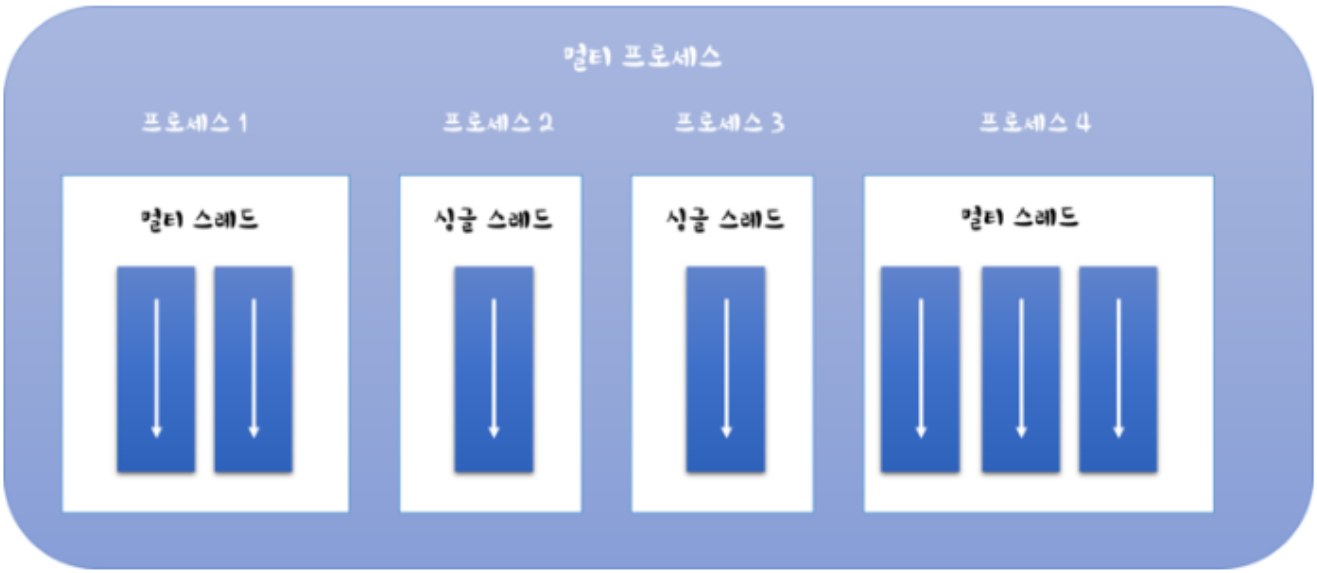
멀티 스레드 개념

12.1 멀티 스레드 개념

12.1.1 프로세스와 스레드

- 운영체제에서는 실행 중인 **하나의 애플리케이션을 프로세스(process)**라고 부른다. 사용자가 애플리케이션을 실행하면 운영체제로부터 실행에 필요한 메모리를 할당받는다.
- **멀티 태스킹(multi tasking)** : 두 가지 이상의 작업을 동시에 처리하는 것. 운영체제는 멀티 태스킹을 할 수 있도록 CPU 및 메모리 자원을 프로세스마다 적절히 할당시키고, 병렬로 실행시킨다.
- **멀티 스레드(multi thread)**
  - 운영체제에서 할당받은 자신의 메모리를 가지고 실행하기 때문에 서로 독립적이다.
  - 하나의 프로세스 내부에 생성되기 때문에 하나의 스레드가 예외를 발생시키면 프로세스 자체가 종료될 수 있다.

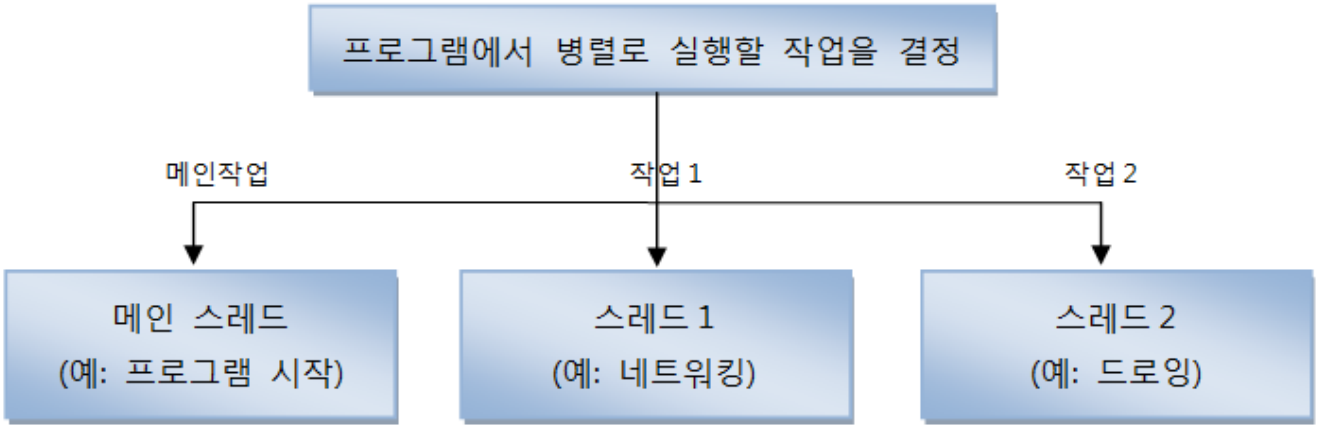


12.1.2 메인 스레드

: 모든 자바 애플리케이션은 메인 스레드(main thread)가 main() 메소드를 실행하면서 시작된다.

- **메인 스레드는 필요에 따라 작업 스레드들을 만들어서 병렬로 코드를 실행할 수 있다.**
- 멀티 스레드 애플리케이션에서는 실행 중인 스레드가 하나라도 있으면, 프로세스는 종료되지 않는다.

12.2 작업 스레드 생성과 실행



: 몇 개의 작업을 병렬로 실행할지 결정하고 각 작업별로 스레드로 생성해야 한다.

12.2.1 Thread 클래스로부터 직접 생성

- **Thread 클래스 생성**

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

- **Runnable 구현 클래스**

```
class Task implements Runnable {
    public void run() {
        스레드가 실행할 코드;
    }
}
```

- **예제1. 메인 스레드만 이용한 경우**

```
package thread_class_direct_production;

import java.awt.*;

public class BeepPrintExample1 {
    public static void main(String[] args) {
        // Toolkit 객체 얻기
        Toolkit toolkit = Toolkit.getDefaultToolkit();

        for(int i=0; i<5; i++) {
            toolkit.beep();     // 비프음 발생
            try {
                Thread.sleep(500);  // 0.5초간 일시 정지
            } catch (Exception e) {

            }
        }

        for(int i=0; i<5; i++) {
```

```
                System.out.println("띵");
                try {
                    Thread.sleep(500);  // 0.5초간 일시 정지
                } catch (Exception e) {

                }
            }
        }
    }
```

> 비프음을 5번 발생시키고 "띵"을 5번 출력시킨다.

