1. **Access Modifiers**

Access modifiers in java are used to control the visibility of a field, method, class and constructor.

* 1. **Private**

Private members cannot be inherited

Private fields cannot be used outside the class

Private methods cannot be used outside the class

Private inner class cannot be used outside the class

Outer class Cannot be private

* 1. **Default**

Visible within the package only, not visible outside the package.

Default field can be inherited within package.

Default field can be used within the package.

Default method can be used within the package.

Default inner class can be used within the package.

* 1. **Protected**

Protected fields and methods can be inherited to any sub class.

Protected field cannot be used outside the package directly.

Protected method cannot be used outside the package directly.

Protected inner class cannot be used outside the package.

* 1. **Public**

public fields or methods can be used anywhere

1. **Inner class**

* It can access all the members (data members and methods) of outer class including private
* It logically group classes and interfaces in one place only.
* Code Optimization: It requires less code to write
  1. **Member inner class**

A non-static class that is created inside a class but outside a method is called member inner class.

* Inner class that is accessing the private data member of outer class.
* Outer.Inner in=new Outer().new Inner(); 🡪 To call inner class methods and variables.
  1. **Anonymous inner class**

A class that have no name is known as anonymous inner class in java. It should be used if you must override method of class or interface. Java Anonymous inner class can be created by two ways using Class and interfaces.

* 1. **Local inner class**

A class that is created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

* Local variable can't be private, public or protected.
* Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible.
  1. **Static nested class**

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

* It can access static data members of outer class including private.
* Static nested class cannot access non-static (instance) data member or method.

1. **Object class**

Object class is parent class for all classes. It’s contains 11 methods

* public final Class getClass()
* public int hashCode()
* public boolean equals(Object obj)
* protected Object clone() throws CloneNotSupportedException
* public String toString()
* public final void notify()
* public final void notifyAll()
* public final void wait(long timeout)throws InterruptedException
* public final void wait(long timeout,int nanos)throws InterruptedException
* public final void wait()throws InterruptedException
* protected void finalize()throws Throwable

1. **Strings**

* String objects are immutable in java. That is, once you create String objects, you can’t modify them.
* StringBuffer and StringBuilder objects are mutable.
* String classes implement Serializable and CharSequence interface.
* Only String and StringBuffer objects are thread safe.
* StringBuilder objects are not thread safe.
* hashCode() and equals() methods are override only in String class but not in StringBuffer and StringBuilder classes
* There is no reverse() and delete() methods in String class. But, StringBuffer and StringBuilder have reverse() and delete() methods.
* Whenever you create a string object using string literal, that object is stored in the string constant pool and whenever you create a string object using new keyword, such object is stored in the heap memory.
* Two equal objects according to **equals**() method must return same hash code values.
* Intern method checks whether the String equals to this String Object is in the pool or not. If it is available, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

1. **Enums**

Enums in java are mainly used for grouping similar kind of constants as a one unit.

* Every constant of enum is public, static and final by default.
* Enums can have any number of fields. methods and constructors and Each constant will have their own copy of fields and methods.
* As every enum extends Enum class, it should not extend any other class.
* Enum constructors are private by default.
* All enum types are Comparable and Serializable by default.
* ordinal() method is used get the order of an enum constant in an enum type.

1. **Exceptions**

* In try block, keep those statements which may throw exceptions during run time.
* This block handles the exceptions thrown by try block.
* Whether exception is thrown or not and thrown exception is caught or not, this block will be always executed.
* The order of catch blocks, Sub classes of Exception must come first and super classes late.
* finally block overrides any return values from try and catch blocks.
* When a method is throwing unchecked type of exceptions, then you need not to mention it using throws keyword. But for a method throwing checked type of exceptions, you must declare it with throws keyword or enclose the statement which is throwing an exception in try-catch block.
* If super class method is not throwing any exceptions, then it can be overrided with any unchecked type of exceptions but cannot be overrided with checked type of exceptions.
* **ClassNotFoundException** is a run time exception which is thrown when an application tries to load a class at run time using Class.forName() or loadClass() or findSystemClass() methods and the class with specified name are not found in the classpath.
* **NoClassDefFoundError** is an error which is thrown when Java Runtime System tries to load the definition of a class and class definition is no longer available.

