JAVA Programming

Course Instructor: Dr. N Nandini Devi

School of Computer Science

UPES Dehradun

TOPICs to be discussed

- Introduction to Threads
- Concept of Multi-Threading
- Properties of Threads
- Use of Multi-Threading
- Thread class and its methods
- Life-Cycle of a Thread
- Types of Threads
- Thread Priority

Let’s START …!!!

Concurrent Programming

- This means that tasks appear to run simultaneously, but under the hood, the system might really be switching back and forth between the tasks.
- The point of concurrent programming is that it is beneficial even on a single processor machine.
- **Need of Concurrent Programming:**
  - Threads are useful only when the task is relatively large and pretty much self contained. When the user needs to perform only a small amount of combination after a large amount of separate processing, there's some overhead to starting and using threads. So if the task is really small, one never get paid back for the overhead.
  - Also, as mentioned above, threads are most useful when the users are waiting. For instance, while one is waiting for one server, the other can be reading from another server.

Concurrent Programming

- **Advantages:**
  - Loose Coupling: Since a separate class can be reused, it promotes loose coupling.
  - Constructors: Arguments can be passed to constructors for different cases. For example, describing different loop limits for threads.
  - Race Conditions: If the data has been shared, it is unlikely that a separate class would be used as an approach and if it does not have a shared data, then no need to worry about the race conditions.
- **Disadvantages:**
  - It was a little bit inconvenient to call back to the main application. A reference had to be passed along the constructor, and even if there is access to reference, only public methods(pause method in the given example) in the main application can be called.

Threads in a Program

- **Process:**

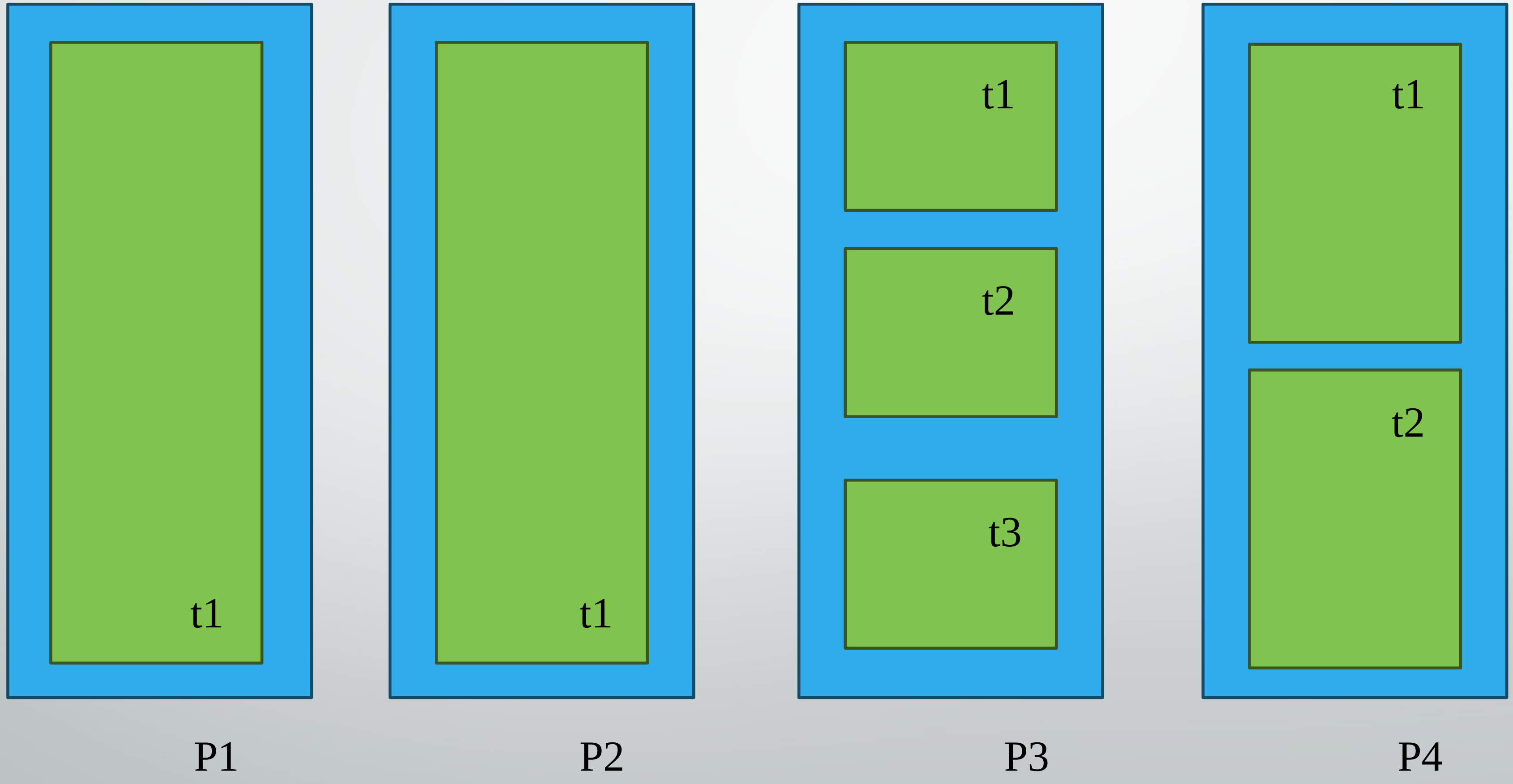
  A process is an independent program in execution, with its own memory space. It is an isolated unit, often running multiple tasks, each of which could involve resources like CPU, memory, and file handles.

