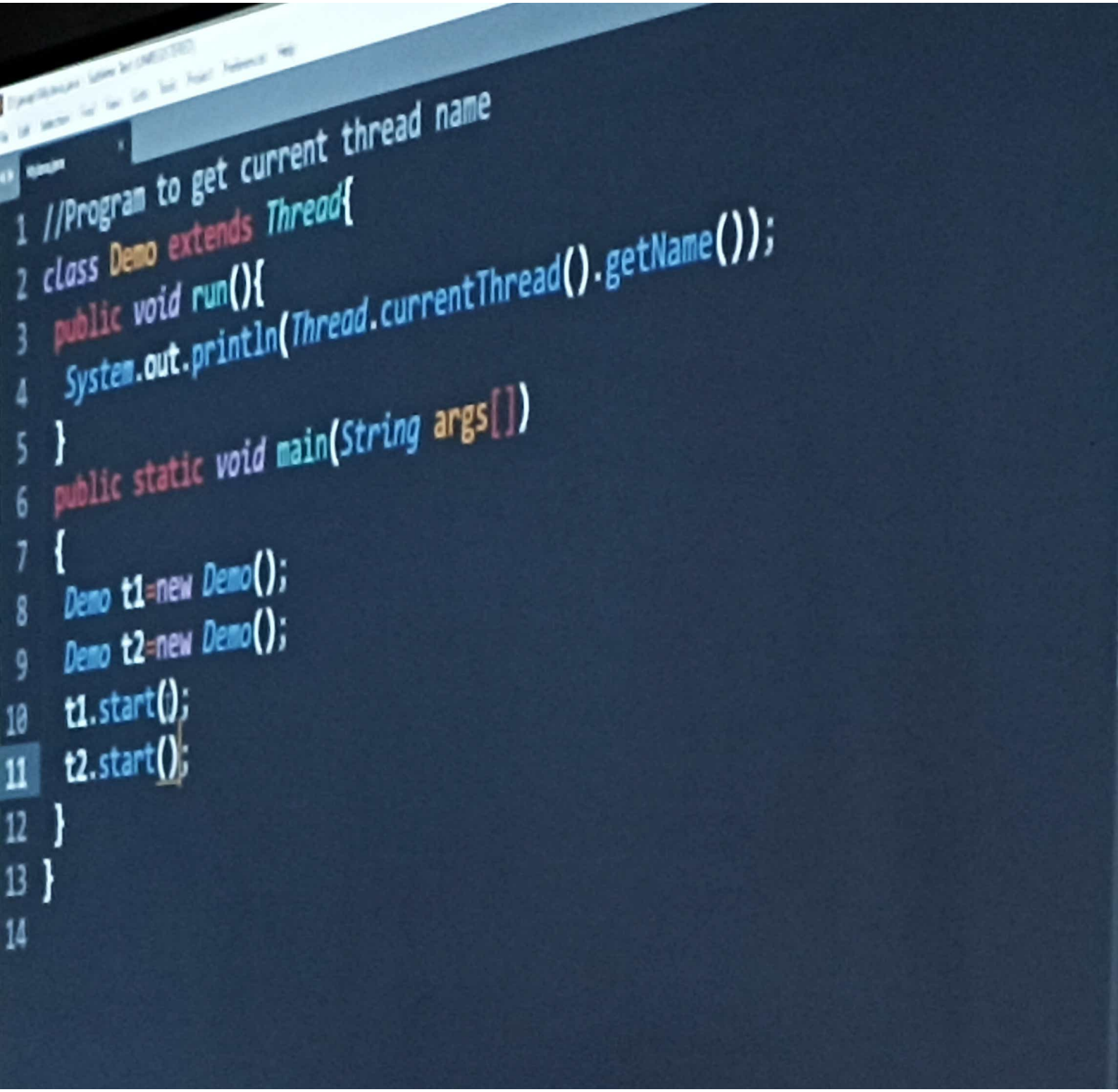
Setting and Getting name of thread

- Every thread in java has some name, It may be the default name or generated by JVM or customize a name provided by the programmer.
- We can set and get the name of thread by using **setName()** and **getName()** respectively

```
public final String getName()
public final void setName(String name)
```

Ex:

```
//Program to get current thread name
class Demo extends Thread{
public void run(){
 System.out.println(Thread.currentThread().getName());
}
public static void main(String args[])
{
 Demo t1=new Demo();
 Demo t2=new Demo();
 t1.start();
 t2.start();
}
}
```



```java
//Program to get current thread name
class Demo extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 public static void main(String args[])
 {
  Demo t1=new Demo();
  Demo t2=new Demo();
  t1.start();
  t2.start();
 }
}
```

