THREAD STATES & TRANSITIONS

Instructor:

Table of contents

Learning Approach

Strongly suggested for a better learning and understanding of this course:

- Noting down the ***key concepts*** in the class
- ***Analyze*** all the examples / code snippets provided
- Study and understand the ***self study topics***
- ***Completion*** and ***submission*** of all the assignments, on time
- Completion of the ***self review*** questions in the lab guide
- ***Study*** and understand all the artifacts
- ***Completion*** of the project on time inclusive of individual and group activities

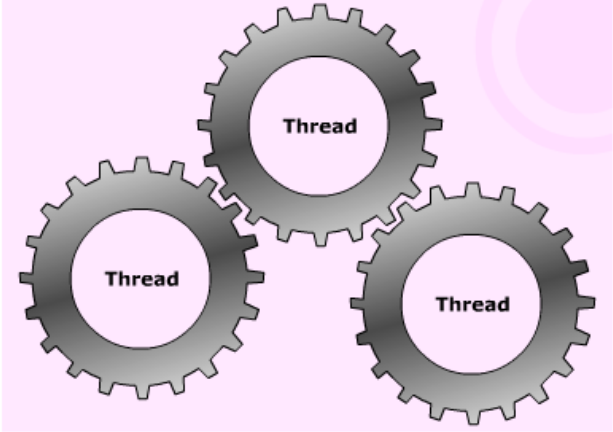
Section 1

INTRODUCTION TO THREAD

Introduction to Threads

- A **process** is a program that is executing.
- Each process has its own run-time resources, such as their own data, variables and memory space.
- Each process executes **several tasks** at a time and each task is carried out by separate **thread**.

- A **thread** is a **path of code execution** through a program, and each thread has its own *local variables*, *program counter*, and *lifetime*.



Process and Thread

Characteristics of Threads

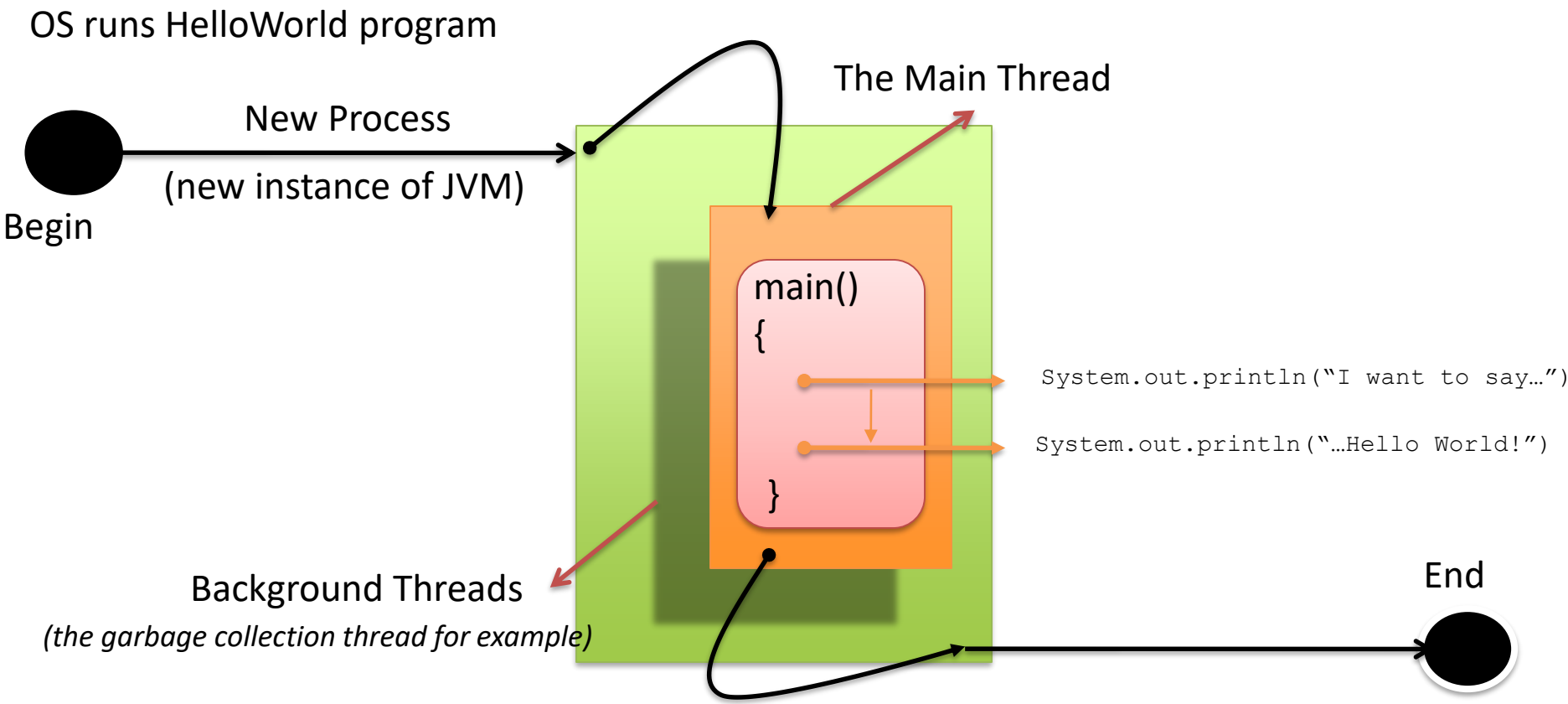
- **A thread** has its own complete set of **basic run-time resources** to run it independently.
- **A thread is the smallest unit of executable code** in an application that performs a particular job or task.
- **Several threads can be executed at a time**, facilitating execution of several tasks of a single application simultaneously.
- **Threads are independent**, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Let’s try HelloWorld again!