- **Threads:**

  A thread is a lightweight, smaller unit within a process. Threads share the same memory space within a process and are often used for executing tasks concurrently within a program.

- In a system, we can have multiple processes running concurrently/simultaneously. **(Multi-processing)**
- A single process may have one or more than one threads. In the former case, the process is often termed **“Single-threaded process”**, whereas the latter one is called a **“Multi-threaded process”**.

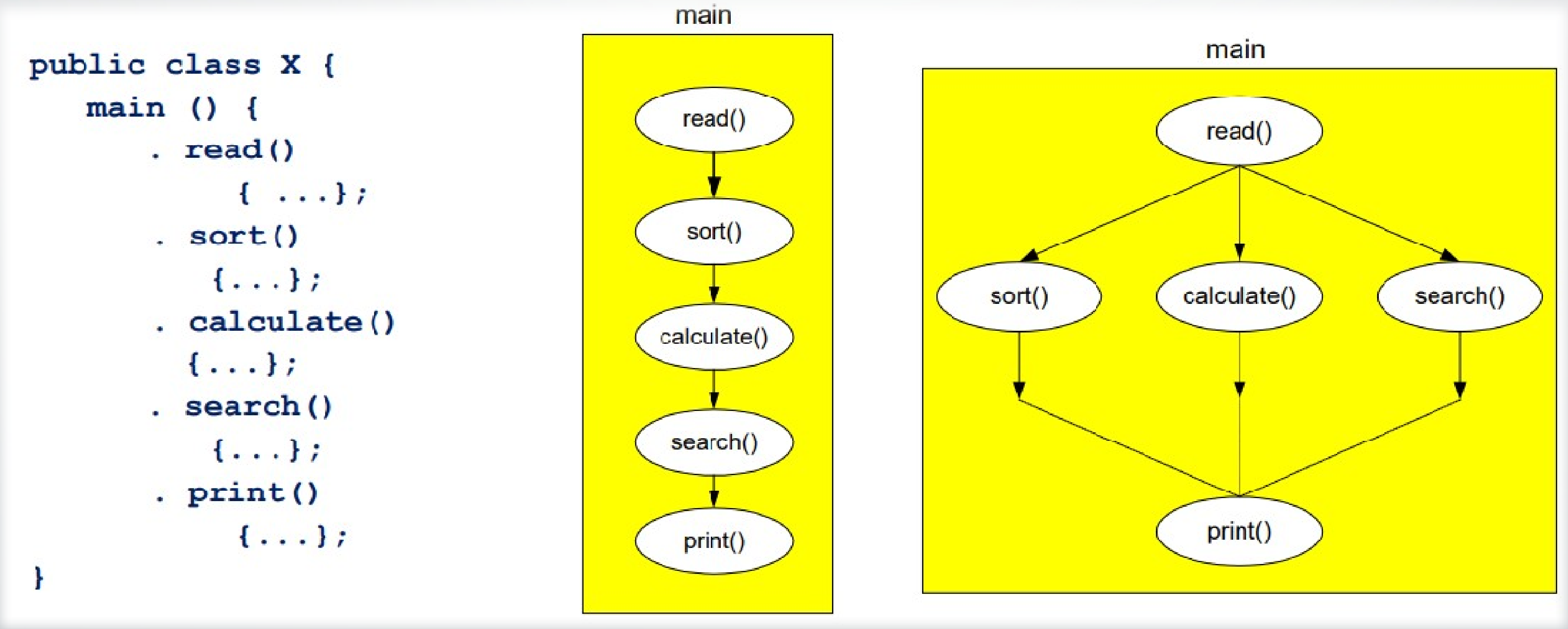
Threads in a Program



Multi-Threading in Java

- **Multi-threading** is a technique that enables concurrent execution of two or more parts of a program, enhancing performance and responsiveness.

- Java **Multi-threading** is mostly used in Games, Animations, etc.



```
public class X {
   main () {
      . read()
          { ...};
       . sort()
         {...};
       . calculate()
        {...};
      . search()
         {...};
       . print()
          {...};
}
```

Properties of Threads

- When we execute a **Java** application,

  - The JVM creates a Thread object whose task is defined by the **main()** method.
  - It starts the thread.
  - The thread executes the statements of the program one by one until the method returns and the thread dies.

- An application may have multiple threads **(Multi-Threading)**, where the threads satisfy following properties:

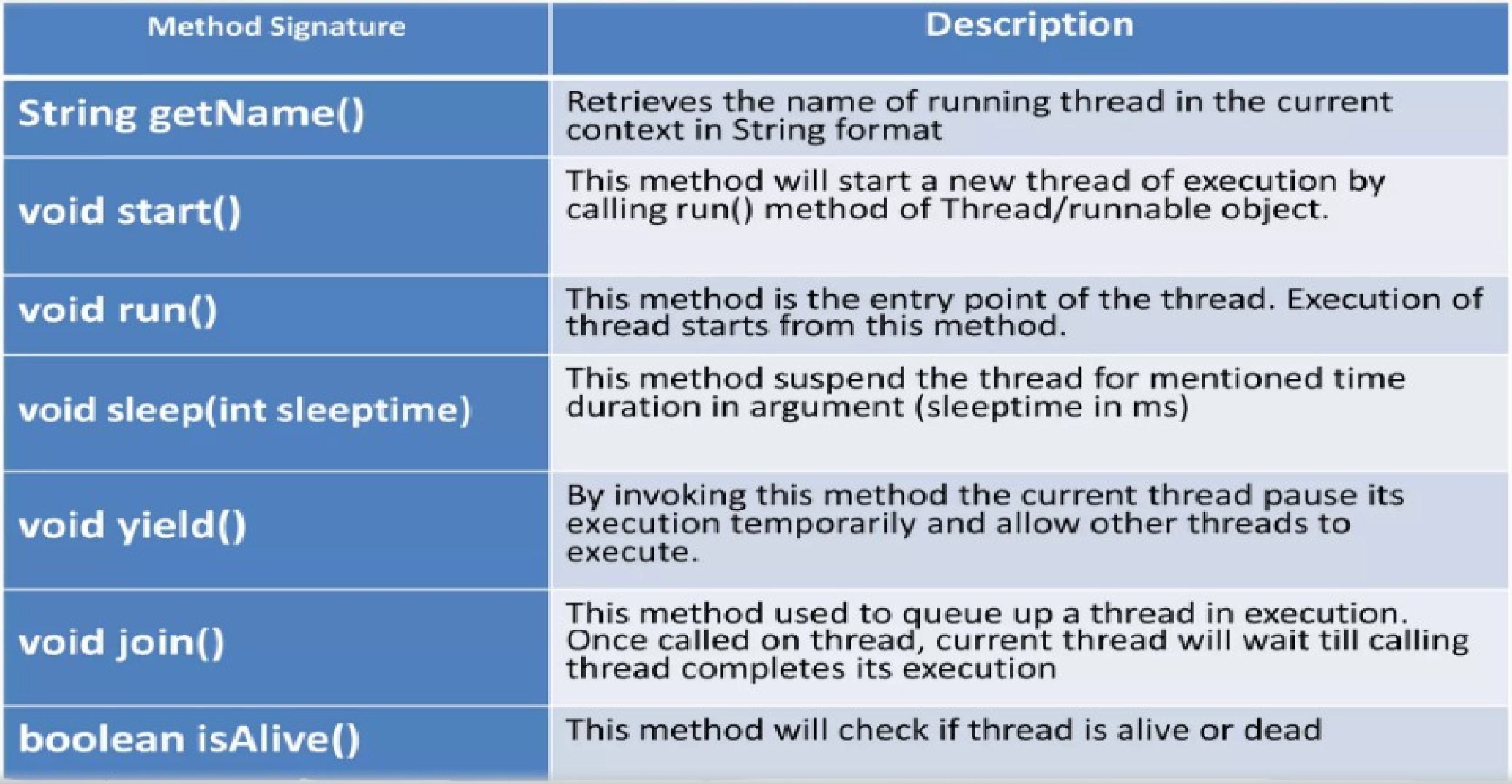
  - ❑ Each thread has its private runtime-stack.
  - ❑ If two threads execute the same method, each will have its own copy of the local variables the method uses.
  - ❑ However, all threads see the same dynamic memory (heap)
  - ❑ Two different threads can act on the same object and same static fields concurrently.

Use of Multi-Threading

- **Multi-Threading** is mainly useful in the following scenarios:
  - ❑ When a **Java** application has some long-running task inside it, to maintain the responsiveness of the program, **Multi-Threading** becomes useful.
  - ❑ If some tasks are separable in a running program, then, for cancellation of few tasks (when required), **Multi-Threading** comes into the picture.
  - ❑ When a particular program has few tasks/problems which can be run parallelly, then the **Multi-Threading** can/should be used.
  - ❑ To monitor status of some resources, like database management, etc., **Multi-Treading** can be handy.
  - ❑ Some APIs and system demand **Multi-Threading** explicitly. For example, **Java Swing**.