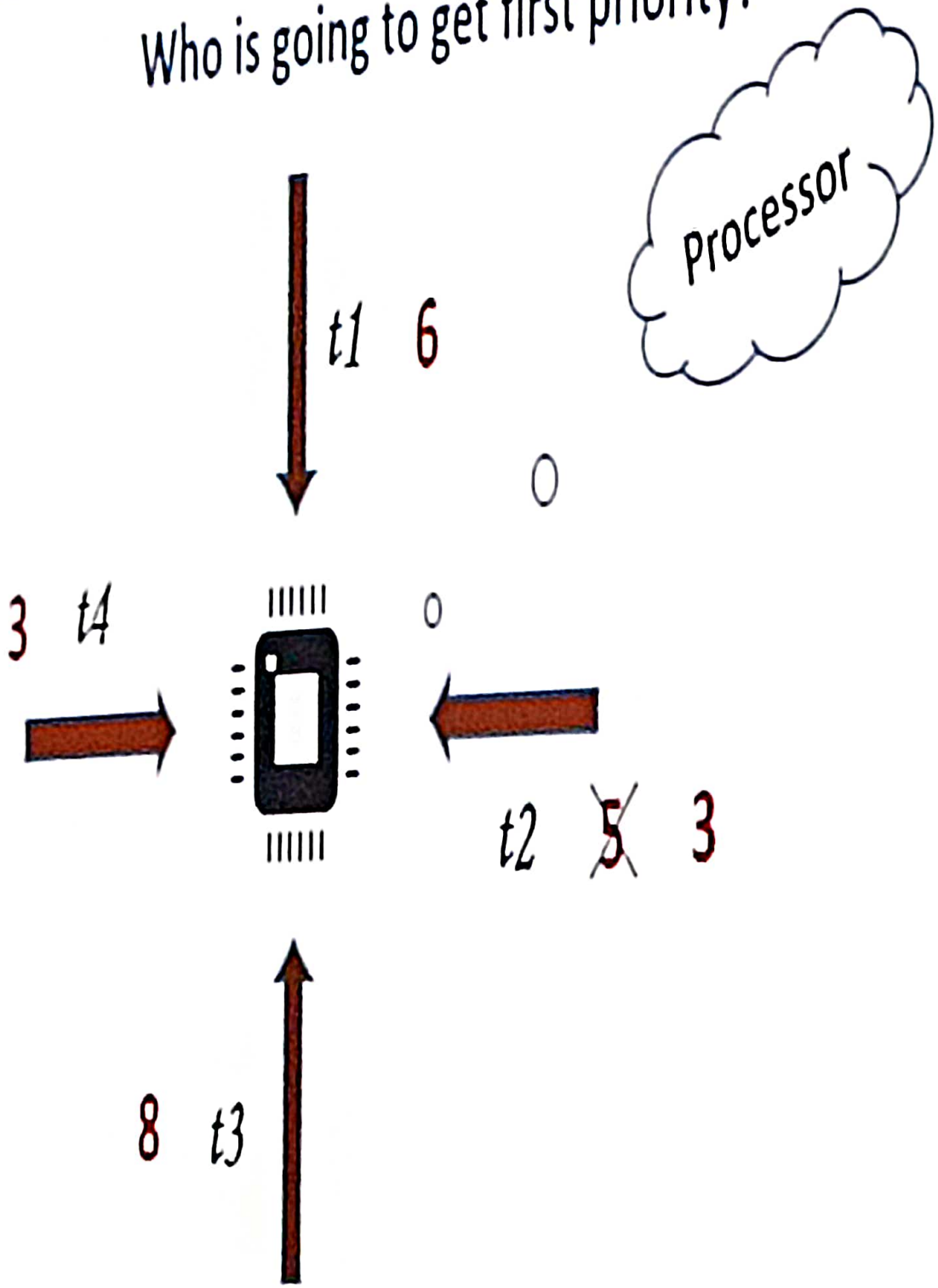
```java
//Program to set name of thread
class SetName extends Thread{
  public void run(){
   System.out.println("run() method");
  }
 public static void main(String args[]){
  SetName t1=new SetName();
  SetName t2=new SetName();
  System.out.println("Name of t1: "+t1.getName());
  System.out.println("Name of t2: "+t2.getName());

  t1.start();
  t2.start();
  t1.setName("Ashish");
  System.out.println("After changing name of t1: "+t1.getName());
  System.out.println("After changing name of t2: "+t2.getName());
 }
}

```

Thread Priorities

- Every thread in java has some priority either **default priority** or **custom priority**
- Priority that helps to the thread scheduler to determine the order in which threads are scheduled
- Higher priority thread will run before than lower priority thread
- Priority range is between **1 to 10**, where 10 being the **highest priority**, 1 being the **lowest priority** and 5 being the **default priority**
- If you specify a priority that is out of range then an **IllegalArgumentException** exception will be thrown

Thread Priorities

Who is going to get first priority?



Thread Priorities

Thread priorities are static member variables of java.lang.Thread class

They are

- Thread.MIN_PRIORITY i.e. 1
- Thread.NORM_PRIORITY i.e. 5
- Thread.MAX_PRIORITY i.e. 10

The priority of the main() thread is Thread.NORM_PRIORITY i.e. 5

Thread class has following two methods to get and set priority

public final int getPriority()

public final void setPriority(int priority)

Ex:

```
//Program to demonstrate getPriority() and setPriority()
class ThreadDemo extends Thread
{
   public void run()
   {
      System.out.println("Inside run method");
   }
   public static void main(String[]args)
   {
      ThreadDemo t1 = new ThreadDemo();
      ThreadDemo t2 = new ThreadDemo();
      ThreadDemo t3 = new ThreadDemo();
      System.out.println("t1 thread priority : " + t1.getPriority()); // Default 5
      System.out.println("t2 thread priority : " + t2.getPriority()); // Default 5
      System.out.println("t3 thread priority : " + t3.getPriority()); // Default 5
      t1.setPriority(2);
      t2.setPriority(5);
      t3.setPriority(8);

      // t3.setPriority(21); will throw IllegalArgumentException
      System.out.println("t1 thread priority : " + t1.getPriority()); //2
```

```
//Program to demonstrate getPriority() and setPriority()
class ThreadDemo extends Thread

  public void run()
  {
     System.out.println("Inside run method");

  }
  public static void main(String[]args)
  {
     ThreadDemo t1 = new ThreadDemo();
     ThreadDemo t2 = new ThreadDemo();
     ThreadDemo t3 = new ThreadDemo();
     System.out.println("t1 thread priority : " + t1.getPriority()); // Default 5
     System.out.println("t2 thread priority : " + t2.getPriority()); // Default 5
     System.out.println("t3 thread priority : " + t3.getPriority()); // Default 5
     t1.setPriority(2);
     t2.setPriority(5);
     t3.setPriority(8);

     // t3.setPriority(21); will throw IllegalArgumentException
     System.out.println("t1 thread priority : " + t1.getPriority()); //2
```

```
        System.out.println("t2 thread priority : " + t2.getPriority()); // Default 5
        System.out.println("t3 thread priority : " + t3.getPriority()); // Default 5
        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw IllegalArgumentException
        System.out.println("t1 thread priority : " + t1.getPriority()); //2
        System.out.println("t2 thread priority : " + t2.getPriority()); //5
        System.out.println("t3 thread priority : " + t3.getPriority()); //8

        // Main thread
        System.out.print(Thread.currentThread().getName());
        System.out.println("Main thread priority : "+ Thread.currentThread().getPriority(

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println("Main thread priority : "+ Thread.currentThread().getPriority(
    }
}
```

Preventing thread from thread execution

We can prevent a thread by using following methods in java

- yield()
- join()
- sleep()

yield() method

- yield() method causes a pause to the current executing thread to give the chance for other waiting threads having some priority.
- If there is no waiting thread and all threads have low priority then the same thread can continue its execution.
- If multiple threads are waiting with the same priority then which waiting thread will get a chance first, we can't aspect, it depends on the thread scheduler.

```
public static native void yield()
```

```java
//yield() method
class Mythread extends Thread{
  public void run(){
   for(int i=1;i<=10;i++)
   {
   System.out.println("child thread");
   Thread.yield(); //if no yeild(), there me be more no of times main thread will
   }
  }
}
class Demo{
  public static void main(String args[]){
   Mythread t=new Mythread();
   t.start();
   for(int i=1;i<=10;i++)
   {
   System.out.println("main thread");
   }
  }
}
```

```
//yield() method
class Mythread extends Thread{
  public void run(){
   for(int i=1;i<=10;i++)
   {
   System.out.println("child thread");
   Thread.yield(); //if no yeild(), there me be more no of times main thread will b
   }
  }
}
class Demo{
 public static void main(String args[]){
  Mythread t=new Mythread();
  t.start();
   for(int i=1;i<=10;i++)
   {
   System.out.println("main thread");
   }
```

join() method

If a thread wants to wait until completing some other thread then we should go for **join()** method.

