Chapter 6

Concurrent Programming (1)

References :

[1] รังสิพรรณ มฤคทัต, กระบวนทัศน์ในการเขียนโปรแกรม (บทที่ 6)

[2] Tucker & Noonan, Programming Languages: Principles and Paradigms (Chapter 17)

[3] Sebesta, Concepts of Programming Languages (Chapter 13)

[4] Oracle, Java Documentation

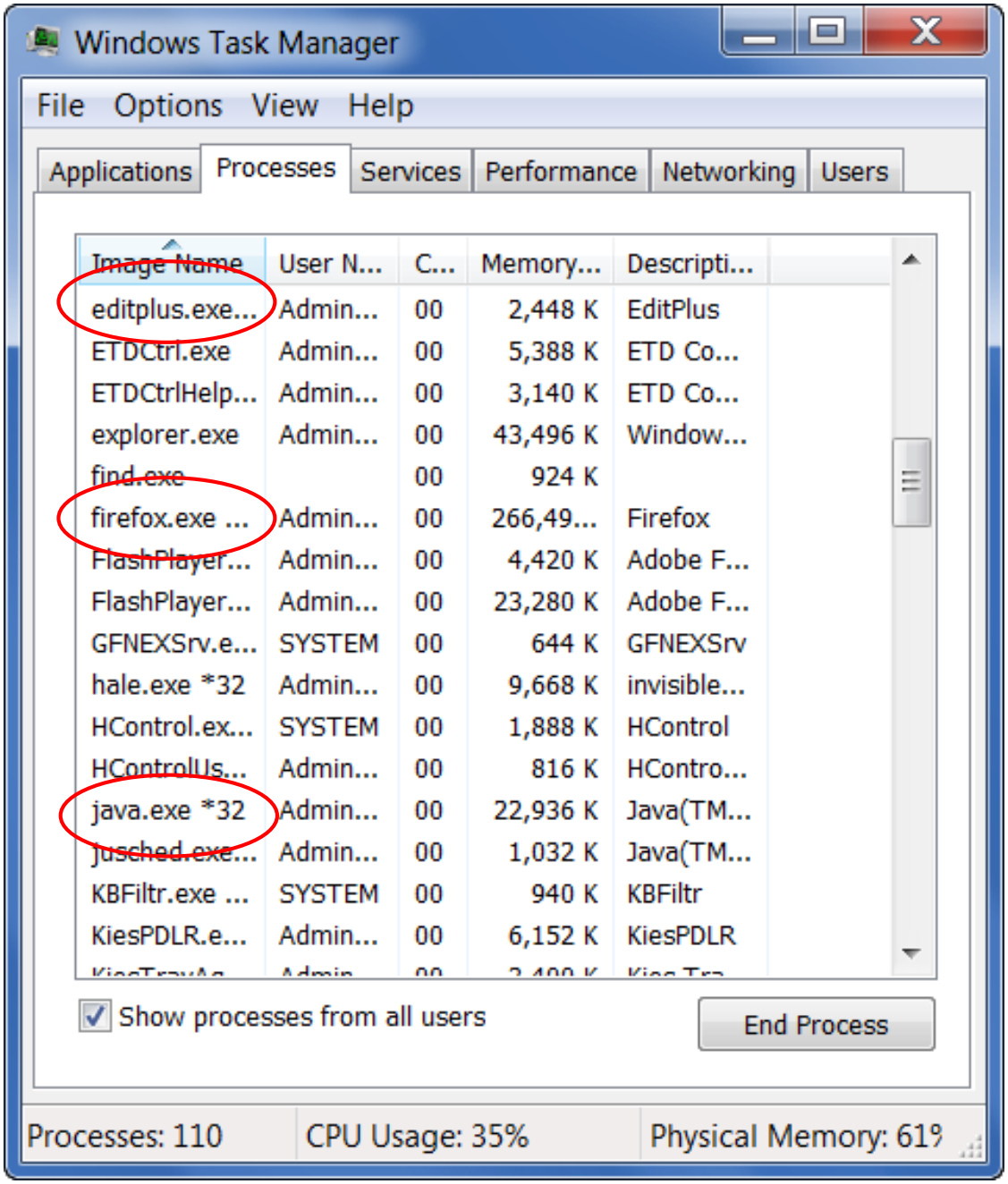
Chapter Objectives

At the end of this chapter, you should be able to:

- Explain life cycle of thread
- Compare thread creation by using Thread and Runnable
- Hand trace Java programs with threads
- Write Java programs with multiple threads running concurrently

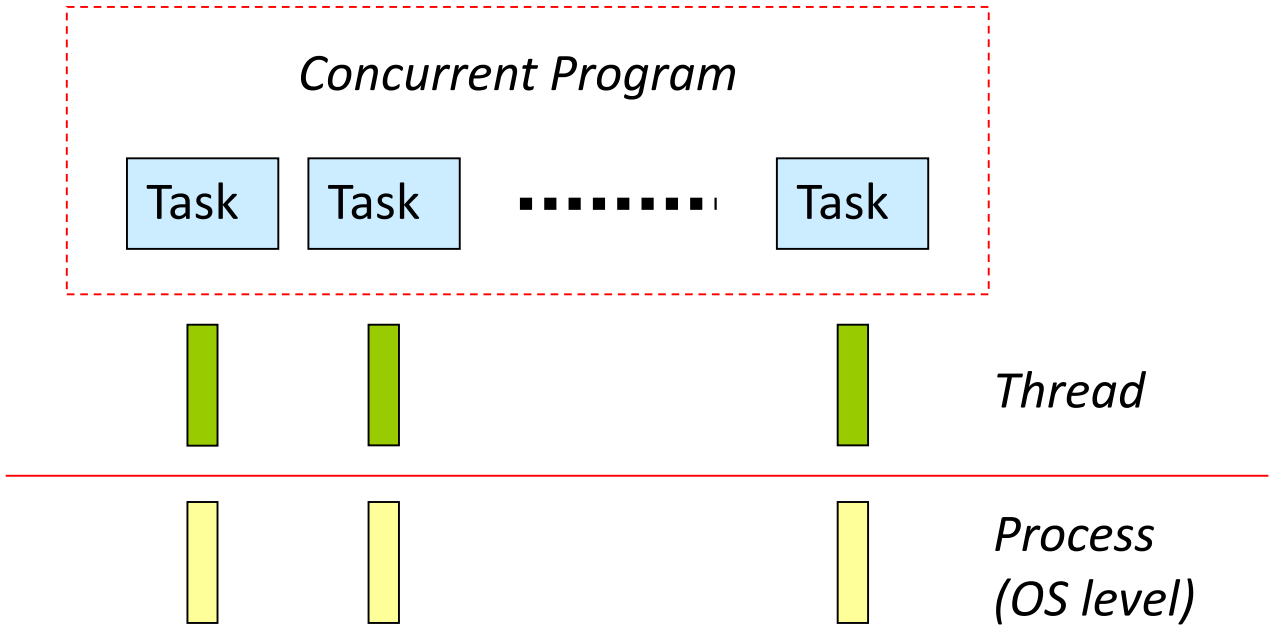
Sequential *vs.* Concurrent Programs

- Sequential program
  - Execute only 1 sequence of statements ➔ **single-thread**
  - Thread = a flow of control in program execution
  - Process = a unit of execution (typically in OS-level)
    - Each process has its own state or execution context and attached resources such as memory, files, network ports
    - Thread and process may be used interchangeably
- Concurrent program
  - Process-oriented paradigm
  - Execute multiple sequences of statements simultaneously
    ➔ **multithread**


Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Image Name User N... C... Memory... Descripti...
editplus.exe... Admin... 00 2,448 K EditPlus
ETDCtrl.exe Admin... 00 5,388 K ETD Co...
ETDCtrlHelp... Admin... 00 3,140 K ETD Co...
explorer.exe Admin... 00 43,496 K Window...
find.exe 00 924 K
firefox.exe ... Admin... 00 266,49... Firefox
FlashPlayer... Admin... 00 4,420 K Adobe F...
FlashPlayer... Admin... 00 23,280 K Adobe F...
GFNEXSrv.e... SYSTEM 00 644 K GFNEXSrv
hale.exe *32 Admin... 00 9,668 K invisible...
HControl.ex... SYSTEM 00 1,888 K HControl
HControlUs... Admin... 00 816 K HContro...
java.exe *32 Admin... 00 22,936 K Java(TM...
jusched.exe... Admin... 00 1,032 K Java(TM...
KBFiltr.exe ... SYSTEM 00 940 K KBFiltr
KiesPDLR.e... Admin... 00 6,152 K KiesPDLR
Show processes from all users
End Process
Processes: 110 CPU Usage: 35% Physical Memory: 61%