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("I want to say...");
        System.out.println("...Hello World!");
    }
}
```

```
C:\java HelloWorld ↵
I want to say...
...Hello World!
```

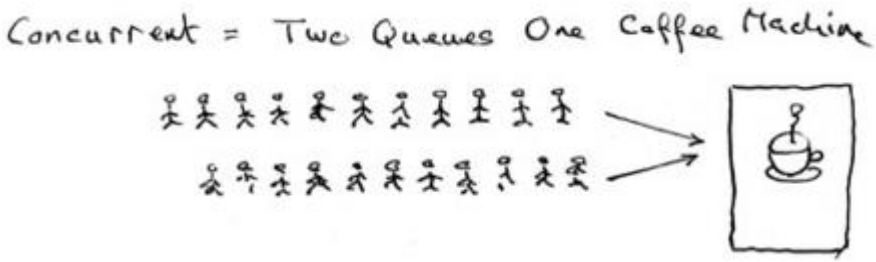
What's happening?



Even a simple HelloWorld program is running in a multithreaded environment

The concept of concurrent programm

- Concurrent programming is a process of **running several tasks at a time**.



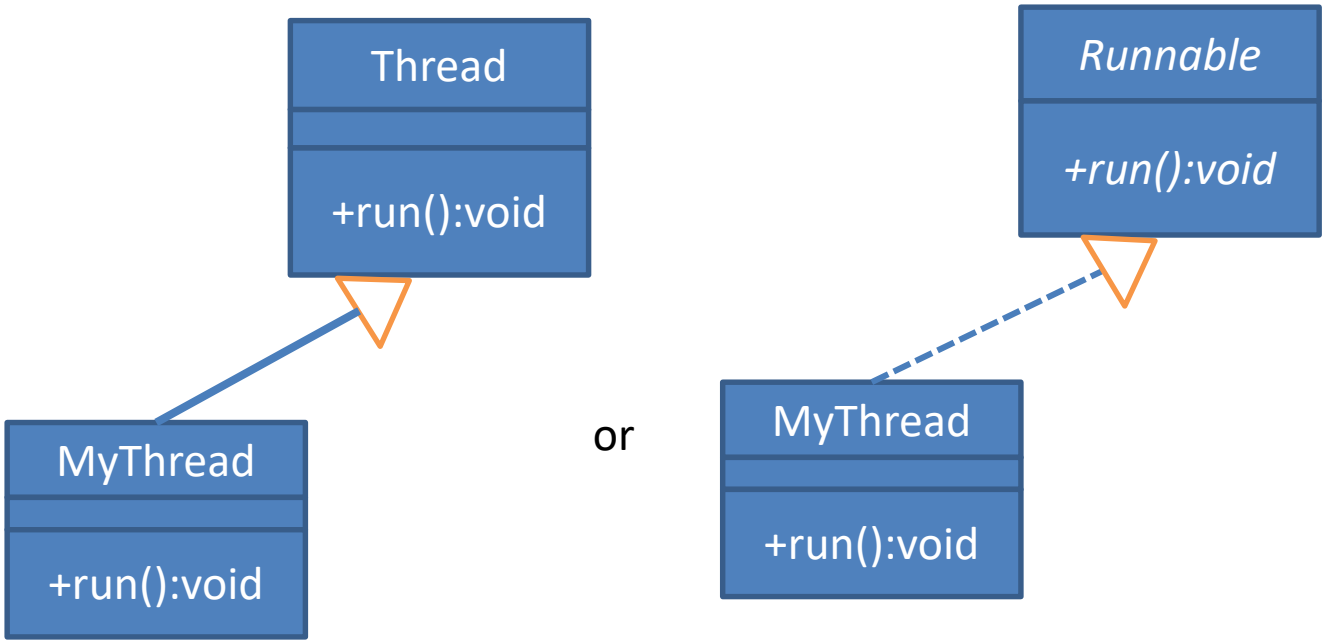
- **In Java**, it is possible to execute simultaneously[đồng thời] a invoked function without waiting for the invoked function to terminate.
- The invoked function runs independently and concurrently with the invoking program, and can share variables, data and so on with it.

Section 2

CREATING THREADS

How to create a Thread?

- Which way you can create a thread?
  - ✓ Inherits the **Thread** class
  - ✓ Implements the **Runnable** interface



Subclassing the Thread class

- **Step 1:** Subclassing Thread class
- **Step 2**: Override run() method