E.g. If a thread **t1** wants to wait until completing **t2** and **t1** has to call **t2.join()**. So t1 executes **t2.join()** then immediately **t1** will be entered into the waiting state until **t2** completes. Once **t2** completed its execution then **t1** can continue its execution.

Methods:

- public final void **join()** throws InterruptedException
- public final void **join(long ms)** throws InterruptedException
- public final void **join(long ms, int ns)** throws InterruptedException

Ex:

```
//join() method
class Mythread extends Thread{
 public void run(){
  for(int i=1;i<=10;i++)
  {
  System.out.println("child thread");
       try{
         Thread.sleep(100);
       }
       catch(InterruptedException ie){}
  }
 }

class ThreadDemo{
public static void main(String args[]) throws InterruptedException
{
 Mythread t=new Mythread();
 t.start();
 t.join();//main thread call child thread to join, and will wait until completing child thread
  for(int i=1;i<=10;i++)
  {
  System.out.println("main thread");
  }
}
```

```java
class MyJava{
    public static void main(String[] args) {
        try{
            for (int i=1;i<=10 ;++i ) {
            System.out.print(i+"\t");
            Thread.sleep(500);
            }
        }
        catch(InterruptedException i){}
    }
}
```

A thread interrupt another thread

Raj is going to sleep for 1 Hour

Joy Venu

interrupt()

Raj

A thread interrupt another thread

A thread can interrupt a sleeping thread or waiting thread by using the interrupt() method of Thread class.

public void interrupt()

Ex:

```
class MyClass extends Thread {
    public void run()
    {
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println("Child Thread");
                // Here current threads goes to sleeping state and Another thread gets th
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println("InterruptedException is occur");
        }
    }


class ThreadDemo {
    public static void main(String[] args) throws InterruptedException
    {
        MyClass thread = new MyClass();
```

```
        catch (InterruptedException e) {
            System.out.println("InterruptedException is occur");
        }
    }


lass ThreadDemo {
    public static void main(String[] args) throws InterruptedException
    {
        MyClass thread = new MyClass();
        thread.start();
        // main thread calls interrupt() method on child thread
        thread.interrupt();
        for (int i = 0; i < 5; i++) {
                System.out.println("Main Thread");
                Thread.sleep(1000);
          }
    }
}
```

A thread interrupt another thread

- Whenever we are calling **interrupt()** method, if a target thread is not in **sleeping state** or **waiting state** then there is no impact of **interrupt call** immediately, interrupt calls will be waited until the target thread enters into the sleeping or waiting thread.
- If target threads is waiting or sleeping thread then immediately **interrupt call** will interrupt the target thread.
- If the target thread never enters into a **sleeping state** or **waiting state** in its lifetime or waiting or sleeping time is over, then there is no impact of **interrupt call**. This is the only case where an interrupt call will be wasted.

Synchronization

The process of allowing multiple threads to modify an object in a sequence is called synchronization.

Synchronization

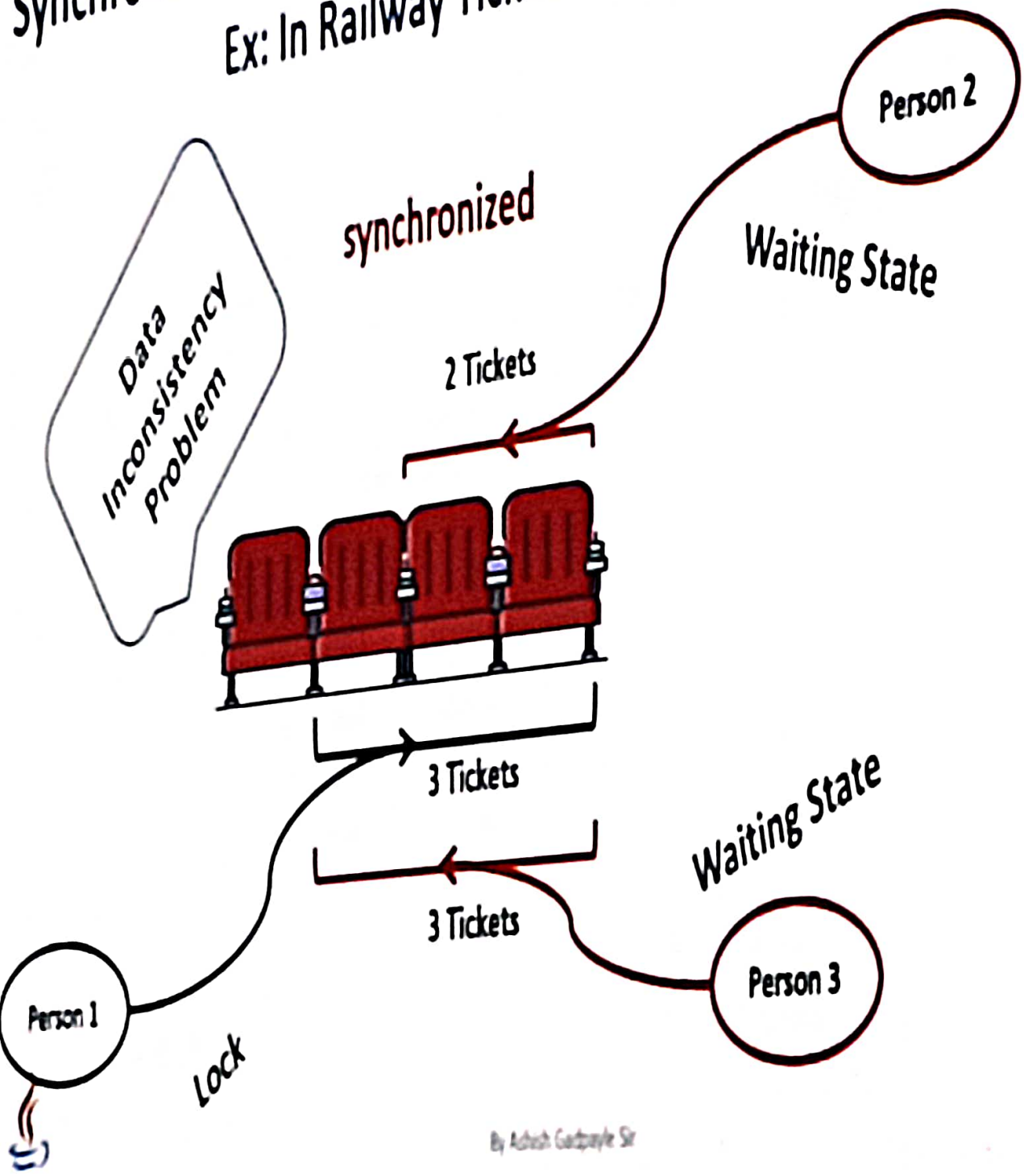
The process of allowing multiple threads to modify an object in a sequence is called synchronization.