Task / Thread / Process

- Task = unit of program executed by a thread
- Process = unit of execution in OS-level
- Mapping between task – thread – process may be 1:1 or n:n, depending on level of concurrency



Concurrency

- From fine-grained to coarse-grained
  - Instruction level : multiple machine instructions
  - Statement level : multiple statements
  - Unit level : multiple tasks
  - Program level : multiple programs
- Languages that support concurrency are Ada 95, Fortran 90, Java, Occam, etc.

Concurrency Unit Control

- To execute concurrent program, we need concurrent processing units which may be physical or logical
  - Physical concurrency : threads run on physically separate processors
  - Logical concurrency : threads run on single processor
    - Each takes turn to use the processor & thread switching is very fast
    - Interleaving or timeslicing execution
- This course focuses on single-processor setup

Common Terms

- Multithreaded program
  - Threads run on single processor
  - Threads as lightweight processes, i.e. multiple program's threads are mapped to one OS process
    - Share process's CPU time, state, address space & heap, access to files & I/O devices
    - But each has its own flow of control, runtime stack, local variables
  - Concurrent threads typically execute separate tasks
    E.g. game characters, GUI components

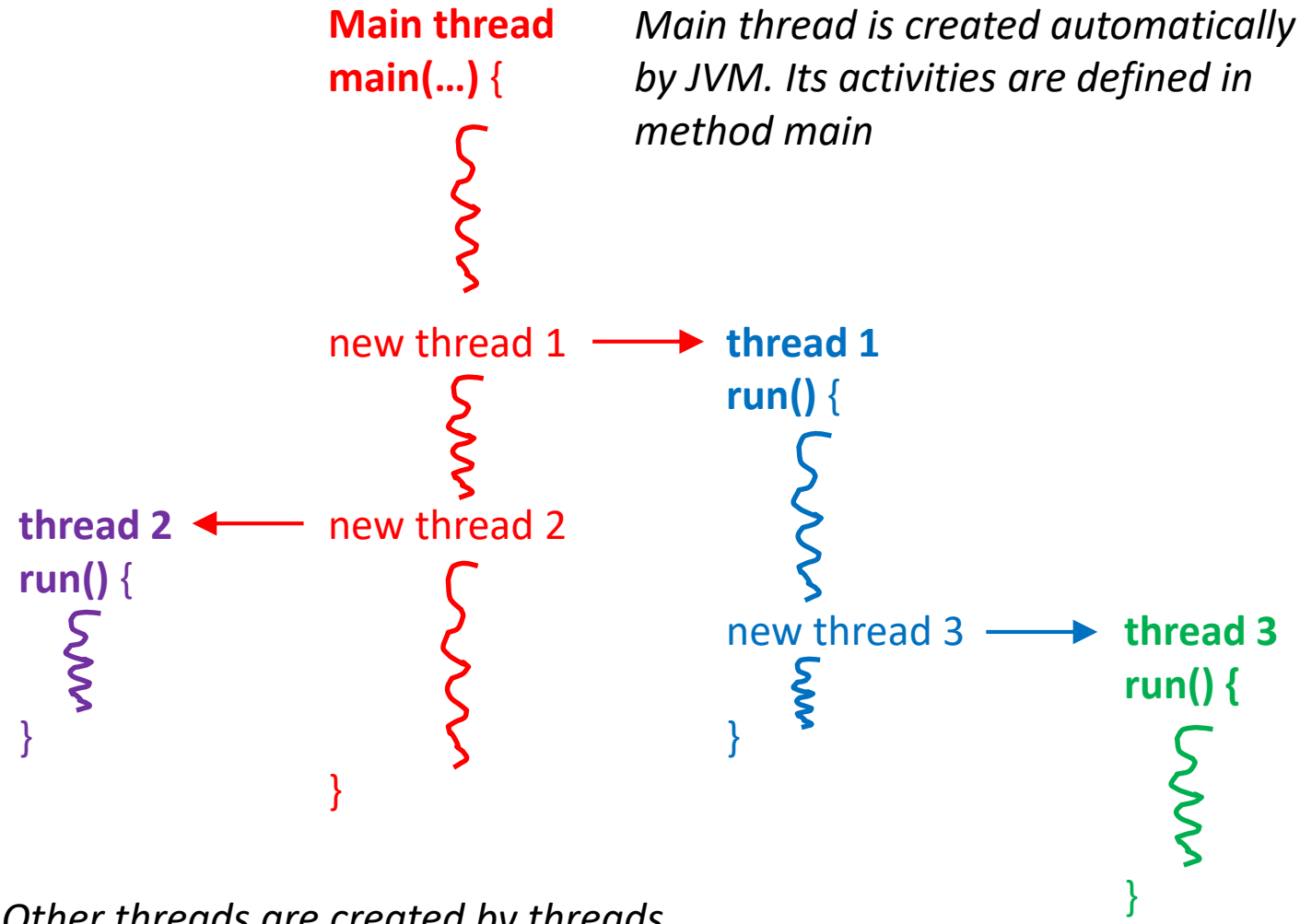
- Parallel program
  - Threads run on multiple processors
  - Often require program or data partitioning, from one sequential task into multiple smaller (& similar) tasks
    E.g. matrix calculation, encryption/decryption
  - Aim for execution time
- Distributed program
  - Threads run on network of autonomous processors
  - Threads execute independent tasks / programs
  - E.g. remote method invocation (RMI), web services

