12 Multi-Threads

Object-Oriented Programming

Program, Process, Multiprocessing

- Program
  - 실행되지 않은 상태의 소프트웨어
  - Disk에 파일형태로 존재, Memory에 load 후 실행
- Process
  - 실행 중인 program의 instance (OS에 의해 생성)
  - 자신만의 독립적 memory space (code, data, heap, stack)
- Multiprocessing
  - 여러 개의 process를 동시에 실행하여, parallel로 작업 수행
  - ex) 웹 서버를 multiprocessing하면 각 사용자의 요청을 동시에 서비스

Thread and Multithreading

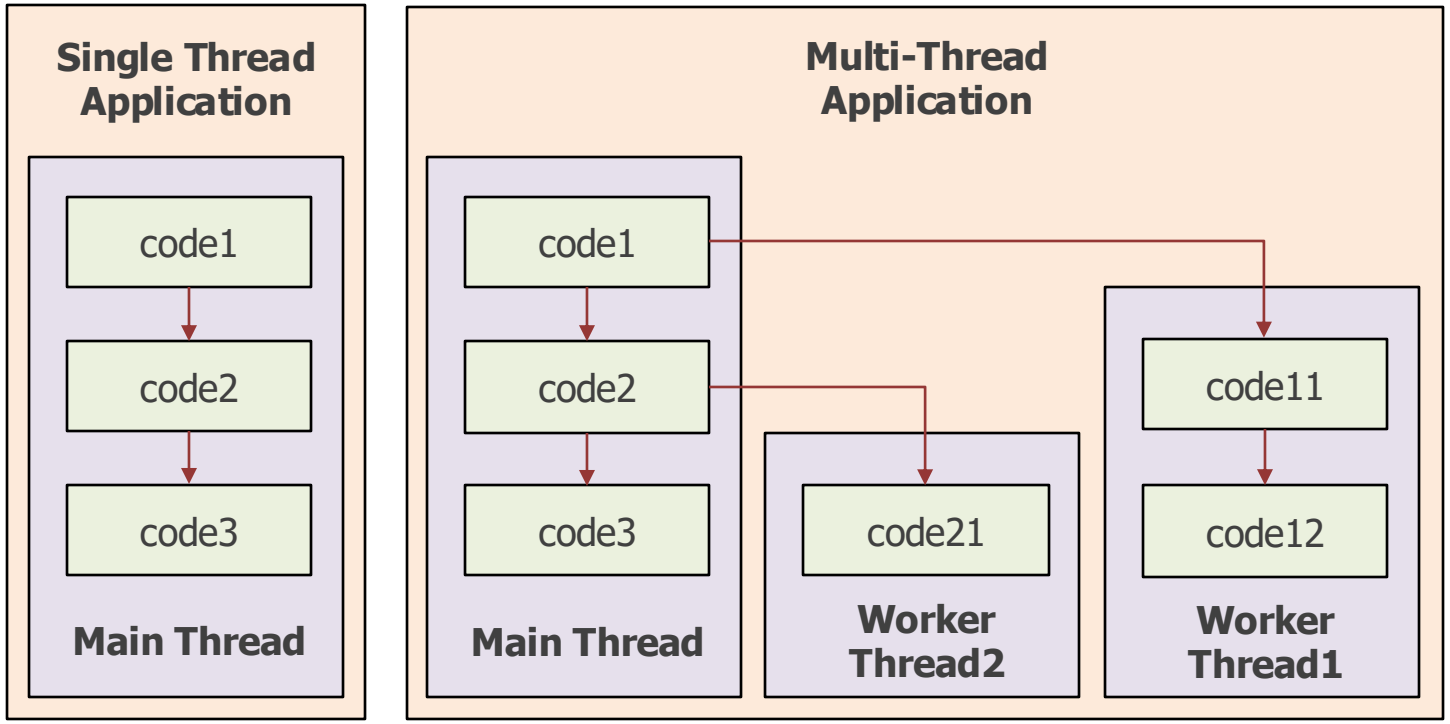
- Thread
  - Process 내에서 실행되는 작업의 단위
  - 하나의 process는 여러 개의 thread를 가질 수 있음
  - ex) Video Game
    - Main thread: game의 main loop 처리, game logic, input과 event 처리
    - Rendering thread: 3D graphics or 2D graphics를 display에 rendering
    - AI thread: NPC (Non-Player Character)의 AI 행동 계산
- Multithreading
  - 하나의 process 내에서 여러 thread를 생성, 동시에 여러 작업을 수행
  - thread간 memory 공유
  - Synchronization, deadlock, race condition 등 위험 존재
  - ex) Text Editor: UI, auto-completion, syntax analysis, file backup, ...