1. **Multithreading**

Multithreading in Java is a process of executing multiple threads simultaneously

It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

You can perform many operations together, so it saves time.

Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

**Thread:** A thread is a smallest unit of processing. Exception occurs in one thread, it doesn't affect other threads. It uses a shared memory area.

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| Void start() | It is used to start the execution of the thread.It does not return any value. IllegalThreadStateException - This exception throws if the start() method is called more than one times. |
| Void run() | It is used to do an action for a thread.It does not return any value. |
| Static void sleep() | It sleeps a thread for the specified amount of time.It does not return any value. IllegalArgumentException: If the value of millis is negative or the value of nanos is not in the range 0-999999.InterruptedException: If any thread has interrupted the current thread. |
| currentThread() | It returns a reference to the currently executing thread object. |
| join() | It waits for a thread to die. |
| getPriority() | It returns the priority of the thread.The thread's priority is in the range of 1 to 10. The default priority of a thread is 5. |
| setPriority() | It changes the priority of the thread. |
| getName() | It returns the name of the thread. |
| setName() | It changes the name of the thread. |
| yield() | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| Notify() | It is used to give the notification for only one thread which is waiting for a particular object. |
| notifyAll() | It is used to give the notification to all waiting threads of a particular object. |

If you want to execute some code before JVM shuts down, use shutdown hook. runnableObj. addShutdownHook(thread);

**Synchronization:** Synchronization in java is a strategy or a method to avoid thread interference and hence protecting the data from inconsistency. Through synchronization, we can make the threads to execute particular method or block in sync not simultaneously.

* synchronized keyword can be used with methods or blocks but not with the variables.
* Whenever an object is created to any class, an object lock is created and is stored inside the object
* Any thread wants to enter into synchronized methods or blocks of any object, they must acquire object lock associated with that object and release the lock after they are done with the execution
* To enter into static synchronized methods or blocks, threads have to acquire class lock associated with that class as static members are stored inside the class memory.

**Synchronized Blocks:** Some part of the method to be synchronized not the whole method. synchronized block takes one argument and it is called mutex. if synchronized block is defined inside non-static definition blocks like non-static methods, instance initializer or constructors, then this mutex must be an **instance** of that class. If synchronized block is defined inside static definition blocks like static methods or static initializer, then this mutex must be like **ClassName.class**.

Class Shared{

static void staticMethod()

synchronized (Shared.class){}

}

void NonStaticMethod(){

synchronized (this){}

}

void anotherNonStaticMethod(){

synchronized (new Shared()){}

}

}

* You can use synchronized keyword only with methods but not with variables, constructors, static initializer and instance initializers.
* Constructors, Static initializer and instance initializer can’t be declared with synchronized keyword, but they can contain synchronized blocks.
* Both static and non-static methods can use synchronized keyword. For static methods, thread need class level lock and for non-static methods, thread need object level lock.
* It is possible that both static synchronized and non-static synchronized methods can run simultaneously.

**Deadlock:** Deadlock in java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources(Locks) they hold.

* ThreadMXBean interface of java.lang.Management package used to detect the threads which have entered into deadlock

ThreadMXBean bean = ManagementFactory.getThreadMXBean();

long ids[] = bean.findMonitorDeadlockedThreads();

ThreadInfo threadInfo[] = bean.getThreadInfo(ids);

for (ThreadInfo threadInfo1 : threadInfo){

System.out.println(threadInfo1.getThreadName());

}

* If a thread calls notify() method and more than one threads are waiting for the object lock, then only one thread will be notified randomly.
* When a thread calls notifyAll() method on an object, it notifies all the threads which are waiting for this object lock. But, only one thread will acquire this object lock depending upon priority.
* When you call sleep() method on a thread, thread goes to sleep with holding the object lock with it. But, if you call wait() method, thread releases the object lock and goes for sleep. This is the main difference between wait() and sleep() methods.
* Thread interruption in java is a mechanism in which a thread which is either sleeping or waiting can be made to stop sleeping or waiting.