Advantage : To overcome **data inconsistency** problem

Disadvantage : It increases **waiting time** of thread.

Synchronization

Ex: In Railway Ticket Reservation System



Synchronization

- Internally **synchronization** is implemented by using the **lock** concept. In Java every object has a unique **lock**, If a thread wants to execute a **synchronized** method on the given object first it has to **lock** that object.
- Once thread gets the **lock** then it is allowed to execute any **synchronized** method in that object. Once method execution completed automatically thread release that **lock**.
- Acquiring and releasing **locks** internally takes care by **JVM** and here the **Processor** is not responsible for this lock concept activity.

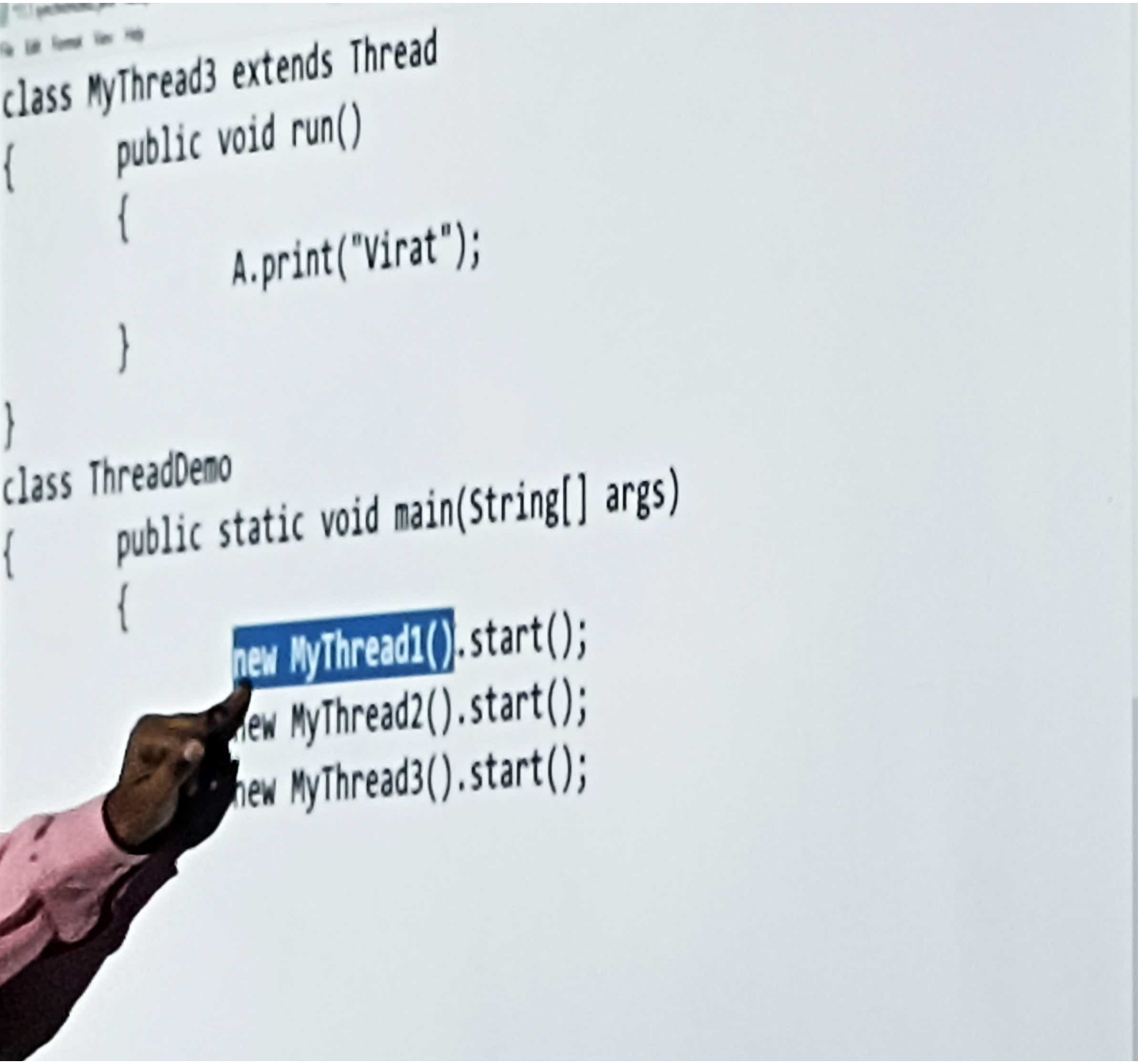
Synchronization

- While a thread is executing **synchronized** methods on the given object, here the remaining threads are not allowed to execute any **synchronized** method **simultaneously** on the same object.
- But here remaining threads are allowed to execute **non-synchronized** methods **simultaneously**.

Ex:

```
class A
{
	public static synchronized void print(String name)
	{
		for (int i=0;i<3;i++)
		{
			System.out.println("Batting = "+name);
			try{
				Thread.sleep(500);

			}
			catch(InterruptedException ie)
			{
				ie.printStackTrace();

			}
		}
	}
}
class MyThread1 extends Thread
{
	public void run()
	{
		A.print("Sachin");
	}
}
```

```
        }

    }

lass MyThread1 extends Thread
    public void run()
    {
        A.print("Sachin");

    }

lass MyThread2 extends Thread
    public void run()
    {
        A.print("Dhoni");

    }

lass MyThread3 extends Thread
    public void run()
    {
        A.print("Virat");
    }
```


class MyThread3 extends Thread
{ public void run()
{
A.print("Virat");
}
}
class ThreadDemo
{ public static void main(String[] args)
{
new MyThread1().start();
new MyThread2().start();
new MyThread3().start();

```
class A
{       public static synchronized void print(String name)
        {       for (int i=0;i<3;i++)
                {
                        System.out.println("Batting = "+name);
                        try{
                                Thread.sleep(500);

                        }
                        catch(InterruptedException ie)
                        {
                                ie.printStackTrace();

                        }
                }
        }
}
class MyThread1 extends Thread
{       public void run()
        {
                A.print("Sachin");
        }
}
class MyThread2 extends Thread
{       public void run()
        {
                A.print("Dhoni");
```

Daemon Threads

The threads which are executed at background and give the support to foreground thread is called **daemon threads**.

