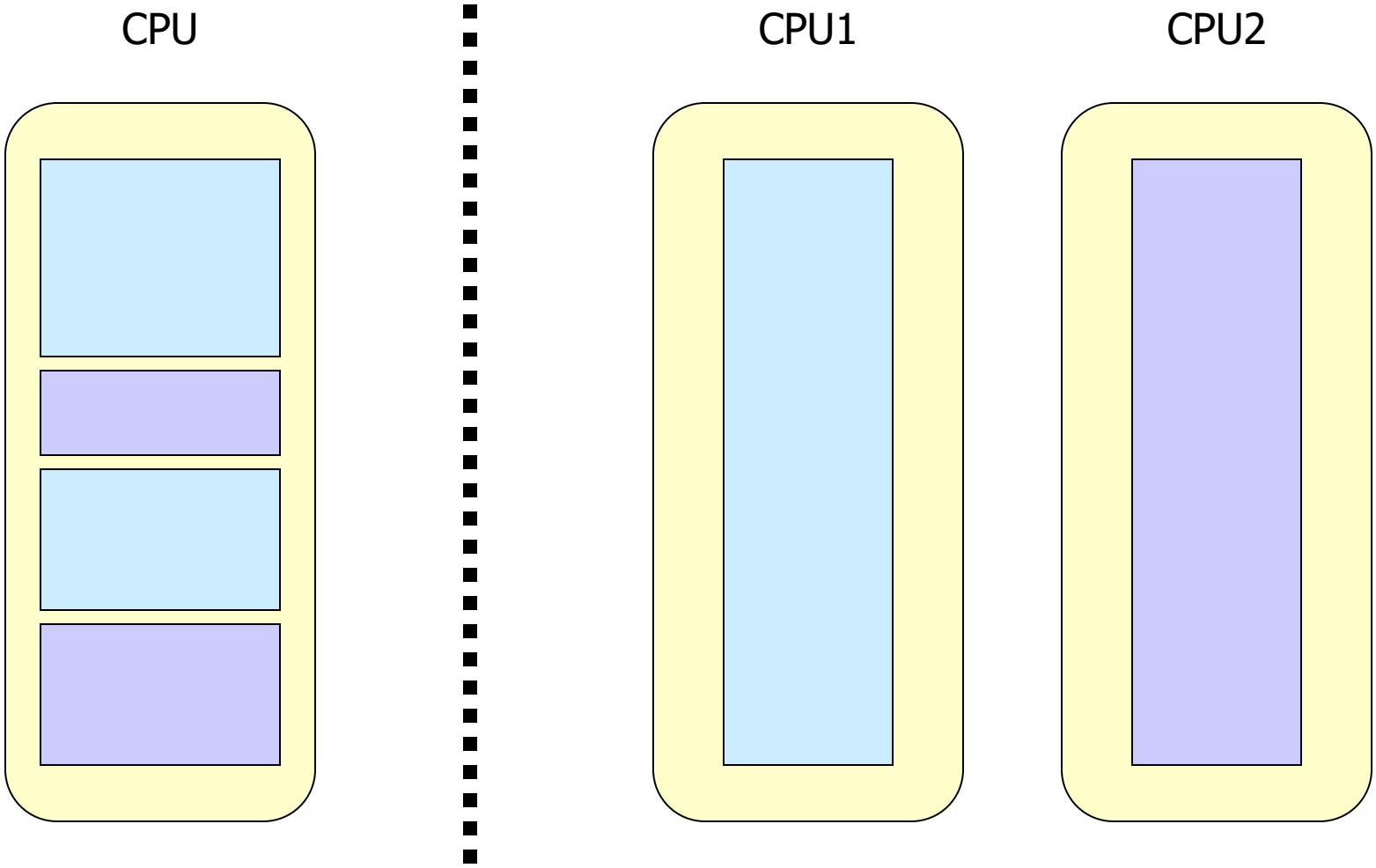
OBJECT ORIENTED PROGRAMMING USING JAVA

Multithreading in java

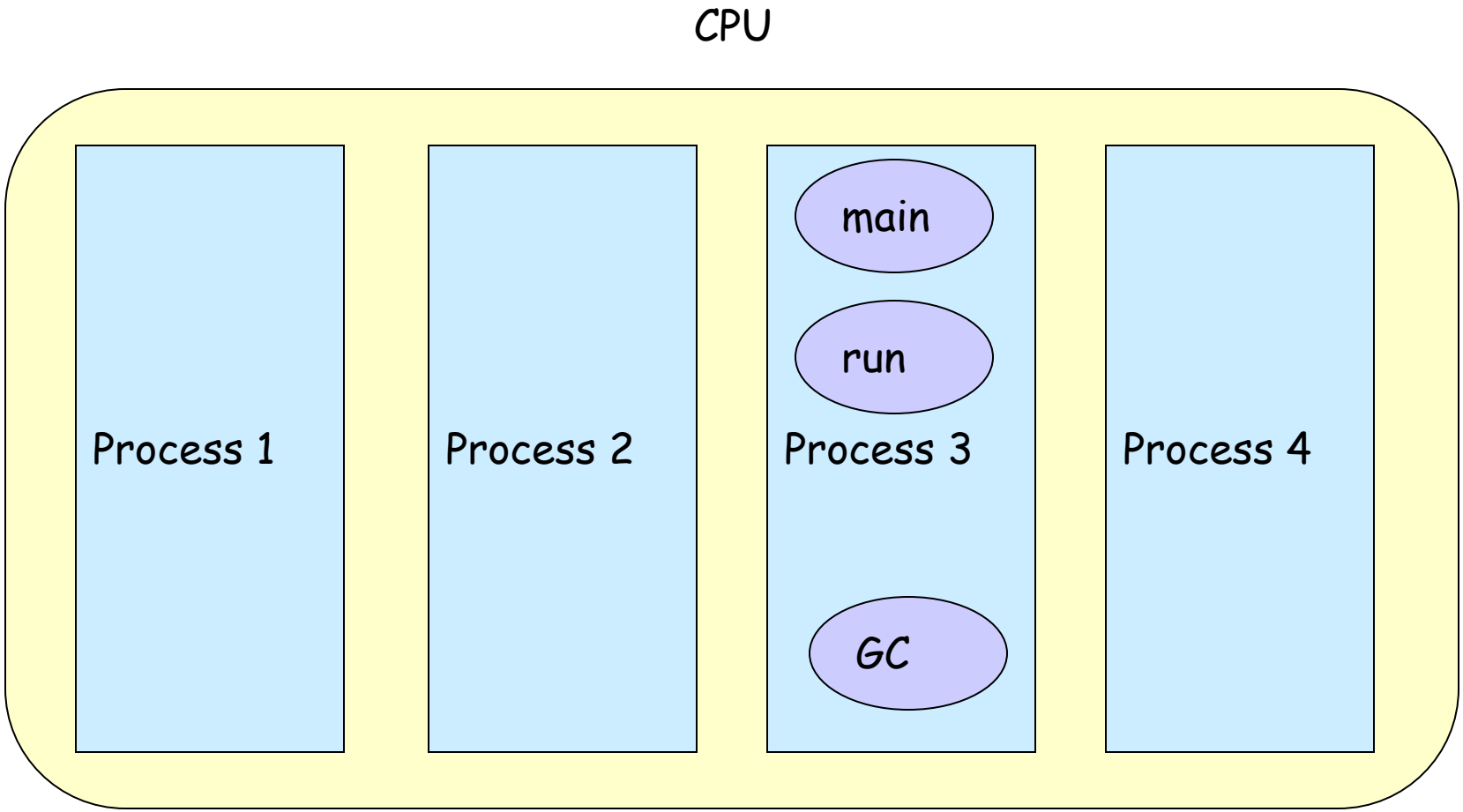
Multitasking and Multithreading

- Multitasking:
  - refers to a computer's ability to perform multiple jobs concurrently
  - more than one program are running concurrently, e.g., UNIX
- Multithreading:
  - A thread is a single sequence of execution within a program
  - refers to multiple threads of control within a single program
  - each program can run multiple threads of control within it, e.g., Web Browser