1. **Collections**

* Default initial capacity of an ArrayList is 10.
* ArrayList is not synchronized.
* Insertion order.
* capacity= n+n/2+1
* Array To ArrayList:
* Arrays.asList(array)
* Collections.addAll(list, array)
* Arrays.stream(array).collect(Collectors.toList())
* ArrayList To Array: list.toArray(array);
* HashMap:
* Initial Capacity Of HashMap 16.
* The default load factor is 0.75f.
* Threshold = 16 \* 0.75 = 12, based o threshold capacity will incerese 2^5

1. **Cloning**

Cloning is a process of creating an exact copy of an existing object in the memory. In java, clone() method of java.lang.Object class is used for cloning process.The objects which implement Cloneable interface are only eligible for cloning process.

The default version of clone() method creates the shallow copy of an object.

**Shallow Copy:**

If it contains any objects as fields then, only references to those objects are copied not the compete objects.

**Deep Copy:**

In the deep copy all fields of the original objects are copied exactly.

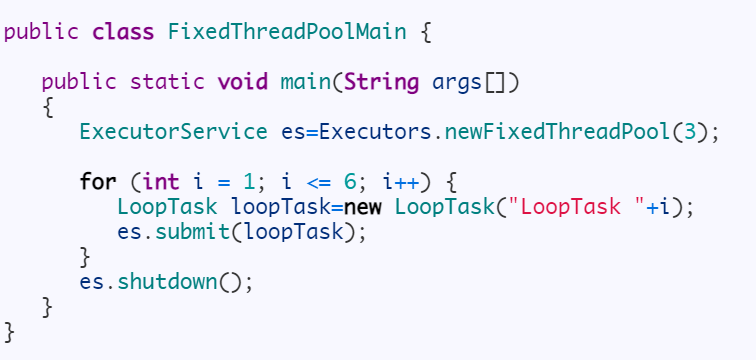
1. **HashCode**

* Whenever it is invoked on the same object more than once during an execution of a Java application, hashCode() must consistently return the same value.
* If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same value.

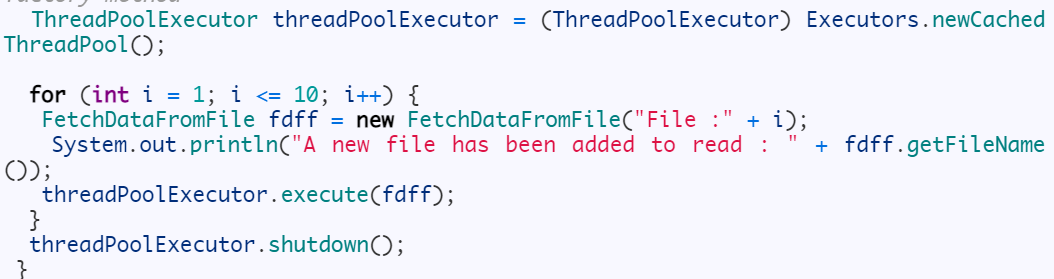
1. **Executor Framework**

This Executor API de-couples the execution of a task from the actual task to be executed with the help of an Executor.