Task and Multitasking

- Task
  - 수행해야 할 작업이나 명령어의 단위를 의미
  - thread 또는 process에 의해 실행 됨
  - OS에서 scheduling 단위로 사용
  - Process나 Thread 보다 추상적인 개념
- Multitasking
  - OS가 여러 task (process 또는 thread) 를 동시에 실행
  - CPU resource의 효율적 사용을 위해 task가 번갈아 실행됨
  - 종류
    - Process-based multitasking (= multiprocessing)
    - Thread-based multitasking (= multithreading)

Main Thread and Worker Threads

- Single Thread Application
  - Main Thread 하나만 존재
    - code1, code2, code3
- Multi Thread Application
  - Main Thread
    - code1 (Worker Thread1 분기)
    - code2 (Worker Thread2 분기)
    - code3
  - Worker Thread1
    - code11
    - code12
  - Worker Thread2
    - code21



Example: main thread only

```
import java.awt.Toolkit;

public class BeepPrintExample {
	public static void main(String[] args) {
		Toolkit toolkit = Toolkit.getDefaultToolkit();
		for(int i=0; i<5; i++) {
			toolkit.beep(); // beep음 발생
			try { Thread.sleep(500); } // 0.5초 일시 정지
			catch(Exception e) {}
		}
		for(int i=0; i<5; i++) {
			System.out.println("띵");
			try { Thread.sleep(500); }
			catch(Exception e) {}
		}
	}
}
```

OUTPUT:
(0.5초 간격으로) beep 5번
(0.5초 간격으로) 아래 print
띵
띵
띵
띵
띵

Example: main + worker threads

```
import java.awt.Toolkit;

public class BeepPrintExample2 {
	public static void main(String[] args) {
		Thread thread = new Thread(new Runnable() {
			@Override
			public void run() {
				Toolkit toolkit = Toolkit.getDefaultToolkit();
				for(int i=0; i<5; i++) {
					toolkit.beep();
					try { Thread.sleep(500); } catch(Exception e) {}
				}
			}
		});
		thread.start(); // worker thread 실행
		for(int i=0; i<5; i++) {
			System.out.println("띵");
			try { Thread.sleep(500); } catch(Exception e) {}
		}
	}
}
```

worker thread code

worker thread 생성

main thread code

Creating Thread

1) Thread class로 직접 생성
   - java.lang.Thread class 에서 worker class를 직접 생성
   - Runnable implements object를 parameter로 하는 constructor를 call

```
Thread thread = new Thread(Runnable target);
```

2) Thread Child Class로 생성
   - Thread class inherit, run() method를 override, 실행 code를 run()에 작성

```
public class WorkerThread extends Thread {
   @Override
   public void run() {
      // thread가 실행할 code
   }
}
// thread object 생성
Thread thread = new WorkerThread();
```

```
// anonymous class 이용
Thread thread = new Thread() {
   @Override
   public void run() {
      // thread가 실행할 code
   }
};
thread.start();
```

Thread's Name

- Worker thead의 default name: Thread-n
- 다른 name으로 설정: Thread class의 setName() method

```
thread.setName("myThread");
```

- Naming: Debugging할 때 어떤 thread가 작업을 하는지 조사하는데 유용
- Thread.currentThread(): 현재 작업하는 thread의 reference return
- getName(): thread의 name return

```
Thread thread = Thread.currentThread();
System.out.println(thread.getName());
```

