Threads and Multithreading

Prof. Anita Agrawal
BITS Pilani k.K.Birla Goa campus

Multithreading

- Multithreading is a feature that allows concurrent execution of two or more threads of a process for maximum utilization of CPU.

- So, threads are light-weight processes within a process.

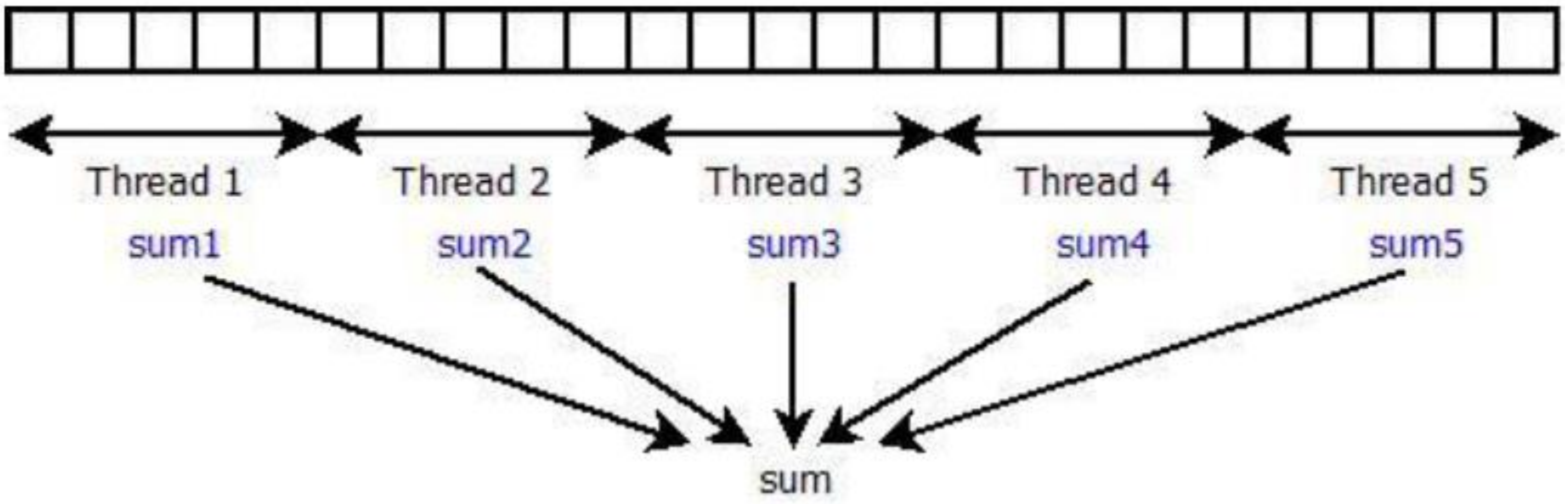
- A thread shares with other threads belonging to the same process the
  - Code section
  - Data section
  - Operating-system resources, such as open files