Concurrent Programming Outline

- Individual thread
  - Properties, states, life cycle
  - Thread execution
    - Sleep
    - Interrupt
    - Stop
- Interaction between threads (Ch.7)
  - Synchronization

Thread Creation

- During class loading, JVM creates main thread to run the class hosting main( ) method
  - Single-threaded program has only this main thread
  - In multithreaded program, main thread creates other threads to work
- Parent and child are independent of each other
  - Parent thread continues after creating child
  - Parent and child execute their own code (i.e. methods) at the same time



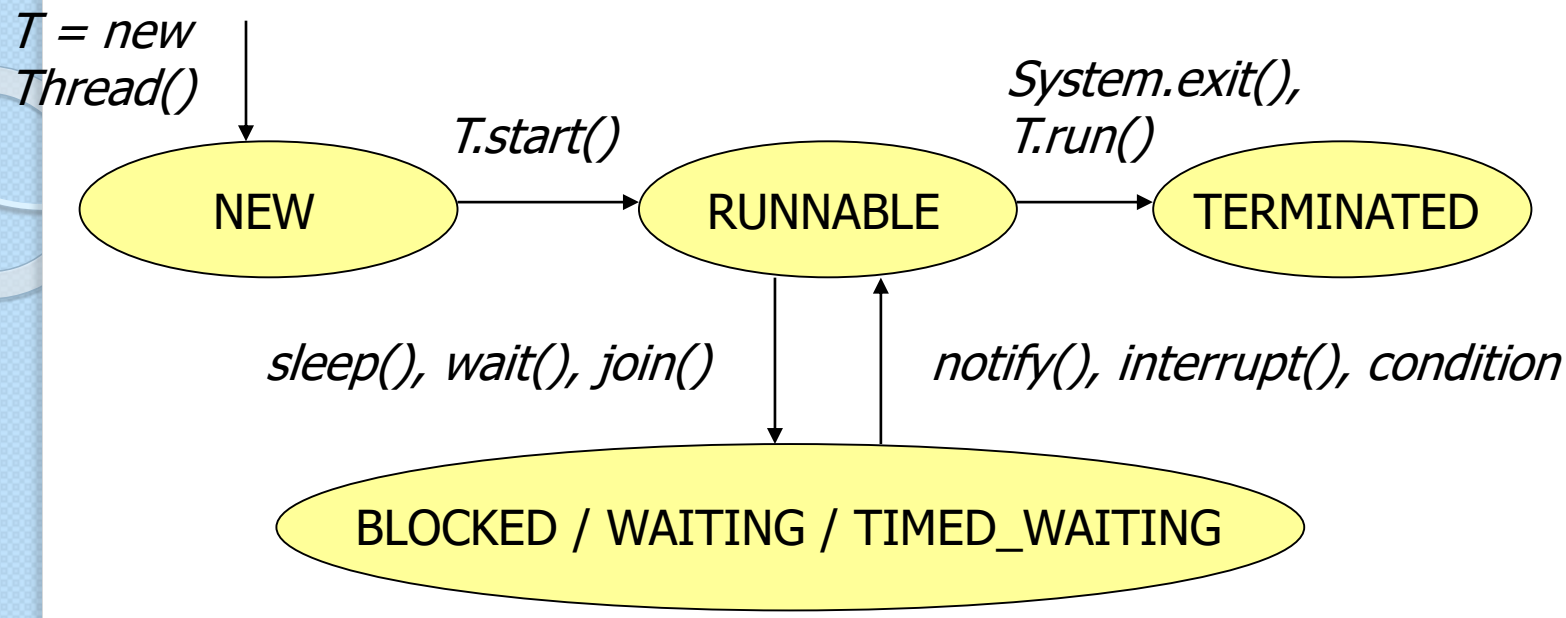
Main thread is created automatically by JVM. Its activities are defined in method main

