Threads

Threads allow separation of loosely coupled parts within a program into concurrently executing tasks. Each of these tasks is called a thread. More formally, a thread is a sequential flow of control in a program. With multiple threads, a program may have multiple sequential flows of control which perform several tasks at the same time.

Many computers have a single CPU which switches among threads. In order to do this, the available processor time is divided among the threads that need it. Each thread is executed for a period of time and it is suspended when its allocated time slice elapses. After the currently executing thread is suspended, another thread waiting for CPU resumes running. By this way, an impression of concurrently executing tasks (or multiple CPUs) is given.

In many cases, threads simplify program design due to its inherent support for separation of concerns. With threading, it is also possible to perform operations that take a large amount of time in the background while doing other tasks. For example, while doing some processor intensive task, the user interface may remain responsive. In addition, the tasks of varying priority may be distinguished by assigning different priorities to each task.

Thread Creation

In order to create a thread in a program, an instance of `Thread` class should be created and a `run()` method should be provided to the `Thread` instance. The `run()` method may be supplied within a class extending the `Thread` class and overriding `run()` method, or an instance of a class implementing `Runnable` interface (implementing `run()` method) could be passed to the `Thread` class instance within its constructor as follows:

Option 1: Declare a subclass of `Thread` class

```
class MyThread extends
Thread {
...
        public void run(){
        // statements to
perform the task
        }
...
};

// create an instance of
MyThread
// in a class method
MyThread myThread = new
MyThread();
...
```

Option 2: Pass an instance of a class implementing `Runnable` interface

```
class MyTask implements Runnable{
...
        public void run(){
        // statements to perform
the task
        }
...
};

// create an instance of MyThread
// in a class method
Thread myThread = new Thread(new
MyTask());
```

In order put the created thread into execution, the thread's

`start()` method could be invoked as:

```
myThread.start();
```

Note: Do not call `run()` method directly to execute the thread. This will correspond to ordinary method call.

Thread Execution

Calling `start()` method of a thread instance make a thread eligible for execution and returns immediately. The execution continues from the next statement while the thread contends for the CPU with other threads if any. The *thread scheduler*, a special system process, is responsible for putting the thread into execution. At some point in the future, the thread scheduler pauses the execution of the current thread and allows one of the waiting threads to execute.

A newly created thread is executed by calling the `run()` method provided to it. After the `run()` method returns, the thread completes its execution and it is considered dead. A dead thread cannot be restarted by calling `start()` method again, but it can be used as an ordinary object (to read its attributes, call its methods etc). In order to start a new thread, a new instance of the thread class has to be created.

Every thread has a priority between `1` and `10`. The default value of the priority for a newly created thread is `5`. The thread scheduler chooses the thread to execute according to the priority assigned to that thread. The threads with higher priority have higher chance to seize CPU than other threads. The priority of a thread can be changed by calling `setPriority` method as:

```
myThread.setPriority(<new priority value>);
```

While a thread is executing, the thread scheduler may pause the execution it and switch to another thread if the time slice allocated to the current thread expires or a higher priority thread is waiting to be executed. In this way, the threads share the CPU among themselves.

A thread may request stopping execution of another thread by calling the `interrupt()` method of that thread. Therefore, within thread execution, it should be checked whether the thread is interrupted or not by calling `isInterrupted()` method.

Yielding, Sleeping, and Joining

A time-consuming thread may also allow other threads to execute by calling the `yield()` method of the `Thread` class. If a thread calls `yield()` method, thread scheduler may pause the current thread and switch to one of the other waiting threads. Therefore, the thread calling `yield()` will wait until the thread scheduler switches back to it. The following example, illustrates the behavior of yield().