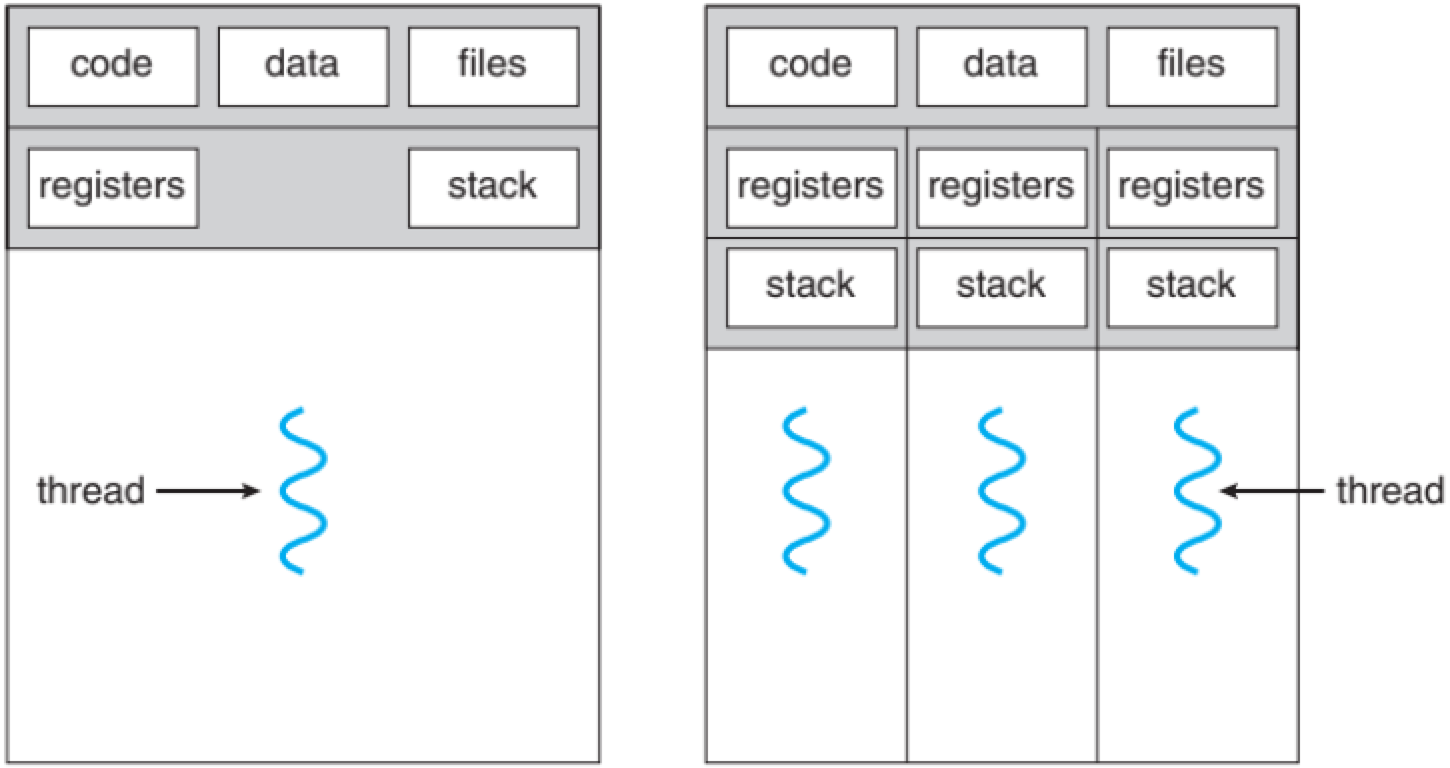
Example 1: A media player

- A media player, where
  - one thread is used for opening the media player,
  - one thread for playing a particular song and
  - one thread for adding new songs to the playlist.

Example 2: Array sum

We can think of threads as child processes that share the parent process resources but execute independently.