- **예제2. 메인 스레드와 작업 스레드가 동시에 실행**

  BeepTask.java

```
package thread_class_direct_production;

import java.awt.*;

// Runnable 구현 클래스를 만든다
public class BeepTask implements 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) {}
        }
    }
}
```

  BeepPrintExample2.java

```
package thread_class_direct_production;

public class BeepPrintExample2 {
    public static void main(String[] args) {    // 메인 스레드
        Runnable beepTask = new BeepTask();     // Runnable 객체 생성
        Thread thread = new Thread(beepTask);   // beepTask 스레드 선언
        thread.start();     // beepTask 스레드 실행

        for(int i=0; i<5; i++) {
            System.out.println("띵");
            try {
                Thread.sleep(500);
            } catch (Exception e) { }
        }
    }
}
```

12.2.2 Thread 하위 클래스로부터 생성

: 작업 스레드가 실행할 작업을 Runnable로 만들지 않고, Thread의 하위 클래스로 작업 스레드를 정의하면서 작업 내용을 포함시킬 수도 있다.

```
public class WorkerThread extends Thread {
    @Override
    public void run() {
        // 스레드가 실행할 코드
    }
}
Thread thread = new WorkerThread(); // 스레드 생성

// 익명 객체로 작업 스레드 객체를 생성할 수도 있다.
Thread thread = new Thread(new Runnable(){
    @Override
    public void run() {
        // 스레드가 실행할 코드
    }
}
```

- **예제**

  BeepThread.java(비프음을 들려주는 스레드)

  ```
  package thread_down_class_production;

  import java.awt.*;

  public class BeepThread extends Thread {    // Thread 클래스를 상속한다.
      @Override
      public void run() {
          Toolkit toolkit = Toolkit.getDefaultToolkit();
          for(int i=0; i<5; i++) {
              toolkit.beep();
              try { Thread.sleep(500); } catch (Exception e) {}
          }
      }
  }
  ```

  BeepPrintExample.java

```
package thread_down_class_production;

public class BeepPrintExample {
    public static void main(String[] args) {
        Thread thread = new BeepThread();   // 스레드 생성
        thread.start();     // 스레드 실행

        for(int i=0; i<5; i++) {
            System.out.println("띵");
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }
}
```

12.2.3 스레드의 이름

: 우리가 직접 생성한 스레드는 자동적으로 "Thread-n"이라는 이름으로 설정된다. n은 스레드의 번호를 말한다. Thread 클래스의 **setName() 메소드**를 사용하면 스레드의 이름을 설정할 수 있다. 반대로 스레드의 이름을 알고 싶을 경우에는 **getName() 메소드**를 호출하면 된다. 만약 스레드의 객체의 참조를 가지고 있지 않다면, Thread의 정적 메소드인 currentThread)로 코드를 실행하면 참조를 얻을 수 있다.

- **예제(메인 스레드 이름 출력 및 UserThread 생성 및 시작)**

  ThreadA.java(스레드 A)

```
package thread_name;

public class ThreadA extends Thread {
    public ThreadA() {
        setName("ThreadA");     // 스레드 이름 설정
    }

    public void run() {
        for(int i=0; i<2; i++) {
            System.out.println(getName() + " 가 출력한 내용");
        }
    }
}
```

  ThreadB.java(스레드 B)

```java
package thread_name;

public class ThreadB extends Thread {
    // 스레드 이름 설정 안함
    @Override
    public void run() {
        for(int i=0; i<2; i++) {
            System.out.println(getName() + " 가 출력한 내용");
        }
    }
}
```

ThreadNameExample.java(main 스레드)

```java
package thread_name;

public class ThreadNameExample {
    public static void main(String[] args) {
        Thread mainThread = Thread.currentThread();
        System.out.println("프로그램 시작 스레드 이름: " + mainThread.getName());

        ThreadA threadA = new ThreadA();
        System.out.println("작업 스레드 이름: " + threadA.getName());
        threadA.start();

        ThreadB threadB = new ThreadB();
        System.out.println("작업 스레드 이름: " + threadB.getName());
        threadB.start();
    }
}
```

실행 결과

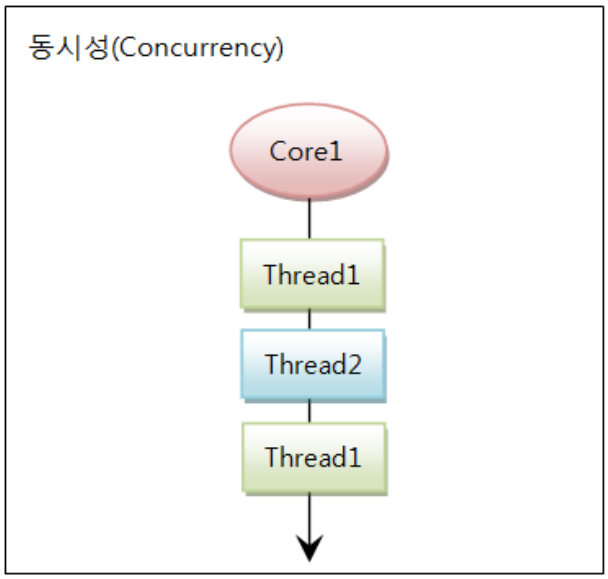
```
프로그램 시작 스레드 이름: main
작업 스레드 이름: ThreadA
작업 스레드 이름: Thread-1
Thread-1 가 출력한 내용
Thread-1 가 출력한 내용
ThreadA 가 출력한 내용
ThreadA 가 출력한 내용
```

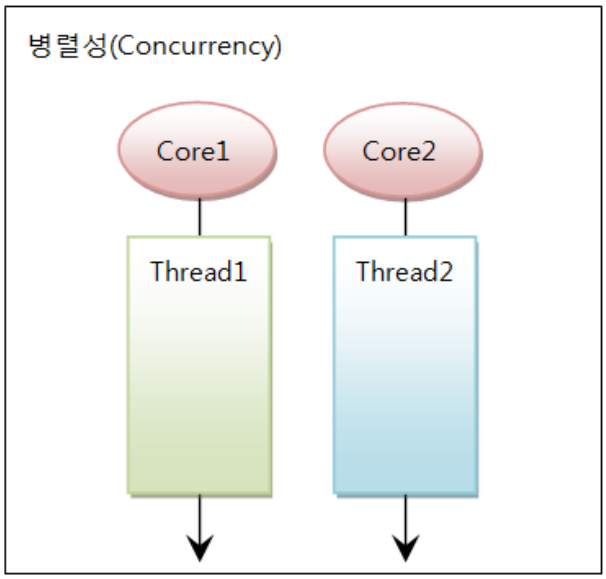
12.3 스레드 우선순위

: 멀티 스레드는 동시성(Concurrency) 또는 병렬성(Parallelism)으로 실행된다.

