ADVANCED JAVA (collections, java8 , Annotations, Serialization,Multithreading, Synchronization)

— Collections

- An object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- A collections framework is a unified architecture for representing and manipulating collections

[Collection Framework]

- Interfaces - Collection, Set, List, Queue, Deque, Map
- Implementations - ArrayList, LinkedList, HashSet, TreeSet, LinkedHashSet, ArrayDeque, HashMap, TreeMap, LinkedHashMap
- Algorithms - Sorting, Shuffling, Searching, Data Manipulation, Composition, Min/Max

--Hierarchy of Collection Framework [capitals are Interfaces, Togglecase is class]

ITERABLE <-- COLLECTION <-- LIST,QUEUE,SET
LIST <-- ArrayList, LinkedList, Vector, Stack
QUEUE <-- PriorityQueue
QUEUE <-- DEQUE <-- ArrayDeque
SET <-- HashSet,LinkedHashSet
SET <-- SORTEDSET <-- TreeSet

MAP <-- SORTEDMAP <-- TreeMap
MAP <-- HasMap <-- LinkedHashMap

[Benefits]

- Reduces Programming Effort
- Increases Program Speed and Quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs
- Fosters software reuse

Arrays

- Fixed size
- Sequential memory allocation

Vector

- Dynamic Array
- Synchronized. Thread Safe

Hashtable
- Key Value store (Objects)
- Synchronized. Thread Safe

Properties
- Key Value store (String)

List - indexed and ordered
- Vector
- ArrayList
- LinkedList

Set - maintains unique values and sorted
- TreeSet
- HashSet
- LinkedHashSet

Map - key value store
- TreeMap
- HashMap
- LinkedHashMap

Queue - FIFO
- PriorityQueue
- Deque

Collections Framework Hierachy

- Iterable
  - Collection
    - List - dynamic sizing, ordered, index based, supports
      - Vector - synchronized
      - ArrayList - faster frequent reads, random access
      - LinkedList - frequent insertions and updations
    - Set - does not allows duplicates, sorted, no index
      - HashSet - faster search, works based on hashing techinque, not ordered/sorted, allows null values
      - TreeSet - sorting, not allows null values
        - Comparable
          - compareTo(object ob), natural ordering of the objects , orders only 1 field
        - Comparator
          - compare(Object ob1, Object ob2), custom ordering

of the objects, allows ordering multiple fields
- LinkedHashSet - ordered, maintains insertion order, follows Hashing technique
- Queue - FIFO
-Deque - doubly queue, operation on both sides
- PriorityQueue
- BlockingQueue

- Map - maintains data as key, value pair, not allows duplicate keys
- Hashtable - not allows null keys, values, synchronized
- HashMap - allows null keys and values , faster search with key(any type primitive or object)
- TreeMap - not allows null keys and allows null values
- LinkedHashMap - allows null keys and values

- Map.Entry
- key
- value

- Iterator -
- ListIterator - goes forward and backward
- forEach

- Arrays.asList(empArr) -- convert array to a list
- empList.toArray() -- convert list to array
- empSet.toArray() -- convert set to array
- convert array to set
1st convert array to list and then convert to set
Set set = new HashSet(Arrays.asList(empArr));

--The Collection framework represents a unified architecture for storing and manipulating a group of objects.
It has Interfaces and its implementations, i.e., classes and Algorithms(search, sort, aggregate etc)
Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

-- Advantages:
Code reusuability
Enables to perform operations on group of objects
improves program speed and performance

-- Earlier , these 2 container classes were there

Dictionary (key/value)
Vector (Dynamic array)

Java Collection framework provides many interfaces (List, Queue, Deque, Set) and
classes (ArrayList, LinkedList, Vector, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

--Hierarchy of Collection Framework [capitals are Interfaces, Togglecase is class]
ITERABLE <-- COLLECTION <-- LIST,QUEUE,SET
LIST <-- ArrayList, LinkedList, Vector, Stack
QUEUE <-- PriorityQueue
QUEUE <-- DEQUE <-- ArrayDeque
SET <-- HashSet,LinkedHashSet
SET <-- SORTEDSET <-- TreeSet