Concurrency vs. Parallelism



Threads and Processes

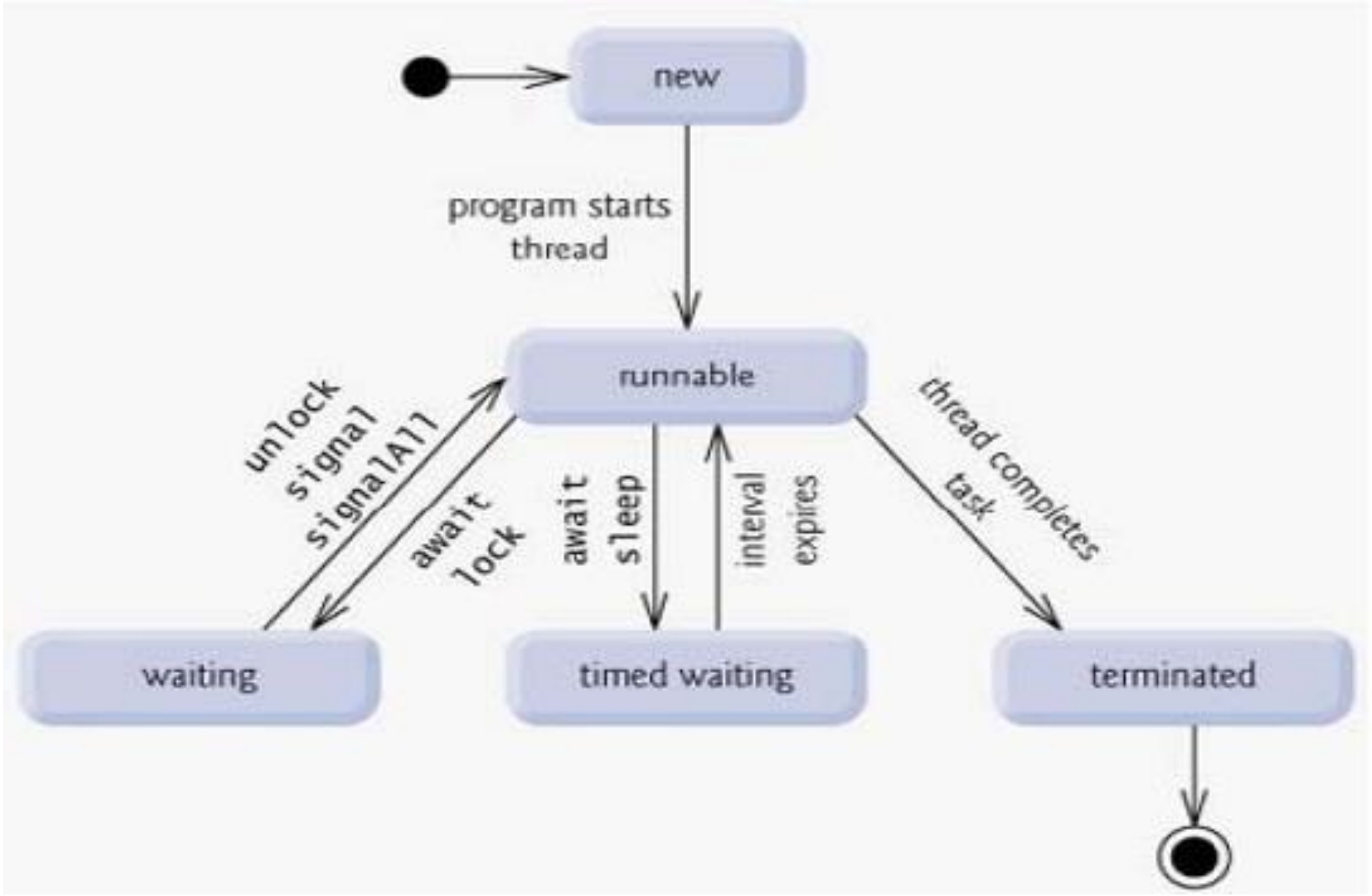


Multitasking and Multithreading

- **Multithreading in java** is a process of executing multiple threads simultaneously.
- Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- But we use multithreading than multiprocessing because threads share a common memory area.
- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation etc.

- **Advantages of Java Multithreading**
- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- 2) You **can perform many operations together so it saves time**.
- 3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

The Java Thread Model



The Main Thread

- When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:
- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions

The Main Thread

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: "
   + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change:
   " + t);
try {
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread
   interrupted");
} }
```

Current thread:Thread[main,5,main]
After name change:Thread[My Thread,5,main]
5
4
3
2
1

Creating Thread

- **How to create thread**

- There are two ways to create a thread:

1. By extending Thread class

2. By implementing Runnable interface.

- Thread class:
- Thread class provide constructors and methods to create and perform operations on a thread.
- Commonly used Constructors of Thread class:
- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

- Thread class:

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

Output:thread is running...

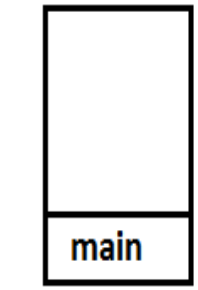
- **Runnable interface:**
- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 } }
```

Output: thread is running...

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitely create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

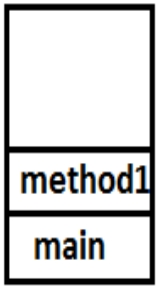
1) When we enter main() method



stack 1 (main thread's)

As soon as main is called by JVM it is pushed on Stack

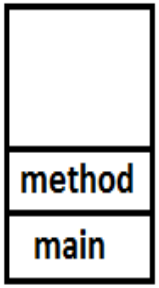
2) When main() calls method1() method



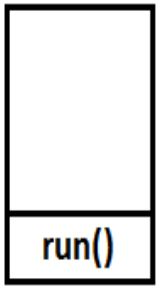
stack 1 (main thread's)

As soon as main calls method1 (), method1 pushed on Stack

3) When methd1() calls thread.start()



stack 1
(main thread's)



stack 2
(Thread-1's)

method1() creates new thread by calling thread.start (), as threads have their own stack - new stack is created.

```java
class MyThread extends Thread{
    public void run(){
        System.out.println("in run() method");
    }
}

public class MyClass {
    public static void main(String...args){
        System.out.println("In main() method");
        method1();
    }

    static void method1(){
        MyThread obj=new MyThread();
        obj.start();
    }
}

/*OUTPUT

currentThreadName= main
in run() method
currentThreadName= Thread-1

*/
```