- **동시성(Concurrency)** : 멀티 작업을 위해 하나의 코어에서 멀티 스레드가 번갈아가며 실행하는 성질

- **병렬성(Parallelism)** : 멀티 작업을 위해 멀티 코어에서 개별 스레드를 동시에 실행하는 성질





- **스레드 스케줄링** : 스레드를 어떤 순서에 의해 동시성으로 실행할 것인가를 결정하는 것
  - **우선순위(Priority) 방식** : 우선순위가 높은 스레드가 실행 상태를 더 많이 가지도록 스케줄링하는 방식
  - **순환 할당 방식(Round-Robin)** : 시간 할당량(Time Slice)을 정해서 하나의 스레드를 정해진 시간만큼 실행하고 다시 다른 스레드를 실행하는 방식
- **우선순위 방식**

  : 우선순위는 1에서부터 10까지 부여가 된다. 우선순위를 부여하지 않으면 모든 스레드들은 기본적으로 5의 우선순위를 할당받는다. 만약 우선순위를 변경하고 싶으면 `setPriority()` 메소드를 이용하면 된다.

12.4 동기화 메소드와 동기화 블록

12.4.1 공유 객체를 사용할 때의 주의할 점

: 멀티 스레드의 스레드들이 객체를 공유해서 작업해야 하는 경우에 스레드 A를 사용하던 객체가 스레드 B에 의해 상태가 변경될 수 있기 때문에 주의해야 한다.

- 예제

  Calculator.java(공유 객체)

```
package careful_shared_object;

public class Calculator {
    private int memory;

    public int getMemory() {
        return memory;
    }

    // 계산기 메모리에 값을 저장하는 메소드
    public void setMemory(int memory) {
```

```
        // 매개값을 memory 필드에 저장
        this.memory = memory;

        try {
            // 스레드를 2초간 일시 정지
            Thread.sleep(2000);
        } catch(InterruptedException e) {}
        System.out.println(Thread.currentThread().getName() + ": " + this.memory);
    }
}
```

User1.java(User1 스레드)

```
package careful_shared_object;

public class User1 extends Thread{
    private Calculator calculator;

    public void setCalculator(Calculator calculator) {
        this.setName("User1");          // 스레드 이름 User1로 설정
        this.calculator = calculator;   // 공유 객체인 Calculator 을 필드에 저장
    }

    @Override
    public void run() {
        calculator.setMemory(100);      // 공유 객체인 Calculator 의 메모리에 100을 저장
    }
}
```

User2.java(User2 스레드)

```
package careful_shared_object;

public class User2 extends Thread {
    private Calculator calculator;

    public void setCalculator(Calculator calculator) {
        this.setName("User2");          // 스레드 이름 User2로 저장
        this.calculator = calculator;   // 공유 객체인 Calculator 을 필드에 저장
    }

    @Override
    public void run() {
        calculator.setMemory(50);       // 공유 객체인 Calculator 의 메모리에 50을 저장
    }
}
```

MainThreadExample.java(메인 스레드가 실행하는 코드)

```
package careful_shared_object;
```

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

        User1 user1 = new User1();
        user1.setCalculator(calculator);
        user1.start();  // user1 스레드 실행

        User2 user2 = new User2();
        user2.setCalculator(calculator);
        user2.start();  // user2 스레드 실행
    }
}
```

실행 결과

```
User1: 50
User2: 50
```

> user1은 100, user2은 50 으로 저장되어야 했으나, 둘다 50으로 저장되어 있는 오류를 볼 수 있다.

12.4.2 동기화 메소드 및 동기화 블록

: 멀티 스레드 프로그램에서 단 하나의 스레드만 실행할 수 있는 코드 영역을 **임계 영역(critical section)**이라고 한다. 임계 영역을 지정하기 위해 **동기화(synchronized) 메소드와 동기화 블록**을 사용한다. 동기화 메소드를 만드는 방법은 메소드 선언에 **synchronized 키워드**를 붙이면 된다.

```
public synchronized void method() {
    임계 영역;  // 단 하나의 스레드만 실행
}
```

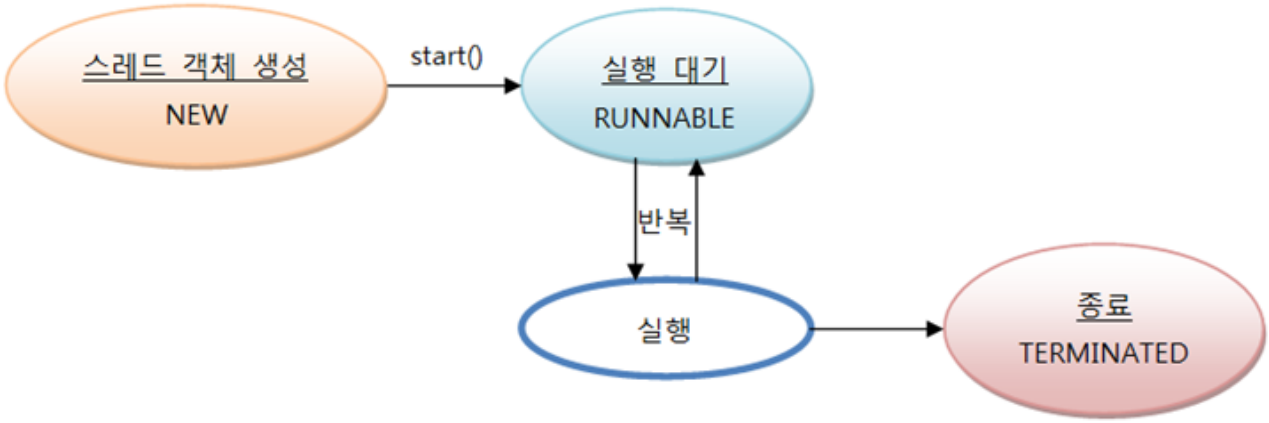
- 일부 내용만 임계 영역으로 만들고 싶다면 **동기화(synchronized) 블록**을 만들면 된다.