```
// Extending Thread class
class MyThread extends Thread {
    public void run()    {// Overriding the Run()
        // implementation the logic
    }
}
```

- **Step 3**: Create a thread object and start the thread

```
public static void main(String args[]) {
    //Creating thread object
    MyThread t = new MyThread();
    t.start(); //Starting a thread
}
```

Implementing *Runnable* interfac

- **Step 1**: Implement the ***Runnable*** interface
- **Step 2**: Implement *run()* method

```
// Declaring a class that implements Runnable interface
class MyRunnable implements Runnable {
    public void run()    {// Overriding the Run()
        // implementation the logic
    }
}
```

- **S**

```
public static void main(String args[]) {
    Runnable r = new MyRunnable();
    Thread thObj=new Thread(r);
    thObj.start(); //Starting a thread
}
```

Create a Thread: Demo

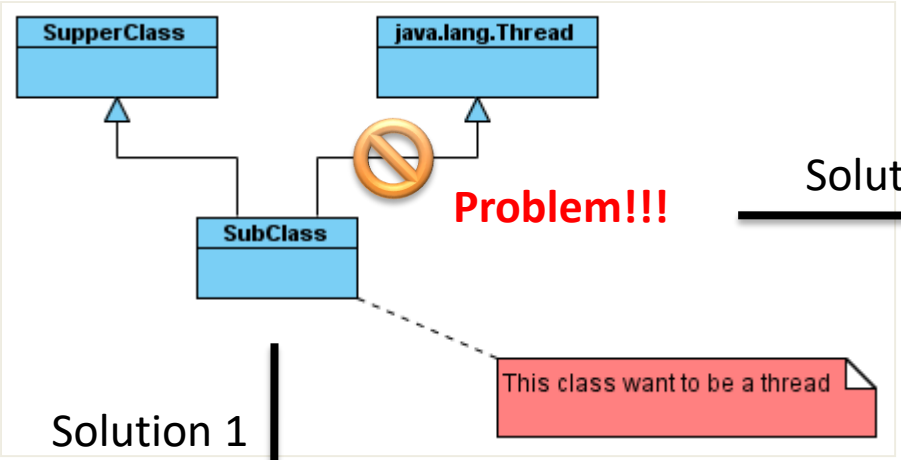
Code snippet