MAP <-- SORTEDMAP <-- TreeMap
MAP <-- HasMap <-- LinkedHashMap

--Collection Interface
The Collection interface is the interface which is implemented by all the classes in the collection framework.
It is the root interface in the collection hierarchy.
This interface is basically used to pass around the collections and manipulate them where the maximum generality is desired.
It declares the methods that every collection will have.
Methods of Collection interface
boolean add(E e)
boolean addAll(Collection<? extends E> c)
public boolean remove(Object element)
public boolean removeAll(Collection<?> c)
default boolean removeIf(Predicate<? super E> filter)
public boolean retainAll(Collection<?> c)
public int size()
public void clear()
public boolean contains(Object element)
public boolean containsAll(Collection<?> c)
public Iterator iterator()
public Object[] toArray()
public <T> T[] toArray(T[] a)
public boolean isEmpty()
default Stream<E> parallelStream()
default Stream<E> stream()

default Spliterator<E> spliterator()
public boolean equals(Object element)
public int hashCode()

--Iterator interface
Iterator interface provides the facility of iterating the elements in a forward direction only.
Methods of Iterator interface :
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
The Iterable interface is the root interface for all the collection classes.
The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
It contains only one abstract method. i.e.,
Iterator<T> iterator()
It returns the iterator over the elements of type T.

-- Collections class
Collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

Java Collection class supports the polymorphic algorithms that operate on collections.
Java Collection class throws a NullPointerException if the collections or class objects provided to them are null.

ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime.
The important points about Java ArrayList class are:
Java ArrayList class can contain duplicate elements.
Java ArrayList class maintains insertion order.
Java ArrayList class is non synchronized.
Java ArrayList allows random access because array works at the index basis.
In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

--Hierarchy of ArrayList class

ITERABLE <-- COLLECTION <-- LIST<-- AbstractList <-- ArrayList

-- Constructors of ArrayList
Constructor Description
ArrayList() It is used to build an empty array list.
ArrayList(Collection<? extends E> c) It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity) It is used to build an array list that has the specified initial capacity.

--Use of forEach:
There are two forEach() method in Java 8, one defined inside Iterable, and the other inside java.util.stream.Stream class.
If the purpose of forEach() is just iteration then you can directly call it like list.forEach() or set.forEach() but
if you want to perform some operations like filter or map then it better first get the stream and then perform that operation and finally call forEach() method.

--Java LinkedList class uses a doubly linked list to store the elements.
Java LinkedList class can contain duplicate elements.
Java LinkedList class maintains insertion order.
Java LinkedList class is non synchronized.
In Java LinkedList class, manipulation is fast because no shifting needs to occur.
Java LinkedList class can be used as a list, stack or queue.
--Doubly Linked List
In the case of a doubly linked list, we can add or remove elements from both sides.

Difference between Array and ArrayList
/*
Array ArrayList
Array is static in size. ArrayList is dynamic in size.
An array is a fixed-length data structure. ArrayList is a variable-length data structure. It can be resized itself when needed.
Must provide size while initializing Not mandatory
Fast operation because of the fixed sizze The Resize operation slows down the ArrayList in performnace
An array can store both objects and primitives type. We cannot store primitive type in ArrayList. It automatically converts primitive type to object.
Array can be multi-dimensional. ArrayList is always single-dimensional.

--Similarities
Array and ArrayList both are used for storing elements.
Array and ArrayList both can store null values.
They can have duplicate values.
They do not preserve the order of elements.
*/

/*
-- Difference between ArrayList and LinkedList
ArrayList
LinkedList
1) ArrayList internally uses a dynamic array to store the elements.
LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array.
Manipulation(insertions,updations) with LinkedList is faster than ArrayList because it uses a
doubly linked list, so no bit
shifting is required in memory.
If any element is removed from the array, all the bits are shifted in memory.
3) An ArrayList class can act as a list only because it implements List only.
LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing(searching) data.
LinkedList is better for manipulating(add/remove) data.

