Introduction multithreading in Java

Overview

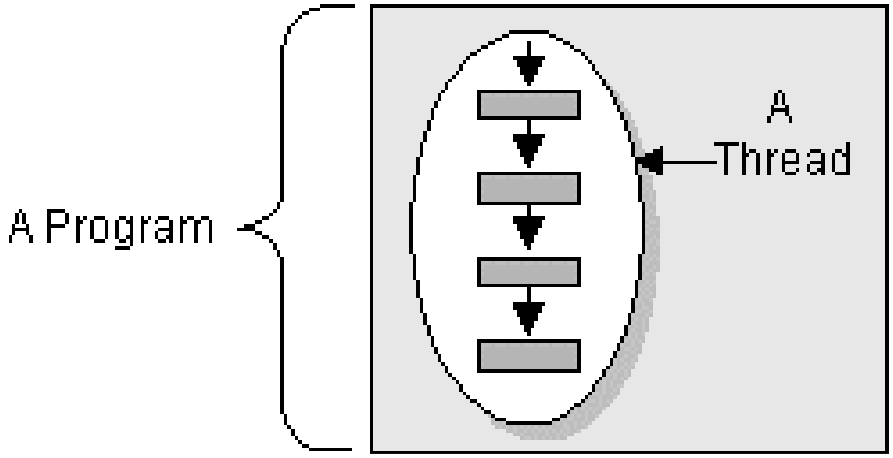
- Multi-tasking, multi-threading
- Java and multithreading

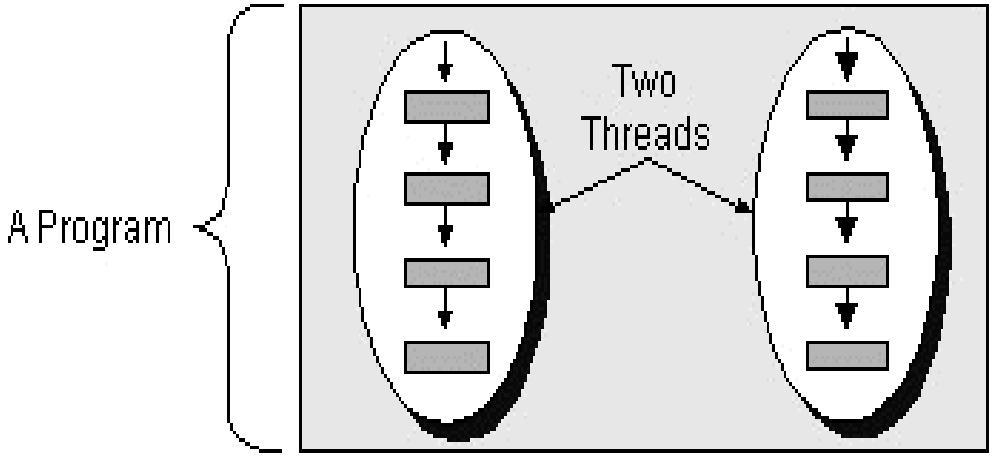
Multi-tasking, multi-threading

- **Multi-tasking,** more programs are executed simultaneously-concurrently (on one or more processors/cores) , based on *processes* (*heavyweight)* with separate address space
- **Multi-threading**, in a program –*process (lightweight),* more *threads* (code sequences) are executed simultaneously – concurrently or in parallel
- **Definition:** A thread-sometimes called an *execution context* or a *lightweight process*-is a single sequential flow of control within a program
- A ***thread*** is created when we have a time consuming activity

- Using **multi-threading** a Java program may concurrently/parallel:
  - Process an animation
  - Play music
  - Communicate with a server, etc.

- **Multiple threads** exist within the context of a process such that they are executed independently but share their process resources


A Program
A
Thread


A Program
Two
Threads

Perform Multiple Tasks Using…

- **Process (***heavyweight)*
  - **Definition**: – executable program loaded in memory
  - Has own address space
    - Variables & data structures (in memory)
  - Each process may execute a different program
  - Communicate via operating system, files, network
  - May contain multiple threads
  - Also known as “heavyweight process”

Perform Multiple Tasks Using…

- **Thread (*lightweight)***
  - **Definition:** – sequentially executed stream of instructions
  - Shares address space with other threads
  - Has own execution context
    - Program counter, call stack (local variables)
  - Communicate via shared access to data
  - Multiple threads in process execute same program
  - Also known as “lightweight process”

Programming with Threads

- **Concurrent/parallel programming**
  - Writing programs divided into independent tasks
  - Tasks may be executed in *parallel* on multi-processors/cores, or *concurrently* on one processor managed by the OS
- Many computers have multiple processors/cores.

Find out via:

Runtime.getRuntime( ).availableProcessors( );

- **Multithreading**
  - Executing a program with multiple threads in parallel/concurrent
  - Special form of multiprocessing

Java and multithreading

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- In fact, Java uses threads to enable the entire environment to be *asynchronous*. This helps reduce inefficiency by preventing the waste of CPU cycles.

- Single-threaded systems use an approach called an *event loop* with *polling (interrogation).*
- In this model, a *single thread* of control runs in an *infinite loop*, *polling a single event queue* to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler.

- Until this event handler returns, nothing else can happen in the system.
- This wastes CPU time.
- It can also result in one part of a program dominating the system and preventing any other events from being processed.
- In general, ***in a singled-threaded*** environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, *the entire program stops running*.

- The benefit of Java's multithreading is that the *main loop/polling mechanism is eliminated*. One thread can pause without stopping other parts of your program.
- For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. **Multithreading** allows animation loops as to sleep for a second between each frame without causing the whole system to pause.
- When *a thread is blocked in a Java* program, *only the single thread* that is blocked *pauses*.
- All other threads continue to *run*.

- Starting with **Java 1.5** we have a complementary multithreading mechanism offered by Java with the ***concurrent*** package
- For multithreading, **implicit** in ***java.lang*** package (for all Java versions from the begining) we have:
  - **Thread** class
  - **ThreadGroup** class
  - **Runnable** interface

    -It provides a thread API and all the generic behavior for threads. These behaviors include **starting**, **sleeping**, **running**, **yielding**, **joining, waiting**, **having a priority, etc**.
  - To implement a thread using the **Thread** class, you need to override the ***run( )*** method that performs the thread's task.
  - The ***run( )*** method gives to the thread something to do.
  - Its code implements the thread's running behavior. It can do anything that can be encoded in Java statements.

Thread class

- The **Thread** class implements a generic thread that, by default, does nothing. The implementation of its ***run( )*** method is *empty*.
- **Steps used** to implement a thread with the **Thread** class:

1. create a **derived class** (subclass) from **Thread**:

    public class MyThread extends Thread { }

2. **override** its empty ***run( )*** method so that it does something:

    public void run( ) {//……}

- The ***run( )*** method is the heart of any Thread and where the action of the Thread takes place. It is automatically called when a thread is executed (like ***main( )*** when we execute a stand-alone application) using a *start( )* method with the involved thread.

3. create an instance of the derived class MyThread:

MyThread th=new MyThread( );

4. execute the thread by calling the ***start( )*** method offered by the **Thread** class with the thread ***th*** object:

th.start( );

5. in the *start( )* method we are not allowed to call the *run( )* method, being a mistake if we specify in an explicit mode the execution of the *run( )* method.

Some Thread methods

- **Method Meaning**
- *getName( );* Obtain thread's name.
- *setName( );* Sets thread name.
- *getPriority( );* Obtain thread's priority.
- *isAlive( );* Determine if a thread is still running.
- *join( );* Wait for a thread to terminate.
- *run( );* Entry point for the thread.
- *sleep( );* Suspend a thread for a period of time.
- *start( );* Start a thread by calling its *run( )* method.