Creating Multiple Threads

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}
catch(InterruptedException e){
System.out.println(e);}
    System.out.println(i);
  }  }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
     t1.start();
  t2.start();
 }  }
```

Using isAlive( ) and join( )

- sometimes one thread needs to know when another thread is ending. In java, **isAlive()** and **join()** are two different methods to check whether a thread has finished its execution.
- The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.
- *final* boolean **isAlive()**
- But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

Using isAlive( ) and join( )

- *final* void **join()** throws **InterruptedException**

  Using **join()** method, we tell our thread to wait until the specified thread completes its execution.

- There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

- *final* void **join**(long *milliseconds*) throws **InterruptedException**

Priority of a Thread (Thread Priority):

- Each thread have a priority. Priorities are represented by a number between 1 and 10.
- In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. 3 constants defiend in Thread class:
- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

Priority of a Thread (Thread Priority):

- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Priority of a Thread (Thread Priority):

```
class TestMultiPriority1 extends Thread{
 public void run()
{
System.out.println(" thread name is:"+Thread.currentThread().getName());

System.out.println(" thread priority is:"+Thread.currentThread().getPriority());
   }
 public static void main(String args[]){
 TestMultiPriority1 m1=new TestMultiPriority1();
 TestMultiPriority1 m2=new TestMultiPriority1();
 m1.setPriority(Thread.MIN_PRIORITY);
 m2.setPriority(Thread.MAX_PRIORITY);
 m1.start();
 m2.start();
   }  }
```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization. As you will see, Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor. A monitor is an object that is used as a mutually exclusive lock.

Synchronization

- Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- **Using Synchronized Methods**
- **Using Synchronized block**

Synchronization

- Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- **Using Synchronized Methods**
- **Using Synchronized block**

- # Inter-thread communication in Java

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

- Inter-thread communication in Java
- It is implemented by following methods of **Object class**:
- wait()
- notify()
- notifyAll()

- Inter-thread communication in Java

1) wait() method

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

2) notify() method

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation

- Inter-thread communication in Java

3) notifyAll() method

- Wakes up all threads that are waiting on this object's monitor

Suspending, Resuming, and Stopping Threads

- While the suspend( ), resume( ), and stop( ) methods defined by Thread class seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs and obsolete in newer versions of Java.
- the wait( ) and notify( ) methods that are inherited from Object can be used to control the execution of a thread.