Code	Output without `yield()`	Output with `yield()`

`public class MyThread extends Thread {`
`  public MyThread(String name){`
`  super(name);`
`  }`
`  public void run(){`
`  for(int i=0; i`
	`t1`
	`t1`
	`t1`
	`t2`
	`t2`
	`t2`
		`t1`
		`t2`
		`t1`
		`t2`
		`t1`
		`t2`

Note: In the above example, a name is assigned to each thread through its contstructor. The name of the thread can be returned by the `getName()` method.
In order to pause the execution of a thread for a specified amount of time, `sleep()` method can be used as:

```
sleep(<number of milliseconds>);
```

The sleeping thread does not use CPU. After the specified amount of time elapsed, it may be allowed to execute by thread scheduler. That is, the waiting time could be longer than the specified duration. If the thread is interrupted during sleep period, `InterruptedException` is thrown when `sleep()` method returns.
The `join()` method allows a thread to wait until another thread's execution completed. Like, `sleep()`, `InterruptedException` could be thrown if the calling thread is requested to be interrupted. In the following example, the execution continues after `t1` completes its execution.

```
Thread t1 = new Thread(…);
t1.start();
…
t1.join();
// the following statements will be executed after t1 dies.
…
```

SYNCHRONİZATİON

In multi-threaded programs, more than one thread may share (or use) a resource. In such cases, access to the resource by different threads should be synchronized to avoid unexpected results. For this purpose, Java allows marking some methods or some block of code within a method as critical sections such that only one thread is allowed to use the shared resource at a time by executing through the critical sections.
In Java, every object has a lock which can only be controlled by a single thread at a time. That is, if a thread acquires the lock of an object, the other threads wanting to acquire the lock will wait until the lock is released by the thread acquiring the lock. The lock of an object could be acquired in three ways:

A method of a class can be marked as `synchronized`. In this

case entire method body will be a critical section. The threads executing this method should first acquire the lock of the object which is an instance of the class declaring the method. When the method returns, the lock is released. For example, in the following, only one thread could be executing `myMethod1()` or `myMethod2()` for the same `MyClass` instance:

```
class MyClass{
  public synchronized void myMethod1(){
          ...
  }
  public synchronized void myMethod2(){
          ...
  }
  ...
}
```

A code segment in a method can also be marked as `synchronized`. In this case the statements in the enclosing block will be a critical section. The threads executing this block should first acquire the lock of the object which is an instance of the class declaring the method. After the block is executed, the lock is released. For example, in the following, only one thread could be executing `myMethod1()` or synchronized block in `myMethod2()` for the same `MyClass` instance:

```
class MyClass{
  public synchronized void myMethod1(){
          ...
  }
  public void myMethod2(){
          ...
          synchronized (this) {
                  // synchronized block
                  ...
          }
  }
  ...
}
```

A code segment in a method can be marked as `synchronized` using instance of another class. In this case the statements in the enclosing block will be a critical section for the corresponding object. The threads to execute this block should first acquire the lock of the specified object. After the block is executed, the lock is released. For example, in the following, only one thread could be executing `myMethod1()` or synchronized block in `myMethod2()` for the same `MyClass` instance `x`:

```
class MyClass{
  public synchronized void myMethod1(){
          ...
  }
  ...
}

class OtherClass{
  public void myMethod2(MyClass x){
          ...
          synchronized (x){
                  // synchronized block
                  ...
          }
  }
  ...
}
```

Wait and Notify

Synchronized methods and blocks allow a thread to use a resource alone. However, in some cases, within a synchronized block, a thread maybe waiting for another thread to do something. Since this may require the lock of the resource to be acquired by another thread, it will never happen and current thread gets stuck. In order to provide a means for thread cooperation `Object` class has three methods: `wait()`, `notify()`, and `notifyAll()`.
The `wait()` method allows a thread executing in a synchronized block to release the lock of the object and wait for an other object (or objects) to do something. If a thread calls the `wait()` method of an object within the critical section, it releases the lock and waits until one of the other threads calls `notify()` or `notifyAll()` method of the object.
The `notify()` method allows a thread to wake up one of the waiting threads. The thread to be awakened is arbitrarily chosen by the thread scheduler. That thread should also gain the lock of the object before resuming execution. The `notifyAll()` method can be used to wake up all threads waiting. The awakened threads should compete to acquire the lock of the object before proceeding. Like `wait()`, `notify()` and `notifyAll()` methods have to be called within a synchronized block.