دانشکده مهندسی و علوم کامپیوتر

برنامه نویسی پیشرفته

وحیدی اصل

مالتی تریدینگ - 1

Multithreading in Java

- **Multithreading** in java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area.
    - They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

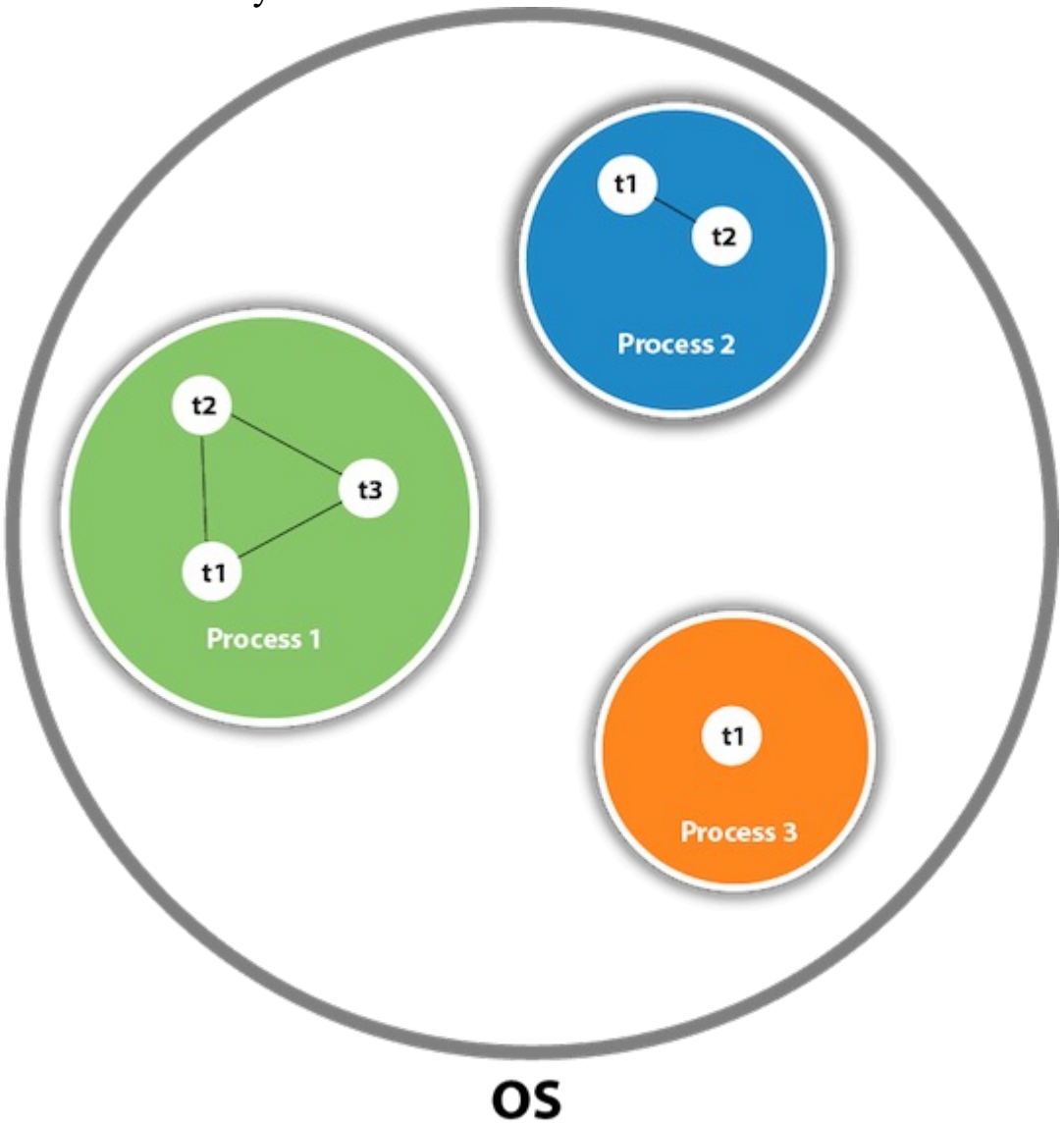
3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

- Multitasking is a process of executing multiple tasks simultaneously.
- We use multitasking to utilize the CPU.
- Multitasking can be achieved in two ways:
  - Process-based Multitasking (Multiprocessing)
    - Each process has an address in memory. In other words, each process allocates a separate memory area.
    - A process is heavyweight.
    - Cost of communication between the process is high.
    - Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.
  - Thread-based Multitasking (Multithreading)
    - Threads share the same address space.
    - A thread is lightweight.
    - Cost of communication between the thread is low.