Daemon Thread in Java

- **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.

Daemon Thread in Java

Points to remember for Daemon Thread in Java

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Daemon Thread in Java

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

ThreadGroup in Java

Java provides a convenient way to group multiple threads in a single object.

In such way, we can suspend, resume or interrupt group of threads by a single method call.

No.	Method	Description
1)	int activeCount()	returns no. of threads running in current group.
2)	int activeGroupCount()	returns a no. of active group in this thread group.
3)	void destroy()	destroys this thread group and all its sub groups.
4)	String getName()	returns the name of this group.
5)	ThreadGroup getParent()	returns the parent of this group.
6)	void interrupt()	interrupts all threads of this group.
7)	void list()	prints information of this group to standard console

JAVA'S COLLECTION FRAMEWORK

JAVA'S COLLECTION FRAMEWORK

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

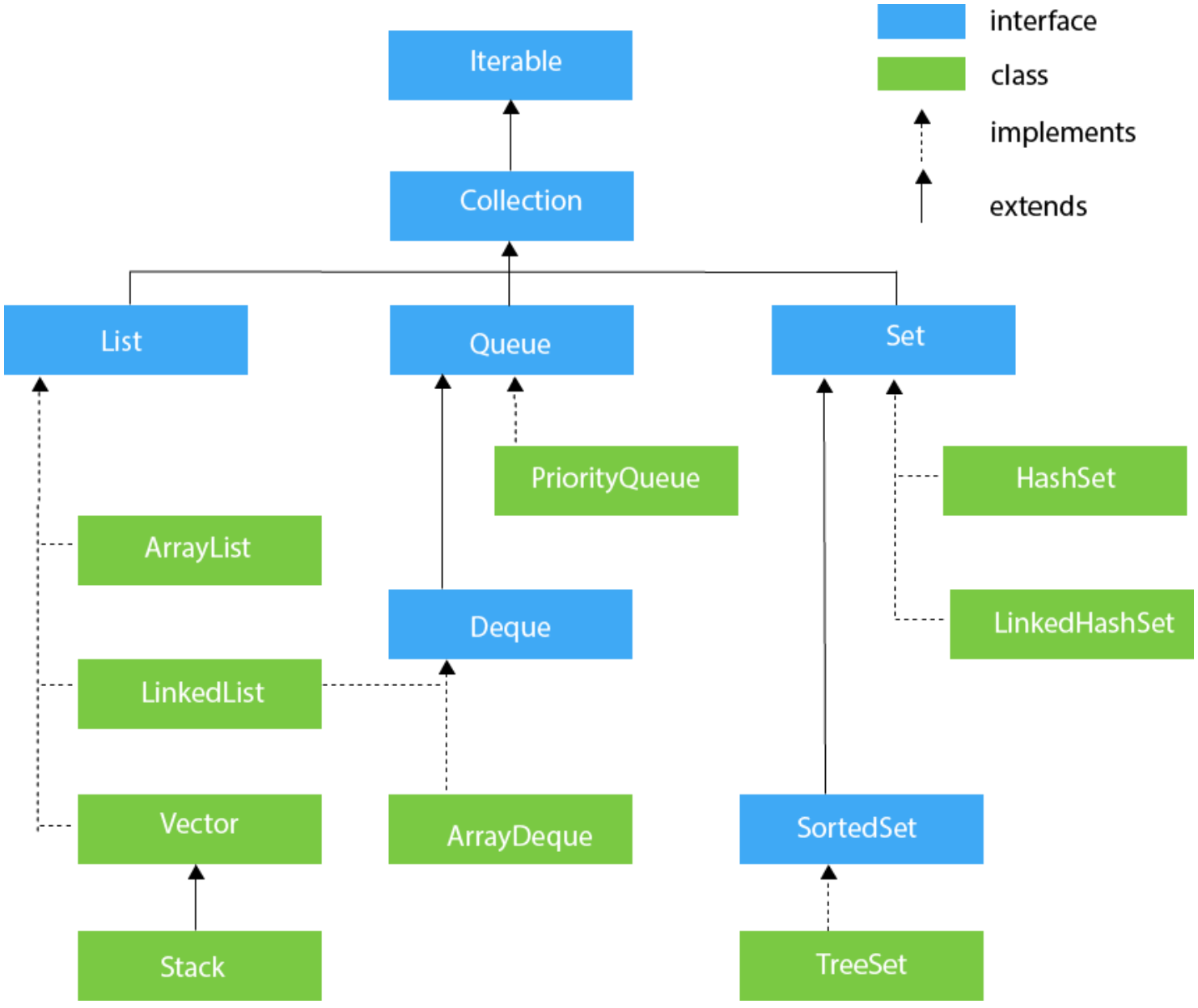
A framework provides readymade architecture and represents a set of classes and interfaces.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

- Interfaces and its implementations, i.e., classes
- Algorithm

JAVA'S COLLECTION'S HIERARCHY



COLLECTION INTERFACE METHODS

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	It returns the total number of elements in the collection.
7	public void clear()	It removes the total number of elements from the collection.
8	public boolean contains(Object element)	It is used to search an element.

COLLECTION INTERFACE METHODS

No.	Method	Description
9	public Iterator iterator()	It returns an iterator.
10	public Object[] toArray()	It converts collection into array.
11	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
12	public boolean isEmpty()	It checks if collection is empty.
13	public boolean equals(Object element)	It matches two collections.
14	public int hashCode()	It returns the hash code number of the collection.

- The Collection interface is implemented by all the classes in the collection framework.
- Collection interface builds the foundation for the collection framework .

ITERATOR INTERFACCE METHODS

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

ITERABLE INTERFACE

It is the root class of all collection classes. It contains only one method

- Iterator<T> iterator()

LIST INTERFACE

- List interface is the child interface of Collection interface.

- It represents a list type data structure in which we can store the **ordered collection** of objects.

- It can have **duplicate** values.

- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

LIST INTERFACE METHODS

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.

SET INTERFACE

- A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

ARRAYLIST CLASS

ArrayList uses a dynamic array data structure to store objects and elements.

ArrayList allows duplicate objects and elements.

ArrayList maintains the insertion order.

ArrayList is non-synchronized.

ArrayList elements/objects can be accessed randomly.

ARRAYLIST EXAMPLE