-- Difference between ArrayList and Vector
Both are same in everything except, Vector is synchronized(thread safe), but ArrayList not
ArrayList has fastermrandom access, for frequent reads

-- When to use ArrayList and LinkedList in Java
ArrayList is better to access data(search operation) time complexity to access elements O(1)
LinkedList is better to manipulate(add and remove operations) data. time complexity to access elements O(n/2)

-- COMPARE ARRAYLISTS
There are following ways to compare two ArrayList in Java:
Java equals() method
Java removeAll() method
Java retainAll() method
Java ArrayList.contains() method
Java contentEquals() method

Java Stream interface : System.out.print("Common elements: "
+firstList.stream().filter(secondList::contains).collect(Collectors.toList()));

-- REVERSE ArrayList
 Collections.reverse(lIST);

-- convert ArrayList to Array AND Array to ArrayList
System.out.println("Converting ArrayList to Array");
String[] item = fruitList.toArray(new String[fruitList.size()]);

System.out.println("Converting Array to ArrayList");
List<String>l2 = new ArrayList<>();
l2 = Arrays.asList(item);

-- Set -Does not allow duplicates, sorted, no imdex
- HahsSet - faster search, works based on hashing techni
- TreeSet -
  - Comparable
    - compareTo(Object obj)
  - Comparator
    - compare(Object obj1, Object obj2
- LinkedHashSet - ordered, maintains insertion order, follows Hashing technique

-- HashSet:
HashSet class is used to create a collection that uses a hash table for storage. no duplicates, no order, allows null
ITERABLE <-- COLLECTION <-- SET <-- AbstractSet <-- HashSet
HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
HashSet stores the elements by using a mechanism called hashing.
HashSet contains unique elements only.
HashSet allows null value.
HashSet class is non synchronized.
HashSet is the best approach for search operations.
The initial default capacity of HashSet is 16, and the load factor is 0.75.

-- Difference between List and Set
List Set
1. The List is an ordered sequence. 1. The Set is an unordered sequence.
2. List allows duplicate elements 2. Set doesn’t allow duplicate elements.

3. Elements by their position can be accessed, index based. 3. Position access to elements is not allowed, not index based.
4. Multiple null elements can be stored. 4. Null element can store only once.
5. List implementations are ArrayList, LinkedList, Vector, Stack 5. Set implementations are HashSet, LinkedHashSet.

-- LinkedHashSet:
LinkedHashSet class is a Hashtable and Linked list implementation of the set interface.
ITERABLE <-- COLLECTION <-- SET <-- AbstractSet <-- HashSet <-- LinkedHashSet

-- TreeSet:
TreeSet class implements the Set interface that uses a tree for storage. no duplicates, no null, by default ascending order
ITERABLE <-- COLLECTION <-- SET <-- SORTEDSET <-- NAVIGABLESET <-- TreeSet
Java TreeSet class contains unique elements only like HashSet.
Java TreeSet class access and retrieval times are quiet fast.
Java TreeSet class doesn't allow null element.
Java TreeSet class is non synchronized.
Java TreeSet class maintains ascending order.

-- Hashing
The hashcode and equals methods must be same all the time
It is allowed if equals is false and hashcode is true but not equals is true and hashcode is false
Whenever equals is true, hashcode must be true

——

Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields
to indicate some additional information which can be used by java compiler and JVM.
It is an alternative option for XML and Java marker interfaces.
1. Built in annotations
2. Custom Annotations

1. Built in annotations

-- builtin annotations used in java code
@Override
@SuppressWarnings
@Deprecated
-- builtin annotations used in custom annotation
@Target
@Retention
@Inherited
@Documented

@Override : To assure that sub class method is overriding the parent class method.
Sometimes, we does the silly mistake such as spelling mistakes etc.
So, it is better to mark @Override annotation that provides assurity that method is overridden.
@SuppressWarnings : is used to suppress warnings issued by the compiler.

@Deprecated : It marks that this method is deprecated so compiler prints warning.
It informs user that it may be removed in the future versions. So, it is better not to use such methods.

2. Custom annotations:
Custom annotations or Java User-defined annotations are easy to create and use. The @interface element is used to declare an annotation.

Points to remember for java custom annotation signature
Method should not have any throws clauses
Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
Method should not have any parameter.
We should attach @ just before interface keyword to define annotation.
It may assign a default value to the method.

— How built-in annotaions are used in real scenario?
In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation.
Creating and Accessing annotation is performed by the implementation provider.
On behalf of the annotation, java compiler or JVM performs some additional operations.

— Types of Annotation
1. Marker Annotation: An annotation that has no method, is called marker annotation.