What is Thread in java

- A thread is a lightweight subprocess, the smallest unit of processing.
- It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



- As shown, a thread is executed inside the process.
- There is context-switching between the threads.
- There can be multiple processes inside the OS, and one process can have multiple threads.

Java Thread class

- Java provides **Thread class** to achieve thread programming.
- Thread class provides constructors and methods to create and perform operations on a thread.
- Thread class extends object class and implements Runnable interface.

Multithreading in Java

S.N.	Modifier and Type	Method	Description
1)	void	start()	It is used to start the execution of the thread.
2)	void	run()	It is used to do an action for a thread.
3)	static void	sleep()	It sleeps a thread for the specified amount of time.
4)	static Thread	currentThread()	It returns a reference to the currently executing thread object.
5)	void	join()	It waits for a thread to die.
6)	int	getPriority()	It returns the priority of the thread.
7)	void	setPriority()	It changes the priority of the thread.
8)	String	getName()	It returns the name of the thread.
9)	void	setName()	It changes the name of the thread.
10)	long	getId()	It returns the id of the thread.
11)	boolean	isAlive()	It tests if the thread is alive.
12)	static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.

Multithreading in Java

S.N.	Modifier and Type	Method	Description
13)	void	suspend()	It is used to suspend the thread.
14)	void	resume()	It is used to resume the suspended thread.
15)	void	stop()	It is used to stop the thread.
16)	void	destroy()	It is used to destroy the thread group and all of its subgroups.
17)	boolean	isDaemon()	It tests if the thread is a daemon thread.
18)	void	setDaemon()	It marks the thread as daemon or user thread.
19)	void	interrupt()	It interrupts the thread.
20)	boolean	isinterrupted()	It tests whether the thread has been interrupted.
21)	static boolean	interrupted()	It tests whether the current thread has been interrupted.
22)	static int	activeCount()	It returns the number of active threads in the current thread's thread group.
23)	void	checkAccess()	It determines if the currently running thread has permission to modify the thread.
24)	static boolean	holdLock()	It returns true if and only if the current thread holds the monitor lock on the specified object.