- When a Java program starts up, one thread begins running immediately.
- This is usually called the ***main*** *thread* of your program, because it is the one that is executed when your program begins. The *main* thread is important for two reasons:

-It is the thread from which other “child” threads will be spawned.

-Often it must be the last thread to finish execution because it performs various shutdown actions.

- Although the ***main*** thread is **created automatically** when your program is started, it can be controlled through a **Thread** object.
- To do so, you must obtain a reference to it by calling the method ***currentThread( )***, which is a **public static** method member of **Thread**.
- Its general form is shown here:

  static Thread currentThread( );

- This method returns a reference to the thread in which it is called.

```
//Controlling the main Thread.

public class CurrentThreadDemo {
        public static void main(String args[ ]) {
Thread t = Thread.currentThread( );//static method
System.out.println("Current thread: " + t);

// change the name of the thread
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"); }
                }//main
}//class
```

```
//name, priority, call method
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

```
//Create a second thread by extending Thread
class NewThread extends Thread {
            NewThread( ) {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start( ); // Start the thread
 }
// This is the entry point for the second thread
@Override
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
```

```
public  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.");
                }//main
}//class
```

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Child Thread: 3
Main Thread: 4
Child Thread: 2
Child Thread: 1
Main Thread: 3
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Runnable interface

- **Steps Implementing multithreading with the Runnable Interface**

1. create a **class** which **implements** the *Runnable* interface:

public class MyRunnable extends Another_Class implements Runnable{ }

2. **define** the ***run( )*** method so that it does something:

public void run( ) {//……}

3. obtain an object of *MyRunnable* type:

MyRunnable obj = new MyRunnable( );

4. this object will be used to create a ***Thread*** object:

Thread th = new Thread (obj);

5. start the thread *th*:

th.start( );

The **concurrent** package provides a new mechanism of Java multithreading using ***Runnable*** interface. We have:

- ***execute(someRunnable)***

– Adds ***someRunnable*** to the queue of **tasks**

- ***taskList.execute(someRunnable)***, that is now used from *concurrent* package

Runnable implementation

```
//Runnable
class FirstRunnable{
        public static void main(String args[ ]){
        System.out.println("Create the runnable object");
        MyRunnable myRunnable=new MyRunnable( );
                System.out.println("Create the thread" );
                Thread thread=new Thread (myRunnable);
                System.out.println("Start the thread");
                thread.start( );
                System.out.println ("Again in main");
                                                }//main
                        }//classFR
```

```
class Display{
        public void display(String message){
                System.out.println(message);}
                }//classDisplay
class MyRunnable extends Display implements Runnable{
                        public void run( ){
                int nrSteps=3;
                display("Run has "+nrSteps+" steps to do");
                for (int i=1;i<=nrSteps;i++) display("Step:"+i);
                display("Run has done its job");
                                }
                        }//classMyRunnable
```

Create the runnable object
Create the thread
Start the thread
Again in main
Run has 3 steps to do
Step:1
Step:2
Step:3
Run has done its job

Thread and Runnable implementation

```
/*This thread extends Thread class. */
class MyThread extends Thread
{
        @Override
        public void run( ){          //do something
                System.out.println("MyThread running");
        }
}
/** This thread implements Runnable interface */
 class MyRunnable implements Runnable
{
        public void run( ){
        System.out.println("MyRunnable running");
        }
}
```

```
/* Starts two threads */
public class ThreadRunnable
{
        public static void main(String[ ] args)
        {
        Thread thread1 = new MyThread( );
        Thread thread2 = new Thread(new MyRunnable( ));