```
public void method() {
    // 여러 스레드가 실행 가능 영역
    ...
        synchronized(공유객체) {
        임계 영역;  // 단 하나의 스레드만 실행
    }
    // 여러 스레드가 실행 가능 영역
    ...
}
```

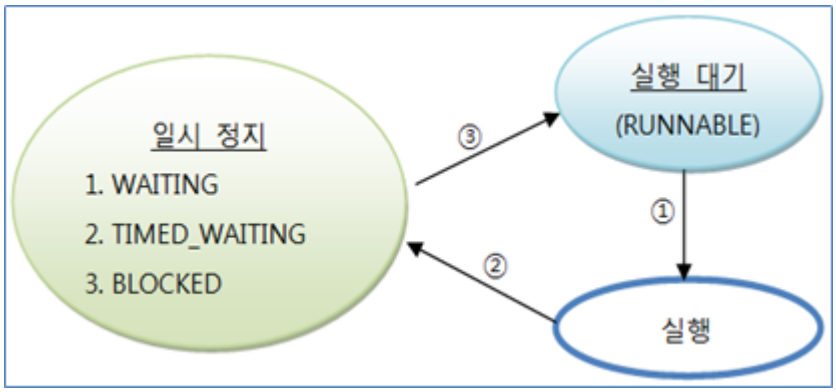
12.5 스레드 상태

1. **실행 대기 상태** : 아직 스케줄링이 되지 않아서 실행을 기다리고 있는 상태

2. **실행(Running) 상태** : 스케줄링으로 선택된 스레드가 비로서 CPU를 점유하고 run() 메소드를 실행
3. **종료 상태** : 실행 상태에서 run() 메소드가 종료되면, 더 이상 실행할 코드가 없기 때문에 스레드의 실행이 멈춤



- **일시 정지 상태** : 스레드가 실행할 수 없는 상태. WAITINGS, TIMED_WAITING, BLOCKED가 있다. 스레드가 다시 실행 상태로 가기 위해서는 일시 정지 상태에서 실행 대기 상태로 가야한다.



: 이러한 스레드의 상태를 코드에서 확인하기 위해 **getState() 메소드**로 호출하면 된다. 스레드 상태에 따라서 Thread.State 열거 상수를 리턴한다.

- **Thread.State 열거 상수**

상태	열거 상수	설명
객체 생성	NEW	아직 start() 메소드가 호출되지 않은 상태
실행 대기	RUNNABLE	실행 상태로 언제든지 갈 수 있는 상태
일시 정지	WAITING TIMED_WAITING BLOCKED	다른 스레드가 통지할 때까지 기다리는 상태 주어진 시간 동안 기다리는 상태 사용하고자 하는 객체의 락이 풀릴 때까지 기다리는 상태
종료	TERMINATED	실행을 마친 상태

- **예제**

StatePrintThread.java(타겟 스레드의 상태를 출력하는 스레드)

```java
package thread_state;

public class StatePrintThread extends Thread{
    private Thread targetThread;

    // 타겟 스레드를 파라미터로 받아서 저장
    // targetThread: 상태를 조사할 스레드
    public StatePrintThread(Thread targetThread) {
        this.targetThread = targetThread;
    }

    public void run() {
        while(true) {
            // 스레드 상태 얻기
            Thread.State state = targetThread.getState();
            System.out.println("타겟 스레드 상태: " + state);

            // 객체 생성 상태일 경우 실행 대기 상태로 만듬
            if(state == Thread.State.NEW) {
                targetThread.start();
            }

            // 종료 상태일 경우 while문을 종료함
            if(state == Thread.State.TERMINATED) {
                break;
            }

            try {
                Thread.sleep(500);
            } catch (Exception e) {}
        }
    }
}
```

TargetThread.java(타켓 스레드)

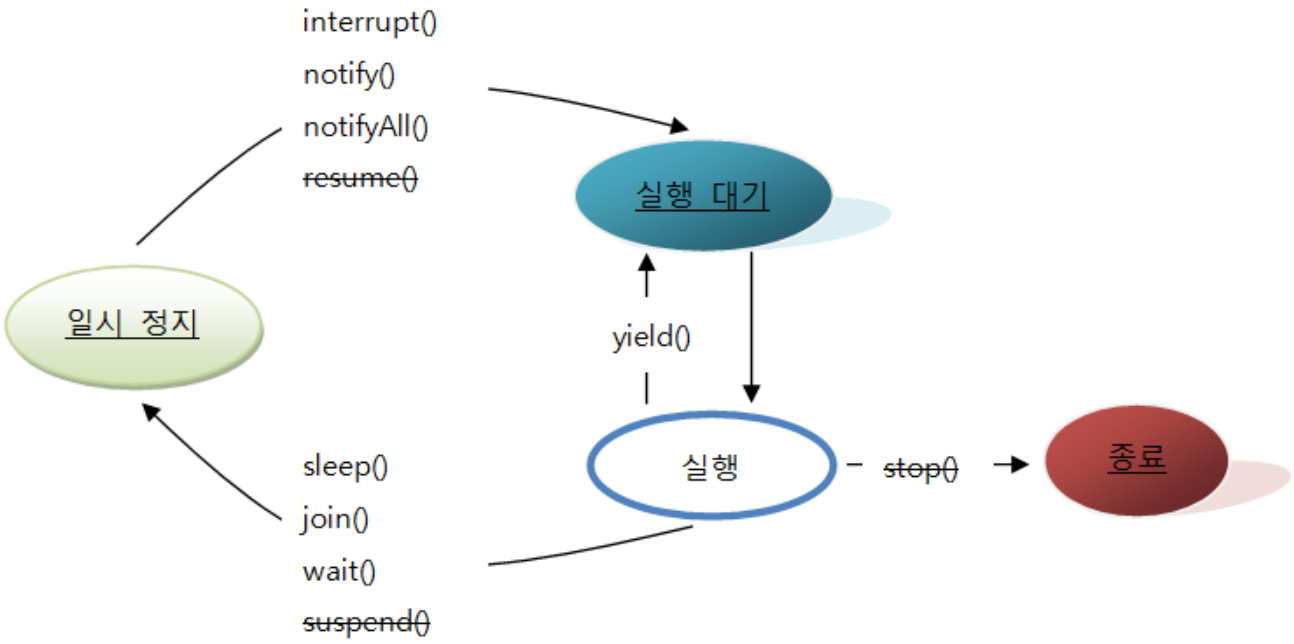
```java
package thread_state;

public class TargetThread extends Thread{
    @Override
    public void run() {
        for(long i=0; i<1000000000; i++){}

        try {
            Thread.sleep(1500);
        } catch (Exception e) {}

        for(long i=0; i<1000000000; i++) {}
    }
}
```

ThreadStateExample.java(**실행 클래스**)

```
package thread_state;

public class ThreadStateExample {
    public static void main(String[] args) {
        StatePrintThread statePrintThread =
                new StatePrintThread(new TargetThread());
        statePrintThread.start();
    }
}
```

12.6 스레드 상태 제어

: 스레드의 상태를 변경하는 것을 스레드 상태 제어라고 한다. 스레드 제어를 제대로 하기 위해서는 스레드의 상태 변화를 가져오는 메소드들을 파악하고 있어야 한다.



> 위 그림에서 취소선을 가진 메소드는 스레드의 안정성을 해친다고 하여 더 이상 사용하지 않도록 권장된 메소드들이다.

- **스레드의 상태 변화 메소드**

메소드	설명
interrupt()	일시 정지 상태의 스레드에서 InterruptedException 예외를 발생시켜, 예외 처리 코드(catch)에서 실행 대기 상태로 가거나 종료 상태로 갈 수 있도록 한다.
notify() notifyAll()	동기화 블록 내에서 wait() 메소드에 의해 일시 정지 상태에 있는 스레드를 실행 대기 상태로 만든다.
sleep(long millis) sleep(long millis, int nanos)	주어진 시간 동안 스레드를 일시 정지 상태로 만든다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다.
join() join(long millis) join(long millis, int nanos)	join() 메소드를 호출한 스레드는 일시 정지 상태가 된다. 실행 대기 상태로 가려면, join() 메소드를 멤버로 가지는 스레드가 종료되거나, 매개값으로 주어진 시간이 지나야 한다.
wait() wait(long millis) wait(long millis, int nanos)	동기화(synchronized) 블록 내에서 스레드를 일시 정지 상태로 만든다. 매개값으로 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다. 시간이 주어지지 않으면 notify(), notifyAll() 메소드에 의해 실행 대기 상태로 갈 수 있다.
yield()	실행 중에 우선순위가 동일한 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.

12.6.1 주어진 시간동안 일시 정지(sleep())

: 이 메소드를 호출한 스레드는 주어진 시간 동안 일시 정지 상태가 되고, 다시 실행 대기 상태로 돌아간다.

