Threads

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

There are two types of multitasking- Thread based and Process based. A process is a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.
In a thread-based multitasking environment, the thread is the smallest unit of dispatch able code. Here a single program can perform two or more tasks simultaneously. Multitasking threads require less overhead than multitasking processes.

Thread v/s Process:

Process	Thread
Processes are heavyweight tasks and each process require their own separate address spaces to run	Threads are lightweight. They share the same address space and cooperatively share the same heavyweight process.
Inter-process communication is expensive and limited	Inter-thread communication is inexpensive
Context switching from one process to another is also costly	Context switching from one thread to the next is low cost.

Note: Java supports Thread based multitasking.

Thread States
New: newly created thread
Ready: It can be ready to run as soon as it gets CPU time.
Running: A thread can be running- allotted CPU.
Suspend: A running thread can be suspended, which temporarily suspends its activity.
Resumed: A suspended thread can then be resumed, allowing it to pick up where it left off.
Waiting: A thread can be blocked when waiting for a resource.
Terminated: At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Adv. of Threading
1. Performance: Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
2. Sharing of resources
3. Reduce complexity of applications
4. Java's multithreading eliminates the event loop/polling mechanism which exists in single threaded system. Here one thread can pause without stopping other parts of the program.

Main Thread
When a Java program begins, one thread begins running automatically. This is usually called the main thread of the program. The main thread is important for two reasons:
1. It is the thread from which other "child" threads will be spawned.
2. Often it must be the last thread to finish execution because it performs various shutdown actions.
To get the reference to the current active thread, we can use the function

```
Thread t = Thread.currentThread();
```

We can display the thread information

```
System.out.println("Current thread: " + t);
```

Displays information as Current thread: Thread [main, 5, main] ; the name of the thread, its priority, and the name of its group. The name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this

thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole.
To change the internal name of the thread we can use

final void setName(String threadName)
final String getName( )

Eg: t.setName("My Thread"); String s = t.getName()
To pause a thread we use

static void sleep(long milliseconds) throws InterruptedException
example : Thread.Sleep(1000);

While dealing with a thread, we need to handle InterruptedException.

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

Creating Threads
Java provides built in class Thread to deal with threads. Multi threaded application can be created by instantiating the Thread class. There are two ways to create threads.

1. Implementing the Runnable interface.
2. Extending/inheriting the Thread class.

Implementing Runnable
The easiest way to create a thread is by implementing the Runnable interface.

Steps: 1. Create a class that implement the Runnable interface.
2. Implement the interface method run (). The code to be executed on the new thread is provided in this method. This method can call another methods, create objects, variables etc.
3. Instantiate the Thread class. Usually this is done from a constructor as

Thread t = new Thread (Runnable obj, String name);

Where, obj is the instance of the class where Runnable interface is implemented and name is the name of the thread.
4. Call start() to execute the new thread

t.start();

This invoke the run() method implemented.

```
class NewThread implements Runnable {
        Thread t;
        NewThread() {
                t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                t.start(); // Start the thread
        }
        public void run() {
        try {
                for(int i = 5; i > 0; i--) {
                    System.out.println("Child Thread: " + i);
                    Thread.sleep(500);
                }
        } catch (InterruptedException e) {
                System.out.println("Child interrupted.");
        }
                System.out.println("Exiting child thread.");
        }
}
class ThreadDemo {
        public static void main(String args[]) {
                NewThread n =  new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

Extending Thread

Steps: 1. Create a class that inherit the Thread Class.

2. Override the Thread Class method run () in the new class. The code to be executed on the new thread is provided in this method. This method can call another methods, create objects, variables etc.

3. Pass name of the thread to the super class.

4. Call start() to execute the new thread which invoke the run() method to begin the execution.

```
    class NewThread extends Thread {
            NewThread() {
                    super("Demo Thread");
                    System.out.println("Child thread: " + this);
                    start(); // Start the thread
            }
            public void run() {
            try {
                    for(int i = 5; i > 0; i--) {
                            System.out.println("Child Thread: " + i);
                            Thread.sleep(500);
                    }
            } catch (InterruptedException e) {
                    System.out.println("Child interrupted.");
            }
                    System.out.println("Exiting child thread.");
            }
    }
    class ExtendThread {
            public static void main(String args[]) {
                    new NewThread(); // create a new thread
                    try {
                            for(int i = 5; i > 0; i--) {
                                    System.out.println("Main Thread: " + i);
                                    Thread.sleep(1000);
```

```
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
        }
}
```

Thread Priority
Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. To set a thread's priority, use the setPriority( ) method, which is a member of Thread class .

```
        final void setPriority(int level)
```

Here, level specifies the new priority setting for the calling thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as final variables within Thread Class.

To obtain the current priority set, use the Thread class method

```
        final int getPriority( )
```

Example:

```
        T1.setPriority(Thread.MAX_PRIORITY);
        int k = T1.getPriority();
```

Synchronization
Mutual Exclusion is achieved using Synchronization. When two or more threads need to access a shared resource, it ensures that the resource is used by only one thread at a time. Synchronization is achieved using the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended (waiting) until the first thread exits the monitor.
For Synchronization java provides a keyword "synchronized".

Using Synchronized Methods
When a method is synchronized, java ensures that the method is invoked only by one thread at a time.

```
        synchronized method(parameter list) {
                // body of the method...
        }
```

Example:

```
class Callme {
        void call(String msg) {
                System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                } catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
class Caller implements Runnable {
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s) {
                target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
        }
        public void run() {
                target.call(msg);
        }
    }
    class Synch {
        public static void main(String args[]) {
                Callme target = new Callme();
                Caller ob1 = new Caller(target, "Hello");
                Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        try {
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
        } catch(InterruptedException e) {
                System.out.println("Interrupted");
        }
    }
    }
```

The synchronized Statement

In situations where we cannot synchronize methods (as we are using methods written by others and source is not available to us) we can synchronize a set of statements as

```
        synchronized(object) {
                // statements to be synchronized
        }
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Example:

```
                public void run() {
                        synchronized(target) { // synchronized block
                                target.call(msg);
                        }
                }
```

Daemon Thread

A thread running in the background and providing service to the program is called a Daemon Thread. Daemon thread is not considered as part of the program and a program get terminates when all non-daemon threads are completed.

```
        class NewThread implements Runnable {
                Thread t;
                NewThread() {
                        t = new Thread(this, "Demo Thread");
                        System.out.println("Child thread: " + t);
                        t.start(); // Start the thread
                        t.setDeamon(true);          // creating a daemon thread.
        }
        public void run() {
        try {
                ..........
```

Inter-thread Communication
These methods are available in object class and only be called from synchronized context. All these methods are final and return types are void.
wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
notify( ) wakes up the first thread that called wait( ) on the same object.
notifyAll( ) wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

Some useful Methods

isAlive( )

Syntax: final boolean isAlive( )
The isAlive( ) method returns true if the thread upon which it is called is still running.
It returns false otherwise.
Example: ob1.t.isAlive()
Join()

Syntax final void join( ) throws InterruptedException

This method the main thread waits until the thread on which it is called terminates.
Example: c1.t.join();