single-threaded process

multithreaded process

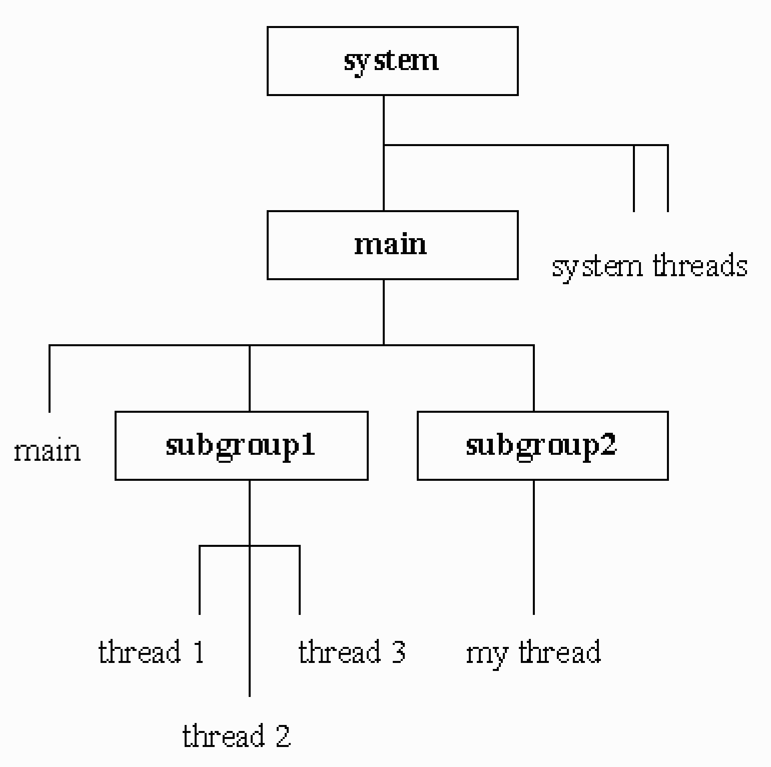
The Main Thread

- When a Java program starts up, one thread begins running immediately

- This is usually called the *mainthread* of your program

- 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

- The main thread is created automatically when your program is started



An application's hierarchical thread-group structure begins with a main thread group just below the system thread group

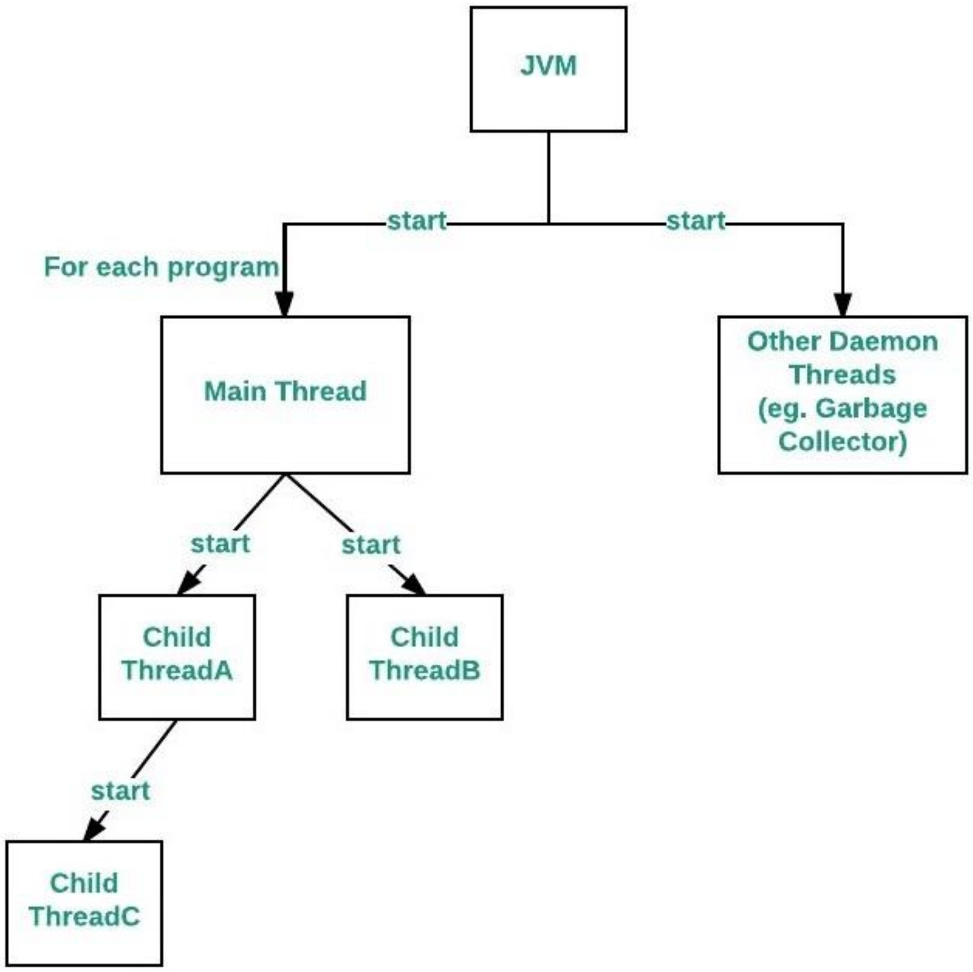
Thread Group

- Java requires every thread and every thread group—save the root thread group, system—to join some other thread group

- **System thread group:** The JVM-created system group organizes JVM threads that deal with object finalization and other system tasks, and serves as the root thread group.