Example: @interface MyAnnotation{}
The @Override and @Deprecated are marker annotations.

2. Single-Value Annotation: An annotation that has one method, is called single-value annotation

```
Example: @interface MyAnnotation{
        int value() default 0;  //  we can give default values also
        }
```

How to apply Single-Value Annotation:

```
@Myannotation(value=10)
```

3. Multi-Value Annotation: An annotation that has more than one method, is called Multi-Value annotation.

```
Example: @interface Myannotation{
        int value1();
        String value2();
        String value3();
        }
```

How to apply Multi-Value Annotation

```
@MyAnnnotation(value1=20,value2="Hi" ,value3="Welcome")
```

— Builtin annotations used in custom annotation are

- @Target
- @Retention
- @Inherited
- @Documented

1. @Target : @Target tag is used to specify at which type the annotation is used. java.lang.annotations.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example to specify annotation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
```

```
int value1;
String value2;
}

Example to specify annotation for a class,method,field
@Target({ElementType.TYPE,ElementType.FIELD,ElementType.METHOD})
@interface MyAnnotation {
int value1;
String value2;
}
```

2. @Retention
@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the .class file, available to java compiler but not to JVM . It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM .

```
Example to specify the RetentionPolicy
@RETENTION(RetentionPolicy.RUNTIME)
@TARGET(ElementTypeTYPE)
@interface MyAnnotation
{
int valye1;
String value2;
}
```

3. @Inherited
By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

4. @Documented
The @Documented Marks the annotation for inclusion in the documentation.

———

-- Serialization is the process of converting an object into a stream of bytes to store or save the object or transmit it to memory, a database, or a file.
Its main purpose is to save or store the state(value) of an object in order to be able

to recreate it when needed.
The reverse process is called deserialization

Note: By default, its not possible to store object. Its possible only if the class implements Serializable interface

-- transient keyword is used to deserialize the variable .
When an object is deserialized the transient variables retains the default value depending on the type of variable declared and hence will not lose its original value.

-- Volatile keyword is used to modify the value of a variable by different threads
Every thread has its own private memory where it stores a copy of the variable.
When a variable is declared as volatile, it lets JVM know that any thread accessing the variable must always sync with its local copy with
the copy in the main memory(heap memory).

The volatile variables are always visible to other threads.
The copy of volatile variable is stored in the main memory, so every time a thread access the variable even for reading purpose,
It can be used as an alternative way of achieving synchronization in Java.
You can use the volatile keyword with variables. Using volatile keyword with classes and methods is illegal.
the local copy is updated each time from the main memory.
Ex: count variable to hold

Note: Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory.
Volatile can only be applied to instance variables, which are of type object or private.
A volatile object reference can be null.

Ex:

```
class SaveObject implements Serializable
{
   int x;
}

public class TestSerialize {
   public static final long serialVersionID = -34523444; // optional as a reference while deserializing the object
   public static void main(String args[]) throws Exception{

      SaveObject saveObj = new SaveObject();
```

```
        saveObj.x = 20;

        File f = new File("obj.txt");
        FileOutputStream fos = new FileOutputStream(f);
        ObjectOutputStream oos = new ObjectOutputStream(fos); //
ObjectOutputStream  does serializaiton
        oos.writeObject(saveObj); // save the object into the file , it is allowed only if
the class of the object implements Serializable

        FileInputStream fis = new FileInputStream(f);
        ObjectInputStream ois = new ObjectInputStream(fis);
        SaveObject objRetrieved = (SaveObject) ois.readObject(); // read as an object
        System.out.println(objRetrieved.x); // the data can be retrived from the  file
where obj is saved.

