**LECTURE NOTES**

**UNIT 4**

|  |
| --- |
| **NESTED CLASSES AND THREADS**  Static Nested Types, Inner Classes |
| Local Inner Classes, Anonymous Inner Classes |
| Inheriting Nested Types, Nesting in Interfaces, Implementation of Nested Types |
| Creating Threads , Using Runnable |
| Synchronization |
| Wait, notifyAll, and notify, Waiting and Notification, Thread Scheduling, |
| Deadlocks , Ending Thread Execution, volatile |
| Thread Management, Security, and ThreadGroup, Threads and Exceptions, debugging threads |

# Multithreading in Java

**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.

### Advantage 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.

### Multitasking

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

* Process-based Multitasking(Multiprocessing)
* Thread-based Multitasking(Multithreading)

### 1) Process-based Multitasking (Multiprocessing)

* Each process have its own address in memory i.e. each process allocates separate memory area.
* Process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

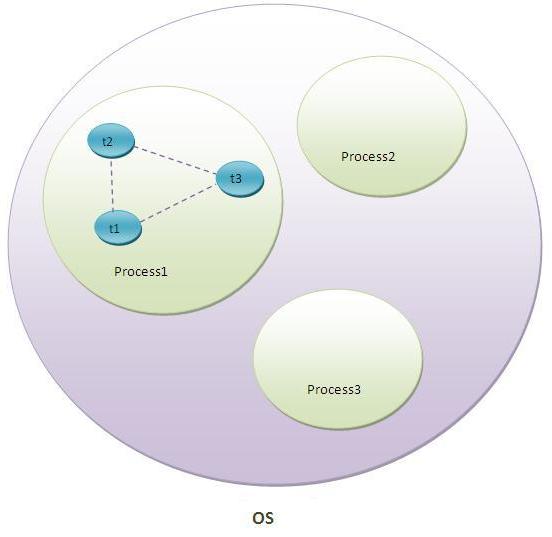
### 2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the thread is low.

### What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



|  |
| --- |
| As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.Life cycle of a Thread (Thread States) A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.  But for better understanding the threads, we are explaining it in the 5 states.  The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:   1. New 2. Runnable 3. Running 4. Non-Runnable (Blocked) 5. Terminated   thread life cycle in java 1) New The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

### 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated

A thread is in terminated or dead state when its run() method exits.

## 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.Thread class extends Object class and implements Runnable interface. |

### Commonly used Constructors of Thread class:

|  |
| --- |
| * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |

### Commonly used methods of Thread class:

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |

### 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(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

### 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. |

### 1)By extending Thread class:

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

Output:thread is running...

### Who makes your class object as thread object?

|  |
| --- |
| **Thread class constructor** allocates a new thread object.When you create object of Multi class,your class constructor is invoked(provided by Compiler) fromwhere Thread class constructor is invoked(by super() as first statement).So your Multi class object is thread object now. |

### 2)By implementing the Runnable interface:

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

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. |

# 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.

## Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under 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 in java

The java sleep() method of Thread class is used to sleep a thread for the specified milliseconds of time.

# Syntax of sleep() method in java

|  |
| --- |
| The Thread class provides two methods for sleeping a thread:   * public static void sleep(long miliseconds)throws InterruptedException * public static void sleep(long miliseconds, int nanos)throws InterruptedException |

## Example sleep method in java

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

Output:1

1

2

2

3

3

4

4

5

5

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.

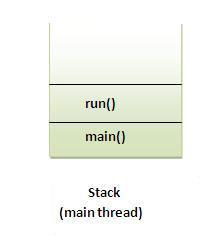
# What if we call run() method directly instead start() method?

|  |
| --- |
| * 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. |

1. **class** TestCallRun1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestCallRun1 t1=**new** TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

 ***Problem if you direct call run() method***

1. **class** TestCallRun2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestCallRun2 t1=**new** TestCallRun2();
10. TestCallRun2 t2=**new** TestCallRun2();
12. t1.run();
13. t2.run();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun2)

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

# 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. |

### Syntax:

|  |
| --- |
| public void join()throws InterruptedException |
| public void join(long milliseconds)throws InterruptedException |

***Example of join() method***

1. **class** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod1)

