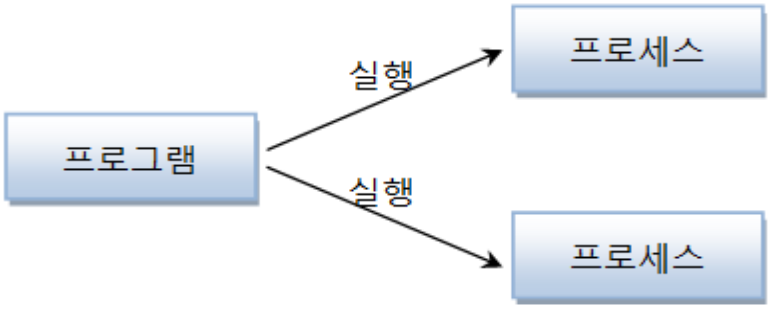
프로그래밍 언어 활용 강의안
(Multi Thread)

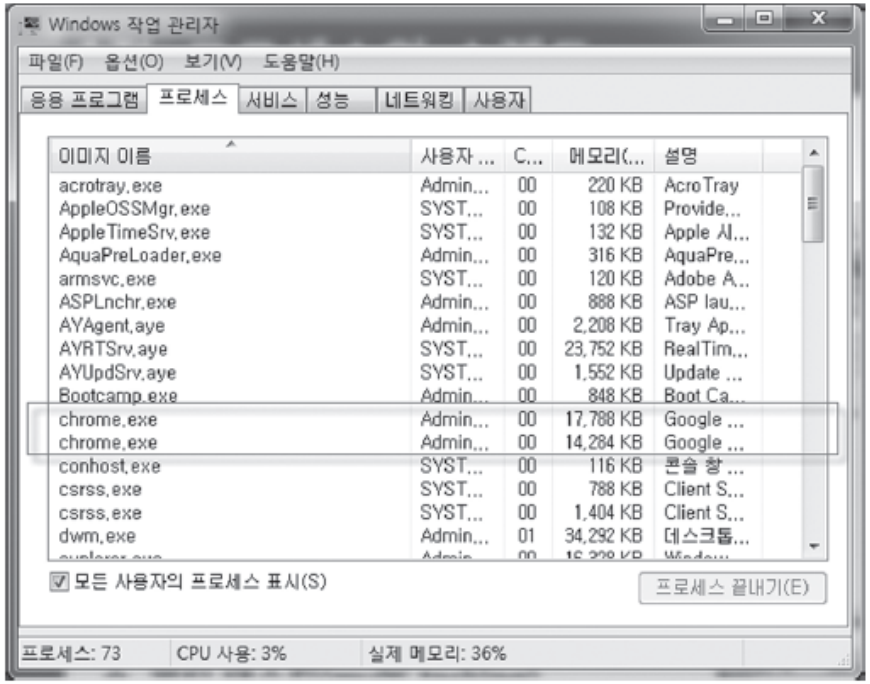
프로세스와 스레드

프로세스(process)

실행 중인 하나의 프로그램

하나의 프로그램이 다중 프로세스 만들기도 함.





프로세스와 스레드

멀티 태스킹(multi tasking)