- **Main Thread group**: JVM-created main thread group, which is system's subthread group (subgroup, for short).
  - main contains at least one thread that executes byte-code instructions in the main() method.
- Subgroup 1 and Subgroup 2 subgroups: Application-created subgroups

- Thread 1, Thread 2, and Thread 3: Subgroup 1’s three application-created threads
- my thread: subgroup 2 group’s one application-created thread


JVM

start

start

For each program

Main Thread

Other Daemon Threads (eg. Garbage Collector)

start

start

Child ThreadA

Child ThreadB

start

Child ThreadC

- You can control the main thread through a **Thread** object

- To do so, you must obtain a reference to it by calling the method

  currentThread ( )

  which is public static member of **Thread** class

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

- Once you have a reference to the main thread, you can control it just like any other thread.

Main Thread Example

- A reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**.
- Next, the program displays information about the thread.
- The program calls **setName( )** to change the internal name of the thread
- Information about the thread is then redisplayed
- Next, the loop downcounts from 5 with a pause of 1s in between two consecutive counts
- The thread execution can be paused by the **sleep( )** method

- The argument to **sleep( )** specifies the delay period in milliseconds
- We use **try/catch** block around the loop

- The **sleep( )** method in **Thread** might throw an **Interrupted Exception**

- This would happen if some other thread wanted to interrupt this sleeping one

Main Thread Example:Output

- The output produced when Thread object **t** is used as an argument to **println( )**
- This displays, in order: the **name of the thread, its priority, and the name of its group**
- By default, the name of the main thread is **main**
- Its priority is 5, which is the default value (Highest: 10, Lowest: 1)
- **main** is also the name of the group of threads to which this thread belongs
- You can obtain the name of a thread by calling **getName( )**

Threading in Java

- Threads can be created by two mechanisms:
  - **By implementing the Runnable interface**
  - **By Extending the Thread class**

- To create a new thread, your program will either **extend Thread** or **implement Runnable** interface

- All the examples that have been used a single thread of execution

Creating a Thread: Implementing Runnable interface

- A class need only implement a single method called

  run()

  public void run()

- Inside **run( )**, you will define the code that constitutes the new thread
- **run( )** can call other methods, use other classes, and declare variables, just like the main thread can
- The only difference is that **run( )** establishes the entry point for another thread of execution within your program

Implementing Runnable Interface

- Create a class that implements **Runnable**
- Instantiate an object of type **Thread**
- **Thread** defines several constructors.
- After the new thread is created, it will not start running until you call its **start( )** method

start( ) executes a call to **run( )**

Thread1.java, Thread7.java

- 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().
- **public void run():** is used to perform action for 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.

Thread: Constructor Overloading

- Thread():
- Thread(Runnable target):
- Thread(Runnable target, String name):
- Thread(String name):
- Thread(ThreadGroup group, Runnable target):
- Thread(ThreadGroup group, Runnable target, String name):
- Thread(ThreadGroup group, Runnable target, String name, long stackSize):
- Thread(ThreadGroup group, String name):

Extending Thread class

The second way to create a thread is

- Create a new class that extends **Thread**
- Then create an instance of that class
- The extending class must override the **run( )** method
- It must also call start() to begin execution of the program

Stages of lifecycle of threads

- A thread goes through various stages in its life cycle.
  - For example, a thread is born, started, runs, and then dies.