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

|  |
| --- |
| As you can see in the above example,when t1 completes its task then t2 and t3 starts executing. |

***Example of join(long miliseconds) method***

1. **class** TestJoinMethod2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod2 t1=**new** TestJoinMethod2();
12. TestJoinMethod2 t2=**new** TestJoinMethod2();
13. TestJoinMethod2 t3=**new** TestJoinMethod2();
14. t1.start();
15. **try**{
16. t1.join(1500);
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod2)

Output:1

2

3

1

4

1

2

5

2

3

3

4

4

5

5

|  |
| --- |
| In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing. |

### getName(),setName(String) and getId() method:

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

1. **class** TestJoinMethod3 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
12. t1.start();
13. t2.start();
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod3)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changling name of t1:Sonoo Jaiswal

running...

### The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax:

|  |
| --- |
| public static Thread currentThread() |

***Example of currentThread() method***

1. **class** TestJoinMethod4 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. }
6. **public** **static** **void** main(String args[]){
7. TestJoinMethod4 t1=**new** TestJoinMethod4();
8. TestJoinMethod4 t2=**new** TestJoinMethod4();
10. t1.start();
11. t2.start();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestJoinMethod4)

Output:Thread-0

Thread-1

# Naming a thread:

|  |
| --- |
| The Thread class provides methods to change and get the name of a thread.   1. **public String getName():** is used to return the name of a thread. 2. **public void setName(String name):** is used to change the name of a thread. |

## Example of naming a thread:

1. **class** TestMultiNaming1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultiNaming1 t1=**new** TestMultiNaming1();
7. TestMultiNaming1 t2=**new** TestMultiNaming1();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
11. t1.start();
12. t2.start();
14. t1.setName("Sonoo Jaiswal");
15. System.out.println("After changing name of t1:"+t1.getName());
16. }
17. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming1)

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:Sonoo Jaiswal

running...

## The currentThread() method:

|  |
| --- |
| The currentThread() method returns a reference to the currently executing thread object. |

### Syntax of currentThread() method:

* **public static Thread currentThread():** returns the reference of currently running thread.

### Example of currentThread() method:

1. **class** TestMultiNaming2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println(Thread.currentThread().getName());
4. }
5. }
6. **public** **static** **void** main(String args[]){
7. TestMultiNaming2 t1=**new** TestMultiNaming2();
8. TestMultiNaming2 t2=**new** TestMultiNaming2();
10. t1.start();
11. t2.start();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiNaming2)

Output:Thread-0

Thread-1

# 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:

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_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. |

### Example of priority of a Thread:

1. **class** TestMultiPriority1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running thread name is:"+Thread.currentThread().getName());
4. System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
6. }
7. **public** **static** **void** main(String args[]){
8. TestMultiPriority1 m1=**new** TestMultiPriority1();
9. TestMultiPriority1 m2=**new** TestMultiPriority1();
10. m1.setPriority(Thread.MIN\_PRIORITY);
11. m2.setPriority(Thread.MAX\_PRIORITY);
12. m1.start();
13. m2.start();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMultiPriority1)

Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

# 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.

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.

# 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.

### Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

### 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. |

# Simple example of Daemon thread in java

*File: MyThread.java*

1. **public** **class** TestDaemonThread1 **extends** Thread{
2. **public** **void** run(){
3. **if**(Thread.currentThread().isDaemon()){//checking for daemon thread
4. System.out.println("daemon thread work");
5. }
6. **else**{
7. System.out.println("user thread work");
8. }
9. }
10. **public** **static** **void** main(String[] args){
11. TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread
12. TestDaemonThread1 t2=**new** TestDaemonThread1();
13. TestDaemonThread1 t3=**new** TestDaemonThread1();
15. t1.setDaemon(**true**);//now t1 is daemon thread
17. t1.start();//starting threads
18. t2.start();
19. t3.start();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread1)

#### Output

daemon thread work

user thread work

user thread work

#### *Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.*

*File: MyThread.java*

1. **class** TestDaemonThread2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("Name: "+Thread.currentThread().getName());
4. System.out.println("Daemon: "+Thread.currentThread().isDaemon());
5. }
7. **public** **static** **void** main(String[] args){
8. TestDaemonThread2 t1=**new** TestDaemonThread2();
9. TestDaemonThread2 t2=**new** TestDaemonThread2();
10. t1.start();
11. t1.setDaemon(**true**);//will throw exception here
12. t2.start();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread2)

Output:exception in thread main: java.lang.IllegalThreadStateException

# Java Thread Pool

**Java Thread pool** 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.

#### Example of Java Thread Pool

Let's see a simple example of java thread pool using ExecutorService and Executors.