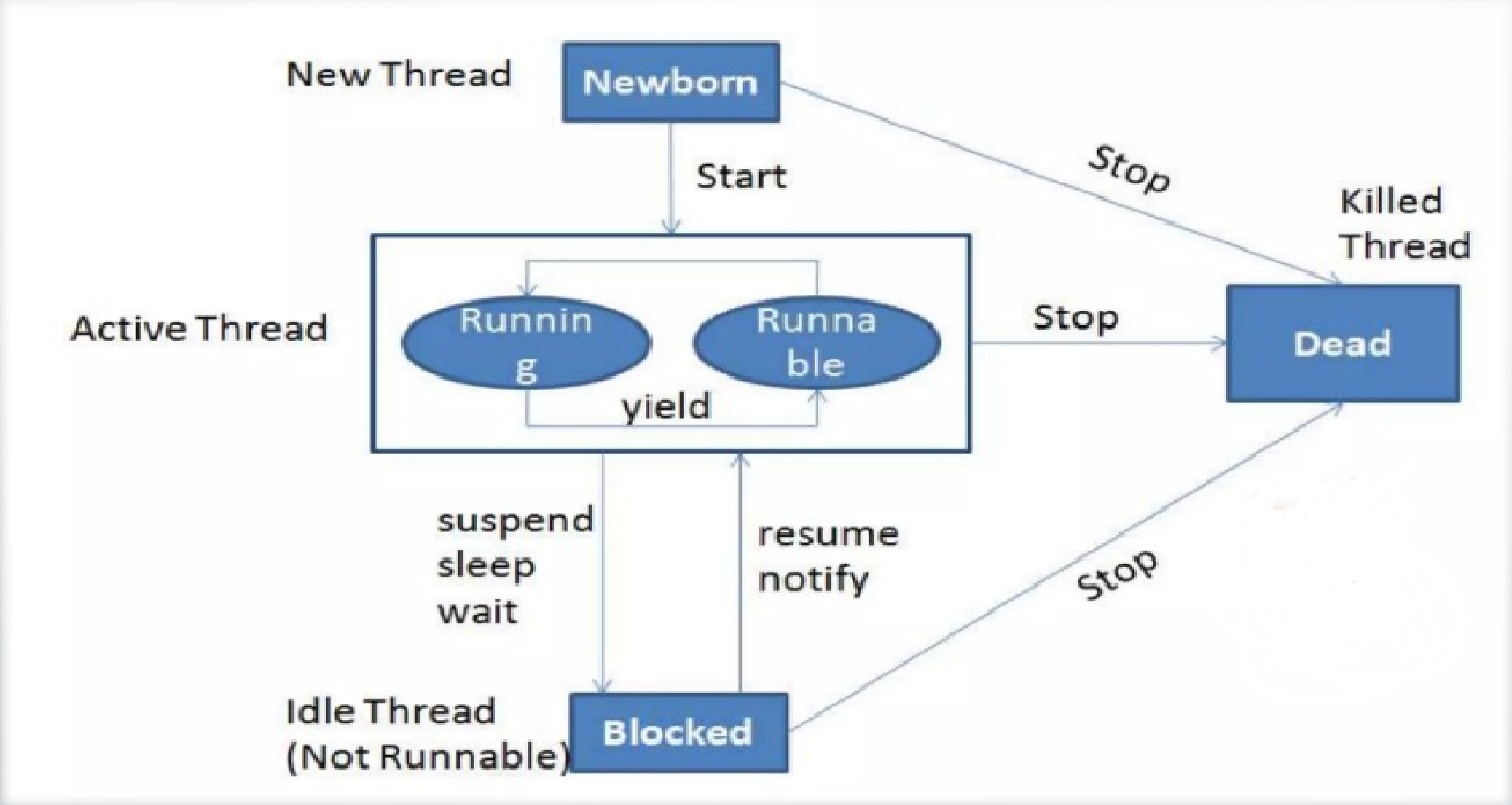
Thread class and its methods

- **Thread** is a class found inside the **java.lang** package.



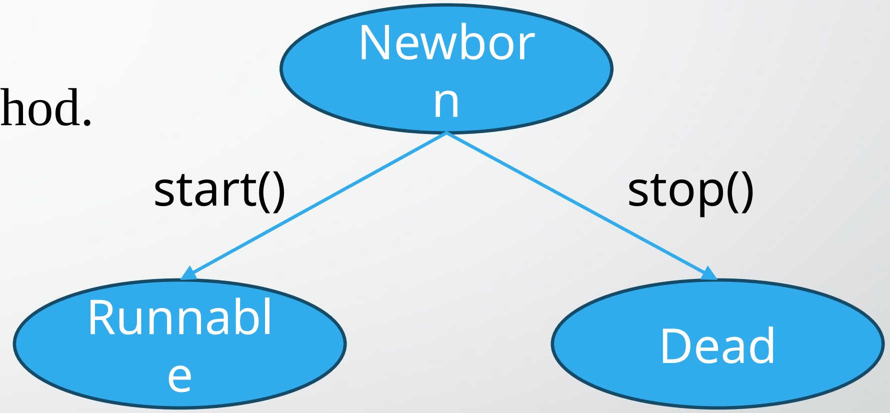
Method Signature	Description
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution
boolean isAlive()	This method will check if thread is alive or dead