    }
}
```

MultiThreading & Synchronization

Multithreading
- process vs Thread
- Thread Lifecycle
  - New
  - Runnable
  - Running
  - Wait
  - Sleep
  - blocked
  - Terminated/Dead
- Ways to create Thread
  - Extending Thread Class
  - Implementing Runnable Interface
  - Implements Callable interface
    - override call method
    - call() returns a value after execution
    - create ExecutorService instance with threadPool (fixed, caches, single)
      ExecutorService exs = Executors.newFixedThreadPool(5);
    - submit the instance of the class implementing callable interface or runnable interface, create future task

Future<String> f1 = exs.submit(c1);
- returns reference of Future object to refer the thread submitted
- get the value returned by thread from future
System.out.println(f2.get());
- shutdown the ExecutorService

- Thread Constructs
  - sleep (invoking on thread, can be interrupted, will not release lock)
  - wait (invoking on object, releases lock once notified)
  - notify (notifies single thread, if 4 threads are waiting, it wakes up the any thread)
  - notifyall (it wakes up all 4 threads)
  - join (waits until it dies)
  - yield (gives slot for other thread and goes back to runnable,)

- Thread Synchronization
  - all operations should happen in Atomic way, Visible, Ordered
  - at a time only 1 thread is allowed to access a synchronised method or block
  - Synchronised block (mostly preferred, since it will not lock the entire method for long time)
    synchronised(account) { // acquire lock, blocked state
    }
  - synchronised method
    synchronised void fundTransfer() {}

- how to make collections thread safe
  - Collections.synchronised(list)
  - Concurrent Collections(use thread safe collections)
    - BlockingQueue
    - ConcurrentMap
    - ConcurrentHashMap
    - ConcurrentSkipListMap

- Thread contention
  Multiple threads trying to access a locked resource

- Deadlock:
  if 2 threads (t1,t2), resources r1, r2 , t1 trying r1(acaquired lock by t2) and t2 trying r2(acquired lock by t1),
  cyclic dependency happens, this is deadlock

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

1) Process-based Multitasking (Multiprocessing)
Each process has an address in memory, runs independently. In other words, each process allocates a separate memory area.
A process is heavyweight.
Cost of communication between the process is high.
Switching from one process to another requires some time for saving and loading registers , memory maps, updating lists, etc.
2) Thread-based Multitasking (Multithreading)
Threads share the same address space(stack).
A thread is lightweight.
Cost of communication between the thread is low.

--Advantages of Java Multithreading
1) It doesn't block the user because threads are independent and you can perform multiple operations at the 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 an exception occurs in a single thread.
4) To utilize the complete CPU(having diff number of core orocessors ex : quad, 6, 8 core)
5) To send asynchronic requests (In android phones,)
6) In web applications, multithreading is used.
7) In Gaming

-- Thread : A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
Only one thread at a time can run in a single process.

-- Life cycle of a Thread
New ： the thread is in new state if an instance is created for the thread class but before the invocation of start() method
Runnable : Once the start() method is invoked, but hread scheduler has not selected it
Running : the thread scheduler has selected it
sleep:
wait:
Non-Runnable (Blocked)： not eligible to run , but still alive
Terminated : dead state when its run() method exits.

-- How to create thread
There are three ways to create a thread:
1. By extending Thread class
2. By implementing Runnable interface.

3. Implements Callable

1. Extending Thread class

Java provides Thread class to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

--Java Thread Methods : start(), run(), sleep(), currentThread(), join(), destroy() etc

--Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

A new thread starts(with new callstack).

The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run() method will run.

Note: run() method must be defined in the class which extends Thread class

2. Implementing 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.

Note: The Runnable interface is used to create Thread in scenarios like below