*File: WorkerThrad.java*

1. **import** java.util.concurrent.ExecutorService;
2. **import** java.util.concurrent.Executors;
3. **class** WorkerThread **implements** Runnable {
4. **private** String message;
5. **public** WorkerThread(String s){
6. **this**.message=s;
7. }
8. **public** **void** run() {
9. System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
10. processmessage();//call processmessage method that sleeps the thread for 2 seconds
11. System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
12. }
13. **private** **void** processmessage() {
14. **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) { e.printStackTrace(); }
15. }
16. }

*File: JavaThreadPoolExample.java*

1. **public** **class** TestThreadPool {
2. **public** **static** **void** main(String[] args) {
3. ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
4. **for** (**int** i = 0; i < 10; i++) {
5. Runnable worker = **new** WorkerThread("" + i);
6. executor.execute(worker);//calling execute method of ExecutorService
7. }
8. executor.shutdown();
9. **while** (!executor.isTerminated()) {   }
11. System.out.println("Finished all threads");
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThreadPool)

[download this example](http://www.javatpoint.com/src/multi/threadpool.zip)

#### Output:

pool-1-thread-1 (Start) message = 0

pool-1-thread-2 (Start) message = 1

pool-1-thread-3 (Start) message = 2

pool-1-thread-5 (Start) message = 4

pool-1-thread-4 (Start) message = 3

pool-1-thread-2 (End)

pool-1-thread-2 (Start) message = 5

pool-1-thread-1 (End)

pool-1-thread-1 (Start) message = 6

pool-1-thread-3 (End)

pool-1-thread-3 (Start) message = 7

pool-1-thread-4 (End)

pool-1-thread-4 (Start) message = 8

pool-1-thread-5 (End)

pool-1-thread-5 (Start) message = 9

pool-1-thread-2 (End)

pool-1-thread-1 (End)

pool-1-thread-4 (End)

pool-1-thread-3 (End)

pool-1-thread-5 (End)

Finished all threads

# Synchronization in Java

Synchronization in java is the capability of control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

### Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

## Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

## Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. static synchronization.
2. Cooperation (Inter-thread communication in java)

### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

### Understanding the concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

### Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

1. Class Table{
3. **void** printTable(**int** n){//method not synchronized
4. **for**(**int** i=1;i<=5;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){System.out.println(e);}
9. }
11. }
12. }
14. **class** MyThread1 **extends** Thread{
15. Table t;
16. MyThread1(Table t){
17. **this**.t=t;
18. }
19. **public** **void** run(){
20. t.printTable(5);
21. }
23. }
24. **class** MyThread2 **extends** Thread{
25. Table t;
26. MyThread2(Table t){
27. **this**.t=t;
28. }
29. **public** **void** run(){
30. t.printTable(100);
31. }
32. }
34. **class** TestSynchronization1{
35. **public** **static** **void** main(String args[]){
36. Table obj = **new** Table();//only one object
37. MyThread1 t1=**new** MyThread1(obj);
38. MyThread2 t2=**new** MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }

Output: 5

100

10

200

15

300

20

400

25

500

### 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.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

1. //example of java synchronized method
2. **class** Table{
3. **synchronized** **void** printTable(**int** n){//synchronized method
4. **for**(**int** i=1;i<=5;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){System.out.println(e);}
9. }
11. }
12. }
14. **class** MyThread1 **extends** Thread{
15. Table t;
16. MyThread1(Table t){
17. **this**.t=t;
18. }
19. **public** **void** run(){
20. t.printTable(5);
21. }
23. }
24. **class** MyThread2 **extends** Thread{
25. Table t;
26. MyThread2(Table t){
27. **this**.t=t;
28. }
29. **public** **void** run(){
30. t.printTable(100);
31. }
32. }
34. **public** **class** TestSynchronization2{
35. **public** **static** **void** main(String args[]){
36. Table obj = **new** Table();//only one object
37. MyThread1 t1=**new** MyThread1(obj);
38. MyThread2 t2=**new** MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }

Output: 5

10

15

20

25

100

200

300

400

500

### Same Example of synchronized method by using annonymous class

In this program, we have created the two threads by annonymous class, so less coding is required.