Other threads are created by threads running in the program. Their activities are defined in method run

Thread Life Cycle

Generally, there are 5 stages

- New : thread is created, but not ready to run
- Runnable : thread is ready to run & queuing to use processor (there is only 1 processor)
- Running : thread is running on processor
- Blocked / Waiting / Timed Waiting
  - In Java, Blocked means waiting for synchronization and Waiting means waiting for certain conditions from other threads
- Terminated : thread stops & is destroyed



- Transition from RUNNABLE to RUNNING is handled by JVM, so there is no explicit RUNNING state in Java
- Thread is TERMINATED when it runs to completion, is destroyed, or when the program exits

Active Object

- Java objects are divided into active and passive
  - Active object has its own thread to manipulate itself
    = object from class that extends **Thread**
- Class **Thread**
  - public **Thread**( )
  - public **Thread**(String)
  - public final String **getName**( )
  - public final void **setName**(String)
  - public void **run**( )
  - public void **start**( )
  - public static native Thread **currentThread**( )

Example : using Thread

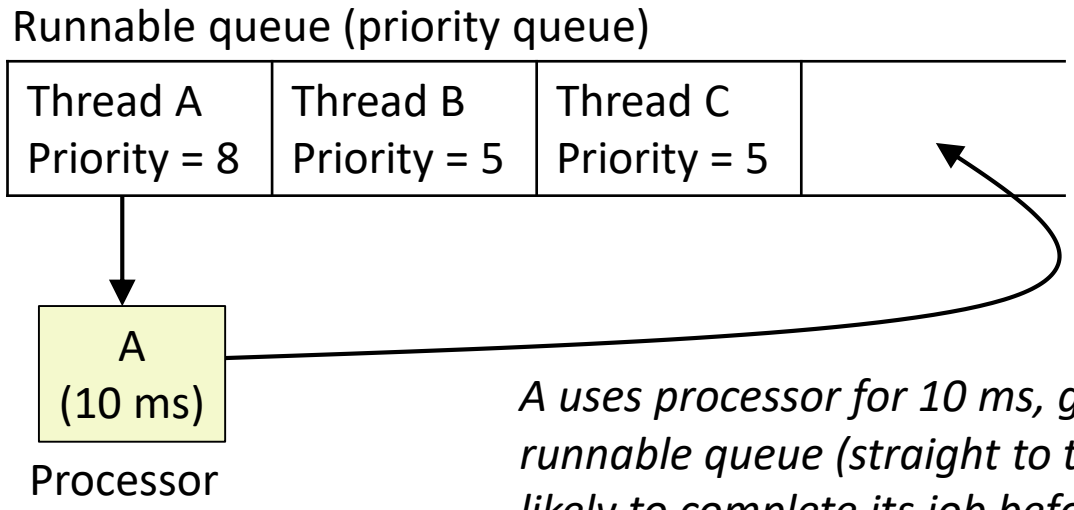
```
class MyThread extends Thread {
        MyThread( String name )          { super(name); }

        // thread's activities when it gets the processor
        public void run( )               { … }
}

MyThread T1 = new MyThread("A");
MyThread T2 = new MyThread("B");

T1.start( ); T2.start( );                // call run( ) automatically

T1.run( ); T2.run( );                    // T1's and T2's run( ) are called
                                         // in succession
                                         // not concurrent execution
```