```
public class TwoThreadsTest {
    public static void main(String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
}
class SimpleThread extends Thread {
    public SimpleThread(String str) {
     super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
            }
        }
        System.out.println("DONE! " + getName());
    }
}
```

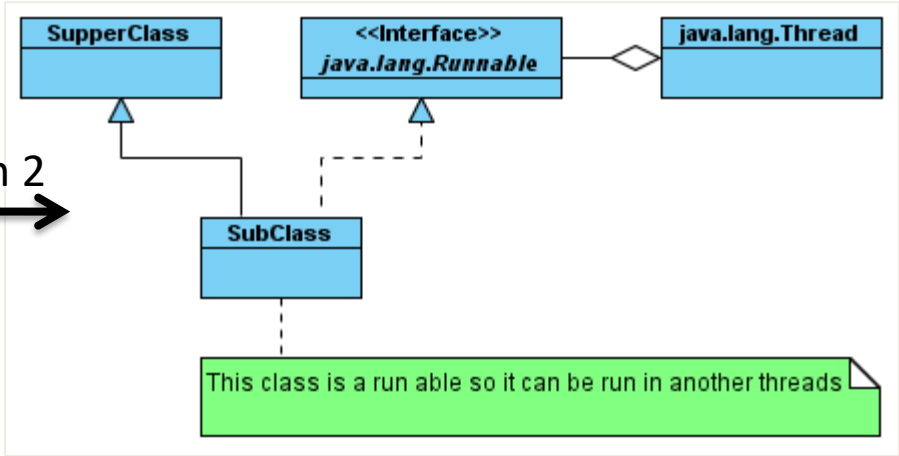
Mar... Pro... Ser... Dat... Sni... Co...

```
<terminated> TwoThreadsTest [Java Application] C:\Program Files\J
0 Fiji
0 Jamaica
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
5 Jamaica
4 Fiji
5 Fiji
6 Jamaica
6 Fiji
7 Fiji
7 Jamaica
8 Fiji
8 Jamaica
9 Fiji
9 Jamaica
DONE! Fiji
DONE! Jamaica
```

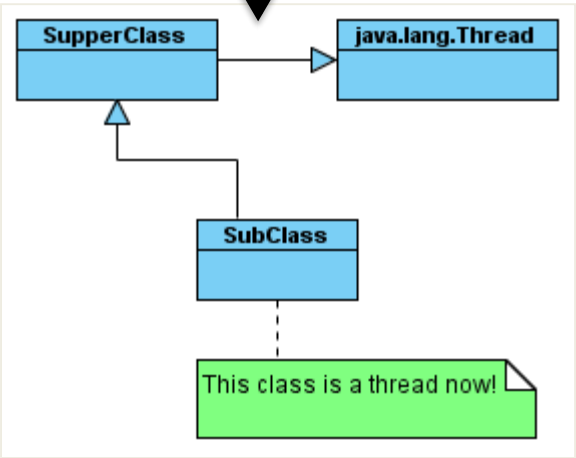
Implementing Runnable



Solution 2



Solution 1

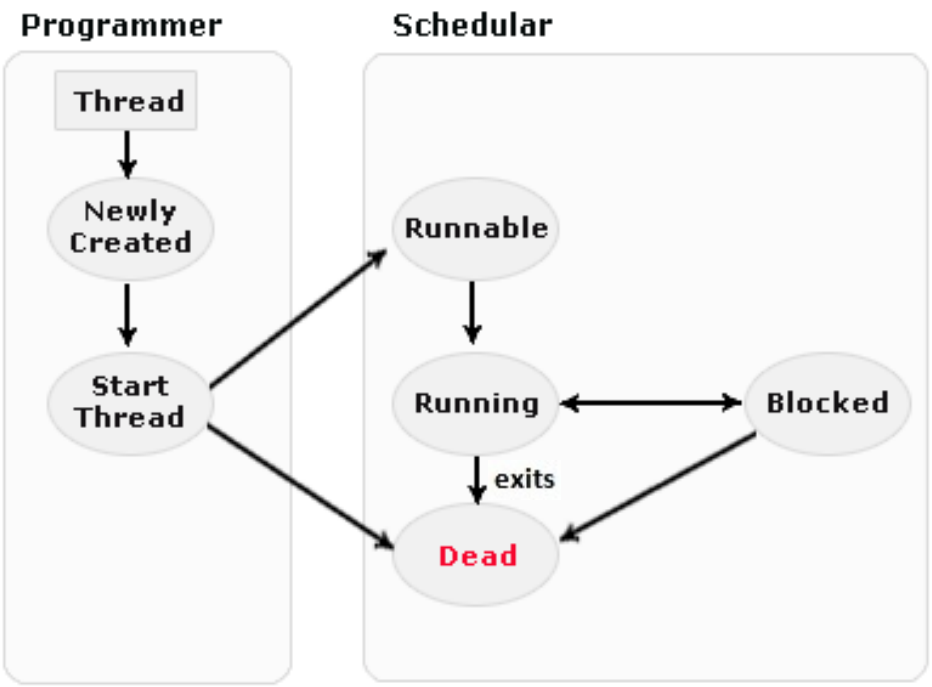


Section 3

THREAD STATES AND TRANSITIONS

Thread States

- Difference state of a thread are:



1. New state
2. Runnable (Ready-to-run) state
3. Running state
4. Blocked
5. Dead state

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:
  - ✓ Thread()
  - ✓ Thread(String name)
  - ✓ Thread(Runnable r)
  - ✓ Thread(Runnable r,String name)

Methods of the Thread class

Method	Return Type	Description
static currentThread( )	Thread	*Returns an object reference to the thread in which it is invoked.*
getName( )	String	*Retrieve the name of the thread object or instance.*
start( )	void	*starts the execution of the thread. JVM calls the run() method on the thread.*
run( )	void	*This method is the entry point to execute thread, like the main method for applications.*
static sleep( )	void	*Suspends a thread for a specified amount of time (in milliseconds).*
isAlive( )	boolean	*This method is used to determine the thread is running or not.*
static activeCount( )	int	*This method returns the number of active threads in a particular thread group and all its subgroups.*
interrupt( )	void	*The method interrupt the threads on which it is invoked.*
static yield( )	void	*By invoking this method the current thread pause its execution temporarily and allow other threads to execute.*
join( )	void	*This method and join(long millisec) Throws InterruptedException. These two methods are invoked on a thread instance, the currently running thread will block until the Thread instance has finished executing.*

Starting a Thread (1/4)

- **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.

Code snippet

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running...");
    }

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
    }
}
```