1. //Program of synchronized method by using annonymous class
2. **class** Table{
3. **synchronized** **void** printTable(**int** n){//synchronized method
4. **for**(**int** i=1;i<=5;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){System.out.println(e);}
9. }
11. }
12. }
14. **public** **class** TestSynchronization3{
15. **public** **static** **void** main(String args[]){
16. **final** Table obj = **new** Table();//only one object
18. Thread t1=**new** Thread(){
19. **public** **void** run(){
20. obj.printTable(5);
21. }
22. };
23. Thread t2=**new** Thread(){
24. **public** **void** run(){
25. obj.printTable(100);
26. }
27. };
29. t1.start();
30. t2.start();
31. }
32. }

Output: 5

10

15

20

25

100

200

300

400

500

# Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### Points to remember for Synchronized block

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.

**Syntax to use synchronized block**

1. **synchronized** (object reference expression) {
2. //code block
3. }

### Example of synchronized block

Let's see the simple example of synchronized block.

***Program of synchronized block***

1. **class** Table{
3. **void** printTable(**int** n){
4. **synchronized**(**this**){//synchronized block
5. **for**(**int** i=1;i<=5;i++){
6. System.out.println(n\*i);
7. **try**{
8. Thread.sleep(400);
9. }**catch**(Exception e){System.out.println(e);}
10. }
11. }
12. }//end of the method
13. }
15. **class** MyThread1 **extends** Thread{
16. Table t;
17. MyThread1(Table t){
18. **this**.t=t;
19. }
20. **public** **void** run(){
21. t.printTable(5);
22. }
24. }
25. **class** MyThread2 **extends** Thread{
26. Table t;
27. MyThread2(Table t){
28. **this**.t=t;
29. }
30. **public** **void** run(){
31. t.printTable(100);
32. }
33. }
35. **public** **class** TestSynchronizedBlock1{
36. **public** **static** **void** main(String args[]){
37. Table obj = **new** Table();//only one object
38. MyThread1 t1=**new** MyThread1(obj);
39. MyThread2 t2=**new** MyThread2(obj);
40. t1.start();
41. t2.start();
42. }
43. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronizedBlock1)

Output:5

10

15

20

25

100

200

300

400

500

### Same Example of synchronized block by using annonymous class:

***//Program of synchronized block by using annonymous class***

1. **class** Table{
3. **void** printTable(**int** n){
4. **synchronized**(**this**){//synchronized block
5. **for**(**int** i=1;i<=5;i++){
6. System.out.println(n\*i);
7. **try**{
8. Thread.sleep(400);
9. }**catch**(Exception e){System.out.println(e);}
10. }
11. }
12. }//end of the method
13. }
15. **public** **class** TestSynchronizedBlock2{
16. **public** **static** **void** main(String args[]){
17. **final** Table obj = **new** Table();//only one object
19. Thread t1=**new** Thread(){
20. **public** **void** run(){
21. obj.printTable(5);
22. }
23. };
24. Thread t2=**new** Thread(){
25. **public** **void** run(){
26. obj.printTable(100);
27. }
28. };
30. t1.start();
31. t2.start();
32. }
33. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronizedBlock2)

Output:5

10

15

20

25

100

200

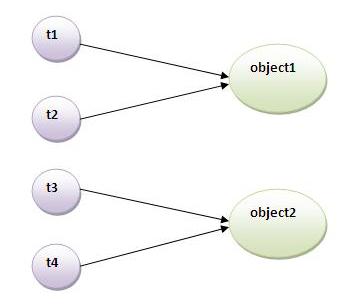
300

400

500

# 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.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

### Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

1. **class** Table{
3. **synchronized** **static** **void** printTable(**int** n){
4. **for**(**int** i=1;i<=10;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){}
9. }
10. }
11. }
13. **class** MyThread1 **extends** Thread{
14. **public** **void** run(){
15. Table.printTable(1);
16. }
17. }
19. **class** MyThread2 **extends** Thread{
20. **public** **void** run(){
21. Table.printTable(10);
22. }
23. }
25. **class** MyThread3 **extends** Thread{
26. **public** **void** run(){
27. Table.printTable(100);
28. }
29. }