                thread1.start( );
                thread2.start( );
        }
}
```

Result:

MyThread running

MyRunnable running

Remark:

It is not recommended to use the implementation with the *Thread* class.

- Not a big problem for getting started
  - but a bad habit for industrial strength development
- The methods of the worker class and the *Thread* class get all tangled up
- Makes it hard to migrate to Thread Pools and other more efficient approaches

The Life Cycle of a Thread

- A thread could be in the following states:

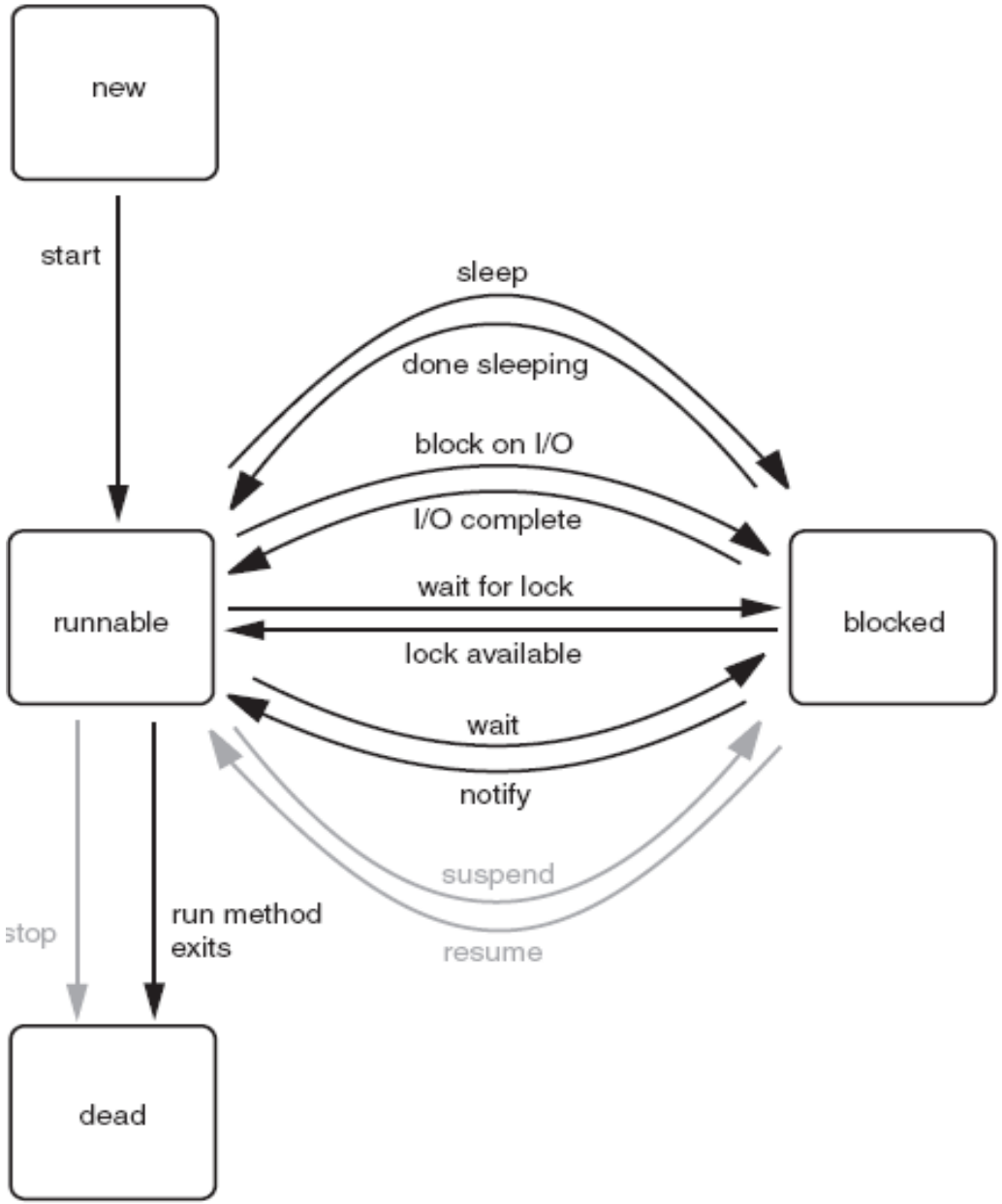
-*new* - thread allocated & waiting for *start*

-*runnable* - thread can execute

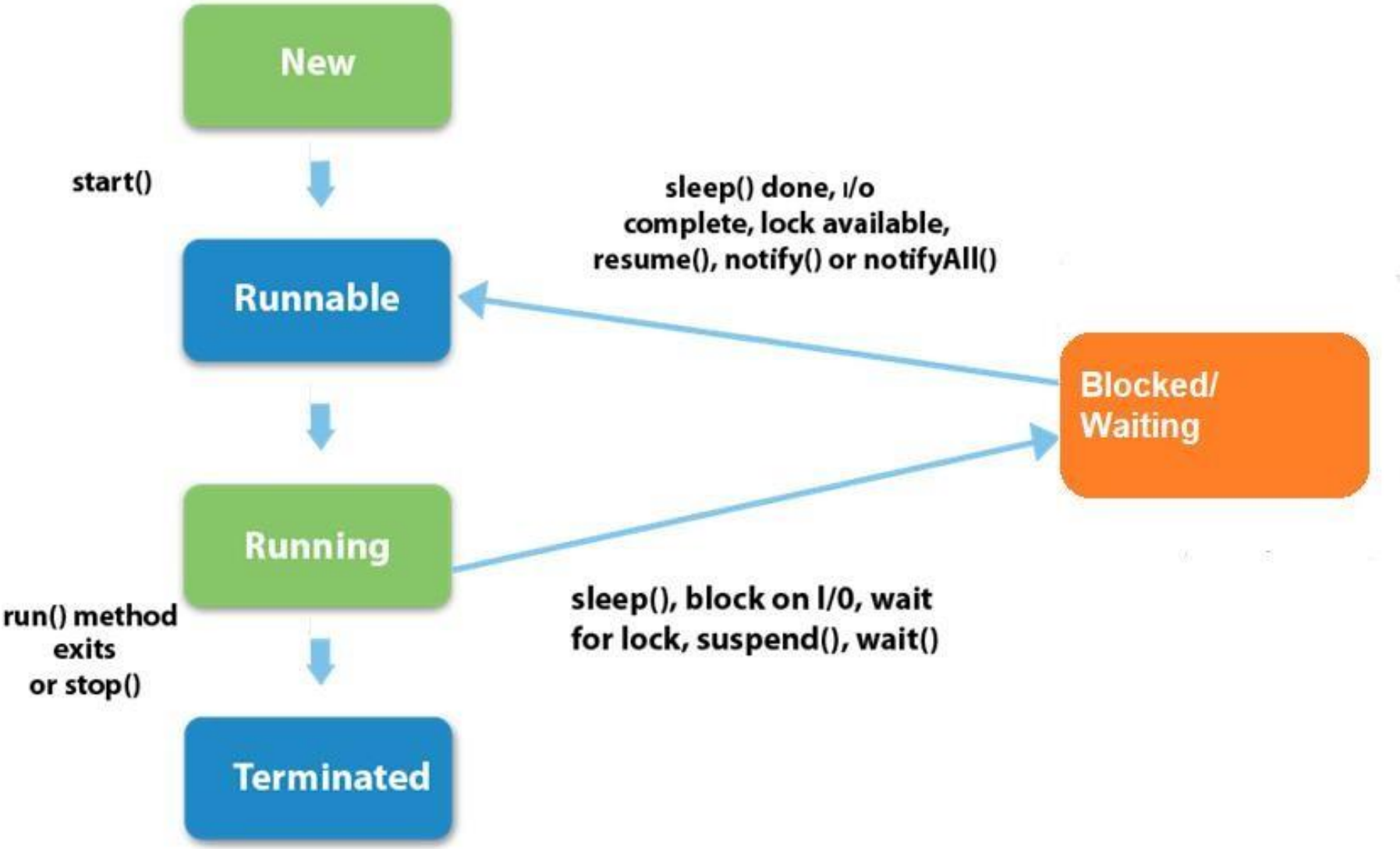
-*blocked* - thread waiting for event (I/O, etc.)

-*ready* – thread ready to be executed or finished

-*dead* - thread finished


new
start
sleep
done sleeping
block on I/O
I/O complete
wait for lock
lock available
wait
notify
suspend
resume
runnable
blocked
stop
run method
exits
dead

The Life Cycle of a Thread



Thread main methods- details

- ***start( )*** - The start method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's *run()* method. After the start method has returned, the thread is "running". Yet, it's somewhat more complex than that. As the previous figure shows, a thread that has been started is actually in the Runnable state. Many computers have a single processor, thus making it impossible to run all "running" threads at the same time. The Java runtime system must implement a scheduling scheme that shares the processor between all "running" threads. So at any given time, a "running" thread actually may be waiting for its turn in the CPU.
- ***stop( )*** – forces to stop the thread and the thread will be stopped no matter in what state it was.
- ***suspend( )*** – stops the thread temporary - deprecated
- ***resume( )*** – executes the thread that was temporary suspended - deprecated
- ***sleep( )*** – blocks the execution of a thread for a time period. After that time the thread will be automatically executed or if we want to execute faster we will use the *interrupt()* method - deprecated
- ***run( )*** – executes the main tasks for the thread

- ***join( )*** – will wait till the child thread will finish the execution; the controll will be pased after a time or finishing the action to the parent thread
- ***yield( )*** – imposes to pass the thread in a waiting queue. A concurrent thread will be executed. The *yield( )* method gives other threads of the same priority a chance to run. If there are no equal priority threads that are runnable, then the yield is ignored - deprecated
- ***interrupt( )*** – will send an interruption to take the thread from the wait state in a running state
- ***destroy( )*** – will pass the thread in a dead state, but will not de-allocate the data.
- ***isAlive( )*** - returns *true* if the thread has been started and not stopped. If the *isAlive( )* method returns *false*, you know that the thread either is a *New* thread or is *Dead*. If the *isAlive( )* method returns *true*, you know that the thread is either *Runnable* or *Not Runnable*. You cannot differentiate between a *New Thread* or a *Dead* thread. Nor can you differentiate between a *Runnable* thread and a *Not Runnable* thread.
- ***isActive( )*** – returns *true* if the thread is in a *running* or *blocked* state and *false* if is *new* or *dead*
- ***wait( )*** and ***notify( )*** or ***notifyAll( )*** are used in a synchronization process

Creating Multiple Threads

- So far, you have been using only two threads: the main thread and one child thread.
- However, your program can spawn as many threads as it needs.
- For example, the following program creates three child threads:

```
//Create multiple threads.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
        NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);//Thread(Runnable target, String name)
System.out.println("New thread: " + t);
t.start( ); // Start the thread
        }
// This is the entry point for thread.
         public void run( ) {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);      }
        } catch (InterruptedException e) {
System.out.println(name + "Interrupted");}
System.out.println(name + " exiting.");    }//run
}//class
```

```
public class MultiThreadDemo {
        public static void main(String args[ ]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
        try {
// wait 10 seconds for other threads to end
Thread.sleep(10000);
         }
catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
                                    }
System.out.println("Main thread exiting.");
                                }//main
}//class
```

The output from this program is shown here (differences depending on systems):

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in ***main( )***.
This causes the ***main*** thread to sleep for **ten seconds** and ensures that it will finish last.

Using *isAlive( )* and *join( )*

- As mentioned, often you will want that the *main* thread to finish last. In the preceding examples, this is accomplished by calling ***sleep( )*** within ***main( )***, with a long enough delay to ensure that all child threads terminate prior to the *main* thread. However, this is hardly a satisfactory solution, and it also raises a larger question:
- How can one thread know when another thread has ended?
- Fortunately, ***Thread*** class provides a means by which you can answer this question.

- Two ways exist to determine whether a thread has finished. First, you can call ***isAlive( )*** on the thread. This method is defined by ***Thread***, and its general form is shown here:

  final boolean isAlive( );

- The ***isAlive( )*** method returns ***true*** if the thread upon which it is called is still running.
- It returns ***false*** otherwise.

- While ***isAlive( )*** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called ***join( )***, shown here:

final void join( ) throws InterruptedException;

- This method waits until the thread on which it is called, terminates.
- Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.

- **Additional forms** of ***join( )*** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.
- Here is an improved version of the preceding example that uses ***join( )*** to ensure that the ***main*** thread is the last to stop. It also demonstrates the ***isAlive( )*** method.