```
try {
    Thread.sleep(1000);
} catch(InterruptedException e) {
    // interrupt() 메소드가 호출되면 실행
}
```

> 일시 정지 상태에서 주어진 시간이 되기 전에 interrupt() 메소드가 호출되면 InterruptedException이 발생하기 때문에 예외 처리가 필요하다.

12.6.2 다른 스레드에게 실행 양보(yield())

: 다른 스레드에게 실행을 양보하고 자신은 실행 대기 상태로 가는 메소드. 이 메소드를 호출한 스레드는 실행 대기 상태로 돌아가고 동일한 우선순위 또는 높은 우선순위를 갖는 다른 스레드가 실행 기회를 가질 수 있도록 한다.

```
public void run() {
    while(ture) {
        if(work) {  // 만약 work의 값이 false이면
            System.out.println("ThreadA 작업 내용");
        } else {    // while문을 실행 대기 상태로 바꿔준다.
            Thread.yield();
        }
    }
}
```

12.6.3 다른 스레드의 종료를 기다림(join())

: 예를 들어 계산 작업을 하는 스레드가 모든 계산 작업을 마쳤을 때, 계산 결과값을 받아 이용하는 경우가 이에 해당된다.

- 예제

  SumThread.java(1부터 100까지 합을 계산하는 스레드)

  ```
  package thread_state_control.join_method;

  public class SumThread extends Thread {
      private long sum;

      public long getSum() {
          return sum;
      }

      public void setSum(long sum) {
          this.sum = sum;
      }

      public void run() {
          for(int i=1; i<=100; i++) {
              sum += i;
          }
      }
  }
  ```

  JoinExample.java(다른 스레드가 종료될 때까지 일시 정지 상태 유지)

  ```
  package thread_state_control.join_method;

  public class JoinExample {
      public static void main(String[] args) {
          SumThread sumThread = new SumThread();
  ```

```
        sumThread.start();

        try {
            // sumThread가 종료할 때까지 메인 스레드를 일시 정지 시킴
            sumThread.join();
        } catch (InterruptedException e) { }

        System.out.println("1~100 합: " + sumThread.getSum());
    }
}
```

12.6.4 스레드 간 협업(wait(), notify(), notifyAll())

: 공유 객체는 두 스레드가 작업할 내용을 각각 동기화 메소드로 구분해 놓는다. 한 스레드가 작업을 완료하면 **notify() 메소드**를 호출해서 일시 정지 상태에 있는 다른 스레드를 실행 대기 상태로 만들고, 자신은 두 번 작업을 하지 않도록 **wait() 메소드**를 호출하여 일시 정지 상태로 만든다.

- **예제**

  ThreadA.java(**WorkObject의 methodA()를 실행하는 스레드**)

```
package thread_cooperation;

public class ThreadA extends Thread {
    private WorkObject workObject;

    public ThreadA(WorkObject workObject) {
        // 공유 객체를 매개값으로 받아 필드에 저장
        this.workObject = workObject;
    }

    @Override
    public void run() {
        // 공유 객체의 methodA()를 10번 반복 호출
        for(int i=0; i<10; i++) {
            workObject.methodA();
        }
    }
}
```

  ThreadB.java(**WorkObject의 methodB()를 실행하는 스레드**)

```
package thread_cooperation;

public class ThreadB extends Thread{
    private WorkObject workObjecct;

    public ThreadB(WorkObject workObjecct) {
        // 공유 객체를 매개값으로 받아 필드에 저장
        this.workObjecct = workObjecct;
    }
```

```
    @Override
    public void run() {
        // 공유 객체의 methodB()를 10번 반복 호출
        for(int i=0; i<10; i++) {
            workObjecct.methodB();
        }
    }
}
```

WorkObject.java(**두 스레드의 작업 내용을 동기화 메소드로 작성한 공유 객체**)

```
package thread_cooperation;

public class WorkObject {
    public synchronized void methodA() {
        System.out.println("ThreadA의 methodA() 작업 실행");
        notify();   // 일시 정지 상태에 있는 ThreadB를 실행 대기 상태로 만듬
        try {
            wait(); // ThreadA를 일시 정지 상태로 만듬
        } catch (InterruptedException e) { }
    }

    public synchronized void methodB() {
        System.out.println("ThreadB의 methodB() 작업 실행");
        notify();   // 일시 정지 상태에 있는 ThreadA를 실행 대기 상태로 만듬
        try {
            wait(); // ThreadB를 일시 정지 상태로 만듬
        } catch (InterruptedException e) { }
    }
}
```

WaitNotifyExample.java(**두 스레드를 생성하고 실행하는 메인 스레드**)

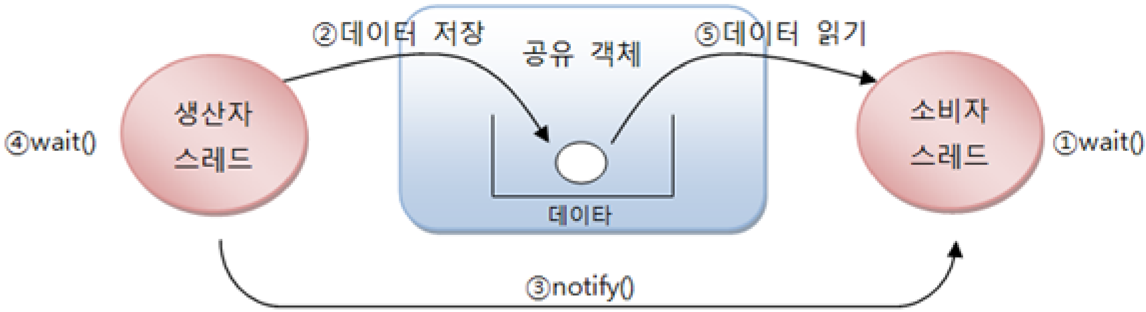
```
package thread_cooperation;

public class WaitNotifyExample {
    public static void main(String[] args) {
        WorkObject sharedObject = new WorkObject();     // 공유 객체