Stages of Lifecycle of Threads

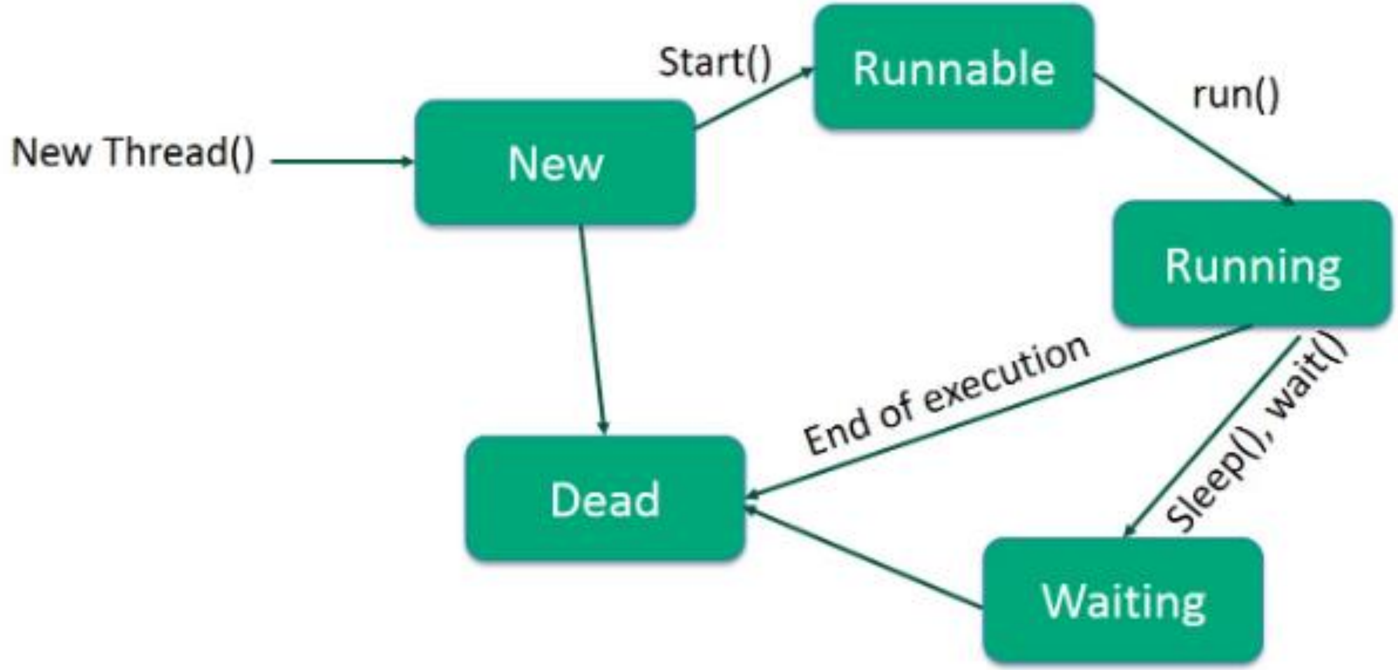
- **New** - When we create an instance of Thread class, a thread is in a new state. A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.

- **Running -** The Java thread is in running state. A thread in this state is considered to be executing its task.

- **Waiting-** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** - A Java thread can be blocked when waiting for a resource.
- **Terminated** - A runnable thread enters the terminated state when it completes its task or otherwise terminated.
  - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.


Start()
Runnable
run()
New Thread()
New
Running
End of execution
Sleep(), wait()
Dead
Waiting

Choosing an approach

- Which one is better….Implementing Runnable or extending class????

- If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

Using isAlive() and join()

- Often you will want the main thread to finish last
  - In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**

- How can one thread know when another thread has ended?
  - **Thread** provides a means by which you can answer this question

    isAlive( ) method defined by **Thread**

    final boolean isAlive( )

- **isAlive( )** is occasionally useful

Using isAlive and Join

- Method that you will more commonly use to wait for a thread to finish is called **join( )**
  - final void join( ) throws InterruptedException
- **join( )**method waits until the thread on which it is called terminates
- It’s name comes from the concept of the calling thread waiting until the specified thread *joins* it
- **join( )** allows you to specify a maximum amount of time that you want to wait for the specified thread to terminate

- public void run(): is used to perform action for a thread.
- public void start(): starts the execution of the thread. JVM calls the run() method on the thread.
- public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- public void join(): waits for a thread to die.
- public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
- public int getPriority(): returns the priority of the thread.
- public int setPriority(int priority): changes the priority of the thread.

- public String getName(): returns the name of the thread.
- public void setName(String name): changes the name of the thread.
- public Thread currentThread(): returns the reference of currently executing thread.
- public int getId(): returns the id of the thread.
- public Thread.State getState(): returns the state of the thread.

- public boolean isAlive(): tests if the thread is alive.
- public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
- public void suspend(): is used to suspend the thread(deprecated).
- public void resume(): is used to resume the suspended thread(deprecated).

- public void stop(): is used to stop the thread(depricated).
- public boolean isDaemon(): tests if the thread is a daemon thread.
- public void setDaemon(boolean b): marks the thread as daemon or user thread.
- public void interrupt(): interrupts the thread.
- public boolean isInterrupted(): tests if the thread has been interrupted.
- public static boolean interrupted(): tests if the current thread has been interrupted.