```
//Using join() to wait for threads to finish.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
        NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start( ); // Start the thread
        }

// This is the entry point for thread.
public void run( ) {
        try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);}
} catch (InterruptedException e) {
System.out.println(name + " interrupted."); }
System.out.println(name + " exiting.");
        }//run
}//class
```

```
public class DemoJoin {
        public static void main(String args[ ]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "+ ob1.t.isAlive( ));
System.out.println("Thread Two is alive: "+ ob2.t.isAlive( ));
System.out.println("Thread Three is alive: "+ ob3.t.isAlive( ));

// wait for threads to finish
        try {
System.out.println("Waiting for threads to finish.");
ob1.t.join( );
ob2.t.join( );
ob3.t.join( );
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive( ));
System.out.println("Thread Two is alive: "+ ob2.t.isAlive( ));
System.out.println("Thread Three is alive: "+ ob3.t.isAlive( ));
System.out.println("Main thread exiting.");
        }//main
}//class
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```


As you can see, after the calls to ***join( )*** , the threads have stopped executing.

Iterative Server vs. Concurrent Server

- An ***iterative*** server is a server program that handles one client at a time. If one or more client connection requests reach the server while the latter is in communication with a client, these requests have to wait for the existing communication to be completed.
- The pending client connection requests are handled on a First-In-First-Serve basis. However, such a design is not efficient.

Code for a Client Program that Requests a Server to Add Integers

```
//Summation Client
import java.io.*;
import java.net.*;
public class summationClient{
        public static void main(String[ ] args){
try{
        String serverHost = "localhost";
        int serverPort = 8008;
        int count = 7;
        if (args.length > 0) {serverHost = args[0];}
        if (args.length > 1) {serverPort = Integer.parseInt(args[1]);}
        if (args.length > 2) {count = Integer.parseInt(args[2]);}//counter used by the server
long startTime = System.currentTimeMillis( );
Socket clientSocket = new Socket(serverHost, serverPort);
PrintStream ps = new PrintStream(clientSocket.getOutputStream());
ps.println(count);//send to server that returns the sum: from 1 to count
BufferedReader br = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
int sum = Integer.parseInt(br.readLine( ));
System.out.println(" sum = "+sum);
long endTime = System.currentTimeMillis( );
System.out.println("\n Time consumed for receiving the feedback from the server:"
+ (endTime-startTime)+" milliseconds");
clientSocket.close( );}
catch(Exception e){e.printStackTrace( );}
        }//main
}//class
```

Code for an Iterative Server that Adds (from *1* to a *count* value) and returns the *sum*

```
//iterative summation Server
import java.io.*;
import java.net.*;

public class summationServer{
		@SuppressWarnings("resource")
		public static void main(String[ ] args){
try{
		int serverPort = 8008;
		if (args.length > 0) {serverPort = Integer.parseInt(args[0]);}
ServerSocket calcServer = new ServerSocket(serverPort);
while (true){
Socket clientSocket = calcServer.accept( );
BufferedReader br = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream( )));
int count = Integer.parseInt(br.readLine( ));//Server receives the counter from the client
int sum = 0;
for (int ctr = 1; ctr <= count; ctr++){
sum += ctr;
Thread.sleep(200);}
PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
ps.println(sum);//Server sends the result to client
ps.flush( );
clientSocket.close( );}
		}catch(Exception e){e.printStackTrace( );}
				}
}
```

Concurrent Server

- An alternative design is the idea of using a concurrent server, especially to process client requests with variable service time. When a client request is received, the server process spawns a separate thread, which is exclusively meant to handle the particular client.
- So, if a program has to sleep after each addition, it would be the particular thread (doing the addition) that will sleep and not the whole server process, which was the case with an iterative server.
- Note that the code for **the client program is independent** of the design choice for the server.
- In other words, one should be able to use the same client program with either an iterative server or a concurrent server

Classical Concurrent Server Program and the Implementation of a Summation Thread

```
//Concurrent Server
import java.io.*;
import java.net.*;
class summationThread extends Thread{
Socket clientSocket;
summationThread(Socket cs){ clientSocket = cs; }
        @Override
        public void run( ){
        try{
BufferedReader br = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream( )));
int count = Integer.parseInt(br.readLine( ));//counter from client
int sum = 0;
for (int ctr = 1; ctr <= count; ctr++){
sum += ctr;
Thread.sleep(200);          }//for
PrintStream ps = new PrintStream(clientSocket.getOutputStream( ));
ps.println(sum);//sum to client
ps.flush( );
clientSocket.close( );}catch(Exception e){e.printStackTrace( );}
            }//run
}//end_class_sT
```

```
public class summationCServer{
        @SuppressWarnings("resource")
        public static void main(String[ ] args){
try{
        int serverPort = 8008;
        if (args.length > 0) {serverPort = Integer.parseInt(args[0]);}
ServerSocket calcServer = new ServerSocket(serverPort);

while (true){
Socket clientSocket = calcServer.accept( );//accepts all clients
summationThread thread = new summationThread(clientSocket);
thread.start( );
        }
}
catch(Exception e){e.printStackTrace( );}
                                }//main
                        }//class_CS
```

Result:

- sum = 28

- Time consumed for receiving the feedback from the server:1427 milliseconds

Screenshots of Execution of a Concurrent Summation Server and its Clients

```
C:\res\tutorial\sockets\server>javac summationServerConcurrent.java

C:\res\tutorial\sockets\server>java summationServer 3456
```

```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 100
 sum = 5050
 Time consumed for receiving the feedback from the server: 20031 milliseconds

C:\res\tutorial\sockets\client>
```

```
C:\res\tutorial\sockets\client>java summationClient localhost 3456 5
 sum = 15
 Time consumed for receiving the feedback from the server: 1000 milliseconds

C:\res\tutorial\sockets\client>
```

Thread Priority

- Most computer configurations have a single CPU, so threads actually run one at a time in such a way as to provide an ***illusion of concurrency***.
- Execution of multiple threads on a single CPU, in some order, is called ***scheduling***.
- The Java runtime supports a very simple, deterministic scheduling algorithm known as ***fixed priority scheduling***. This algorithm schedules threads based on their ***priority*** relative to other runnable threads.

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread **priorities are integers** that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless; a **higher-priority thread** doesn't run any faster than a lower-priority thread if **it is the only** thread running.
- Instead, a thread's **priority is used to decide when to switch** from one running thread to the next.

- This is called a *context switch.* The rules that determine when a context switch takes place are simple:

- A *thread* can ***voluntarily*** *relinquish control.* This is done by **explicitly** yielding, sleeping, or blocking on pending I/O.

In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- A *thread* can be ***preempted*** *(be on the second place)* by a *higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted, no matter what it is doing by a higher-priority thread.

Basically, **as soon as a higher-priority thread wants to run, it does**. This is called *preemptive multi-threading.*

What is pre-emptive and non-preemptive scheduling?

- Tasks are usually assigned with **priorities**.