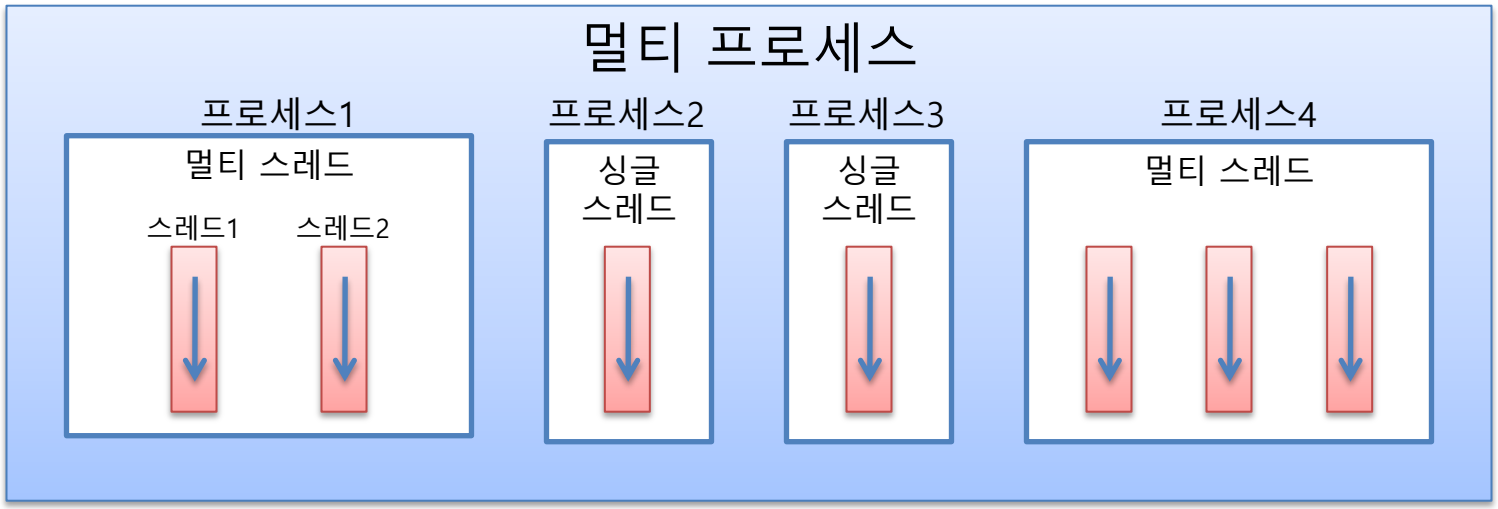
두 가지 이상의 작업을 동시에 처리하는 것

멀티 프로세스

독립적으로 프로그램들을 실행하고 여러 가지 작업 처리

멀티 스레드

한 개의 프로그램을 실행하고 내부적으로 여러 가지 작업 처리



메인(main) 스레드

모든 자바 프로그램은 메인 스레드가 main() 메소드 실행하며 시작
main() 메소드의 첫 코드부터 아래로 순차적으로 실행
실행 종료 조건
- 마지막 코드 실행
- return 문을 만나면

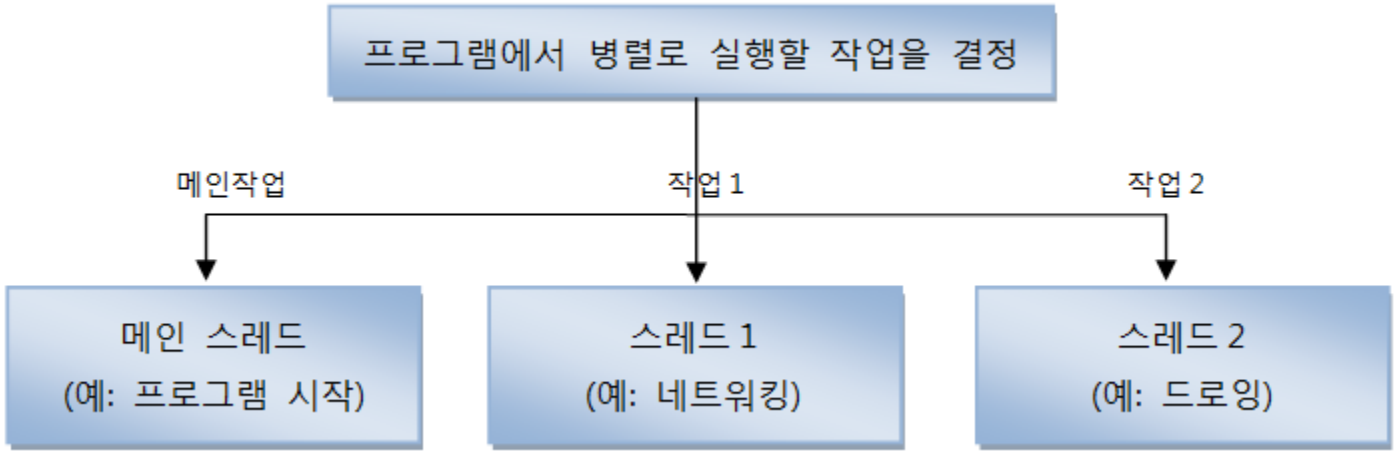
main 스레드는 작업 스레드들을 만들어 병렬로 코드를 실행
- 멀티 스레드 생성해 멀티 태스킹 수행