Ex:

- Default Exceptional Handler
- Garbage Collector
- ThreadSchedular

Daemon Thread

There two types of threads in java we have

- User thread
- Daemon thread

Daemon Thread Methods:

public final void **setDaemon(boolean)**

public final boolean **isDaemon()**

Note: *These threads are low priority threads and whenever user threads completes their execution, all the daemon threads are automatically stopped.*

Ex:

```
                System.out.println("Daemon Thread "+i);
                try{
                        Thread.sleep(500);
                }
                catch(InterruptedException ie){
                        ie.printStackTrace();
                }
        }
    }

lass ThreadDemo

    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.setDaemon(true);//setting daemon nature to Thread
        t.start();
        //main thread logic
        for (int i=0;i<5 ;i++)
        {
            System.out.println("Main Thread "+i);
```

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();

        t.start();
        t.setDaemon(true);//setting daemon nature to Thread
        //main thread logic
        for (int i=0;i<5 ;i++)
        {
            System.out.println("Main Thread "+i);
            try{
                Thread.sleep(500);

            }
            catch(InterruptedException ie)
            {
                ie.printStackTrace();
            }
        }
    }
```

```
/daemon thread
lass MyThread extends Thread

    public void run()
    {       for (int i=0;i<10 ;i++)
            {
                    System.out.println("Daemon Thread "+i);
                    try{
                            Thread.sleep(500);
                    }
                    catch(InterruptedException ie){
                            ie.printStackTrace();
                    }
            }
    }

lass ThreadDemo

    public static void main(String[] args)
```

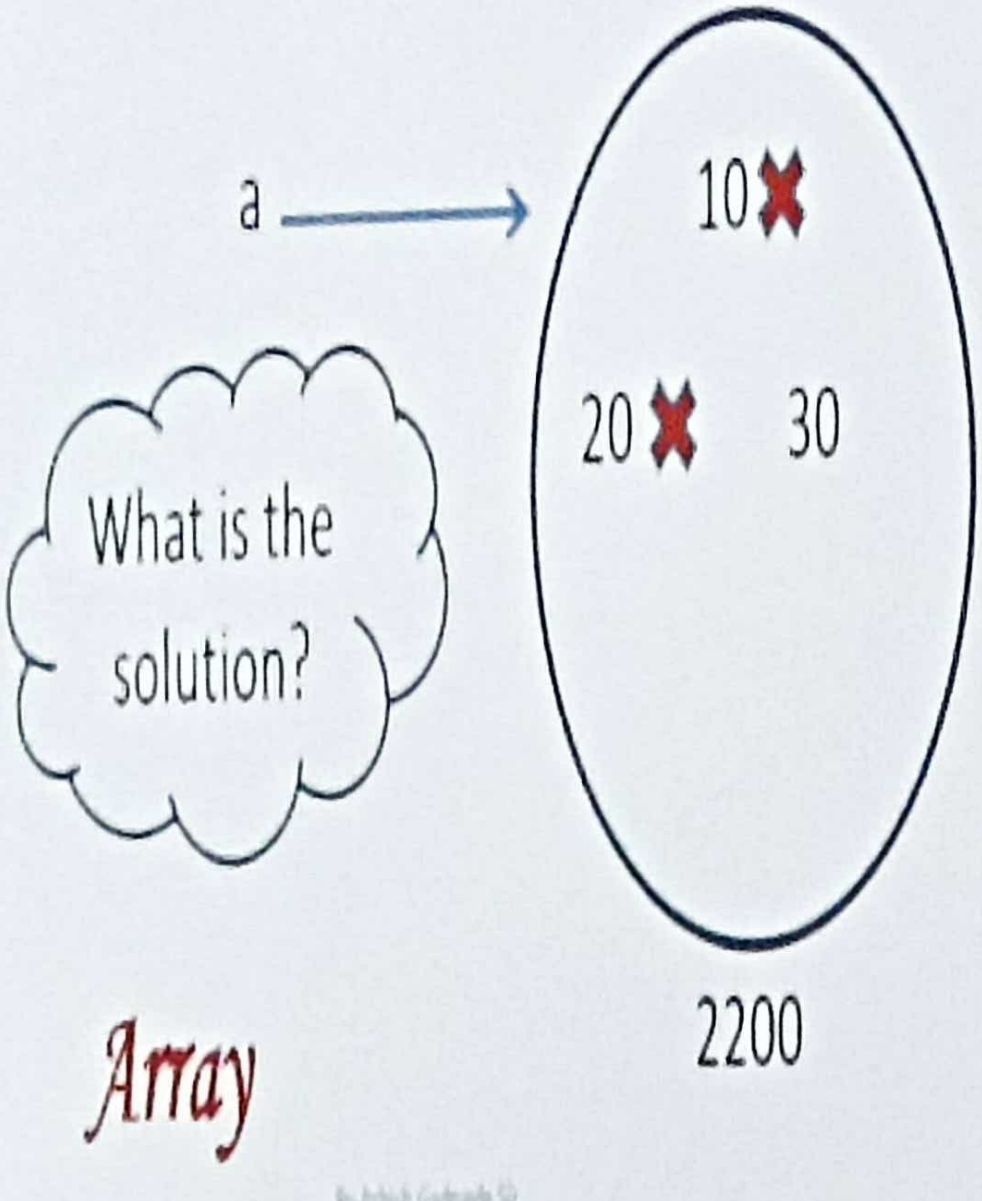
```
public static void main(String[] args)
{
        MyThread t = new MyThread();
        t.setDaemon(true);//setting daemon nature to Thread
        t.start();
        //main thread logic
        for (int i=0;i<5 ;i++)
        {
                System.out.println("Main Thread "+i);
                try{
                        Thread.sleep(500);
                }
                catch(InterruptedException ie)
                {
                        ie.printStackTrace();
                }
        }
}
```

Why Collection Framework

What is variable?

Variable is a name given to memory location or storage area