Life Cycle of Threads



Newborn State

- A thread is just created/born and yet to be scheduled for running.
- At this point, we can do one of the following:
  - ❑ Schedule it for running using the **start()** method.
  - ❑ Kill it using the **stop()** method



Creation of threads:

- A thread can be created using one of the following ways:
  - ❑ by extending the Thread class
  - ❑ by implementing Runnable interface

Creating a Thread (Example)

```
class ThreadA extends Thread {
   public void run(){
      System.out.println("ThreadA is running...");
   }
}
class ThreadB implements Runnable {
   public void run(){
      System.out.println("ThreadB is running...");
   }
}
class MainClass {
   public static void main(String[] args) {
      ThreadA t1 = new ThreadA();
      t1.start();

      ThreadB obj = new ThreadB();
      Thread t2 = new Thread(obj);
      t2.start();
   }
}
```

Output:

```
ThreadA is running . . .
ThreadB is running . . .
```

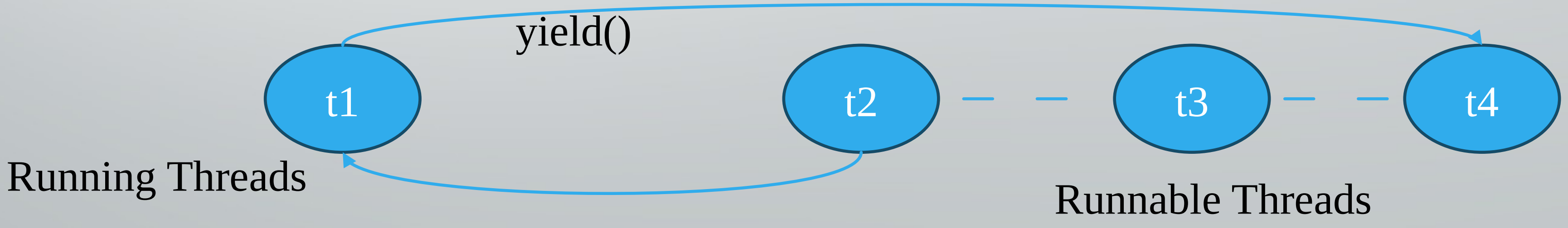
Runnable and Running States

Runnable State:

- A thread is ready for execution.
- The thread joins the ques with other existing threads in the program.
- The thread waits for the availability of the processor/resources.

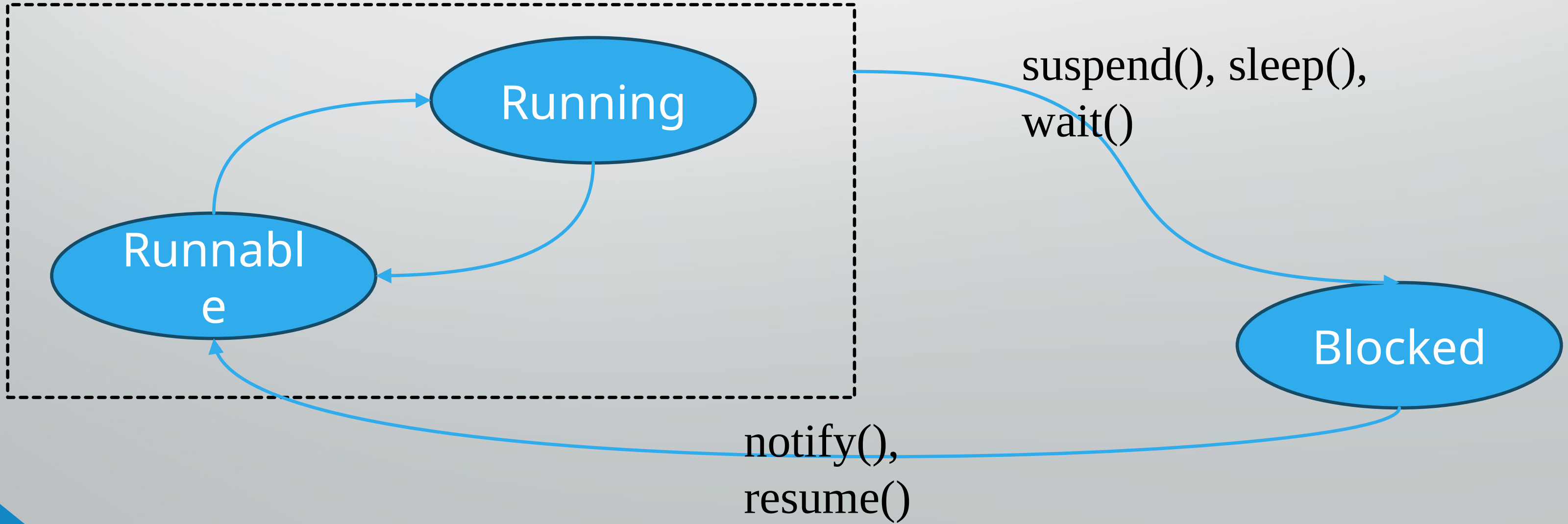
Running State:

- A thread is executing.
- The processor has given its time to the thread for its execution.
- The thread runs until its execution is over or the control is taken over by some other thread.



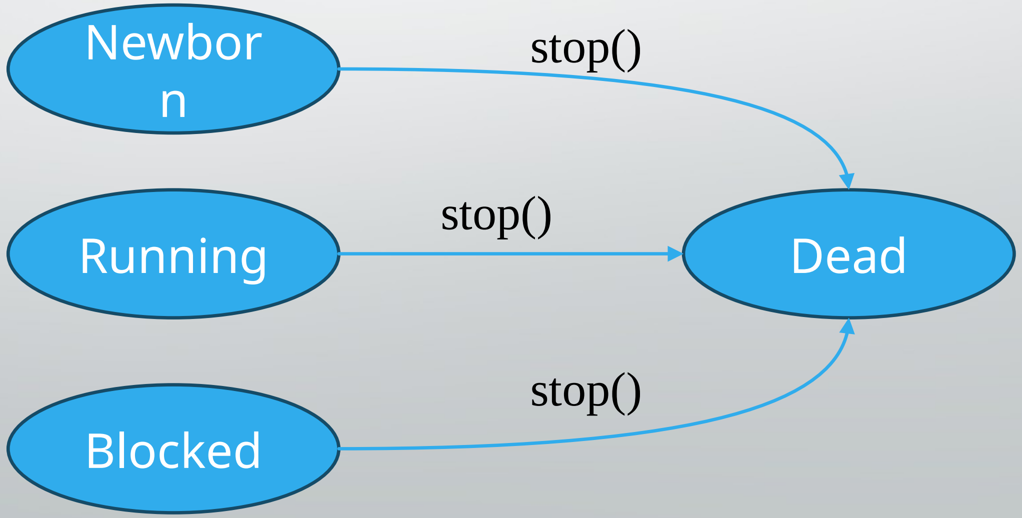
Blocked State

- A thread is said to be in **blocked state** if it is prevented from entering the runnable and the running state.
- This happens when a thread is suspended, sleeping, or waiting for some condition to be satisfied.
- A **blocked** thread is considered as **“not runnable”**, but **“not dead”** also. That means, the thread is fully qualified to be runnable again.



Dead State

- Every thread has a life cycle.
- A running thread ends its life when it has completed its execution of the **run()** method. This is called “natural death”.
- A thread can be killed by calling the **stop()** method when the thread is in its **“newborn”** state, or in **“running”** state, or even in the **“blocked**” state. This is referred to as “premature death”. (depreciated)



Types of Threads

- In **Java**, Threads are mainly of two types:
  - □ User-defined
  - □ Daemon Threads

Daemon Thread:

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread.
- The JVM will not exit if non-Daemon threads are executing.
- The JVM will exit only if Daemon threads are executing.
- Daemon threads die when the JVM exits.
- The user can also set a thread as Daemon by using **setDaemon(true)** before calling the **start()** method.