34. **class** MyThread4 **extends** Thread{
35. **public** **void** run(){
36. Table.printTable(1000);
37. }
38. }
40. **public** **class** TestSynchronization4{
41. **public** **static** **void** main(String t[]){
42. MyThread1 t1=**new** MyThread1();
43. MyThread2 t2=**new** MyThread2();
44. MyThread3 t3=**new** MyThread3();
45. MyThread4 t4=**new** MyThread4();
46. t1.start();
47. t2.start();
48. t3.start();
49. t4.start();
50. }
51. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronization4)

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

### Same example of static synchronization by annonymous class

In this example, we are using annonymous class to create the threads.

1. **class** Table{
3. **synchronized** **static**  **void** printTable(**int** n){
4. **for**(**int** i=1;i<=10;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){}
9. }
10. }
11. }
13. **public** **class** TestSynchronization5 {
14. **public** **static** **void** main(String[] args) {
16. Thread t1=**new** Thread(){
17. **public** **void** run(){
18. Table.printTable(1);
19. }
20. };
22. Thread t2=**new** Thread(){
23. **public** **void** run(){
24. Table.printTable(10);
25. }
26. };
28. Thread t3=**new** Thread(){
29. **public** **void** run(){
30. Table.printTable(100);
31. }
32. };
34. Thread t4=**new** Thread(){
35. **public** **void** run(){
36. Table.printTable(1000);
37. }
38. };
39. t1.start();
40. t2.start();
41. t3.start();
42. t4.start();
44. }
45. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronization5)

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

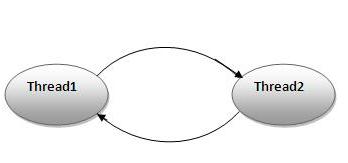
### Synchronized block on a class lock:

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:

1. **static** **void** printTable(**int** n) {
2. **synchronized** (Table.**class**) {       // Synchronized block on class A
3. // ...
4. }
5. }

# Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



### Example of Deadlock in java

1. **public** **class** TestDeadlockExample1 {
2. **public** **static** **void** main(String[] args) {
3. **final** String resource1 = "ratan jaiswal";
4. **final** String resource2 = "vimal jaiswal";
5. // t1 tries to lock resource1 then resource2
6. Thread t1 = **new** Thread() {
7. **public** **void** run() {
8. **synchronized** (resource1) {
9. System.out.println("Thread 1: locked resource 1");
11. **try** { Thread.sleep(100);} **catch** (Exception e) {}
13. **synchronized** (resource2) {
14. System.out.println("Thread 1: locked resource 2");
15. }
16. }
17. }
18. };
20. // t2 tries to lock resource2 then resource1
21. Thread t2 = **new** Thread() {
22. **public** **void** run() {
23. **synchronized** (resource2) {
24. System.out.println("Thread 2: locked resource 2");
26. **try** { Thread.sleep(100);} **catch** (Exception e) {}
28. **synchronized** (resource1) {
29. System.out.println("Thread 2: locked resource 1");
30. }
31. }
32. }
33. };

36. t1.start();
37. t2.start();
38. }
39. }

Output: Thread 1: locked resource 1

Thread 2: locked resource 2

# 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.It is implemented by following methods of **Object class**:

* wait()
* notify()
* notifyAll()

### 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.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void wait()throws InterruptedException | waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | waits for the specified amount of time. |

### 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. Syntax:

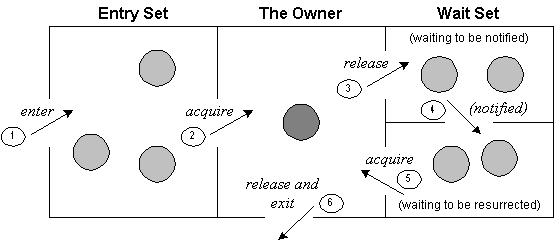
public final void notify()

### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

### Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

### 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?

Let's see the important differences between wait and sleep methods.