At times it is necessary to run a certain task that has a higher priority before another task although it is running.

Therefore, the running task is **interrupted** for some time and resumed later when the priority task has finished its execution. This is called **preemptive** scheduling.

Example: Round Robin

In **non-preemptive** scheduling, a running task is executed till **completion**. It cannot be interrupted.
Example: First In First Out

- In cases where **two threads** with the **same priority** are competing for CPU cycles, the situation is a bit complicated.
- For operating systems such as **preemptive Windows OS**, threads of equal priority are **time-sliced** automatically in **round-robin** fashion.
- For **other types of operating systems**, threads of equal priority **must voluntarily yield** control to their peers.
- If they don’t, the others threads will not run.

Context switch rules (preemptive OS)

1. The **higher the integer**, the *higher the priority*. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution.

2. A thread with **higher priority** *will interrupt* other threads with *lower priority*. Only when that threads stops, yields, or becomes not runnable for some reason, a lower priority thread will start executing (preemptive mechanism)

3. If two threads of the **same priority** are waiting for the CPU, the scheduler chooses one of them to run in a *round-robin* or *other* fashion, depending on the OS. In Windows now it is used a *time-slicing* process.

- When a Java thread is created, it **inherits its priority** from the *thread that created* it. You can also modify a thread's priority at any time after its creation using the ***setPriority()*** method:

  final void setPriority(int level);

- Here, *level* specifies the new priority setting for the calling thread.
- Thread priorities are **integers** ranging between **MIN_PRIORITY** (1) and **MAX_PRIORITY (10)** (constants defined in the *Thread* class).
  - *public final static int MAX_PRIORITY; //10*
  - *public final static int MIN_PRIORITY; //1*
  - *public final static int NORM_PRIORITY; //5*
- The priority in Java is relative, the **order is established** considering the *context switch* process.

- You can obtain the current priority setting by calling the ***getPriority( )*** method of ***Thread*** class, shown here:

  final int getPriority( );

- Implementations of Java may have radically different behavior when it comes to scheduling.
- The Windows OS works, more or less, as you would expect. However, other versions may work quite differently.
- Most of the **inconsistencies** arise when you have threads that are relying on *preemptive* behavior, instead of *cooperatively* giving up CPU time.

- The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.
- The following example demonstrates **two threads at different priorities**, which **do not run** on a *preemptive* platform **in the same way** as they run on a *non-preemptive* platform.
- **One thread** is set *two levels above* the normal priority, as defined by **Thread.NORM_PRIORITY**, and the **other** is set to *two levels below* it. The threads are started and allowed to *run for ten seconds*. Each thread executes a loop, counting the number of iterations. After *ten seconds*, the **main thread stops** *both threads*.
- The *number of times* that each thread made it through the loop is then *displayed*.

```
//Demonstrate thread priorities.
class Clicker implements Runnable {
long click = 0;
Thread t;
private volatile boolean running = true;
//private boolean running = true;

public Clicker(int p) {
t = new Thread(this);
t.setPriority(p); }

public void run( ) {
while (running) {
click++; }
        }//run

public void stop( ) {
running = false;}//stop

public void start( ) {
t.start( ); }//start
}//Clicker_class
```

```
public class HiLoPriority {
        public static void main(String args[ ]) {
Thread.currentThread( ).setPriority(Thread.MAX_PRIORITY);
Clicker hi = new Clicker(Thread.NORM_PRIORITY + 2);
Clicker lo = new Clicker(Thread.NORM_PRIORITY - 2);
lo.start( );
hi.start( );
try {
Thread.sleep(10000);
    } catch (InterruptedException e)
                {System.out.println("Main thread interrupted.");}

lo.stop( );
hi.stop( );
// Wait for child threads to terminate.
try {
hi.t.join( );
lo.t.join( );
    } catch (InterruptedException e)
                {System.out.println("InterruptedException caught");}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
                        }//main
}//class_HiLoPri
```

- The output of this program, shown as follows when run under different Windows OS, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O.
- The *higher-priority thread* got approximately **90 percent** of the CPU time in W98. Other results:
- *Low-priority thread: 4408112-W98*|980922144-W7
- *High-priority thread: 589626904-W98*|988280225-W7
- **Other results W7**:
- *Low-priority thread: 223657882*
- *High-priority thread: 438326511*
- **W10/I7 processor:**
- Low-priority thread: 5288348591
- High-priority thread: 5296335061
- Of course, the **exact output produced** by this program *depends on* the speed of your CPU and the number of other tasks running in the system.
- When this same program is run under a ***non-preemptive*** system, different results will be obtained**.**

- One other note about the preceding program. Notice that *boolean* ***running*** is preceded by the keyword ***volatile***.
- Although ***volatile*** it is used here to ensure that the value of ***running*** is examined each time the following loop iterates:

```
while (running) {
click++;
        }
```

- Without the use of ***volatile***, Java is free to optimize the loop in such a way that a local copy of ***running*** is created.
- The use of ***volatile*** prevents this optimization, telling Java that ***running*** may change in ways not directly apparent in the immediate code.

- But ***volatile*** tells the compiler:
  - other threads might see *reads/writes* of this variable
  - don't change/reorder or eliminate the reads and writes

- An optimizing compiler should, if it sees:

  x++; *x*--;

- replace it with a number

  if *x* isn't volatile

Insufficient atomicity

- Very frequently, you will want a sequence of actions to be performed atomically or indivisibly
  - not interrupted or disturbed by actions by any other thread
- *x*++ isn't an atomic operation
  - it is a *read* followed by a *write*
- Can be an intermittent error
  - depends on exact interleaving
- *concurrent* package from Java 5 has a performant mechanism to manage the atomicity

```
public class InsufficentAtomicity implements
   Runnable {
    static int x = 0;

   public void run( ) {
        int tmp = x;
        x = tmp+1;
    }

  public static void main(String[ ] args) {
        for (int i = 0; i < 3; i++)
        new Thread(new InsufficentAtomicity( )).start( );
        System.out.println(x); // may not be 3
    }
}
```

Thread Groups

- Every Java thread is a member of a *thread group*.
- Thread groups provide a mechanism for collecting *multiple threads into a single object* and **manipulating** those threads *all at once*, rather than individually.
- For example, you can **start or suspend all the threads** within a group *with a single method* call.
- Java thread groups are implemented by the ***ThreadGroup*** class in the **java.lang** package.
- The root of all threads is the thread group ***system*** created initially by JVM.
- In this group JVM creates the ***main*** thread group that will call the *main( )* method with a *run( )* method.
- Now the programmer may creates the own threads that belongs to the ***main*** group

Daemon Threads

- Java threads types:
  - User
  - Daemon:
    - Provide general services
    - Typically never terminate
    - Call *setDaemon( )* before *start( )*
- Program termination
  - If all non-daemon threads terminate, JVM shuts down

- The **runtime system** puts a thread into a *thread group* during thread *construction*.
- A ***Daemon*** *thread offers services* to other threads.
- When you create a thread, you can either allow the runtime system to put the **new thread** in some reasonable *default group* or you can explicitly set the *new thread's group*.
- The thread is a *permanent member* of whatever thread group it joins upon its **creation**-you *cannot move* a thread to a new group after the thread has been created.

- If you create **a new thread** without specifying its group in the constructor, the **runtime system automatically places** the new thread in the *same group as the thread that created it* (known as the *current thread group* and the *current thread*, respectively).

- If you wish to put your **new thread** in a thread group *other than the default*, you **must specify the thread group explicitly** when you **create** the thread.

- The ***Thread*** class has three constructors that let you set a new thread's group:

public Thread(ThreadGroup group, Runnable runnable);

public Thread(ThreadGroup group, String name);

public Thread(ThreadGroup group, Runnable runnable, String name);

- Each of these constructors creates a new thread, initializes it based on the *Runnable* and *String* parameters, and makes the new thread a member of the specified group.