```
import java.util.*;
class ArrayListImplementation{
public static void main(String args[]){

ArrayList<String>list=new ArrayList<String>();//Creatingarraylist
System.out.println("Initial size of list: "+list.size());
list.add("Raghu");
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
list.add("Dayakar");
list.add(1,"Dayakar");
//TraversinglistthroughIterator

System.out.println("list contents: "+list);
System.out.println("Individual Elements in list:");
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
list.remove("Dayakar");
list.remove(0);
System.out.println("list contents after removing elements:"+list);
System.out.println("Size of list: "+list.size());
}
}
```

```
Initial size of list: 0
list contents: [Raghu, Dayakar, Vijay, Ravi, Ajay, Dayakar]
Individual Elements in list:
Raghu
Dayakar
Vijay
Ravi
Ajay
Dayakar
list contents after removing elements:[Vijay, Ravi, Ajay, Dayakar]
Size of list: 4
```

LINKEDLIST CLASS

- LinkedList implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.
- In LinkedList, the manipulation is fast because no shifting is required.

LINKEDLIST EXAMPLE

```
import java.util.*;
class LinkedListImplementation{
public static void main(String args[]){

LinkedList<String>list=new LinkedList<String>();//Creatingarraylist
System.out.println("Initial size of list: "+list.size());
list.add("Raghu");
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
list.add("Dayakar");
list.add(1,"Dayakar");
list.addFirst("First");
list.addLast("Last");
//TraversinglistthroughIterator

System.out.println("list contents: "+list);
System.out.println("Individual Elements in list:");
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
list.remove("Dayakar");
list.remove(0);
list.removeFirst();
list.removeLast();
System.out.println("list contents after removing elements:"+list);
System.out.println("Size of list: "+list.size());
}
}
```