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

### Example of inter thread communication in java

Let's see the simple example of inter thread communication.

1. **class** Customer{
2. **int** amount=10000;
4. **synchronized** **void** withdraw(**int** amount){
5. System.out.println("going to withdraw...");
7. **if**(**this**.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. **try**{wait();}**catch**(Exception e){}
10. }
11. **this**.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
15. **synchronized** **void** deposit(**int** amount){
16. System.out.println("going to deposit...");
17. **this**.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
23. **class** Test{
24. **public** **static** **void** main(String args[]){
25. **final** Customer c=**new** Customer();
26. **new** Thread(){
27. **public** **void** run(){c.withdraw(15000);}
28. }.start();
29. **new** Thread(){
30. **public** **void** run(){c.deposit(10000);}
31. }.start();
33. }}

Output: going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw 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. Let's first see the methods provided by the Thread class for thread interruption. |

## The 3 methods provided by the Thread class for interrupting a thread

|  |
| --- |
| * **public void interrupt()** * **public static boolean interrupted()** * **public boolean isInterrupted()** |

## Example of interrupting a thread that stops working

|  |
| --- |
| In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception. |

1. **class** TestInterruptingThread1 **extends** Thread{
2. **public** **void** run(){
3. **try**{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }**catch**(InterruptedException e){
7. **throw** **new** RuntimeException("Thread interrupted..."+e);
8. }
10. }
12. **public** **static** **void** main(String args[]){
13. TestInterruptingThread1 t1=**new** TestInterruptingThread1();
14. t1.start();
15. **try**{
16. t1.interrupt();
17. }**catch**(Exception e){System.out.println("Exception handled "+e);}
19. }
20. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread1)

[download this example](http://www.javatpoint.com/src/multi/interrupt1.zip)

Output:Exception in thread-0

java.lang.RuntimeException: Thread interrupted...

java.lang.InterruptedException: sleep interrupted

at A.run(A.java:7)

## Example of interrupting a thread that doesn't stop working

|  |
| --- |
| In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working. |

1. **class** TestInterruptingThread2 **extends** Thread{
2. **public** **void** run(){
3. **try**{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }**catch**(InterruptedException e){
7. System.out.println("Exception handled "+e);
8. }
9. System.out.println("thread is running...");
10. }
12. **public** **static** **void** main(String args[]){
13. TestInterruptingThread2 t1=**new** TestInterruptingThread2();
14. t1.start();
16. t1.interrupt();
18. }
19. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread2)

[download this example](http://www.javatpoint.com/src/multi/interrupt2.zip)

Output:Exception handled

java.lang.InterruptedException: sleep interrupted

thread is running...

## Example of interrupting thread that behaves normally

|  |
| --- |
| If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later. |

1. **class** TestInterruptingThread3 **extends** Thread{
3. **public** **void** run(){
4. **for**(**int** i=1;i<=5;i++)
5. System.out.println(i);
6. }
8. **public** **static** **void** main(String args[]){
9. TestInterruptingThread3 t1=**new** TestInterruptingThread3();
10. t1.start();
12. t1.interrupt();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread3)

Output:1

2

3

4

5

## What about isInterrupted and interrupted method?

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

1. **public** **class** TestInterruptingThread4 **extends** Thread{
3. **public** **void** run(){
4. **for**(**int** i=1;i<=2;i++){
5. **if**(Thread.interrupted()){
6. System.out.println("code for interrupted thread");
7. }
8. **else**{
9. System.out.println("code for normal thread");
10. }
12. }//end of for loop
13. }
15. **public** **static** **void** main(String args[]){
17. TestInterruptingThread4 t1=**new** TestInterruptingThread4();
18. TestInterruptingThread4 t2=**new** TestInterruptingThread4();
20. t1.start();
21. t1.interrupt();
23. t2.start();
25. }
26. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestInterruptingThread4)

Output:Code for interrupted thread

code for normal thread

code for normal thread

code for normal thread

# 

# Java Inner Class

**Java inner class** or nested class is a class i.e. declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

#### Syntax of Inner class

1. **class** Java\_Outer\_class{
2. ...
3. **class** Java\_Inner\_class{
4. ...
5. }
6. ...
7. }

## Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Do You Know ?