ThreadNameExample

```
public class ThreadNameExample {
	public static void main(String[] args) {
		Thread mainThread = Thread.currentThread();
		System.out.println(mainThread.getName() + " 실행");
		for(int i=0; i<3; i++) {
			Thread threadA = new Thread() {
				@Override
				public void run() {
					System.out.println(getName() + " 실행");
				}
			};
			threadA.start();
		}
		Thread chatThread = new Thread() {
			@Override
			public void run() {
				System.out.println(getName() + " 실행");
			}
		};
		chatThread.setName("chat-thread");
		chatThread.start();
	}
}
```

OUTPUT:
main 실행
Thread-1 실행
Thread-2 실행
Thread-0 실행
chat-thread 실행

OUTPUT:
main 실행
Thread-0 실행
Thread-1 실행
Thread-2 실행
chat-thread 실행

OUTPUT:
main 실행
Thread-1 실행
Thread-0 실행
Thread-2 실행
chat-thread 실행

OUTPUT:
main 실행
Thread-1 실행
chat-thread 실행
Thread-0 실행
Thread-2 실행

OUTPUT:
main 실행
Thread-1 실행
Thread-0 실행
chat-thread 실행
Thread-2 실행

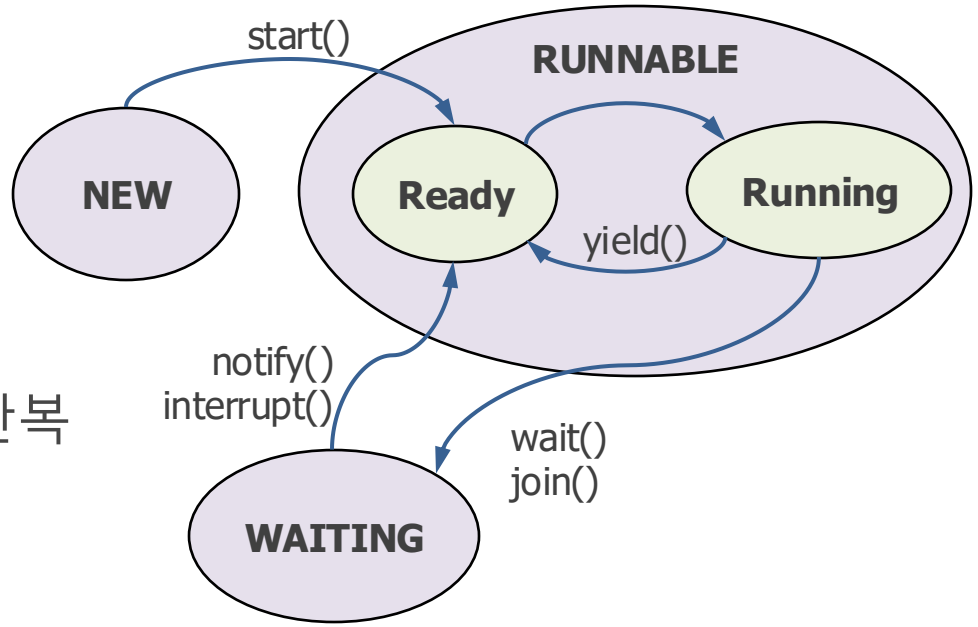
OUTPUT:
main 실행
Thread-0 실행
Thread-2 실행
chat-thread 실행
Thread-1 실행

Thread's States (1/2)

- NEW
  - Thread 생성됨. 아직 start 되지 않은 상태
  - start() method에 의해 RUNNABLE의 Ready로 전환