```
Initial size of list: 0
list contents: [First, Raghu, Dayakar, Vijay, Ravi, Ajay, Dayakar, Last]
Individual Elements in list:
First
Raghu
Dayakar
Vijay
Ravi
Ajay
Dayakar
Last
list contents after removing elements:[Vijay, Ravi, Ajay, Dayakar]
Size of list: 4
```

HASHSET CLASS

- HashSet class implements Set Interface.
- It represents the collection that uses a hash table for storage.
- Hashing is used to store the elements in the HashSet.
- It contains unique items.

HASHSET EXAMPLE

```
import java.util.*;
class HashSetImplementation{
public static void main(String args[]){
HashSet<String>set=new HashSet<String>();//Creatingarraylist
System.out.println("Initial size of list: "+set.size());
set.add("Raghu");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
set.add("Dayakar");
set.add("Dayakar");
//TraversinglistthroughIterator

System.out.println("list contents: "+set);
System.out.println("Individual Elements in list:");
Iterator itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
System.out.println("Size of list: "+set.size());
}
}
```

```
Initial size of list: 0
list contents: [Vijay, Dayakar, Raghu, Ravi, Ajay]
Individual Elements in list:
Vijay
Dayakar
Raghu
Ravi
Ajay
Size of list: 5
```

TREESET CLASS

- Java TreeSet class implements the Set interface that uses a tree for storage.
- Like HashSet, TreeSet also contains unique elements.
- The access and retrieval time of TreeSet is quite fast.
- The elements in TreeSet stored in ascending order.

TREESET EXAMPLE

```
import java.util.*;
class TreeSetImplementation{
public static void main(String args[]){
TreeSet<String>set=new TreeSet<String>();//Creatingarraylist
System.out.println("Initial size of list: "+set.size());
set.add("Raghu");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
set.add("Dayakar");
set.add("Dayakar");
//TraversinglistthroughIterator

System.out.println("list contents: "+set);
System.out.println("Individual Elements in list:");
Iterator itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
System.out.println("Subset of Elements:");
System.out.println(set.subSet("Ajay","Ravi"));
System.out.println("Size of list: "+set.size());
} }
```

```
Initial size of list: 0
list contents: [Ajay, Dayakar, Raghu, Ravi, Vijay]
Individual Elements in list:
Ajay
Dayakar
Raghu
Ravi
Vijay
Subset of Elements:
[Ajay, Dayakar, Raghu]
Size of list: 5
```

QUEUE INTERFACE

- Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PRIORITY QUEUE CLASS

- The PriorityQueue class provides the facility of using queue.
- But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.
- PriorityQueue holds the elements or objects which are to be processed by their priorities.
- PriorityQueue doesn’t allow null values to be stored in the queue.

PRIORITY QUEUE CLASS

```
import java.util.*;
class PriorityQueueImplementation{
public static void main(String args[]){
PriorityQueue<String> queue=new
PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
System.out.println("head using element()
method:"+queue.element());
System.out.println("head using peek()
method:"+queue.peek());
System.out.println("iterating the queue
elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two
elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

```
head using element() method:Amit
head using peek() method:Amit
iterating the queue elements:
Amit
Jai
Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay
```

ArrayDeque CLASS

- ArrayDeque class implements the Deque interface.

- It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

ArrayDeque CLASS

```
import java.util.*;
public class ArrayDequeImplementation{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output

Gautam
Karan
Ajay

HASHTABLE

- Java Hashtable class implements a hashtable, which maps keys to values.

- It inherits Dictionary class and implements the Map interface.

Points to remember

- A Hashtable is an array of a list.

- Each list is known as a bucket.

- The position of the bucket is identified by calling the hashcode() method.

HASHTABLE

- A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

HASHTABLE

Method	Description
void clear()	Use to reset the hash table.
Object clone()	It returns a shallow copy of the Hashtable.
Enumeration elements()	It returns an enumeration of the values in the hash table.
Set<Map.Entry> entrySet()	It returns a set view of the mappings contained in the map.
boolean equals(Object o)	Use to compare the specified Object with the Map.
void forEach(BiConsumer action)	It performs the action for each entry in the map until all entries have been processed or the action throws an exception.
V getOrDefault(Object key, V defaultValue)	This method returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map
Enumeration keys()	This method returns an enumeration of the keys in the hashtable.
Set keySet()	It returns a Set view of the keys contained in the map.

HASHTABLE EXAMPLE

```
import java.util.*;
class HashtableImplementation{
public static void main(String args[]){
        Hashtable<Integer,String> hm=new
                        Hashtable<Integer,String>();
  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");
for(Map.Entry m:hm.entrySet()){
                System.out.println(m.getKey()+"
"+m.getValue());
}
} }
```

```
103 Rahul
 102 Ravi
 101 Vijay
100 Amit
```

PROPERTIES CLASS

- The java.util.Properties class is the subclass of Hashtable.

- The **properties** object contains key and value pair both as a string.

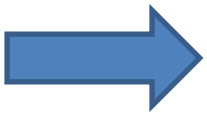
- It can be used to get property value based on the property key.

- The Properties class provides methods to get data from the properties file and store data into the properties file.

- Moreover, it can be used to get the properties of a system.

PROPERTIES CLASS EXAMPLE

Lets create a properties file



db.properties
user=system
password=oracle

Now, let's create the java class to read the data from the properties file Test.java