        ThreadA threadA = new ThreadA(sharedObject);    // ThreadA 생성
        ThreadB threadB = new ThreadB(sharedObject);    // ThreadB 생성

        threadA.start();    // ThreadA 실행
        threadB.start();    // ThreadB 실행
    }
}
```

실행 결과

```
ThreadA의 methodA() 작업 실행
```

```
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
ThreadA의 methodA() 작업 실행
ThreadB의 methodB() 작업 실행
```

- **데이터를 저장하는 스레드(생산자 스레드)가 데이터를 저장하면, 데이터를 소비하는 스레드(소비자 스레드)가 데이터를 읽고 처리하는 교대 작업을 구현하는 예제**



DataBox.java(두 스레드의 작업 내용을 동기화 메소드로 작성한 공유 객체)

```
package thread_cooperation;

public class DataBox {
    private String data;

    public synchronized String getData() {
        // data 필드가 null 이면 소비자 스레드를
        // 일시 정지 상태로 만든다
        if(this.data == null) {
            try {
                wait();
            } catch (InterruptedException e) {

            }
        }
        String returnValue = data;
        System.out.println("ConsummerThread가 읽은 데이터: " + returnValue);
```

```
        // data 필드를 null 로 만들고 생산자 스레드를 실행 대기 상태로 만든다.
        data = null;
        notify();

        return returnValue;
    }

    public synchronized void setData(String data) {
        // data 필드가 null 이 아니면 생상자 스레드를
        // 일시 정지 상태로 만든다.
        if(this.data != null) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }

        // data 필드에 값을 저장한다.
        this.data = data;
        System.out.println("ProducerThread가 생성한 데이터: " + data);
        // 소비자 스레드를 실행 대기 상태로 만든다.
        notify();
    }
}
```

ProducerThread.java(데이터를 생산(저장)하는 스레드)

```
package thread_cooperation;

public class ProducerThread extends Thread {
    private DataBox dataBox;

    // 공유 객체를 필드에 저장
    public ProducerThread(DataBox dataBox) {
        this.dataBox = dataBox;
    }

    @Override
    public void run() {
        for(int i=1; i<=3; i++) {
            String data = "Data-" + i;
            // 새로운 데이터를 저장
            dataBox.setData(data);
        }
    }
}
```

ConsummerThread.java(데이터를 소비하는(읽는) 스레드)

```
package thread_cooperation;

public class ConsumerThread extends Thread {
```

```
    private DataBox dataBox;

    // 공유 객체를 필드에 저장
    public ConsumerThread(DataBox dataBox) {
        this.dataBox = dataBox;
    }

    @Override
    public void run() {
        for(int i=1; i<=3; i++) {
            // 새로운 데이터를 읽음
            String data = dataBox.getData();
        }
    }
}
```

WaitNotifyExampe2.java(두 스레드를 생성하고 실행하는 메인 스레드)

```
package thread_cooperation;

public class WaitNotifyExample2 {
    public static void main(String[] args) {
        DataBox dataBox = new DataBox();

        ProducerThread producerThread = new ProducerThread(dataBox);
        ConsumerThread consumerThread = new ConsumerThread(dataBox);

        producerThread.start();
        consumerThread.start();
    }
}
```

실행 결과

```
ProducerThread가 생성한 데이터: Data-1
ConsummerThread가 읽은 데이터: Data-1
ProducerThread가 생성한 데이터: Data-2
ConsummerThread가 읽은 데이터: Data-2
ProducerThread가 생성한 데이터: Data-3
ConsummerThread가 읽은 데이터: Data-3
```

12.6.5 스레드의 안전한 종료(stop 플래그, interrupt())

: 경우에 따라서 실행 중인 스레드를 즉시 종료하고 싶을 때, **stop() 메소드**를 사용한다. 하지만 이 메소드는 deprecated 되었다. 왜냐하면 stop() 메소드로 스레드를 갑자기 종료하게 되면 스레드가 사용 중이던 자원들이 불안정한 상태로 남겨지기 때문이다.

stop 플래그를 이용하는 방법

- 예제

  PrintThread1.java(무한 반복해서 출력하는 스레드)

```
package thread_safe_stop;

public class PrintThread1 extends Thread{
    private boolean stop;

    public void setStop(boolean stop) {
        this.stop = stop;
    }

    public void run() {
        // stop 이 true 가 되면 자원을 정리한 후 실행을 종료시킨다.
        while(!stop) {
            System.out.println("실행 중");
        }
        System.out.println("자원 정리");
        System.out.println("실행 종료");
    }
}
```

  StopFlagExample.java(1초 후 출력 스레드를 중지시킴)

```
package thread_safe_stop;

public class StopFlagExample {
    public static void main(String[] args) {
        PrintThread1 printThread1 = new PrintThread1();
        printThread1.start();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}

        // 스레드를 종료시키기 위해 stop 필드를 true 로 변경
        printThread1.setStop(true);
    }
}
```

  실행 결과

```
실행 중
실행 중
실행 중
실행 중
실행 중
실행 중
실행 중
실행 중
실행 중
자원 정리
실행 종료
```

interrupt() 메소드를 이용하는 방법

: 이 메소드는 스레드가 일시 정지 상태에 있을 때 InterruptedException 예외를 발생시키는 역할을 한다. 이것을 이용하면 run() 메소드를 정상 종료시킬 수 있다.

- **예제**

  PrintThread2.java(무한 반복해서 출력하는 스레드)

```java
package thread_safe_stop.interrupt_method;

public class PrintThread2 extends Thread{
    @Override
    public void run() {
        try {
            while(true) {
                System.out.println("실행 중");
                Thread.sleep(1);
                // interrupt() 메소드를 실행시키면 catch 문으로 이동한다.
            }
        } catch (InterruptedException e) {}

        System.out.println("자원 정리");
        System.out.println("실행 종료");
    }

}
```

  InterruptExample.java(1초 후 출력 스레드를 중지시킴)

```java
package thread_safe_stop.interrupt_method;

public class InterruptExample {
    public static void main(String[] args) {
        Thread thread = new PrintThread2();
        thread.start();
```

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}

            // 스레드를 종료시키기 위해
            // InterruptedException 을 발생시킨다
            thread.interrupt();
        }
    }
```

실행 결과

```
실행 중
실행 중
실행 중
실행 중
...
실행 중
자원 정리
실행 종료
```

> **주의할 점**
>
> : 스레드가 미래에 일시 정지 상태가 되면 InterruptedException 예외가 발생한다는 것이다. 그래서 짧은 시간이나마 일시 정지시키기 위해 **Thread.sleep(1)**을 사용한 것이다. Thread.sleep(1) 말고도 **Thread.interrupted()**를 사용해서 스레드를 멈추도록 할 수 있다.