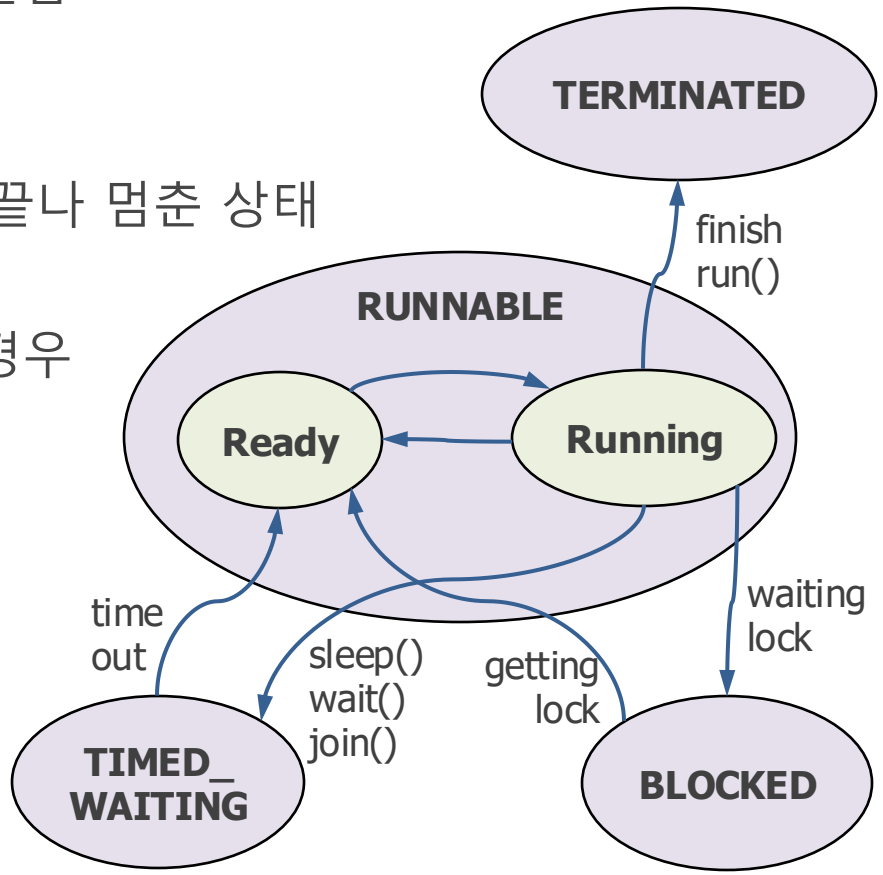
- RUNNABLE: CPU scheduling에 의해 아래 두 state를 반복
  - Ready: 실행을 기다리고 있는 상태.
  - Running: CPU를 점유, run() method를 실행하는 상태
    - Scheduling 또는 yield()로 Ready로 전환

- WAITING: Thread가 다른 thread의 작업 종료를 무한히 대기
  - wait(), join() 에 의해 진입
  - notify(), interrupt()에 의해 해제, RUNNABLE로 돌아감



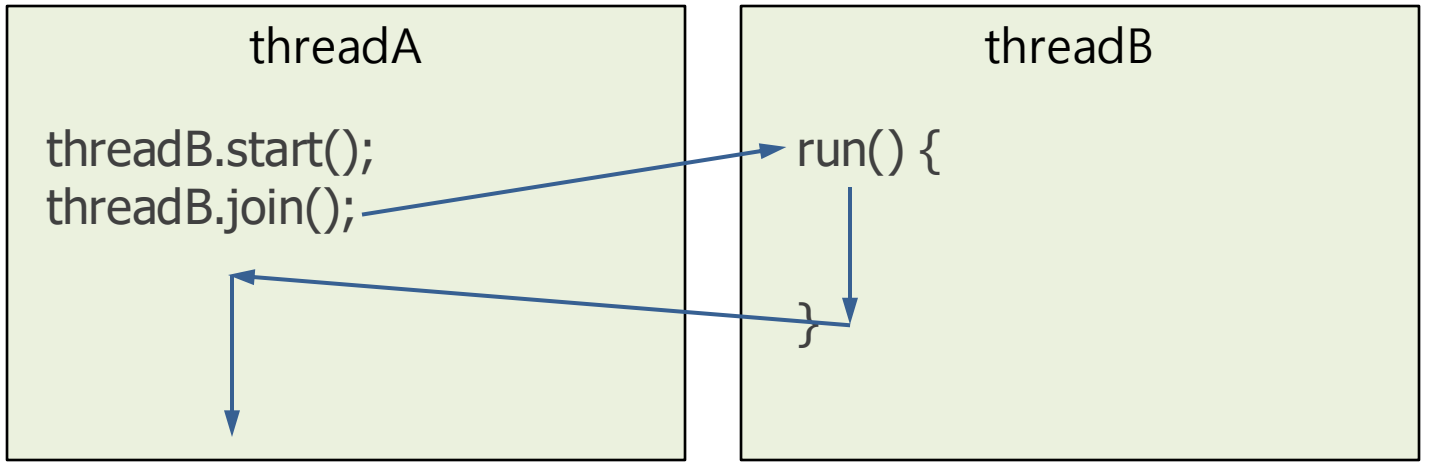
Thread's States (2/2)