Starting a Thread (2/4)

- Be sure that, you **start a Thread**, **not a Runnable**. You call start() on a Thread instance, not on a Runnable instance.

Code snippet

```
class FooRunnable implements Runnable {
    public void run() {
        for (int x = 1; x < 6; x++) {
         System.out.println("Runnable running");
        }
    }
}
```

```
public class TestThreads {
    public static void main(String[] args) {
        FooRunnable r = new FooRunnable();
        Thread t = new Thread(r);
        t.start();
    }
}
```

```
Runnable running
Runnable running
Runnable running
Runnable running
Runnable running
```

Starting a Thread (3/4)

Watch

- If you see code that calls the run() method on a Runnable (or even on a Thread instance), **that's perfectly legal**.
  - ✓ But it **doesn't** mean the run() method will run in a **separate thread**!
  - ✓ Calling a run() method directly just means you're invoking a method from whatever **thread is currently executing**, and the **run() method goes onto the current call stack rather than at the beginning of a new call stack**.
- The following code does not start a new thread of execution:

```
Thread t = new Thread();
t.run(); // Legal, but does not start a new thread
```

Starting a Thread (4/4)

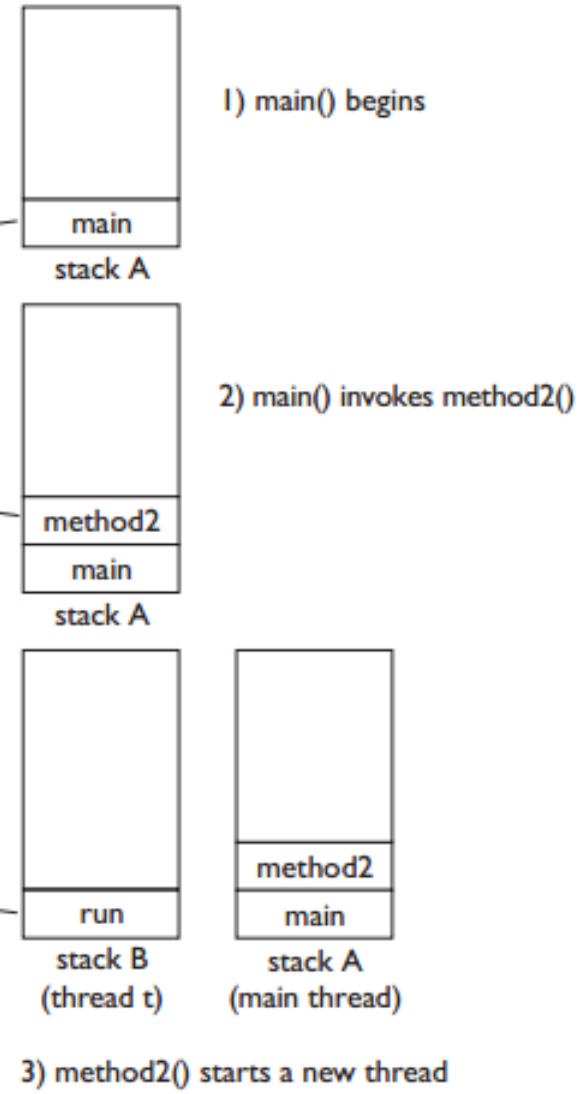
Watch

FIGURE 9-1

Starting a thread

```
public static void main(String [] args) {
  //  running
  //  some code
  //  in main()
  method2();
  //  running
  //  more code
}

static void method2() {
  Runnable r = new MyRunnable();
  Thread t = new Thread(r);
  t.start();
  //  do more stuff
}
```



sleep() Method

- **Thread.sleep()** suspends[ngừng/hoãn] the execution of the current thread for a specified period of time.
  - ✓ It makes the **processor time available** to the **other threads** of an application or **other applications**.
  - ✓ It stops the execution if the active thread for the time specified in milliseconds or nanoseconds.
- It raises **InterruptedException** when it is interrupted using the interrupt () method.

Code snippet

```
public void run()
{
    for(int x=1; x<4; x++) {
        System.out.println("nameRunnable is running  by" +
        /** getName() method give the name of the
        * currently running thread.*/
        Thread.currentThread().getName());
        try{
            Thread.sleep(1000);
        /** sleep() method will force the running thread to
         * wait for 1000 milli seconds.*/
        }catch (InterruptedException ex) { }
    }
```

An example of using sleep() method

join() Method (1/2)

- The join() method is used to hold the execution of currently running thread until the specified thread is dead (finished execution).

Code snippet

```
public class Leader implements Runnable {