```
class A{ }
class Mythread extends A, Thread {  // throws error because java doesn't support multiple inheritance
}
```

Hence this issue can be handled using interface Runnable

class Mythread extends A, implements Runnable (its a functional interface)

3. Implements Callable

-- Callable interface

To return an object after completing its execution

It has call() method like run() method in Runnable

Using Executors class, can create a thread pool and create Future tasks and submit the tasks to the threads which implement Callable interface

-- Executor Framework
Executor, ExecutorService, Executors are part of Java's Executor Framework
Since the creation and management of Threads are expensive and the operating system also imposes restrictions
on how many Threads an application can spawn, it's a good idea is to use a pool of threads to execute tasks in parallel,
instead of creating a new thread every time a request comes in. This not only improves the response time of the application
but also prevent resource exhaustion errors like "java.lang.OutOfMemoryError: unable to create new native thread".

Read more: https://javarevisited.blogspot.com/2017/02/difference-between-executor-executorservice-and-executors-in-java.html#ixzz7Y2aLet1d

-- Executor - It is a core interface used to execute the submitted Runnable tasks.
An Executor is normally used instead of explicitly creating threads.
It has only execute() method.
-- ExecutorService - It is an extension to Executor interface which provides a facility for returning a Future object and terminate,
or shut down the thread pool, and the ability to wait for and look at the status of jobs.
Once the shutdown is called, the thread pool will not accept new tasks but complete any pending task.
It also provides a submit() method which extends Executor.execute() method and returns a Future.
Future<T> submit(Callable<T> task);
Future<?> submit(Runnable task);
The Future object provides the facility of asynchronous execution, which means you don't need to wait until the execution
finishes, you can just submit the task and go around, come back and check if the Future object has the result,
if the execution is completed then it would have a result which you can access by using the Future.get() method.
Just remember that this method is a blocking method i.e. it will wait until execution finish and the result is available
if it's not finished already.

-- Executors - It is a utility class similar to Collections , which provides factory methods
to create diff types of thread pools(ex:fixed and cached thread pools)

Ex: ExecutorService exs = Executors.newFixedThreadPool(5); // creates 5 threads
Future<String> f1 = exs.submit(c1); // it can accept both Callable/Runnable tasks
System.out.println(f1.get());

Advantage: Using thread pool, the threads can be reused once they finish their task.

-- Thread Scheduler in Java
Thread scheduler in java is the part of the JVM that decides which thread should run.
There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
Only one thread at a time can run in a single process.
The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

-- Preemptive scheduling, the highest priority task executes until it enters the waiting or dead states
-- Time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks.
The scheduler then determines which task should execute next, based on priority and other factors.

-- Sleep method : to sleep a thread for the specified amount of time.
As you know well that at a time only one thread is executed.
If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

-- Can we start a thread twice
No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown.
In such case, thread will run once but for second time, it will throw exception.

-- What if we call run() method directly instead start() method?
Each thread starts in a separate call stack.
Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.
It means the thread object is treated as normal class object and thread fucntionality(context switching) does not occur.

-- The join() method
The join() method waits for a thread to die.
In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

-- Naming Thread
The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on.
public String getName(): is used to return the name of a thread.
public void setName(String name): is used to change the name of a thread.

-- Current Thread
The currentThread() method returns a reference of currently executing 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.Default priority of a thread is 5 (NORM_PRIORITY)

3 constants defined in Thread class:
public static int MIN_PRIORITY : value 1
public static int NORM_PRIORITY : value 5
public static int MAX_PRIORITY :value 10

-- Daemon Thread
It is a service provider thread runs automaticaaly(example gc, finalizer)
Its life depends on user threads. when user thread dies, jVM terminates this thread automatically.
It is a low priority thread.
Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.

--jconsole tool
You can see all the detail by typing the jconsole in the command prompt.
The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

-- Thread pool: It represents a group of worker threads that are waiting for the job and reuse many times.
In case of thread pool, a group of fixed size threads are created.

A thread from the thread pool is pulled out and assigned a job by the service provider.
After completion of the job, thread is contained in the thread pool again.

Advantage of Java Thread Pool
Better performance It saves time because there is no need to create new thread.

Real time usage
It is used in Servlet and JSP where container creates a thread pool to process the request.
It is also used in Executor framework in Multithreading

-- Thread Group
A ThreadGroup represents a set of threads.
Grouping multiple threads in a single object so that we can suspend, resume or interrupt group of threads by a single method call.

A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

-- Shutdown Hook
The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly.
Performing clean resource means closing log file, sending some alerts or something else.
So if you want to execute some code before JVM shuts down, use shutdown hook.

The JVM shuts down when:
user presses ctrl+c on the command prompt
System.exit(int) method is invoked
user logoff
user shutdown etc.

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine.
The object of Runtime class can be obtained by calling the static factory method(The method that returns the instance of a class is known as factory method.) getRuntime().
The shutdown sequence can be stopped by invoking the halt(int) method of Runtime class.

-- How to perform single task by multiple threads?
If you have to perform single task by many threads, have only one run() method.

-- How to perform multiple tasks by multiple threads (multitasking in multithreading)?
If you have to perform multiple tasks by multiple threads,have multiple run() methods.

-- Garbage Collection : destroy the unused objects.
In java, garbage means unreferenced objects.
Garbage Collection is process of reclaiming the runtime unused memory automatically.
To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.
gc() method - invoked by Garbage collector(Daemon thread), found in System and Runtime classes, cleans up the objects which are created using new keyword
finalize() method - invoked each time before the object is garbage collected, found in Object class, cleans up the objects which are not created using new keyword

Advantage of Garbage Collection
It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts

How can an object be unreferenced?
By nulling the reference
By assigning a reference to another
By anonymous object etc.

1) By nulling a reference:
Employee e=new Employee();
e=null;
2) By assigning a reference to another:
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
3) By anonymous object:
new Employee();

-- Runtime class : to interact with java runtime environment
Java Runtime class provides methods to execute a process, invoke GC, get total

and free memory etc.
There is only one instance of java.lang.Runtime class is available for one java application.
The Runtime.getRuntime() method returns the singleton instance of Runtime class.
Methods:
static Runtime getRuntime()
void addShutdownHook(Thread hook)
long freeMemory()
long totalMemory()

Synchronization

-- Synchronization
All operations should happen in Atomic way, Visible, Ordered
The capability to control the access of multiple threads to any shared resource.
To allow only one thread to access the shared resource.

Uses
To prevent thread interference.
To prevent consistency problem.

Types of Synchronization
Process Synchronization
Thread Synchronization

-- Thread Synchronization
There are two types of thread synchronization mutual exclusive and inter-thread communication.
1. Mutual Exclusive ： Mutual Exclusive helps keep threads from interfering with one another while sharing data.It can be achieved by using the following three ways
    Synchronized method.
    Synchronized block.
    Static synchronization.
2. Cooperation (Inter-thread communication in java)

-- Synchronized Method
If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.
At a time only 1 thread is allowed to access synchronized method
---
-- Synchronized block
It can be used to perform synchronization on any specific resource of the method.
Scenario: Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.
If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember
Synchronized block is used to lock an object for any shared resource.
Scope of synchronized block is smaller than the method.
A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
The system performance may degrade because of the slower working of synchronized keyword.
Java synchronized block is more efficient than Java synchronized method.

--Static Synchronization
If you make any static method as synchronized, the lock will be on the class not on object.

-- Problem without static synchronization
Suppose there are two objects of a shared class (e.g. Table) named object1 and object2.
In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4
because t1 and t2 both refers to a common object that have a single lock.
But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires
another lock. We don't want interference between t1 and t3 or t2 and t4.
Static synchronization solves this problem.

--Synchronized block on a class lock:
The block synchronizes on the lock of the object denoted by the reference .class name .class.
A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