- TIMED_WAITING: Thread가 정해진 시간동안 대기
  - sleep(), wait() (시간 지정시), join() (시간 지정시) 에 의해 진입
  - 정해진 시간 후에 해제, RUNNABLE로 돌아감
- TERMINATED
  - Running에서 run() method가 종료. Thread의 실행이 다 끝나 멈춘 상태
- BLOCKED
  - Synchronization method (또는 block) 으로 Lock이 걸린 경우
  - Lock과 Unlock으로 진입과 해제



Waiting for Other Thread

- 다른 thread의 종료를 기다렸다가 그 결과를 받아 처리하는 경우
- threadA
  - threadB.start()
  - threadB.join() // threadA는 일시 정지 (WAIT 상태), threadB의 종료를 기다림
  - threadB 종료 후 계속 진행
- threadB
  - run() 실행 후 종료



Example: Waiting for Other Thread

```
public class SumThread extends Thread {
        private long sum;
        public long getSum() { // accessor
                return sum;
        }
        @Override
        public void run() {
                for(int i=1; i<=100; i++) {
                   sum += i;
                }
        }
}
```

```
public class SumThreadJoin {
        public static void main(String[] args) {
                SumThread sumThread = new SumThread();
                sumThread.start(); // start sumThread
                try {
                        sumThread.join(); // main to WAIT
                } catch (InterruptedException e) { }
                System.out.println("1~100 합: " +
                   sumThread.getSum());
        }
}
```

OUTPUT:
1~100 합: 5050

Yielding Execution to Another Thread

```
public void run() {
   while(true) {
      if (work)
         System.out.println("threadA 작업");
   }
}
```

- work가 false인 경우
- while문은 의미 없는 반복
- 이 thread는 아무 것도 하지 않고 있음

```
public void run() {
   while(true) {
      if (work)
         System.out.println("threadA 작업");
      else
         Thread.yield();
   }
}
```

- 의미 없는 반복 없이
- work가 false인 경우 다른 thread에게 yield (양보)

Example: YieldExample (1/2)

```
public class YieldExample {
    public static void main(String[] args) {
        WorkThread workThreadA = new WorkThread("workThreadA");
        WorkThread workThreadB = new WorkThread("workThreadB");
        workThreadA.start();
        workThreadB.start();


        // 5초 후 workThreadA가 yield 시작, workThreadB가 더 많이 실행 됨
        try { Thread.sleep(5000); } catch (InterruptedException e) {}
        workThreadA.work = false;

        // 10초 후 workThreadA, workThreadB가 비슷한 횟수로 실행 됨
        try { Thread.sleep(10000); } catch (InterruptedException e) {}
        workThreadA.work = true;
    }
}
```

Example: YieldExample (2/2)

```
class WorkThread extends Thread {
   public boolean work = true; // 초기 work = true
   public WorkThread(String name) {
      setName(name);
   }
   @Override
   public void run() {
      while(true) {
         if(work) {
            System.out.println(getName() + ": 작업처리");
         } else {
            Thread.yield();
         }
      }
   }
}
```

OUTPUT:

workThreadB: 작업처리
...
workThreadA: 작업처리
...
workThreadB: 작업처리
...
workThreadA: 작업처리
...
workThreadA: 작업처리

Thread Synchronization

- User1Thread와 User2Thread가 하나의 object (Calculator)를 공유하며 작업
- User1Thread
  - Calculator’s memory = 100으로 set
  - 2초간 일시 정지
  - print Calculator’s memory: 50으로 바뀌어 있음 (User2Thread가 변경)
- User2Thread
  - Calculator’s memory = 50으로 set
  - 2초간 일시 정지

Example: Non-SynchronizedExample1 (1/4)

```
public class SynchronizedExample1 {
   public static void main(String[] args) {
      Calculator1 calculator = new Calculator1();

      User1Thread1 user1Thread = new User1Thread1();
      user1Thread.setCalculator(calculator);
      user1Thread.start();

      User2Thread1 user2Thread = new User2Thread1();
      user2Thread.setCalculator(calculator);
      user2Thread.start();
   }
}
```