12.7 데몬 스레드

- **데몬(daemon) 스레드**
  - 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드이다.
  - 주 스레드가 종료되면 데몬 스레드는 강제적으로 자동 종료된다.
  - 주 스레드는 데몬이 될 스레드의 setDaemon(true)를 호출해주면 된다.

```
public static void main(String[] args) {
    AutoSaveThread thread = new AutoSaveThread();
    thread.setDaemon(true);
    thread.start()
    ...
}
```

> 위의 코드에서 메인 스레드가 주 스레드가 되고 AutoSaveThread가 데몬 스레드가 된다.
>
> **주의할 점**
>
> : start() 메소드가 호출되고 나서 setDaemon(true)를 호출하면 IllegalThreadStateException이 발생한다는 것.

12.8 스레드 그룹

- **스레드 그룹(ThreadGroup)** : 관련된 스레드를 묶어서 관리할 목적으로 이용된다. 스레드는 반드시 하나의 스레드 그룹에 포함되는데, 명시적으로 스레드 그룹에 포함시키지 않으면 기본적으로 자신을 생성한 스레드와 같은 스레드 그룹에 속하게 된다.

12.8.1 스레드 그룹 이름 얻기

- 현재 스레드가 속한 스레드 그룹의 이름을 얻고 싶다면 다음과 같은 코드를 사용할 수 있다.

```
ThreadGroup group = Thread.currentThread().getThreadGroup();
String groupName = group.getName();
```

- **getAllStackTraces()**를 이용하면 프로세스 내에서 실행하는 모든 스레드에 대한 정보를 얻을 수 있다.

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
```

12.8.2 스레드 그룹 생성

- 명시적으로 스레드 그룹을 만들고 싶다면 생성자를 이용해서 ThreadGroup 객체를 만들면 된다.

```
// 스레드 그룹 이름만 매개값으로
ThreadGroup tg = new ThreadGroup(String name);
// 부모 스레드 그룹과 이름을 매개값으로
ThreadGroup tg = new ThreadGroup(ThreadGroup parent, String name);
```

> 스레드 그룹 생성 시 부모(parent) 스레드 그룹을 지정하지 않으면 현재 스레드가 속한 그룹의 하위 그룹으로 생성된다.

- 스레드 그룹을 매개값으로 갖는 Thread 생성자

```
Thread t = new Thread(ThreadGroup group, Runnable target);
Thread t = new Thread(ThreadGroup group, Runnable target, String name);
Thread t = new Thread(ThreadGroup group, Runnable target, String name, long
stackSize);
Thread t = new Thread(ThreadGroup group, String name);
```

12.8.3 스레드 그룹의 일괄 interrupt()

: 스레드 그룹에서 제공하는 interrupt() 메소드를 이용하면 그룹 내에 포함된 모든 스레드들을 일괄 interrupt 할 수 있다.

- **ThreadGroup이 가지고 있는 주요 메소드들**

반환형	메소드	설명
int	activeCount()	현재 그룹 및 하위 그룹에서 활동 중인 모든 스레드의 수를 리턴한다.
int	activeGroupCount()	현재 그룹에서 활동 중인 모든 하위 그룹의 수를 리턴한다.
void	checkAccess()	현재 스레드가 스레드 그룹을 변경할 권한이 있는지 체크한다. 만약 권한이 없으면 SecurityException을 발생시킨다.
void	destroy()	현재 그룹 및 하위 그룹을 모두 삭제한다. 단, 그룹 내에 포함된 모든 스레드들이 종료 상태가 되어야 한다.
boolean	isDestroyed()	현재 그룹이 삭제되었는지 여부를 리턴한다.
int	getMaxPriority()	현재 그룹에 포함된 스레드가 가질 수 있는 최대 우선순위를 리턴한다.
void	setMaxPriority(int pri)	현재 그룹에 포함된 스레드가 가질 수 있는 최대 우선순위를 설정한다.
String	getName()	현재 그룹의 이름을 리턴한다.
ThreadGroup	getParent()	현재 그룹의 부모 그룹을 리턴한다.
boolean	parentOf(ThreadGroup g)	현재 그룹이 매개값으로 지정한 스레드 그룹의 부모인지 여부를 리턴한다.
boolean	isDaemon()	현재 그룹이 데몬 그룹인지 여부를 리턴한다.
void	setDaemon(boolean daemon)	현재 그룹을 데몬 그룹으로 설정한다.
void	list()	현재 그룹에 포함된 스레드와 하위 그룹에 대한 정보를 출력한다.
void	interrupt()	현재 그룹에 포함된 모든 스레드들을 interrupt 한다.

- **예제**

WorkThread.java(**InterruptedException이 발생할 때 스레드가 종료되도록 함**)

```
package thread_group_interrupt;

public class WorkThread extends Thread{
    public WorkThread(ThreadGroup threadGroup, String threadName) {
        // 스레드 그룹과 스레드 이름을 설정
        super(threadGroup, threadName);
    }

    @Override
    public void run() {
        while(true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // 예외 발생시 스레드 종료
                System.out.println(getName() + " interrupted");
                break;
            }
        }
        System.out.println(getName() + " 종료됨");
    }
}
```

ThreadGroupExample.java(**스레드 그룹을 이요한 일괄 종료 예제**)

```
package thread_group_interrupt;

public class ThreadGroupExample {
    public static void main(String[] args) {
        // 스레드 그룹 생성
        ThreadGroup myGroup = new ThreadGroup("myGroup");

        // myGroup 스레드 그룹에 두 스레드를 포함
        WorkThread workThreadA = new WorkThread(myGroup, "workThreadA");
        WorkThread workThreadB = new WorkThread(myGroup, "workThreadB");

        workThreadA.start();
        workThreadB.start();

        System.out.println("[ main 스레드 그룹의 list() 메소드 출력 내용 ]");
        // mainGroup 에 현재 스레드의 그룹을 가져온다.
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();
        // mainGroup 의 포함된 스레드와 하위 그룹에 대한 정보를 출력 시킨다.
        mainGroup.list();
        System.out.println();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {}
```