```
static void printTable(int n) {
    synchronized (Table.class) {       // Synchronized block on class A
        // ...
```

```
    }
}
```

-- Deadlock:
if 2 threads (t1,t2), resources r1, r2 , t1 trying r1(acaquired lock by t2) and t2 trying r2(acquired lock by t1),
cyclic dependency happens, this is deadlock

-- Thread contention
Multiple threads trying to access a locked resource

More Complicated Deadlocks
A deadlock may also include more than two threads
Thread 1 locks A, waits for B
Thread 2 locks B, waits for C
Thread 3 locks C, waits for D
Thread 4 locks D, waits for A
Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

-- 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.
It is implemented by following methods of Object class:
1. wait() : It invokes the object. It releases the current thread and waits until it gets notified (by invoking notify() or notifyall() methods)
by any other thread in for the same object
Note: The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise
it will throw exception.
2. notify() : It wakes up any of the single thread which is waiting in the same object's monitor. The choice is arbitrary.
3. notifyAll() : Wakes up all threads that are waiting on this object's monitor.

-- Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?
It is because they are related to lock and object has a lock.

-- Difference between wait and sleep?

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

-- Interrupting a Thread:
If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread,
breaks out the sleeping or waiting state throwing InterruptedException
If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt
the thread but sets the interrupt flag to true.

-- The 3 methods provided by the Thread class for interrupting a thread
public void interrupt()
public static boolean interrupted()
public boolean isInterrupted()

--isInterrupted : The isInterrupted() method returns the interrupted flag either true or false.
-- The static interrupted() method returns the interrupted flag, then it sets the flag to false if it is true.

ReEntrant Monitor
According to Sun Microsystems, Java monitors are reentrant means java thread can reuse the same monitor for
different synchronized methods if method is called from the method.