Daemon Thread (Example)

```
class BackgroundTask extends Thread {
   public void run(){
      while(true){       //Continuous task
         try{
            System.out.println("Background task running...");
            Thread.sleep(2000); //Sleep for 2 seconds
         }catch(InterruptedException e){
            e.printStackTrace();
         }
      }
   }
}
class MainClass {
   public static void main(String[] args) {
      BackgroundTask bt = new BackgroundTask();
      bt.setDaemon(true); //Setting the thread as a daemon

      bt.start();         //Starting the daemon thread
      System.out.println("Main thread finished.");
   }
}
```

Output:

```
Background task running...
Main thread finished.
```

Note: After the **main thread** ends, the JVM will exit, and the daemon thread will also.

Thread Priority

- In **Java**, each thread is assigned some priority, which affects the order in which it is scheduled for running.
- **Java** allows users to set/change the priority of the threads explicitly using the **setPriority(int Number)** method and retrieve the priority of a thread using the **getPriority()** method.
  - ❑ **MIN_PRIORITY = 1**
  - ❑ **NORM_PRIORITY = 5**
  - ❑ **MAX_PRIORITY = 10**
- **Java** assigns the default priority **(NORM_PRIORITY)** to each thread if not assigned the priority explicitly.
- The threads with the same priority level are executed on FCFS (First come first service) basis.

Thread Priority (Example)

```
class A extends Thread {
  public void run(){
    System.out.println("Thread A started...");
    for(int i=0; i<2; i++)
       System.out.println("From Thread A, i="+i);
    System.out.println("Exit from Thread A.");
  }
}
class B extends Thread {
  public void run(){
    System.out.println("Thread B started...");
    for(int j=0; j<2; j++)
       System.out.println("From Thread B, j="+j);
    System.out.println("Exit from Thread B.");
  }
}
class C extends Thread {
  public void run(){
    System.out.println("Thread C started...");
    for(int k=0; k<2; k++)
       System.out.println("From Thread C, k="+k);
    System.out.println("Exit from Thread C.");
  }
}
```

```
class MainClass {
  public static void main(String[] args) {
    A t1 = new A();
    B t2 = new B();
    C t3 = new C();

    t1.setPriority(Thread.MIN_PRIORITY);
    t2.setPriority(t1.getPriority()+1);
    t3.setPriority(Thread.MAX_PRIORITY);

    t1.start();
    t2.start();
    t3.start();

    System.out.println("Exit from main");
  }
}
```

What will be the order of execution of the Threads?

Thread Priority (Example)

- The output of the shown Code will be:

Output:

```
Thread A started...
Thread B started...
Thread C started...
From Thread C, k=0
From Thread C, k=1
Exit from Thread C.
From Thread B, j=0
From Thread B, j=1
Exit from Thread B.
From Thread A, i=0
From Thread A, i=1
Exit from Thread A.
Exit from main
```

Also, can be

Output:

```
Thread A started...
Thread B started...
Exit from main
Thread C started...
From Thread C, k=0
From Thread C, k=1
Exit from Thread C.
From Thread B, j=0
From Thread A, i=0
From Thread B, j=1
From Thread A, i=1
Exit from Thread B.
Exit from Thread A.
```

or

Output:

```
Thread A started...
Exit from main
Thread B started...
Thread C started...
From Thread B, j=0
From Thread B, j=1
From Thread C, k=0
From Thread C, k=1
Exit from Thread C.
From Thread A, i=0
From Thread A, i=1
Exit from Thread B.
Exit from Thread A.
```

,etc.

Summary

Today, we learned about

- Threads in a Program
- Multi-Threading concept in Java (Types of Threads, Properties of Threads)
- Thread class and its useful methods
- Life Cycle of a Thread
- Priority of Threads

Thank You!