Thread Priority

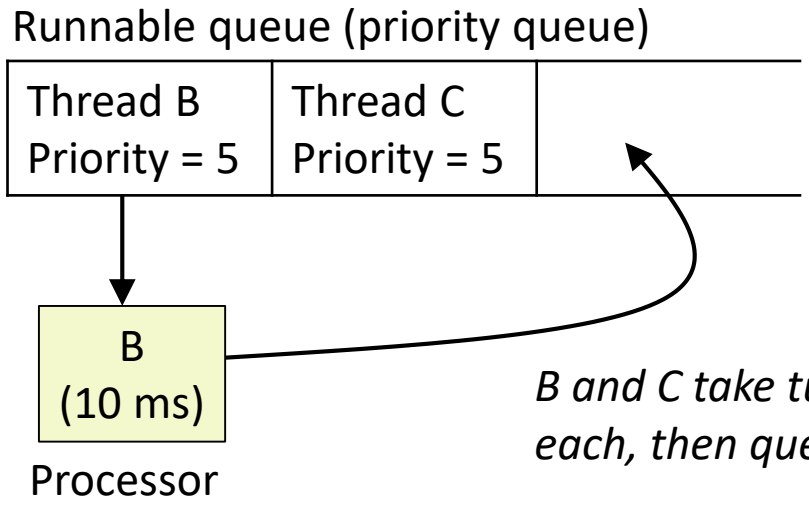
- Thread's priority in Java
  - 3 constants defined in class Thread : MIN_PRIORITY (=1), NORM_PRIORITY (=5, default), MAX_PRIORITY (=10)
  - Upon creation, child has the same priority as its parent's
  - New priority can be assigned but it must be between MIN_PRIORITY and MAX_PRIORITY
- Class Thread
  - public final void **setPriority**(int)
  - public final int **getPriority**( )

Thread Scheduling

- Single processor with timeslicing setup : runnable threads get the processor one-by-one
  - Equal timeslices or quantums for all threads. Thread with highest priority is likely to get the processor first (but this depends on OS and JVM's scheduler)
  - Once the time is up, running thread releases the processor & queues up for the next turn
- Some JVMs use non-timeslicing setup
  - Running thread does its work until completion or until a thread with higher priority requests the processor



A uses processor for 10 ms, gets out, enters runnable queue (straight to the front). So, it is likely to complete its job before the others



B and C take turns to use processor for 10 ms each, then queue up for the next rounds

Sleep

- Exception is thrown if thread is interrupted while sleeping
- In such case, it wakes up, catches the exception, and clears the interrupted flag
  - public static native void **sleep** (long millisec) throws InterruptedException
  - public static native void **sleep** (long millisec, int nanosec) throws InterruptedException
- Sleeping thread is in TIMED_WAITING state

Interrupt

- Mechanism may differ in different languages
  - Issue 1 ➔ general or specific interrupt ?
  - Issue 2 ➔ handler is compulsory (exception-like) or voluntary ?
- Java : general & voluntary approach
  - Interrupted flag is set to true
  - Running thread continues its execution. It may check flag & handle the interrupt later on
  - Blocked / waiting thread wakes up by InterruptedException

  - public void **interrupt** ( ) // interrupt thread
  - public boolean **isInterrupted** ( ) // check flag only
  - public static boolean **interrupted** ( ) // check & reset flag