- To find out what group a thread is in, you can call its ***getThreadGroup()*** method:

  theGroup = myThread.getThreadGroup( );

- Once you've obtained a thread's *ThreadGroup*, you can query the group for information, such as what other threads are in the group. You can also modify the threads in that group, such as suspending, resuming, or stopping them, with a single method invocation.

- The ***ThreadGroup*** class manages groups of threads for Java applications. A *ThreadGroup* can contain any number of threads.
- The threads in a group are **generally related** in some way, **such as** *who created* them, *what function* they perform, or *when they should be started and stopped*.

- ***ThreadGroup*** can **contain** not only threads but also **other** *ThreadGroup*.

- The **top-most thread group** in a **Java** application is the thread group named ***main***.
- You can create threads and thread groups in the main group. You can also create threads and thread groups in subgroups of ***main***

- Methods used to manage a thread group:

    public int activeCount( ); // ret no. of active threads

    void list( );// prints information about this thread group to the standard output

    public int enumarate(Thread list[ ]);//lists the threads

    public int enumarate(ThreadGroup list[ ]);//lists the groups and sub-groups

- The methods that *get* and *set ThreadGroup* attributes **operate at the group level**. They inspect or change the attribute on the *ThreadGroup* object, but **do not affect any of the threads within the group**.
- The following is a list of *ThreadGroup* **methods** that *operate at the group level*:

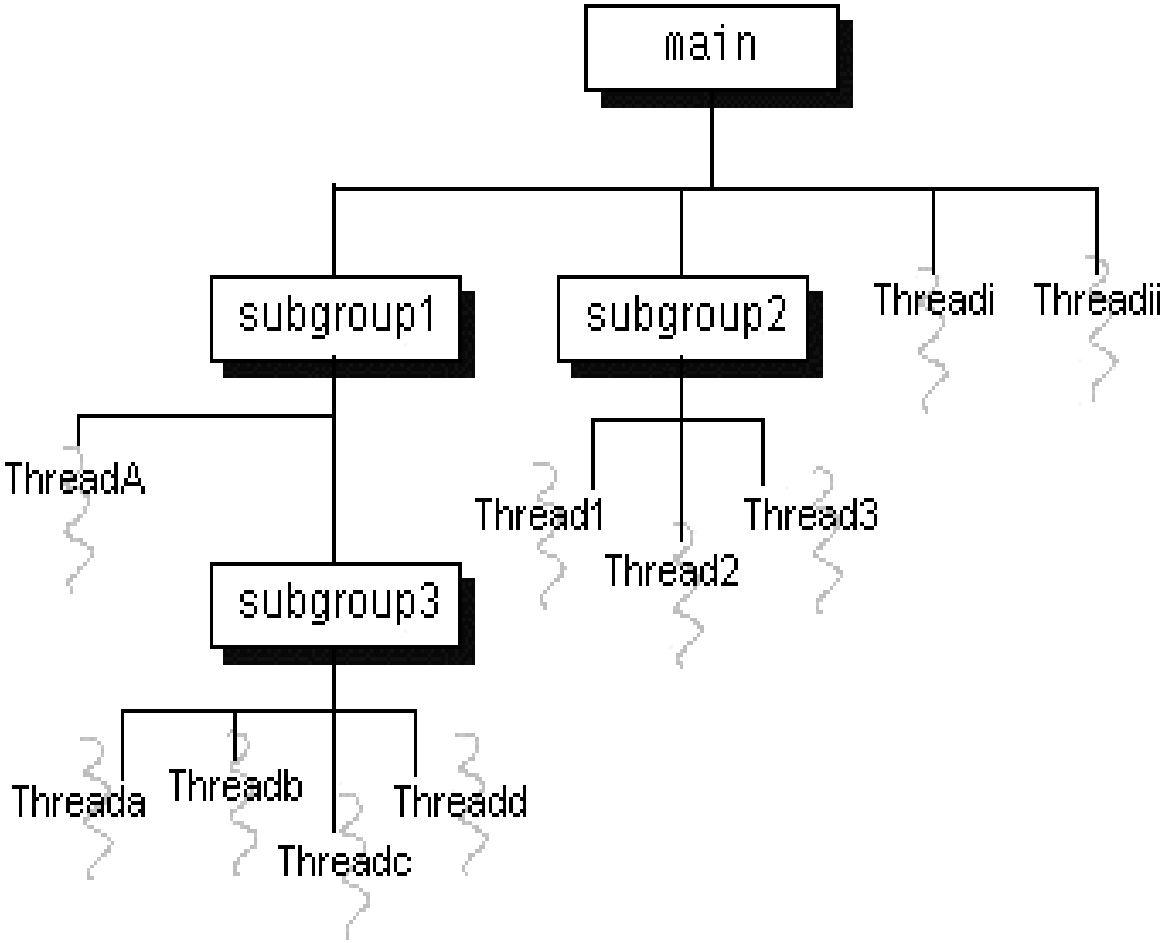
  getMaxPriority( ) and *setMaxPriority( ),* (highest priority for members)

  getDaemon( ) and *setDaemon( ),*

  getName( ),

  getParent( ) and *parentOf( ),*

  toString( ), etc.
- The ***ThreadGroup*** class has three methods that allow you to **modify the current state of all the threads within that group**: *resume( ), stop( ), suspend( )* (now are deprecated)
- These methods apply the appropriate state change to every thread in the thread group and its subgroups.


main
subgroup1
subgroup2
Threadi
Threadii
ThreadA
Thread1
Thread2
Thread3
subgroup3
Threada
Threadb
Threadc
Threadd

```
/*Simple thread*/
class GroupMemberThread extends Thread {

public GroupMemberThread(ThreadGroup group, String threadName){
super(group, threadName);}

@SuppressWarnings("static-access")
        @Override
        public void run( ){
boolean run_it = true;
while (run_it){
System.out.println(this.getName() + " running");

try
{
System.out.println("Putting " + this.getName( ) + " to sleep");
this.sleep(1000);
}catch(InterruptedException e){
System.out.println(this.getName( ) + " interrupted");
run_it = false; }
}//while
        }//run
}//class
```

```
/** This class creates a group of threads. */
public class ThreadGroupExample{
        public static void main(String args[ ]){
System.out.println("creating a thread group");
ThreadGroup group = new ThreadGroup("My Group");

Thread thread1 = new GroupMemberThread(group, "thread1");
thread1.setPriority(Thread.MIN_PRIORITY);
Thread thread2 = new GroupMemberThread(group, "thread2");
thread2.setPriority(Thread.MIN_PRIORITY);
Thread thread3 = new GroupMemberThread(group, "thread3");
thread3.setPriority(Thread.MAX_PRIORITY);
group.list( );
thread1.start( );
thread2.start( );
thread3.start( );

Thread[ ] threads = new Thread[group.activeCount( )];
group.enumerate(threads);
for(int i = 0; i < threads.length; i++){
System.out.println("Interrupting thread "+threads[i].getName( ));
threads[i].interrupt( );}
        }//main
}//class
```

```
creating a thread group
java.lang.ThreadGroup[name=My Group,maxpri=10]
thread1 running
Interrupting thread thread1
thread3 running
Putting thread3 to sleep
thread2 running
Putting thread2 to sleep
Interrupting thread thread2
Interrupting thread thread3
thread3 interrupted
thread2 interrupted
Putting thread1 to sleep
thread1 interrupted
```

Multithreading in current Java applications methods

- Starting with **JDK 1.5** the initial multithreading mechanism based on ***start( )*** method is not so used in current Java applications, where we use the new package: *concurrent*.