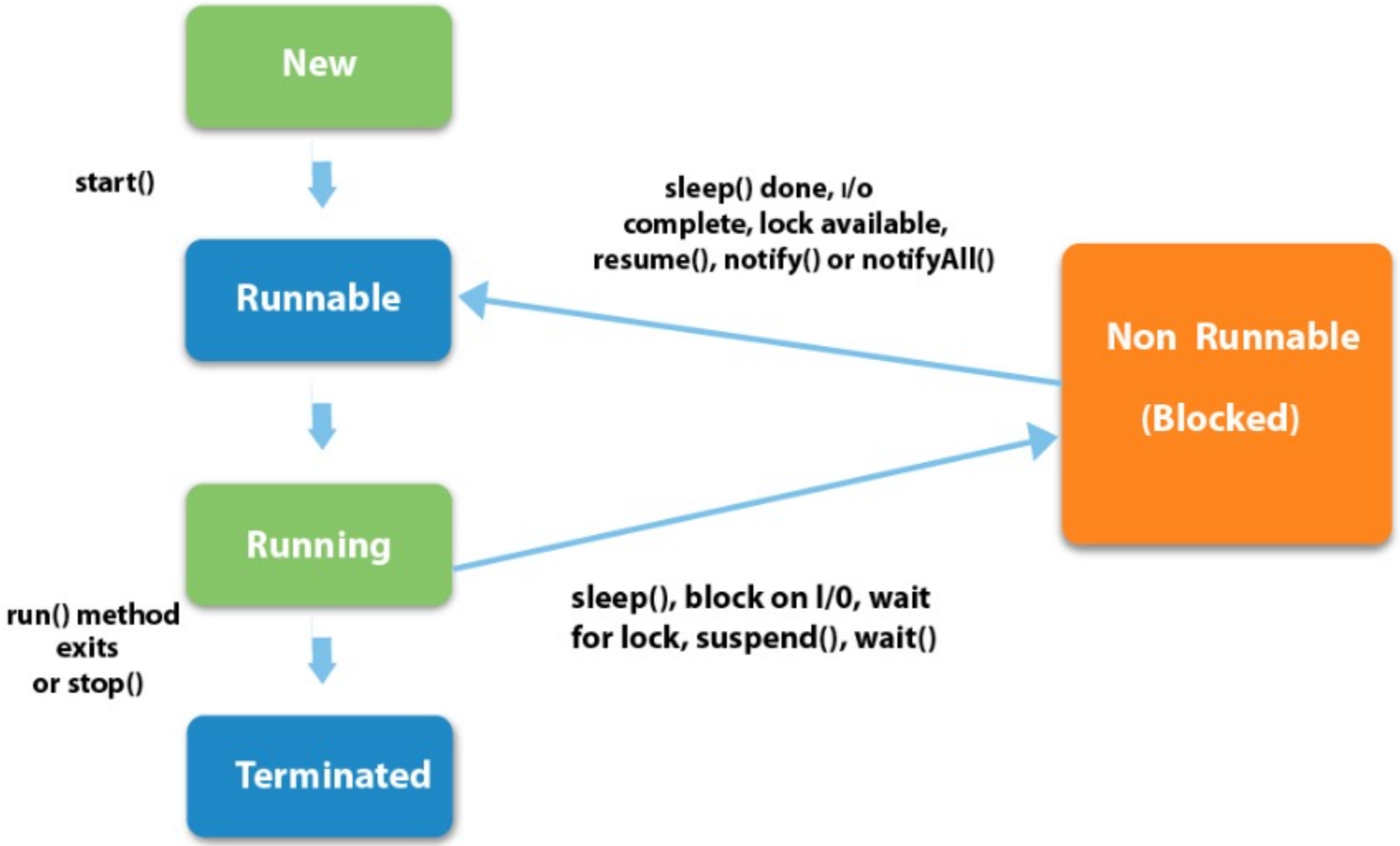
Multithreading in Java

S.N.	Modifier and Type	Method	Description
25)	static void	dumpStack()	It is used to print a stack trace of the current thread to the standard error stream.
26)	StackTraceElement[]	getStackTrace()	It returns an array of stack trace elements representing the stack dump of the thread.
27)	static int	enumerate()	It is used to copy every active thread's thread group and its subgroup into the specified array.
28)	Thread.State	getState()	It is used to return the state of the thread.
29)	ThreadGroup	getThreadGroup()	It is used to return the thread group to which this thread belongs
30)	String	toString()	It is used to return a string representation of this thread, including the thread's name, priority, and thread group.
31)	void	notify()	It is used to give the notification for only one thread which is waiting for a particular object.
32)	void	notifyAll()	It is used to give the notification to all waiting threads of a particular object.
33)	void	setContextClassLoader()	It sets the context ClassLoader for the Thread.
34)	ClassLoader	getContextClassLoader()	It returns the context ClassLoader for the thread.
35)	static Thread.UncaughtExceptionHandler	getDefaultUncaughtExceptionHandler()	It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception.
36)	static void	setDefaultUncaughtExceptionHandler()	It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception

Life cycle of a Thread (Thread States)

- A thread can be in one of the five states.
- According to sun, there are only 4 states in **thread life cycle in java:** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- The life cycle of the thread in java is controlled by JVM.
- The java thread states are as follows:
    1. New
    2. Runnable
    3. Running
    4. Non-Runnable (Blocked)
    5. Terminated

Life cycle of a Thread (Thread States)



Life cycle of a Thread (Thread States)

- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:
  1. New
     - The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
  2. Runnable
     - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
  3. Running
     - The thread is in running state if the thread scheduler has selected it.
  4. Non-Runnable (Blocked)
     - This is the state when the thread is still alive, but is currently not eligible to run.
  5. Terminated
     - A thread is in terminated or dead state when its run() method exits.

How to create thread

- There are two ways to create a thread:
  - By extending Thread class
  - By implementing Runnable interface.

Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface

Commonly used Constructors of Thread class:

•Thread()
•Thread(String name)
•Thread(Runnable r)
•Thread(Runnable r, String name)

Java Thread Example by extending Thread class

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

```
Output:thread is running...
```

How to create thread

- **Runnable interface:**
  - 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.

- **Starting 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.