Example: Non-SynchronizedExample1 (2/4)

```
class User1Thread1 extends Thread {
    private Calculator1 calculator;

    public User1Thread1() {
        setName("User1Thread");
    }

    public void setCalculator(Calculator1 calculator) {
        this.calculator = calculator;
    }

    @Override
    public void run() {
        calculator.setMemory(100);
    }
}
```

Example: Non-SynchronizedExample1 (3/4)

```
class User2Thread1 extends Thread {
   private Calculator1 calculator;

   public User2Thread1() {
      setName("User2Thread");
   }

   public void setCalculator(Calculator1 calculator) {
      this.calculator = calculator;
   }

   @Override
   public void run() {
      calculator.setMemory(50);
   }
}
```

Example: Non-SynchronizedExample1 (4/4)

```
class Calculator1 {
   private int memory;

   public int getMemory() {
      return memory;
   }

   public void setMemory(int memory) {
      this.memory = memory;
      try {
         Thread.sleep(2000);
      } catch(InterruptedException e) {}
      System.out.println(Thread.currentThread().getName() + ": " +  this.memory);
   }
}
```

OUTPUT:
User2Thread: 50
User1Thread: 50

Synchronized Method and Block

- Method나 Block을 한번에 하나의 thread만 실행할 수 있게 lock을 건다
- 그 thread가 synchronized method나 block의 실행을 끝내면 lock을 푼다
- 다른 thread가 lock을 얻어 걸고 synchronized method나 block을 실행

```
public synchronized void method() {
   // 하나의 thread만 실행하는 영역
}


public void method() {
   // 여러 thread가 실행할 수 있는 영역
   synchronized(this) {
      // 하나의 thread만 실행할 수 있는 영역
   }
   // 여러 thread가 실행할 수 있는 영역
}
```

Example: SynchronizedExample2 (1/2)

```
class Calculator2 {
   private int memory;

   public int getMemory() {
      return memory;
   }

   public synchronized void setMemory1(int memory) { // synchronized method
      this.memory = memory;
      try {
         Thread.sleep(2000);
      } catch(InterruptedException e) {}
      System.out.println(Thread.currentThread().getName() + ": " +  this.memory);
   }
```

Example: SynchronizedExample2 (2/2)

```
    public void setMemory2(int memory) {
        synchronized(this) { // synchronized block
            this.memory = memory;
            try {
                Thread.sleep(2000);
            } catch(InterruptedException e) {}
            System.out.println(Thread.currentThread().getName() + ": " +  this.memory);
        }
    }
}
```

OUTPUT:
User1Thread: 100
User2Thread: 50

Thread Control Using wait() and notify()

- Waiting pool에 BLOCKED thread들이 synchronized method (block)을 wait 중
- 한 thread가 작업을 완료하면 notify() 또는 notifyAll() 을 call하여 다른 thread (또는 wait pool의 모든 thread들)을 Ready 상태로 만들고, 자신은 wait()하여 BLOCKED 상태로 만듦
- wait()와 notify()는 synchronized method (block) 내에서만 실행 가능
- 이를 통해 thread들이 번갈아서 synchronized method (block) 을 실행 가능

Example: WaitNotifyExample (1/4)

```
public class WaitNotifyExample {
   public static void main(String[] args) {
      WorkObject workObject = new WorkObject();

      ThreadA threadA = new ThreadA(workObject);
      ThreadB threadB = new ThreadB(workObject);

      threadA.start();
      threadB.start();
   }
}
```

Example: WaitNotifyExample (2/4)

```
class ThreadA extends Thread {
    private WorkObject workObject;

    public ThreadA(WorkObject workObject) {
        setName("ThreadA");
        this.workObject = workObject;
    }

    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            workObject.methodA();
        }
    }
}
```

Example: WaitNotifyExample (3/4)

```
class ThreadB extends Thread {
    private WorkObject workObject;

    public ThreadB(WorkObject workObject) {
        setName("ThreadB");
        this.workObject = workObject;
    }

    @Override
    public void run() {
        for(int i=0; i<10; i++) {
            workObject.methodB();
        }
    }
}
```