As initial (lower) methods are used:

- ***a) Implement Runnable interface: ThreadObj.start( ) –*** start *someRunnable* object from *Runnable*

– Implement *Runnable*, pass to *Thread* constructor, call *start( ):*

Thread t = new Thread(someRunnable);

t.start( );

- ***b) Extend Thread class***

– Put *run( )* method in *Thread* subclass, instantiate, call *start( )*

SomeThread t = new SomeThread(…);

t.start( );

– About same effect as ***taskList.execute(someRunnable)***, that is now used from *concurrent* package, except that you cannot put bound (limit) on number of simultaneous threads.

– Mostly a carryover from pre-Java-5 days; still widely used.

c) Executor Framework

- Starting with Java 5, the ***Executor* framework** was introduced with the *java.util.concurrent.Executor* interface.
- A framework is created to standardize the invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies.

- The ***Executors* class** offers utility methods for the **interfaces**: *Executor, ExecutorService, ScheduledExecutorService, ThreadFactory*, and *Callable*.

- The *Executors* class can be used to easily create a *ThreadPool* (thread pool - queues) in Java, and it also supports the execution of *Callable* implementations.

- Together with *Executors*, the *Callable* and *Future* interfaces are introduced to extend *Runnable* and allow *management, return* and *exception* handling.

Executors support interfaces

- java.util.concurrent.Executor, will offer the method:

void execute (Runnable command);

- java.util.concurrent.ExecutorService, derived from Executor, adds:

Future<?> submit(Runnable task) that will return a *Future* object

- java.util.concurrent. ScheduledExecutorService, derived from ExecutorService adds:

<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit), to execute *Future* or scheduled processes

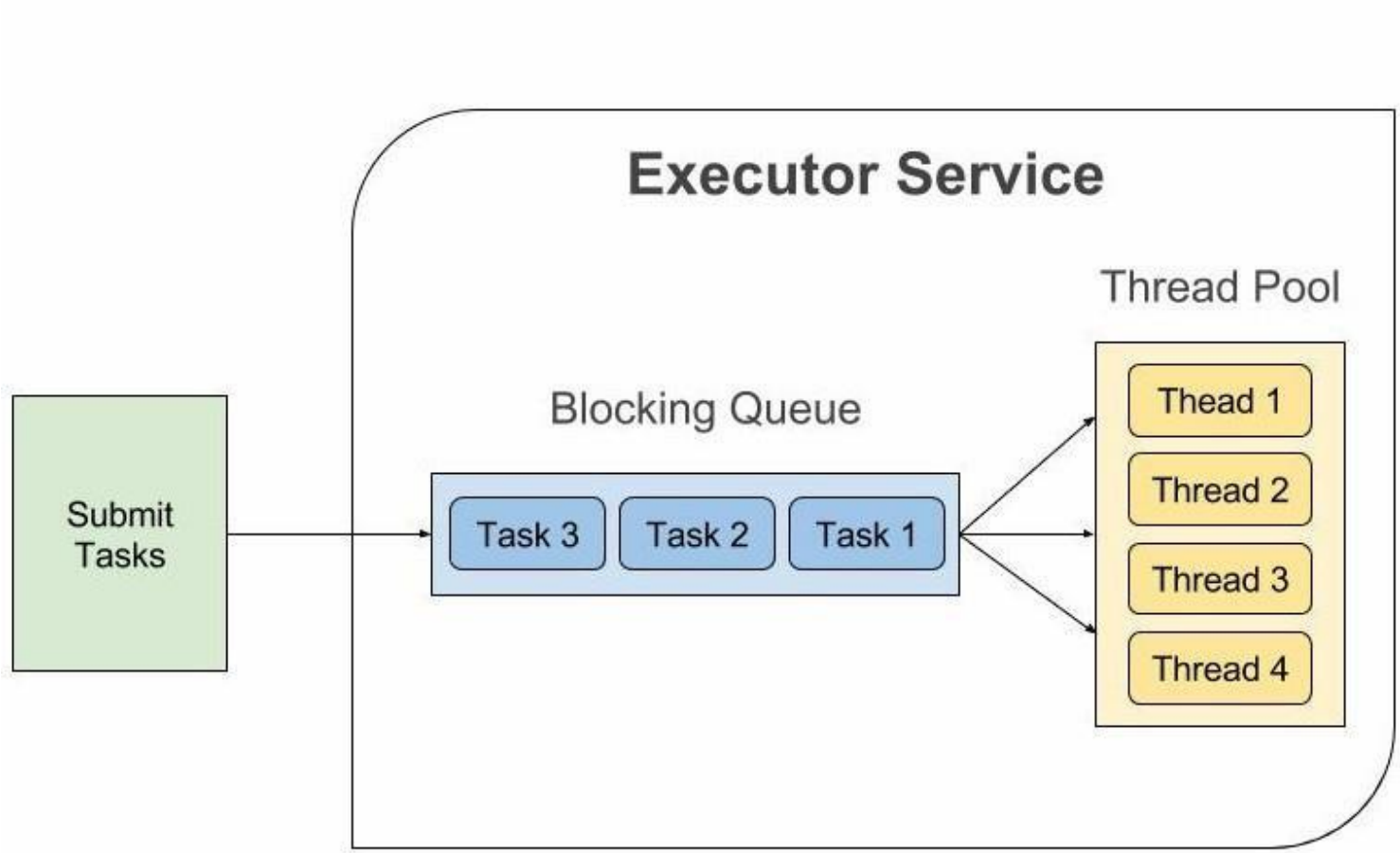
- java.util.concurrent.ThreadFactory –

Java also provides an interface, the *ThreadFactory* interface, to create your own *Thread* object factory. Using *ThreadFactory* you can customize the threads created by *executor* so that they have proper thread names, priority or even they can be set to be daemon also.

- java.util.concurrent.Callable

For implementing *Runnable*, the *run( )* method needs to be implemented which does not return anything, while for a *Callable*, the *call( )* method needs to be implemented which returns a result on completion. Note that a thread can't be created with a *Callable*, it can only be created with a *Runnable*.

*https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/***.html*


Executor Service
Thread Pool
Blocking Queue
Submit Tasks
Task 3
Task 2
Task 1
Thead 1
Thread 2
Thread 3
Thread 4

Runnable, Callable, Future

• **Runnable**

– "*run( )*" method, runs in background. **No return values**, but *run( )* can do *side effects*.

– Use "*execute( )*" to put in task queue

• **Callable**

Callable interface has the *call( )* method. In this method, we have to implement the logic of a task. The *Callable* interface is a generic interface, meaning we have to indicate the type of data the *call( )* method will return.

– "*call( )*" method, runs in background. It **returns a value** that can be retrieved after termination with "*get( )*".

– Use "*submit( )*" to put in task queue.

– Use *invokeAny( )* and *invokeAll( )* to **block until value or values are available**

• **Future**

The new *Future* interface is used with *Callable* and serves as a handle to *wait* and *retrieve* the result of the task or *cancel* the task before it is executed.

A *Future* object is **returned by the** *submit( )* method of an *ExecutorService*.

Think of a *Future* as an object that **holds the result** – it may not hold it right now, but it will do so in the future (once the *Callable* returns). Thus, a *Future* is basically one way the *main* thread can keep track of the progress and result from other threads.