    public static void main(String[] args) {
        Thread t = new Thread(new Leader());
        t.start();
        System.out.print("m1 ");
        try {
         t.join();
        } catch (InterruptedException e) {
         e.printStackTrace();
        }
        System.out.print("m2 ");
    }
    public void run() {
        System.out.print("r1 ");
        System.out.print("r2 ");
    }
}
```

```
Mar... Pro... Ser... Dat... Sni... Co...
<terminated> Leader [Java Application] C:\Program Files\Java\jre7\bi
m1 r1 r2 m2
```

`join()` Method (2/2)

Analysis

- If you have a **thread B** that can't do its work until another **thread A** has completed its work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished (and entered the dead state).

```
Thread t = new Thread();

t.start();

t.join();
```

"the code `t.join()` means "Join me (the current thread) to the end of `t`, so that `t` must finish before **I** (the current thread) can run again"

Practical time

- Exer: Viết 1 thread để gán giá trị ngẫu nhiên cho một biến, running thread và lấy giá trị của biến.

`yield()` Method (1/2)

- By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
  - ✓ Yield is a **Static** method and **Native** too.
  - ✓ Yield tells the **currently executing thread** to give a chance to the threads that have **equal priority in the Thread Pool**.
  - ✓ There is **no guarantee** that Yield will make the currently executing thread to runnable state **immediately**.
  - ✓ It can only make a thread **from Running State to Runnable State**, not in wait or blocked state.

yield() Method (2/2)

Example

Code snippet

```
public class YieldExample {
    public static void main(String[] args) {
        Thread producer = new Producer();
        Thread consumer = new Consumer();

        producer.setPriority(Thread.MIN_PRIORITY); // Min Priority
        consumer.setPriority(Thread.MAX_PRIORITY); // Max Priority

        producer.start();
        consumer.start();
    }
}
```

```
class Producer extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
        System.out.println("I am Producer :
              Produced Item " + i);
        Thread.yield();
        }
    }
}
```

```
class Consumer extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
        System.out.println("I am Consumer
        :           Consumed Item " + i);
        Thread.yield();
        }
    }
}
```

isAlive() Method

- The **main thread** must be the last thread to finish.
- A thread is considered to be alive when it is running. We can check it by use isAlive() method.
- This method helps one thread to check state of another **to do something** or **use join() method**.
- **Syntax**:

```
System.out.println("The thread is alive: "
                    +thread.isAlive());
```

isAlive() Method (2/2)