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

```
Output:thread is running...
```

- If you are not extending the Thread class, your class object would not be treated as a thread object.
- So you need to explicitly create Thread class object.
- We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Scheduler in Java

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.

Sleep method in java

- The sleep() method of Thread class is used to sleep a thread for the specified amount of time.
- Syntax of sleep() method in java
  - The Thread class provides two methods for sleeping a thread:
    - public static void sleep (long miliseconds) throws InterruptedException
    - public static void sleep (long miliseconds, int nanos) throws InterruptedException

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

```
1
1
2
2
3
3
4
4
```

Can we start a thread twice?

- No.
- After starting a thread, it can never be started again.
- If you does so, an *IllegalThreadStateException* is thrown.
- In such case, thread will run once but for second time, it will throw exception.

Can we start a thread twice?

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

Test it Now

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

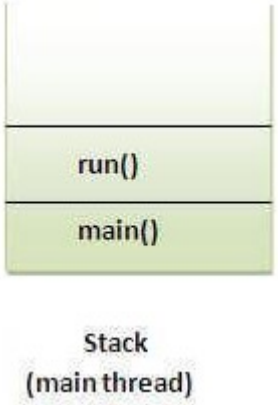
What if we call run() method directly instead of start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
class TestCallRun1 extends Thread{
 public void run(){
  System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

Test it Now

```
Output:running...
```



Problem if you direct call run() method

- As you can see in the program that there is no context-switching, because here t1 and t2 will be treated as normal object not thread object.

```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```

```
Output:1
       2
       3
       4
       5
       1
       2
       3
       4
       5
```

The join() method

- The join() method waits for a thread to die.
- In other words, it causes the currently running threads to stop executing until the thread it joins with, completes its task.
- Syntax:
  - public void join()throws InterruptedException
  - public void join(long milliseconds)throws InterruptedException

The join() method

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

```
Output:1
       2
       3
       4
       5
       1
       1
       2
       2
       3
       3
       4
       4
       5
       5
```

- As you can see in the example, when t1 completes its task then t2 and t3 starts executing.

Naming Thread

- The Thread class provides methods to change and get the name of a thread.
- By default, each thread has a name i.e. thread-0, thread-1 and so on.
- But we can change the name of the thread by using setName() method.
- The syntax of setName() and getName() methods are given below:

    - **public String getName():** is used to return the name of a thread.
    - **public void setName(String name):** is used to change the name of a thread.

getName(),setName(String) and getId() method:

```
class TestJoinMethod3 extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  TestJoinMethod3 t1=new TestJoinMethod3();
  TestJoinMethod3 t2=new TestJoinMethod3();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  System.out.println("id of t1:"+t1.getId());

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

  t1.setName("Sonoo Jaiswal");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

```
Output:Name of t1:Thread-0
       Name of t2:Thread-1
       id of t1:8
       running...
       After changling name of t1:Sonoo Jaiswal
       running...
```

- The **thread ID** is a positive long number generated when the **thread** is created.
- Note : **thread ID** remains unique and unchanged during the lifetime of the **thread**.
- When a **thread** is terminated then its **thread ID** may be reused.

currentThread() method:

- The currentThread() method returns a reference to the currently executing thread object.

```
class TestJoinMethod4 extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 }
 public static void main(String args[]){
  TestJoinMethod4 t1=new TestJoinMethod4();
  TestJoinMethod4 t2=new TestJoinMethod4();

  t1.start();
  t2.start();
 }
}
```

Test it Now

```
Output:Thread-0
       Thread-1
```

Priority of a Thread (Thread Priority)

- Each thread have a priority.
- Priorities are represented by a number between 1 and 10.
- In most cases, thread scheduler schedules the threads according to their priority (known as scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- 3 constants defined in Thread class:
  - public static int MIN_PRIORITY
  - public static int NORM_PRIORITY
  - public static int MAX_PRIORITY
- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

 }
 public static void main(String args[]){
 TestMultiPriority1 m1=new TestMultiPriority1();
 TestMultiPriority1 m2=new TestMultiPriority1();
 m1.setPriority(Thread.MIN_PRIORITY);
 m2.setPriority(Thread.MAX_PRIORITY);
 m1.start();
 m2.start();

 }
}
```

```
Output:running thread name is:Thread-0
       running thread priority is:10
       running thread name is:Thread-1
       running thread priority is:1
```