Types of Task Queues

• ***Executors.newFixedThreadPool(nThreads)***
– Simplest and most widely used type. Makes a list of tasks to be run in the background, but with a warning that there are never more than *nThreads* simultaneous threads running.
• ***Executors.newScheduledThreadPool( )***
– Lets you define tasks that run after a delay, or that run periodically. Replacement for pre-Java-5 “Timer” class.
• ***Executors.newCachedThreadPool( )***
– Optimized version for apps that start many short-running threads. Reuses thread instances.
• ***Executors.newSingleThreadExecutor( )***
– Makes queue of tasks and executes one at a time

***ExecutorService* (subclass) constructors**
– Lets you build FIFO, LIFO, and priority queues

New methods in *ExecutorService*

• ***execute(Runnable)***

– Adds *Runnable* to the queue of tasks

```
Runnable task = ( ) -> { String threadName =
Thread.currentThread( ).getName( );
System.out.println("Hello " + threadName);
};
ExecutorService executor =  Executors.newFixedThreadPool(5);
executor.execute(task); //equivalent with (new Thread(task)).start( );
```

• ***shutdown( )***

– Prevents any more tasks from being added with *execute( )* (or *submit( ))* but lets current tasks finish.

• ***shutdownNow( )***

– Attempts to halt current tasks. But author of tasks must have them respond to interrupts (ie, catch *InterruptedException*), or this is no different from *shutdown( ).*

• ***awaitTermination( )***

– Blocks until all tasks are complete. Must *shutdown( )* first.

• ***submit( ), invokeAny( ), invokeAll( )***

– Variations that use *Callable/Future* instead of *Runnable*.

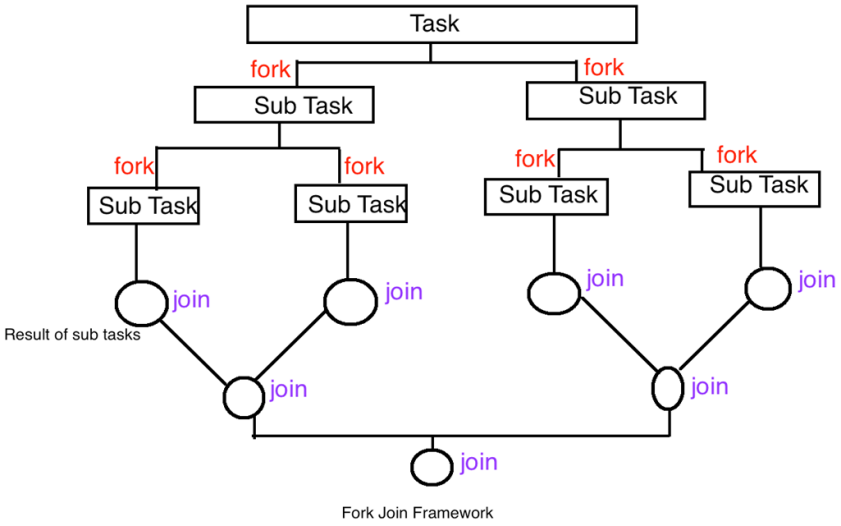
```
Callable<String> callable(String result, long  sleepSeconds) {
return ( ) -> {  TimeUnit.SECONDS.sleep(sleepSeconds);  return result;
                        };
}
ExecutorService executor = Executors.newWorkStealingPool( );
…
List<Callable<String>> callables = Arrays.asList (
callable("task1", 2),
callable("task2", 1),
callable("task3", 3));

String result = executor.invokeAny(callables);  System.out.println(result);
```

Fork/Join Framework

= set of API-s to use the advantages of parallel programming based on multi-cores;

The principle of **Fork/Join** framework (divide et impera):

if (problemSize < threshold)
solve problem directly
else {
break problem into subproblems
recursively solve each problem
combine the results
}



Example *execute( )*

```
//Threads- concurrent package
import java.util.concurrent.*;

class App1 extends SomeClass {
public App1( ) {
ExecutorService taskList = Executors.newFixedThreadPool(100);
taskList.execute(new Counter(this, 6));
taskList.execute(new Counter(this, 5));
taskList.execute(new Counter(this, 4));
taskList.shutdown( );
}
public void pause(double seconds) {
try {
Thread.sleep(Math.round(1000.0 * seconds));
} catch (InterruptedException ie) { }
}//pause
}

class SomeClass{ }
```

```
class Counter implements Runnable {
private final App1 mainApp;
private final int loopLimit;

public Counter(App1 mainApp, int loopLimit) {
this.mainApp = mainApp;
this.loopLimit = loopLimit;}

public void run( ) {
for(int i=0; i<loopLimit; i++) {
String threadName = Thread.currentThread( ).getName( );
System.out.printf("%s: %s%n", threadName, i);
mainApp.pause(Math.random( ));}
        }
}//Counter class

public class ExecutorApp1Test {
public static void main(String[ ] args) {
new App1( );
 }
        }
```

Some Results

pool-1-thread-1: 0
pool-1-thread-2: 0
pool-1-thread-3: 0
pool-1-thread-2: 1
pool-1-thread-2: 2
pool-1-thread-1: 1
pool-1-thread-3: 1
pool-1-thread-2: 3
pool-1-thread-3: 2
pool-1-thread-1: 2
pool-1-thread-1: 3
pool-1-thread-1: 4
pool-1-thread-3: 3
pool-1-thread-2: 4
pool-1-thread-1: 5

Example: https://www.geeksforgeeks.org/callable-future-java/

```
//Java program to illustrate Callable and FutureTask
//for random number generation

import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

@SuppressWarnings("rawtypes")
class CallableExample implements Callable
{
public Object call( ) throws Exception
{
 Random generator = new Random( );
 Integer randomNumber = generator.nextInt(5);

 Thread.sleep(randomNumber * 1000);

 return randomNumber;
}
}
```

```
public class CallableFutureTest {
@SuppressWarnings({ "rawtypes", "unchecked" })
public static void main(String[ ] args) throws Exception {
 // FutureTask is a concrete class that
 // implements both Runnable and Future
 FutureTask[ ] randomNumberTasks = new FutureTask[5];
 for (int i = 0; i < 5; i++)  {
   Callable callable = new CallableExample( );

   // Create the FutureTask with Callable
   randomNumberTasks[i] = new FutureTask(callable);

   // As it implements Runnable, create Thread
   // with FutureTask
   Thread t = new Thread(randomNumberTasks[i]);
   t.start( );
 }
 for (int i = 0; i < 5; i++) {
   // As it implements Future, we can call get()
   System.out.println(randomNumberTasks[i].get());

   // This method blocks till the result is obtained
   // The get method can throw checked exceptions
   // like when it is interrupted. This is the reason
   // for adding the throws clause to main
 }
}
}
```

Example with generics: https://howtodoinjava.com/java/multi-threading/java-callable-future-example/

```
import java.util.*;
import java.util.concurrent.*;

class FactorialCalculator implements Callable<Integer>
{
     private Integer number;
     public FactorialCalculator(Integer number) {
        this.number = number;
    }
     @Override
    public Integer call( ) throws Exception {
        int result = 1;
        if ((number == 0) || (number == 1)) {
            result = 1;
        } else {
            for (int i = 2; i <= number; i++) {
                result *= i;
                TimeUnit.MILLISECONDS.sleep(20);
            }
        }
        System.out.println("Result for number - " + number + " -> " + result);
        return result;
    }
```

```
public class CallableExampleGenerics
{
    public static void main(String[ ] args)
    {
        ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(2);
        List<Future<Integer>> resultList = new ArrayList< >( );
        Random random = new Random( );
        for (int i=0; i<4; i++){
            Integer number = random.nextInt(10);
            FactorialCalculator calculator  = new FactorialCalculator(number);
            Future<Integer> result = executor.submit(calculator);
            resultList.add(result);
        }
        for(Future<Integer> future : resultList){
             try {
                 System.out.println("Future result is - " + " - " + future.get( ) + "; And Task done is " +
future.isDone( ));
             } catch (InterruptedException | ExecutionException e)
             { e.printStackTrace( ); }
         }
         //shut down the executor service now
         executor.shutdown( );
    }
}
```