* What is the internal code generated by the compiler for member inner class ?
* What are the two ways to create annonymous inner class ?
* Can we access the non-final local variable inside the local inner class ?
* How to access the static nested class ?
* Can we define an interface within the class ?
* Can we define a class within the interface ?

## Difference between nested class and inner class in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

# Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

1. non-static nested class(inner class)
   * a)Member inner class
   * b)Annomynous inner class
   * c)Local inner class
2. static nested class

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](http://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Annonymous Inner Class](http://www.javatpoint.com/annonymous-inner-class) | A class created for implementing interface or extending class. Its name is decided by the java compiler. |
| [Local Inner Class](http://www.javatpoint.com/local-inner-class) | A class created within method. |
| [Static Nested Class](http://www.javatpoint.com/static-nested-class) | A static class created within class. |
| [Nested Interface](http://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

# 1)Member inner class

|  |
| --- |
| A class that is declared inside a class but outside a method is known as member inner class. |

### Invocation of Member Inner class

|  |
| --- |
| 1. From within the class 2. From outside the class |

### Example of member inner class that is invoked inside a class

In this example, we are invoking the method of member inner class from the display method of Outer class.

1. **class** TestMemberOuter1{
2. **private** **int** data=30;
3. **class** Inner{
4. **void** msg(){System.out.println("data is "+data);}
5. }
7. **void** display(){
8. Inner in=**new** Inner();
9. in.msg();
10. }
11. **public** **static** **void** main(String args[]){
12. TestMemberOuter1 obj=**new** TestMemberOuter1();
13. obj.display();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMemberOuter1)

Output:data is 30

### Internal code generated by the compiler for member inner class:

|  |
| --- |
| The java compiler creates a class file named Outer$Inner in this case. The Member inner class have the reference of Outer class that is why it can access all the data members of Outer class including private. |

1. **import** java.io.PrintStream;
3. **class** Outer$Inner
4. {
5. **final** Outer **this**$0;
6. Outer$Inner()
7. {   **super**();
8. **this**$0 = Outer.**this**;
9. }
11. **void** msg()
12. {
13. System.out.println((**new** StringBuilder()).append("data is ")
14. .append(Outer.access$000(Outer.**this**)).toString());
15. }
17. }

### Example of member inner class that is invoked outside a class

In this example, we are invoking the msg() method of Inner class from outside the outer class i.e. Test class.

1. <b><i>//Program of member inner class that is invoked outside a class</i></b>
3. **class** Outer{
4. **private** **int** data=30;
5. **class** Inner{
6. **void** msg(){System.out.println("data is"+data);}
7. }
8. }
10. **class** TestMemberInner{
11. **public** **static** **void** main(String args[]){
12. Outer obj=**new** Outer();
13. Outer.Inner in=obj.**new** Inner();
14. in.msg();
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestMemberInner)

Output:data is 30

# 2)Annonymous inner class

A class that have no name is known as annomymous inner class.

Annonymous class can be created by:

1. Class (may be abstract class also).
2. Interface

### Program of annonymous inner class by abstract class

1. **abstract** **class** Person{
2. **abstract** **void** eat();
3. }
5. **class** TestAnnonymousInner{
6. **public** **static** **void** main(String args[]){
7. Person p=**new** Person(){
8. **void** eat(){System.out.println("nice fruits");}
9. };
11. p.eat();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnonymousInner)

Output:nice fruits

### What happens behind this code?

1. Person p=**new** Person(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
5. }
6. }
7. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
8. An object of Annonymous class is created that is reffered by p reference variable of Person type. As you know well that Parent class reference variable can refer the object of Child class.