Code snippet

```
public class IsAliveTest {
    public static void main(String args[]) {
        SimpleThread thread1 = new SimpleThread("One");
        SimpleThread thread2 = new SimpleThread("Two");
        System.out.println("Thread One is alive: "+ thread1.isAlive());
        System.out.println("Thread Two is alive: "+ thread2.isAlive());
        // wait for threads to finish
        try {
            System.out.println("Waiting for threads to finish.");
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
         System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: "+ thread1.isAlive());
        System.out.println("Thread Two is alive: "+ thread2.isAlive());
        System.out.println("Main thread exiting.");
    }
}
```

Section 4

MANAGING THREAD

Thread priority

- **Thread priority** helps the thread scheduler to decide which thread to run.
- Priority also helps the OS to decide the amount of resource that has to be allocated to each thread.
- Thread priority:
  - ✓ Thread.MAX_PRIORITY : constant value of 10
  - ✓ Thread.NORM_PRIORITY : constant value of 5, default
  - ✓ Thread.MIN_PRIORITY : constant value of 1
- Important Methods:
  - ✓ setPriority()
  - ✓ getPriority()

Java Thread Priority

Example 1

Code snippet

```
public class ThreadDefaultPriority extends Thread {
   public void run() {
        System.out.println(getName()+":"+
         Thread.currentThread().getPriority());
   }

   public static void main(String[] args)
                                  throws InterruptedException {
        ThreadDefaultPriority t1 = new ThreadDefaultPriority();
        ThreadDefaultPriority t2 = new ThreadDefaultPriority();

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

```
Thread-1:5
Thread-0:5
```

Java Thread Priority

Example 2

Code snippet

```
public class ThreadPriority extends Thread {
    public void run() {
        String tName = Thread.currentThread().getName();
        Integer tPrio = Thread.currentThread().getPriority();
        System.out.println(tName + " has priority " + tPrio);
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadPriority t0 = new ThreadPriority();
        ThreadPriority t1 = new ThreadPriority();
        ThreadPriority t2 = new ThreadPriority();

        t1.setPriority(Thread.MAX_PRIORITY);
        t0.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.NORM_PRIORITY);

        t0.start();
        t1.start();
        t2.start();
    }
}
```

```
Thread-1 has priority 10
Thread-0 has priority 1
Thread-2 has priority 5
```

Thread priority

watch

- If a thread enters the **runnable** state, and it has a higher priority than any of the threads in the pool and a higher priority than the currently running thread:
  - ✓ the **lower-priority running thread** usually will be bumped **back to runnable** and the **highest-priority thread will be chosen to run.**
- At any given time the currently running thread usually will not have a priority that is lower than any of the threads in the pool.
  - ✓ In most cases, the running thread will be of equal or greater priority than the highest priority threads in the pool.
- Don't rely on thread priorities when designing your multithreaded application.
  - ✓ Use thread priorities as a way to improve the efficiency of your program.

Daemon Threads

- In Java **Daemon threads** are service providers for normal running threads inside the same process. Java Daemon threads does their execution as helper to complete the current task.
- The characteristics of the daemon threads are:
  - ✓ They work in the background providing service to other threads.
  - ✓ They are fully dependent on the user threads.
  - ✓ JVM stops once a thread dies and only daemon thread is alive.
- To specify a thread is deamon or not, **setDaemon()** method is used

Daemon Threads

Example

Code snippet

```
public class Main {
	public static void main(String[] args) {
		BackgroundService service = new BackgroundService();
		try {
			Thread.sleep(10000);
		} catch (InterruptedException ex) {
			System.out.println(ex);
		}
	}
}
```

```
public class BackgroundService extends Thread {
	private int count;
	public BackgroundService() {
		setDaemon(true);
		start();
	}

	@Override
	public void run() {
		while (true) {
			System.out.println("#" + count++);
			// Causes the currently executing thread object to temporarily pause
			// and allow other threads to execute.
			yield();
		}
	}
}
```

Needs for Daemon Thread

- The tasks performed by the Daemon threads are:
  - ✓ *Daemon threads are service providers for other threads running in the same process,*
  - ✓ *Daemon threads are designed as low-level background threads that perform some tasks such as mouse events for Java program.*

Practical time

- **Exer2**: Tạo một Daemon Thread để thêm vào List một danh sách số trong khi User Thread đang thực hiện một tác vụ khác.

Lesson summary

- ◊ **Introduction to Thread**
- ◊ **Creating Threads**
  - ✓ Thread class
  - ✓ Runnable interface
- ◊ **Thread States and Transitions**
- ◊ **Managing Thread**

Thank you