```
int a=10;
a=20;
a=30;
```

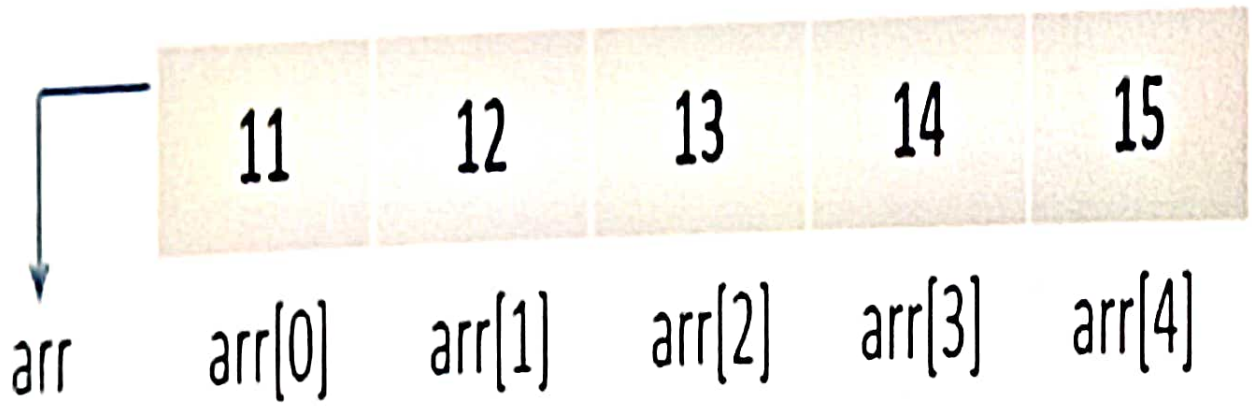


Array

Why Collection Framework

Lets understand about the Arrays

An array is an indexed collection of fixed number of homogeneous data elements



Disadvantages of Array

- Array is fixed
- Homogeneous data collection
- Non standard data structure

- Array is not implemented based on some standard Data Structure that is why the readymade method support is not available for every requirement of array.
- We have to code explicitly which increases the complexity of the program.

To overcome above problems we should go for

Collection Framework

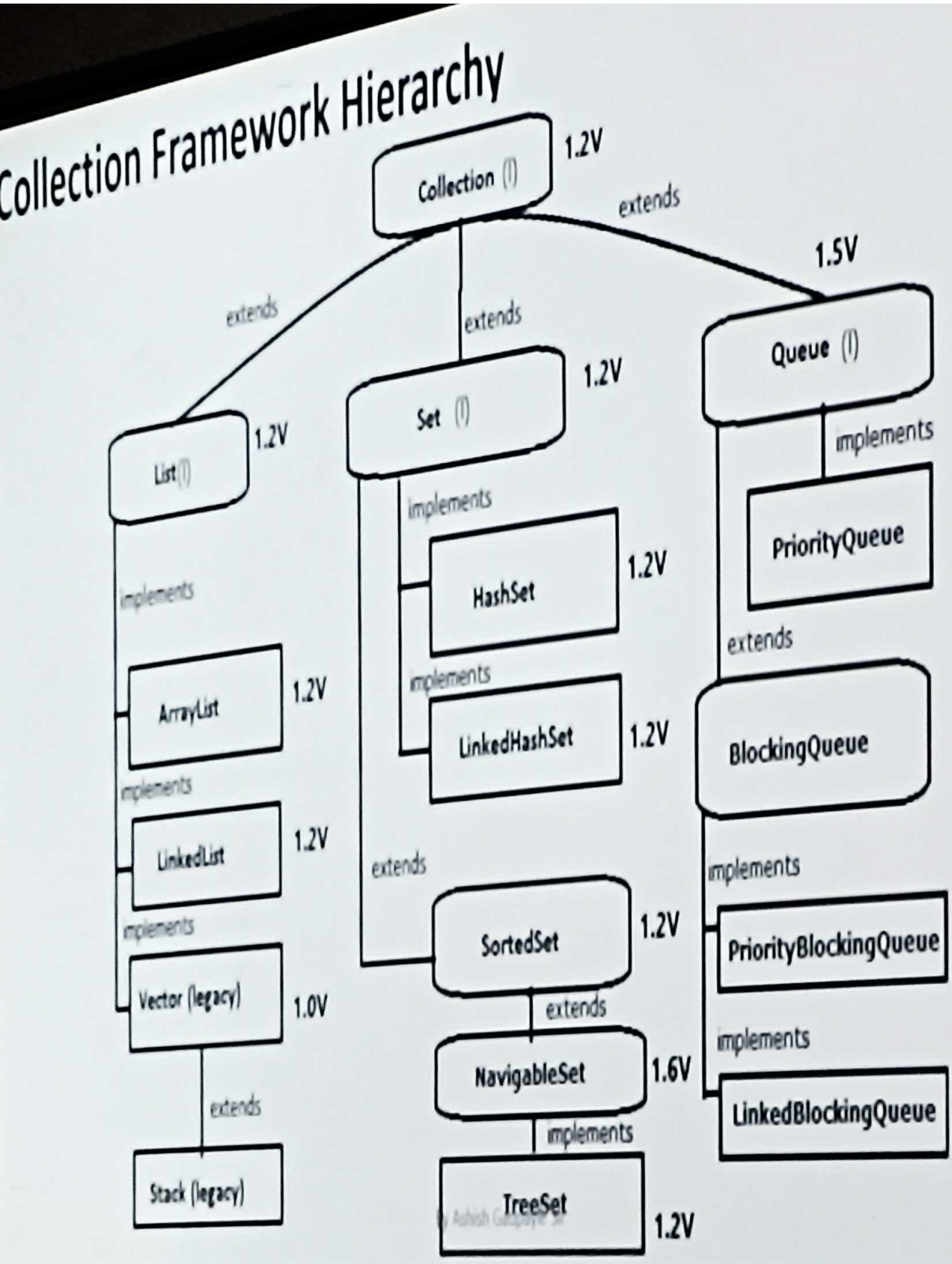
Difference between Arrays and Collections

Array	Collections
Arrays are **fixed in size** that is why we are not allowed to **increase** or **decrease** the size based on our requirement	Collections are **grow-able** in nature and based on our requirement we can **increase** or **decrease** the size
Arrays can hold both **primitives** as well as **objects**	Collections can hold only **objects** but are not primitive
Performance point of view arrays **faster** than collection	Performance point of view collections are **slower** than array

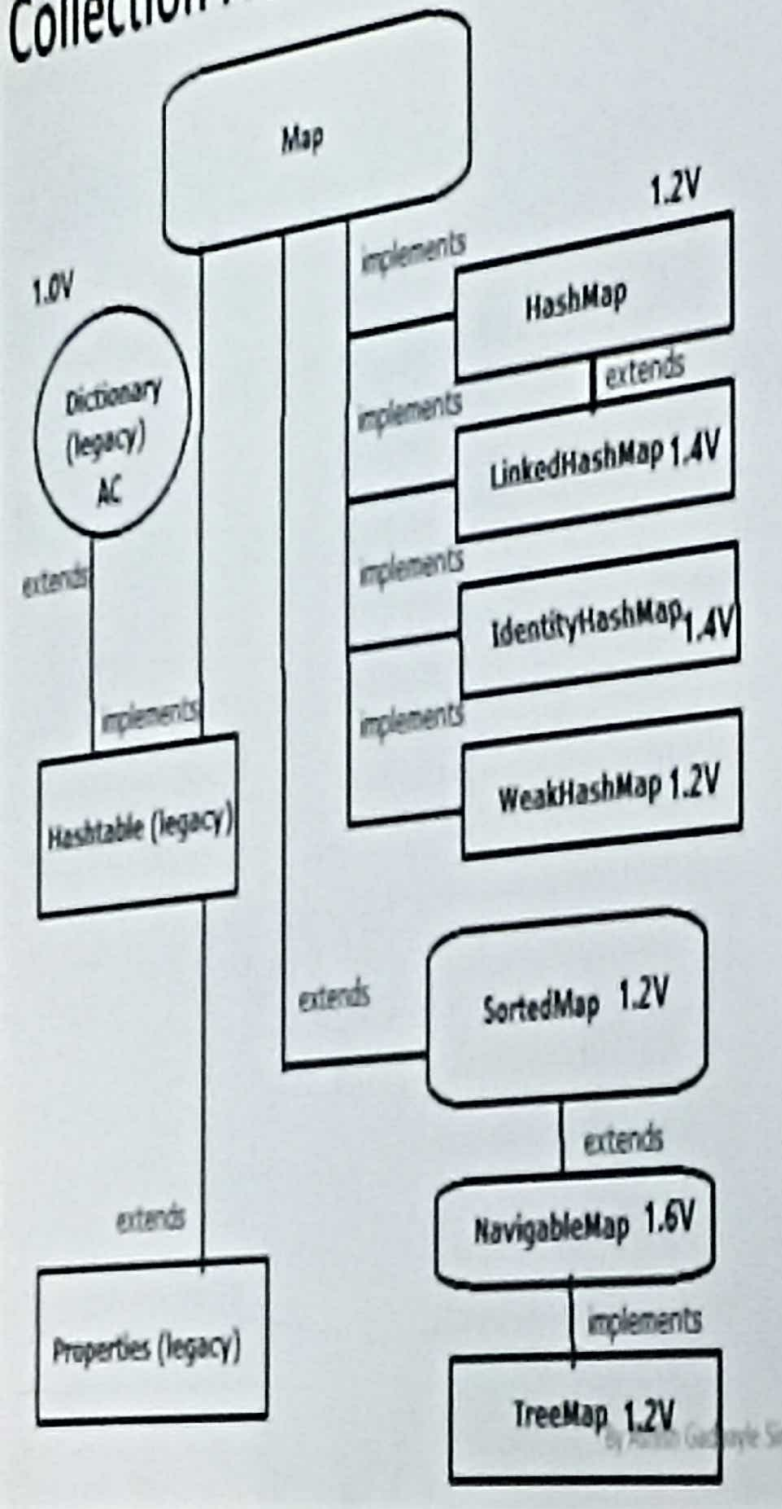
Difference between Arrays and Collections

Array	Collections
Arrays can hold only **homogeneous** elements	Collections can hold both **homogeneous** and **heterogeneous** elements
Memory point of view array is **not recommended** to use	Memory point of view collections are **recommended** to use
There is **no data structure** and readymade method support available	Every collection class implemented based on **some standard data structure** that is why read made method support is available

Collection Framework Hierarchy



Collection Framework Hierarchy



Sorting

- Comparator (I)
- Comparable (I)

Cursor

- Enumeration (I)
- Iterator (I)
- ListIterator (I)

Utility Classes

- Collections
- Arrays

Collection Interface

If we want to represent a group of individual objects as a single entity then we should go for Collection.

Collection Interface Methods