```
            System.out.println("[ myGroup 스레드 그룹의 interrupt() 메소드 호출 ]");
            myGroup.interrupt();
        }
    }
```

실행 결과

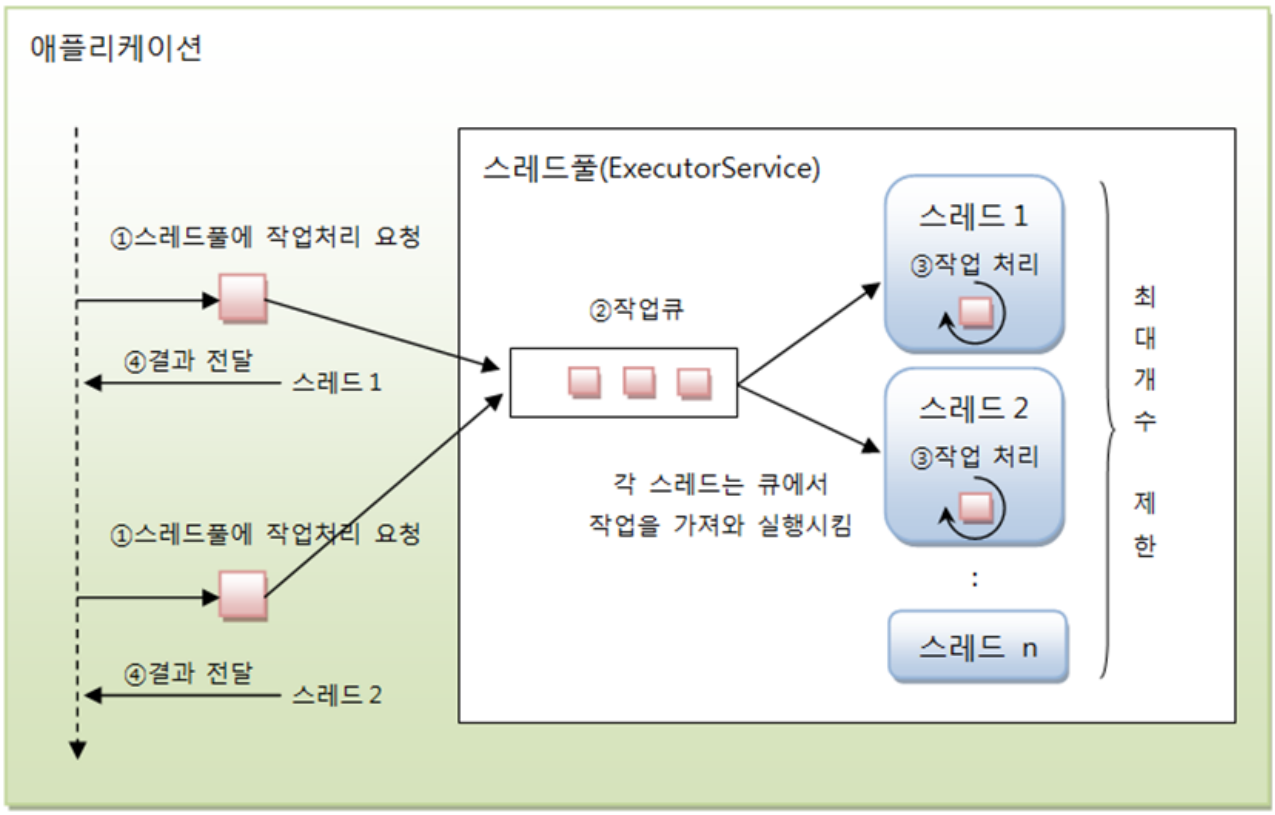
```
[ main 스레드 그룹의 list() 메소드 출력 내용 ]
java.lang.ThreadGroup[name=main,maxpri=10]          // 메인 스레드
    Thread[main,5,main]
    Thread[Monitor Ctrl-Break,5,main]
    java.lang.ThreadGroup[name=myGroup,maxpri=10]
        Thread[workThreadA,5,myGroup]
        Thread[workThreadB,5,myGroup]

[ myGroup 스레드 그룹의 interrupt() 메소드 호출 ]
workThreadA interrupted
workThreadB interrupted
workThreadA 종료됨
workThreadB 종료됨
```

> [스레드이름, 우선순위, 소속 그룹명]으로 출력

12.9 스레드 풀

: 갑작스런 병렬 작업의 폭증으로 인한 스레드의 폭증을 막으려면 스레드풀(ThreadPool)을 사용해야 한다.

- **스레드 풀** : 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐(Queue)에 들어오는 작업들을 하나씩 스레드가 맡아 처리한다.



12.9.1 스레드풀 생성 및 종료

스레드풀 생성

: ExecutorService 구현 객체는 Executors 클래스의 다음 두 가지 메소드 중 하나를 이용해서 간편하게 생성할 수 있다.

메소드명(매개변수)	초기 스레드 수	코어 스레드 수	최대 스레드 수
newCachedThreadPool()	0	0	Integer.MAX_VALUE
newFixedThreadPool(int nThreads)	0	nThreads	nThreads

- **초기 스레드 수** : ExecutorService 객체가 생성될 때 기본적으로 생성되는 스레드 수
- **코어 스레드 수** : 스레드 수가 증가된 후 사용되지 않는 스레드를 스레드풀에서 제거할 때 최소한 유지해야 할 스레드 수
- **최대 스레드 수** : 스레드풀에서 관리하는 최대 스레드 수
- **ExecutorService 구현 객체를 얻는 코드**

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

- **CPU 코어의 수만큼 최대 스레드를 사용하는 스레드풀 생성 코드**

```
ExecutorService executorService = Executors.newFixedThreadPool(
    Runtime.getRuntime().availableProcessors()
);
```

- **위의 메소드들과 다르게 코어 스레드 개수와 최대 스레드 개수를 설정하는 코드**

```
ExecutorService threadPool = new ThreadPoolExecutor(
    3,      // 코어 스레드 개수
    100,    // 최대 스레드 개수
    120L,   // 놀고 있는 시간
    TimeUnit.SECONDS,   // 놀고 있는 시간 단위
    new SynchronousQueue<Runnable>()    // 작업 큐
)
```

스레드풀 종료

: 애플리케이션을 종료하려면 스레드풀을 종료시켜 스레드들이 종료 상태가 되도록 처리해주어야 한다.

- **ExecutorService 종료에 관련한 메소드**

리턴 타입	메소드명(매개변수)	설명
void	shutdown()	현재 처리 중인 작업뿐만 아니라 작업 큐에 대기하고 있는 모든 작업을 처리한 뒤에 스레드풀을 종료시킨다.
List<Runnable>	shutdownNow()	현재 작업 처리 중인 스레드를 interrupt 해서 작업 중지를 시도하고 스레드풀을 종료시킨다. 리턴값은 작업 큐에 있는 미처리된 작업(Runnable)의 목록이다.
boolean	awaitTermination(long timeout, TimeUinit unit)	shutdown() 메소드 호출 이후, 모든 작업 처리를 timeout 시간 내에 완료하면 ture를 리턴하고, 완료하지 못하면 작업 처리 중인 스레드를 interrupt하고 false를 리턴한다.

12.9.2 작업 생성과 처리 요청

작업 생성

하나의 작업은 **Runnable** 또는 **Callable** 구현 클래스로 표현한다. Runnable과 Callable의 차이점은 작업 처리 완료 후 리턴값이 있느냐 없느냐이다.

- **Runnable과 Callable 구현 클래스 작성**

Runnable 구현 클래스	**Callable 구현 클래스**

Runnable tast = new Runnable() {
@Override
public void run() {
// 스레드가 처리할 작업 내용
}
}
	Callable<T> task = new Callable<T>() {
	@Override
	public T call() throws Exception {
	// 스레드가 처리할 작업 내용
	return T;
	}
	}