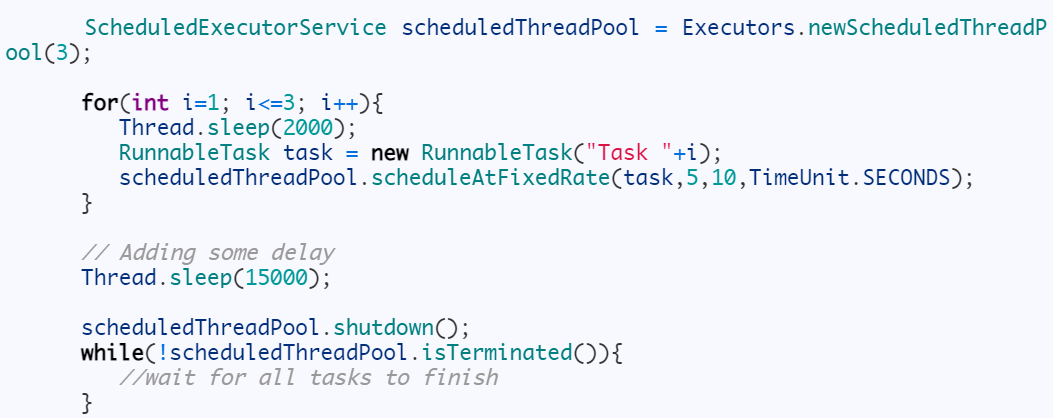
* **Executor** This interface is used to submit new task. It has a method called “execute”.
* **ExecutorService** This interface provides methods to manage lifecycle of tasks as well for executor.
* **ScheduledExecutorService** Provides methods for scheduling tasks at fixed intervals or with initial delay.
* **Executors** This class provides factory methods for creating thread pool.
* **newFixedThreadPool**: If all n threads are busy performing the task and additional tasks are submitted, then they will have to be in the queue until thread is available.

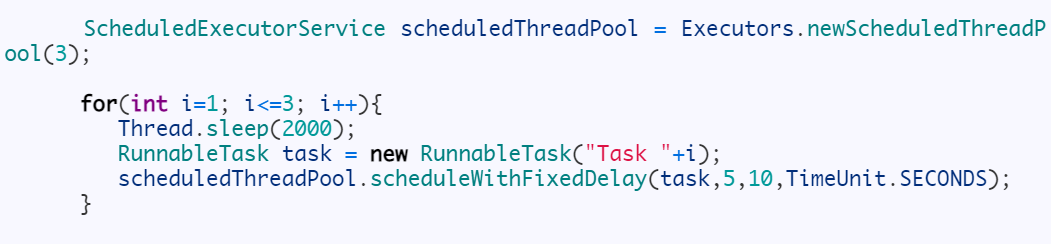


* **newCachedThreadPool**: This method returns an unbounded thread pool. If a thread has been unused for 1 mins(keepAliveTime), then it will tear it down.



* **newSingleThreadedExecutor**: This method returns an executor which is guaranteed to use the single thread.
* **newScheduledThreadPool**: This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

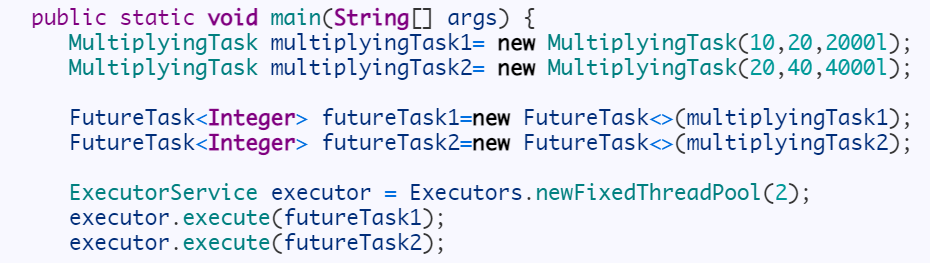




**Note:**You cannot change thread pool size of executors returned by **newSingleThreadExecutor** but you can change thread pool size of executors returned by **newFixedThreadPool(1)** by calling **setCorePoolSize()** of the class **ThreadPoolExecutor**.

**Future Task**

Class should implement callable interface it’s has call method. Future task will return the result back. We will get the value using get() method.



**ExecutorCompletionService**

ExecutorCompletionService returns futures objects based on completion order, so whichever task executes first, will be returned first. You just need to call **executorCompletionService.take()** to get completed Future object.