```
import java.util.*;
import java.io.*;
public class PropertiesImplementation {
public static void main(String[] args)throws Exception{
    FileReader reader=new FileReader("db.txt");
    Properties p=new Properties();
    p.load(reader);
    System.out.println(p.getProperty("username"));
    System.out.println(p.getProperty("password"));
}
}
```

VECTOR CLASS

- Vector uses a dynamic array to store the data elements. It is similar to ArrayList.
- However, It is synchronized and contains many methods that are not the part of Collection framework.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());  } } }
```

OUTPUT

```
Ayush
Amit
Garima
```

STACK CLASS

- The stack is the subclass of Vector.

- It implements the last-in-first-out data structure, i.e., Stack.

- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

STACK CLASS EXAMPLE

```
import java.util.*;
public class StackImplementation{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

OUTPUT

```
Ayush
Garvit
Amit
Ashish
```

BITSET CLASS

- The stack is the subclass of Vector.

- It implements the last-in-first-out data structure, i.e., Stack.

- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

BITSET CLASS contd...

CONSTRUCTOR	Description
BitSet( )	This constructor creates a default object.
BitSet(int size)	This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero.

METHOD	Description
void and(BitSet bitSet)	ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.
void andNot(BitSet bitSet)	For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
int cardinality( )	Returns the number of set bits in the invoking object.
void or(BitSet bitSet)	ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.
void clear(int index)	Zeros the bit specified by index.
void xor(BitSet bitSet)	XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.
void flip(int index)	Reverses the bit specified by the index.

```
import java.util.BitSet;
public class BitSetDemo {
 public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);
    for(int i = 0; i < 16; i++) {
       if((i % 2) == 0) bits1.set(i);
       if((i % 5) != 0) bits2.set(i); }
   System.out.println("Initial pattern in bits1: ");
   System.out.println(bits1);
   System.out.println("\nInitial pattern in bits2: ");
   System.out.println(bits2);
   bits2.and(bits1); // AND bits
   System.out.println("\nbits2 AND bits1: ");
   System.out.println(bits2);
   bits bits2.or(bits1); // OR
   System.out.println("\nbits2 OR bits1: ");
   System.out.println(bits2);
} }
```

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11,
12, 13, 14}
bits2 AND bits1:
 {2, 4, 6, 8, 12, 14}
bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}
```

DATE CLASS

- The java.util.Date class represents date and time in java.

- It provides constructors and methods to deal with date and time in java.

- The java.util.Date class implements Serializable, Cloneable and Comparable<Date> interfaces.

- It is inherited by java.sql.Date, java.sql.Time and java.sql.Timestamp classes.

- Most of the constructors and methods of java.util.Date class has been deprecated.

DATE CLASS contd...

CONSTRUCTOR	Description
Date()	Creates a date object representing current date and time.
Date(long milliseconds)	Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.

METHOD	Description
Boolean after(Date date)	tests if current date is after the given date.
boolean before(Date date)	tests if current date is before the given date.
Object clone()	returns the clone object of current date.
int compareTo(Date date)	compares current date with given date.
long getTime()	returns the time represented by this date object.
void setTime(long time)	changes the current date and time to given time.

DATE CLASS EXAMPLE

Sample code 1:

```
java.util.Date date=new java.util.Date();
System.out.println(date);
```

Output:

Wed Oct 6 08:22:02 IST 2020

Sample code 2:

```
long millis=System.currentTimeMillis();
java.util.Date date=new java.util.Date(millis);
System.out.println(date);
```

Output:

Wed Oct 6 08:22:02 IST 2020

CALENDAR CLASS

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc.

It inherits Object class and implements the Comparable, Serializable and Cloneable interfaces.

Syntax ::

public abstract class Calendar **extends** Object
implements Serializable, Cloneable, Comparable<Calendar>

Calendar CLASS contd...

METHOD	Description
public void add(int field, int amount)	Adds the specified (signed) amount of time to the given calendar field.
public int getWeekYear()	This method gets the week year represented by current Calendar.
Public static Calendar getInstance()	This method is used with calendar object to get the instance of calendar according to current time zone set by java runtime environment
public int get(int field)	In get() method fields of the calendar are passed as the parameter, and this method Returns the value of fields passed as the parameter.
long getTime()	This method gets the time value of calendar object and Returns date.
public String getCalendarType()	Returns in string all available calendar type supported by Java Runtime Environment

CALENDAR CLASS EXAMPLE

```
import java.util.Calendar;
public class CalendarExample1 {
   public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("The current date is : " + calendar.getTime());
   calendar.add(Calendar.DATE, -15);
   System.out.println("15 days ago: " + calendar.getTime());
   calendar.add(Calendar.MONTH, 4);
   System.out.println("4 months later: " + calendar.getTime());
   calendar.add(Calendar.YEAR, 2);
   System.out.println("2 years later: " + calendar.getTime());
System.out.println("At present Calendar's Year: " + calendar.get(Cale
ndar.YEAR));
   }
}
```

```
The current date is : Thu Jan 19 18:47:02 IST 2017
15 days ago: Wed Jan 04 18:47:02 IST 2017
4 months later: Thu May 04 18:47:02 IST 2017
2 years later: Sat May 04 18:47:02 IST 2019
At present Calendar's Year: 2020
```

RANDOM CLASS

- An instance of this class is used to generate a stream of pseudorandom numbers.