Example: WaitNotifyExample (4/4)

```
class WorkObject {
   public synchronized void methodA() {
      Thread thread = Thread.currentThread();
      System.out.println(thread.getName() + ": methodA 작업 실행");
      notify(); // ThreadB를 Ready로
      try {
         wait(); // 자신(ThreadA)을 BLOCKED state로
      } catch (InterruptedException e) { }
   }
   public synchronized void methodB() {
      Thread thread = Thread.currentThread();
      System.out.println(thread.getName() + ": methodB 작업 실행");
      notify(); // ThreadA를 Ready로
      try {
         wait(); // 자신(ThreadB)를 BLOCKED state로
      } catch (InterruptedException e) { }
   }
}
```

OUTPUT:

```
ThreadA: methodA 작업 실행
ThreadB: methodB 작업 실행
ThreadA: methodA 작업 실행
ThreadB: methodB 작업 실행
ThreadA: methodA 작업 실행
…
```

Safe Termination of Thread

- Thread가 예상치 않은 상태나 부작용 없이, 모든 resource를 적절히 정리하고 작업을 마무리하는 것
  - Resource Release (resource 해제) – 파일, 네트워크 소켓 등 thread 사용resource 해제
  - State Cleanup (상태 정리) – thread 작업을 적절히 마무리
  - Consistent Behavior (일관된 동작) – 예측 가능한 종료 과정

Safe Termination Method

1) Volatile variable을 사용한 flag 방식
   - volatile variable: value가 변경될 때마다 모든 thread에서 즉시 반영 보장
2) Interrupt() method
   - thread의 WAIT, BLOCKED state 해제, 작업 중단 가능
3) ExecutorService 사용
   - thread pool 관리
   - work submission (thread pool의 thread에게 작업 의뢰)
   - termination control
   - Using shutdown() and shutdownNow()

Example: SafeStopVolatile

```
class VolatileThread extends Thread {
    private volatile boolean running = true;
    public void run() {
        while (running) {
            // 작업 수행
        }
    }
    public void stopThread() {
        running = false;
    }
}
public class SafeStopVolatile {
    public static void main(String[] args) throws InterruptedException {
        VolatileThread thread = new VolatileThread();
        thread.start();
        Thread.sleep(1000); // 1초 후에 종료
        thread.stopThread();
    }
}
```

Example: SafeStopInterrupt

```
class InterruptibleThread extends Thread {
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                Thread.sleep(100); // 작업 수행 중 차단 상태일 수 있음
            }
        } catch (InterruptedException e) { // interrupt가 발생했을 때 처리
            System.out.println("Thread interrupted");
        }
    }
}
public class SafeStopInterrupt {
    public static void main(String[] args) throws InterruptedException {
        InterruptibleThread thread = new InterruptibleThread();
        thread.start();
        Thread.sleep(1000); // 1초 후에 인터럽트
        thread.interrupt();
    }
}
```

OUTPUT:
Thread interrupted

Example: SafeStopExecutorService

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class SafeStopExecutorService {
   public static void main(String[] args) throws InterruptedException {
      // thread pool 생성 (2개 thread)
      ExecutorService executor = Executors.newFixedThreadPool(2);
      executor.submit(() -> { // pool의 1st thread에게 작업 의뢰
         while (!Thread.currentThread().isInterrupted()) {
            System.out.println(Thread.currentThread().getName() + " running");
         }
      });
      executor.submit(() -> { // pool의 2nd thread에게 작업 의뢰
         while (!Thread.currentThread().isInterrupted()) {
            System.out.println(Thread.currentThread().getName() + " running");
         }
      });
      Thread.sleep(10); // 10 miliseconds (0.001초) 후에 종료
      executor.shutdown(); // ExecutorService 종료
      if (!executor.awaitTermination(10, TimeUnit.MILLISECONDS)) {
         executor.shutdownNow(); // 강제 종료 (interrupt 발생)
      }
   }
}
```

OUTPUT:

```
pool-1-thread-1 running
pool-1-thread-1 running
pool-1-thread-2 running
pool-1-thread-1 running
pool-1-thread-2 running
...
```