boolean retainAll(Collection c): Removes from the list all of its elements that are not contained in the specified collection.

boolean contains(Object o): Returns true if the list contains the specified element.

boolean containsAll(Collection c): Returns true if the collection contains all of the elements in the specified collection.

boolean isEmpty(): Returns true if the list contains no elements.

int size(): Returns the number of elements in the list.

Object[] toArray(): Returns an array containing all of the elements in the list in proper sequence (from first to last element).

Iterator iterator(): Returns an iterator over the elements in the list in proper sequence.

List(I)

- List is a child interface of collection.
- If we want to represent a group of individual objects as a single entity where duplicates are allowed and the insertion order must be preserved then we should go for List.
- Here we can preserve insertion order via. index and we can differentiate duplicate objects by using index.

ist Interface Methods

- void add(int index, Object o)
- boolean addAll(int index, Collection c)
- Object get(int index)
- Object set(int index, E element)
- Object remove(int index)
- int indexOf(Object o)
- int lastIndexOf(Object o)
- List subList(int fromIndex, int toIndex)
- ListIterator listIterator()
- ListIterator listIterator(int index)

ArrayList

- ArrayList is present in the java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation are needed in the array.

- Underlying data structure is Resizable array or Growable array.
- Duplicates are allowed.
- Insertion order preserved.
- Heterogeneous objects are allowed.
- null insertion is possible.

ArrayList - Constructors

- ArrayList al=new ArrayList():

Constructs an empty list with an initial capacity of 10. Once ArrayList reaches max capacity then a new ArrayList object will be created with New_Capacity

New_Capacity = (Initial_Capacity * 3/2) + 1

- public ArrayList(Collection c): Constructs a list containing the elements of the specified collection

- public ArrayList(int initialCapacity): Constructs an empty list with the specified initial capacity

Ex:

```
mport java.util.ArrayList;
lass ArrayListDemo

public static void main(String[] args)
{
      ArrayList al=new ArrayList();
      al.add("R");
      al.add(120);
      al.add("R");
      al.add(null);
      System.out.println(al);
      al.remove(2);
      System.out.println(al);
      al.add(2, "M");
      al.add("N");
      System.out.println(al);
      al.removeAll(al);
      System.out.println(al);
//    al.add(3, "C");  // java.lang.IndexOutOfBoundsException: Index: 3, Size: 0
//    System.out.println(al);
   }
```

```
import java.util.*;
class ArrayListDemo2
{
    public static void main(String[] args) {
        //create an empty array list with an initial capacity
        ArrayList<String> color_list=new ArrayList<>(4);
        //use add() method to add values in the list
        color_list.add("White");
        color_list.add("Black");
        color_list.add("Red");
        color_list.add("White");
        //print out the colors in the array list
        System.out.println("*** Color list ***");
        for (int i = 0; i < 4; i++)
        {
            System.out.println(color_list.get(i).toString());
        }
        //create an empty array sample with an initial capacity
        ArrayList<String> sample=new ArrayList<>(3);
```

```
        //create an empty array sample with an initial capacity
        ArrayList<String> sample=new ArrayList<>(3);
        //use add() method to add values in the list
        sample.add("Yellow");
        sample.add("Red");
        sample.add("White");
        //now add sample with color_list
        color_list.addAll(sample);
        //new size of the list
        System.out.println("New size of the list is: "+color_list.size());
        //print out the colors
        System.out.println("*** Color list ***");
        for (int i = 0; i < color_list.size(); i++)
        {
            System.out.println(color_list.get(i).toString());
        }
}
}
```

```
import java.util.*;
class ArrayListDemo2

  public static void main(String[] args) {
     //create an empty array list with an initial capacity
     ArrayList<String> color_list=new ArrayList<>(4);
     //use add() method to add values in the list
     color_list.add("White");
     color_list.add("Black");
     color_list.add("Red");
     color_list.add("White");
     //print out the colors in the array list
     System.out.println("*** Color list ***");
     for (int i = 0; i < 4; i++)
     {
        System.out.println(color_list.get(i).toString());
     }
     //create an empty array sample with an initial capacity
     ArrayList<String> sample=new ArrayList<>(3);
```

```
    //create an empty array sample with an initial capacity
    ArrayList<String> sample=new ArrayList<>(3);
    //use add() method to add values in the list
    sample.add("Yellow");
    sample.add("Red");
    sample.add("White");
    //now add sample with color_list
    color_list.addAll(sample);
    //new size of the list
    System.out.println("New size of the list is: "+color_list.size());
    //print out the colors
    System.out.println("*** Color list ***");
    for (int i = 0; i < color_list.size(); i++)
    {
        System.out.println(color_list.get(i).toString());
    }
}
```

ArrayList

- Every collection is implemented by default **Serializable** and **Cloneable** interface.

- Usually we can use collections to **hold** & **to transfer** objects from one location to another location (container). To provide support for this type of requirement, every collection class by default implements **Serializable** & **Cloneable** interface.

ArrayList

Every collection is implemented by default **Serializable** and **Cloneable** interface.

Usually we can use collections to **hold** & **to transfer** objects from one location to another location (container). To provide support for this type of requirement, every collection class by default implements **Serializable** & **Cloneable** interface.

ArrayList and **Vector** implements **RandomAccess** interface so that any random element we can access with the same speed.

Set(I)

- Set is a child interface of **Collection**.
- If we want to represent a group of individual objects as a single entity where **duplicates** are not allowed & the insertion order is **not preserved**.
- Set interface doesn't contain any new method and we have to use only **Collection** interface method.

HashSet

- The underlying data structure is **Hashtable**.
- **Duplicate** objects are not allowed.
- Insertion order is **not preserved** and it is based on **Hashcode** of the objects.
- **null** insertion is possible only once.
- **Heterogeneous** objects are allowed.
- Implements **Serializable, Cloneable** but not **RandomAccess** interface.
- **HashSet** is the best choice if our frequent operation is **search** operation.

HashSet - Constructors

public HashSet(): Constructs a new empty set with default initial capacity 16 and fill ratio (load factor) is 0.75

public HashSet(Collection c): The new HashMap is created with a default load factor (0.75) and an initial capacity

public HashSet(int initialCapacity): Constructs a new empty set with specified initial capacity

public HashSet(int initialCapacity, float loadFactor): Constructs a new, empty set with specified initial capacity and the specified load factor

HashSet

- If we are trying to insert a duplicate object then we won't get any **Exception** & the **add**() method simply returns "false".

```
HashSet h=new HashSet();
        h.add("A");          true
        h.add("A");          false //duplicate
```

Fill ratio or load factor:

After filling how much ratio a new **HashSet** object will be created, this ratio is called the **fill ratio** or **load factor**.

Ex: **fill ratio** 0.75 means after filling 75% a new **HashSet** object will be created.

```
/HashSetDemo
mport java.util.*;
lass HashSetDemo {
   public static void main(String[] args) {
      HashSet<String> h = new HashSet<String>();
               h.add("A");
               h.add("B");
               h.add("C");
               h.add("D");
               h.add("D");
               Iterator<String> itr = h.iterator();
               while(itr.hasNext())
               {
                        String str = itr.next();
                        System.out.println(str);
               }
               System.out.println(h);
   }
```

SortedSet

- It is a child interface of a **Set** Interface.

- If we want to represent a group of individual objects according to some **sorting order** where duplicate objects are not allowed then we should go for **SortedSet**.

SortedSet Methods

public Comparator comparator(): Use for the natural ordering of its elements.

public Object first(): Returns the first (lowest) element

public Object last(): Returns the last (highest) element

public SortedSet headSet(Object toElement): Returns a element whose elements are less than **toElement**

public SortedSet tailSet(Object fromElement): Returns a element whose elements are greater than or equal to **fromElement**

public SortedSet subSet(Object fromElement, Object toElement): Returns elements whose elements range >= **fromElement** but < **toElement**

TreeSet

- Underlying Data Structure is a **balanced tree.**
- Duplicates are not allowed.
- All objects will be inserted based on some sorting order, It may be **default natural sorting** order or **customize sorting** order.
- Heterogeneous objects are not allowed.
- **null** insertion is possible (only once).
- **TreeSet** implements **Serializable** & **Cloneable** but not **RandomAccess.**

TreeSet - Constructors

- **public TreeSet():** Constructs a new, empty TreeSet, sorted according to the natural ordering of its elements.
- **public TreeSet(Collection c):** Constructs a new TreeSet containing the elements in the specified collection
- **public TreeSet(Comparator comparator):** Constructs a new, empty TreeSet, sorted according to the specified comparator.
- **public TreeSet(SortedSet s):** Constructs a new TreeSet containing the same elements and using the same ordering as the specified sorted set.

Ex:

Important Points

- If we are depending on **default natural** sorting order then it is compulsory that the object should be **homogeneous** and **Comparable** otherwise we will get a runtime exception saying **ClassCastException**.
- An object is said to be **Comparable** if and only if corresponding class implements the **Comparable** interface.
- **String** class and all wrapper classes already implements **Comparable** interface but **StringBuffer** class doesn't implement **Comparable** interface, in that case we get **ClassCastException**