Killing Thread

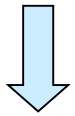
- There are many ways to kill thread

1. Call stop( ) → A.stop( ) to kill thread A
   - ThreadDeath error will be thrown. Once it propagates to JVM, thread will be dead
   - Anyone can kill thread & it may die in the midst of some execution, whose result affects other threads
2. Make thread throw ThreadDeath to kill itself
   - Still unsafe, e.g. if it kills itself inside a monitor
3. Use program logic, e.g.
   - Let thread run in a loop & break upon a condition
   - Return from method run upon a condition

- Class Thread
  - public final native boolean **isAlive** ( )
- ThreadDeath error
  - Inherit from class java.lang.Error
  - Can propagate to JVM by itself
  - May be caught if thread needs to do some cleanup task
    Then it must be re-thrown (to reach JVM)

Implementation of Java Thread

- Use **Thread** or make a new class extending Thread
  - public Thread ( )
  - public Thread (String)
- Implement interface **Runnable** and bind it to **Thread**
  - public interface Runnable {
    public abstract void run ( );
    }



binding Runnable and Thread

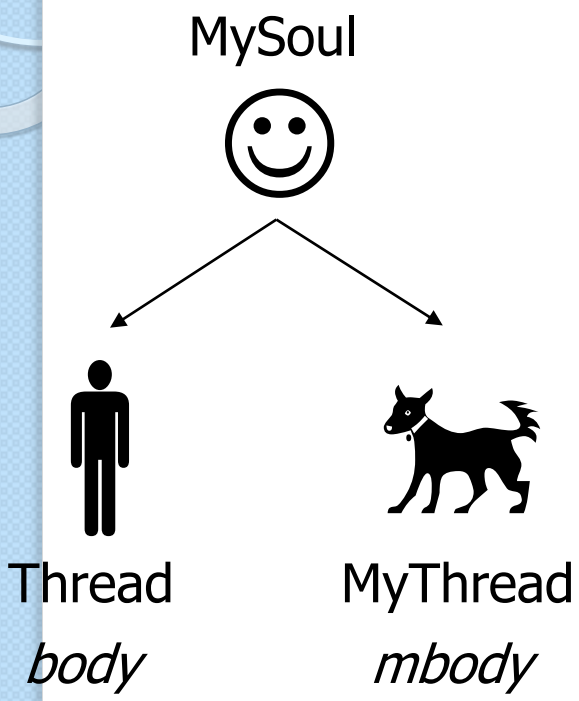
  - public Thread (Runnable)
  - public Thread (Runnable, String)

Interface Runnable

- If a class already inherits from another class
  - Java does not allow multiple inheritance, so we have to implement Runnable instead
  - Runnable object = soul, Thread object = body
  - Runnable must be bound to Thread to make active object

```
class MySoul extends Human implements Runnable {
    public MySoul( )          { … }
    public void run( )        { … }
    public void activity( )   { … }
}
Thread body        = new Thread( new MySoul( ) );
MyThread mbody  = new MyThread( new MySoul( ) );
```

Thread *vs.* Runnable



- When mbody.start( ) is called, JVM does the following:

  If MyThread overrides run( ) ➔ invoke this method

  Otherwise ➔ invoke run( ) that belongs to Runnable

- Thread object & Runnable object are 2 separate objects ∴ they cannot use the other's methods directly

Calling Thread Methods in Runnable

- Get thread currently executing Runnable object

```
class MySoul extends Human implements Runnable {
    public void run( )
    {
        // get thread on-the-fly
        MyThread me = (MyThread) Thread.currentThread( );
        // call MyThread's methods via "me"
    }
}
```

- Or keep Thread as member of Runnable object

```
class MySoul extends Human implements Runnable {
    private MyThread me;
    public MySoul( )
    {
        // keep pointer to thread since creating Runnable
        me = new MyThread(this);  me.start( );
    }
    public void run( )
    {   // call MyThread's methods via "me"  }
}
```

Calling Runnable Methods in Thread

- Keep Runnable object as member of Thread

```
class MyThread extends Thread {
    private MySoul me;
    public MyThread( MySoul s )
    {
        // keep pointer to Runnable since creating thread
        super(s);  me = s;
    }
    public void run( )
    {   // call MySoul's methods via "me"  }
}
```