### The internal code generated by the compiler for annonymous inner class

1. **import** java.io.PrintStream;
2. **static** **class** Emp$1 **extends** Person
3. {
4. Emp$1(){}
6. **void** eat()
7. {
8. System.out.println("nice fruits");
9. }
10. }

### Program of annonymous inner class by interface

1. **interface** Eatable{
2. **void** eat();
3. }
5. **class** TestAnnonymousInner1{
6. **public** **static** **void** main(String args[]){
8. Eatable e=**new** Eatable(){
9. **public** **void** eat(){System.out.println("nice fruits");}
10. };
11. e.eat();
12. }
13. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestAnnonymousInner1)

Output:nice fruits

### What does the compiler for annonymous inner class created by interface

It performs two main tasks behind this code:

1. Eatable p=**new** Eatable(){
2. **void** eat(){System.out.println("nice fruits");}
3. };
5. }
6. }
7. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
8. An object of Annonymous class is created that is reffered by p reference variable of Eatable type. As you know well that Parent class reference variable can refer the object of Child class.

### The internal code generated by the compiler for annonymous inner class created by interface

1. **import** java.io.PrintStream;
2. **static** **class** Emp$1 **implements** Eatable
3. {
4. Emp$1(){}
6. **void** eat(){System.out.println("nice fruits");}
7. }

# 3)Local inner class

|  |
| --- |
| A class that is created inside a method is known as local inner class. If you want to invoke the methods of local inner class, you must instantiate this class inside the method. |

### Program of local inner class

1. **public** **class** localInner1{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **class** Local{
5. **void** msg(){System.out.println(data);}
6. }
7. Local l=**new** Local();
8. l.msg();
9. }
10. **public** **static** **void** main(String args[]){
11. localInner1 obj=**new** localInner1();
12. obj.display();
13. }
14. }

Output:30

### Internal code generated by the compiler for local inner class

|  |
| --- |
| In such case, compiler creates a class named Simple$1Local that have the reference of the outer class. |

1. **import** java.io.PrintStream;
2. **class** Simple$1Local
3. {
4. **final** Simple **this**$0;
6. Simple$1Local()
7. {
8. **super**();
9. **this**$0 = Simple.**this**;
10. }
11. **void** msg()
12. {
13. System.out.println(Simple.access$000(Simple.**this**));
14. }
16. }

#### *Rule: Local variable can't be private, public or protected.*

## Rules for Local Inner class

#### *1) Local inner class cannot be invoked from outside the method.*

#### *2) Local inner class cannot access non-final local variable.*

### Program of accessing non-final local variable in local inner class

1. **class** localInner2{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **int** value=50;//local variable must be final
5. **class** Local{
6. **void** msg(){System.out.println(value);}//C.T.Error
7. }
8. Local l=**new** Local();
9. l.msg();
10. }
11. **public** **static** **void** main(String args[]){
12. localInner2 obj=**new** localInner2();
13. obj.display();
14. }
15. }

Output:Compile Time Error

### Program of accessing final local variable in local inner class

1. **class** localInner3{
2. **private** **int** data=30;//instance variable
3. **void** display(){
4. **final** **int** value=50;//local variable must be final
5. **class** Local{
6. **void** msg(){System.out.println(data+" "+value);}//ok
7. }
8. Local l=**new** Local();
9. l.msg();
10. }
11. **public** **static** **void** main(String args[]){
12. localInner3 obj=**new** localInner3();
13. obj.display();
14. }
15. }

Output:30 50

# 4)static nested class

A static class that is created inside a class is known as static nested class. It cannot access the non-static members.

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

### Program of static nested class that have instance method

1. **class** TestOuter1{
2. **static** **int** data=30;
4. **static** **class** Inner{
5. **void** msg(){System.out.println("data is "+data);}
6. }
8. **public** **static** **void** main(String args[]){
9. TestOuter1.Inner obj=**new** TestOuter1.Inner();
10. obj.msg();
11. }
12. }

Output:data is 30

|  |
| --- |
| In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object. |

### Internal code generated by the compiler for static nested class

1. **import** java.io.PrintStream;
3. **static** **class** Outer$Inner
4. {
5. Outer$Inner(){}
7. **void** msg(){
8. System.out.println((**new** StringBuilder()).append("data is ")
9. .append(Outer.data).toString());
10. }
12. }

### Program of static nested class that have static method

1. **class** TestOuter2{
2. **static** **int** data=30;
4. **static** **class** Inner{
5. **static** **void** msg(){System.out.println("data is "+data);}
6. }
8. **public** **static** **void** main(String args[]){
9. TestOuter2.Inner.msg();//no need to create the instance of static nested class
10. }
11. }

Output:data is 30