- The class uses a 48-bit seed, which is modified using a linear congruential formula.

- The algorithms implemented by Random class use a protected utility method than can supply up to 32 pseudo randomly generated bits on each invocation.

METHOD	Description
doubles()	Returns an unlimited stream of pseudorandom double values.
ints()	Returns an unlimited stream of pseudorandom int values.
longs()	Returns an unlimited stream of pseudorandom long values.
next()	Generates the next pseudorandom number.
nextByte()	Generates random bytes and puts them into a specified byte array.
nextDouble()	Returns the next pseudorandom Double value between 0.0 and 1.0 from the random number generator's sequence
nextFloat()	Returns the next uniformly distributed pseudorandom Float value between 0.0 and 1.0 from this random number generator's sequence
nextInt()	Returns a uniformly distributed pseudorandom int value generated from this random number generator's sequence
nextLong()	Returns the next uniformly distributed pseudorandom long value from the random number generator's sequence.
setSeed()	Sets the seed of this random number generator using a single long seed

RANDOM CLASS EXAMPLES

```
import java.util.Random;

public class Test
{
   public static void main(String[] args)
   {
      Random random = new Random();
      System.out.println(random.nextInt(10));
      System.out.println(random.nextBoolean());
      System.out.println(random.nextDouble());
   }
 }
```

```
OUTPUT::
4
true
 0.1967493434040291
```

FORMATTER CLASS

- Formatter class outputs the formatted output.
- It can format numbers, strings, and time and date.
- It operates in a manner similar to the C/C++ printf() function..
- Formatters are not necessarily safe for multithreaded access.
- Thread safety is optional and is the responsibility of users of methods in this class.

FORMATTER CLASS CONTD..

Sr.No	Constructor & Description
1	**Formatter()** This constructor constructs a new formatter.
2	**Formatter(Appendable a)** This constructor constructs a new formatter with the specified destination.
3	**Formatter(File file)** This constructor constructs a new formatter with the specified file.

Sno	Method & Description
1	void close()This method closes this formatter.
2	void flush()This method flushes this formatter.
4	Formatter format(String format, Object... args)This method writes a formatted string to this object's destination using the specified format string and arguments.
7	Appendable out()This method returns the destination for the output.
8	String toString()This method returns the result of invoking toString() on the destination for the output

FORMATTER CLASS EXAMPLE

```
import java.util.Formatter;
 public class Main {
 public static void main(String args[]) {
Formatter fmt = new Formatter();
 fmt.format("%s Science marks are %d and percentage  is  %f",
“Ravi", 70, 62.3);
System.out.println(fmt.out());
   }
}
```

OUTPUT::
Ravi Science marks are 70 and perceentage is 62.3

SCANNER CLASS

- Scanner class in Java is found in the java.util package.

- The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression.

- It is the simplest way to get input in Java.

- Scanner can also get input from the user in primitive types such as int, long, double, byte, float, short, etc.

- The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt() where XXX represents a datatype

- The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

SCANNER CLASS CONTD..

Sr.No	Constructor & Description
1	**Scanner(File source)** It constructs a new Scanner that produces values scanned from the specified file.
2	**Scanner(InputStream source)** It constructs a new Scanner that produces values scanned from the specified input stream
3	**Scanner(String source)** It constructs a new Scanner that produces values scanned from the specified string
Sno	**Method & Description**
1	void close()This method closes this Scanner.
2	void hasNext(): It returns true if this scanner has another token in its input
4	int nextInt(): It scans the next token as an integer.
5	nextLine() : It is used to get the complete Line of input string
6	String toString()This method returns the string representation of scanner

SCANNER CLASS EXAMPLE

```
import java.util.*;
public class ScannerExample {
public static void main(String args[]){
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
        in.close();
        }
}
```

STRINGTOKENIZER CLASS

- The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

- It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. All tokens are treated as strings

Constructor	Description
StringTokenizer(String str)	creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	creates StringTokenizer with specified string and delimeter.
StringTokenizer(String str, String delim, boolean returnValue)	creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

STRINGTOKENIZER CLASS

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens

STRINGTOKENIZER CLASS

```
import java.util.*;
public class StringTokenizerExample {
public static void main(String args[]){
   StringTokenizer st=new StringTokenizer("I belong to II ALEXA");
                    while(st.hasMoreTokens())
      System.out.println(st.nextToken());

    }

}
```

OUTPUT::
I
belong
to
II
ALEXA