프로세스의 종료
- 싱글 스레드: 메인 스레드가 종료하면 프로세스도 종료
- 멀티 스레드: 실행 중인 스레드가 하나라도 있다면, 프로세스 미종료

작업 스레드 생성과 실행

멀티 스레드로 실행하는 어플리케이션 개발

몇 개의 작업을 병렬로 실행할지 결정하는 것이 선행되어야 함.



작업 스레드 생성 방법

- Thread 클래스로부터 직접 생성
  - Runnable을 매개값으로 갖는 생성자 호출
- Thread 하위 클래스로부터 생성
  - Thread 클래스 상속 후 run 메소드 재정의 해 스레드가 실행할 코드 작성

작업 스레드 생성과 실행

스레드의 이름

메인 스레드 이름: main
작업 스레드 이름 (자동 설정) : Thread-n

```
thread.getName();
```

작업 스레드 이름 변경

```
thread.setName("스레드  이름");
```

코드 실행하는 현재 스레드 객체의 참조 얻기

```
Thread thread = Thread.currentThread();
```

스레드 우선순위

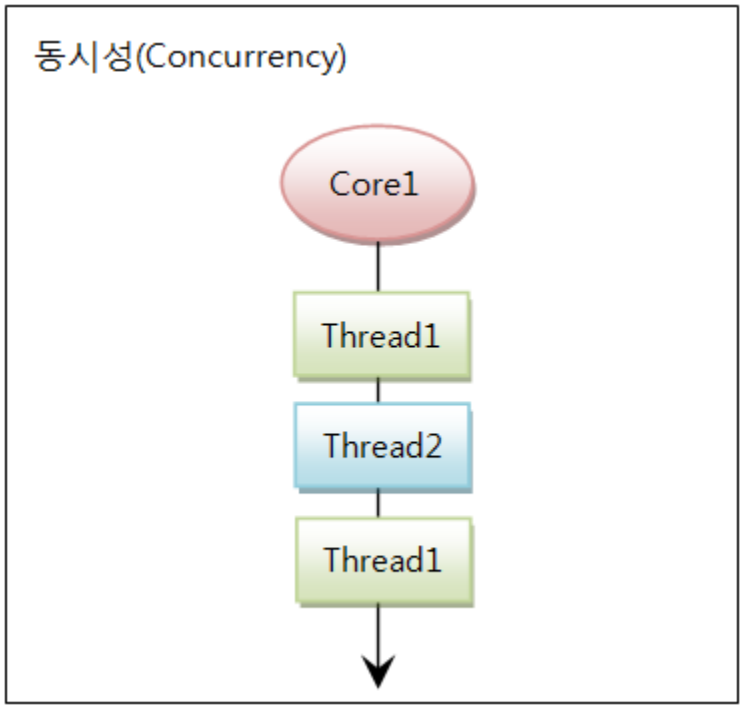
동시성과 병렬성

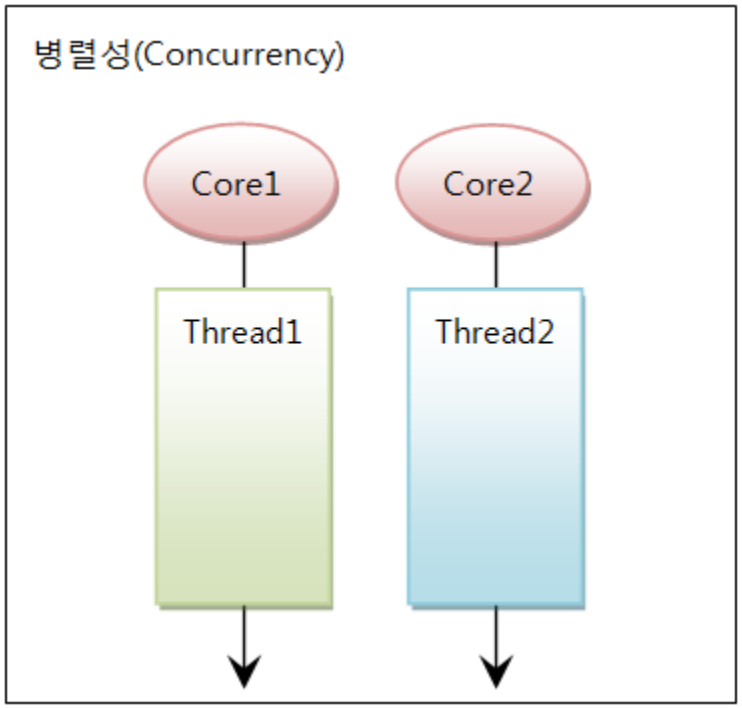
동시성

멀티 작업 위해 하나의 코어에서 멀티 스레드가 번갈아 가며 실행하는 성질

병렬성

멀티 작업을 위해 멀티 코어에서 개별 스레드를 동시에 실행하는 성질





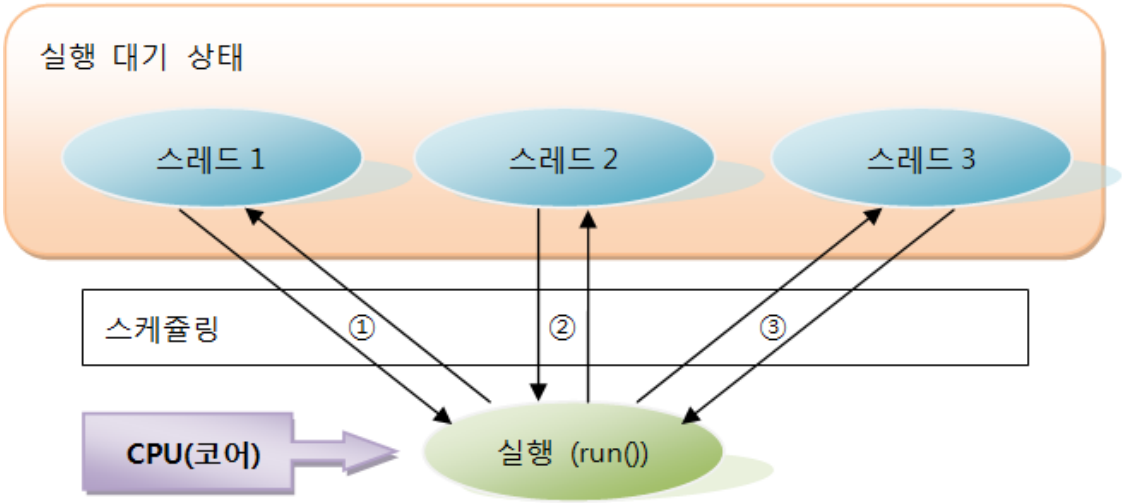
스레드 우선순위

스레드 스케줄링

스레드의 개수가 코어의 수보다 많을 경우

스레드를 어떤 순서로 동시성으로 실행할 것인가 결정→ 스레드 스케줄링

스케줄링 의해 스레드들은 번갈아 가며 run() 메소드를 조금씩 실행



스레드 우선순위

자바의 스레드 스케줄링

우선 순위(Priority) 방식과 순환 할당(Round-Robin) 방식 사용

우선 순위 방식 (코드로 제어 가능)

우선 순위가 높은 스레드가 실행 상태를 더 많이 가지도록 스케줄링

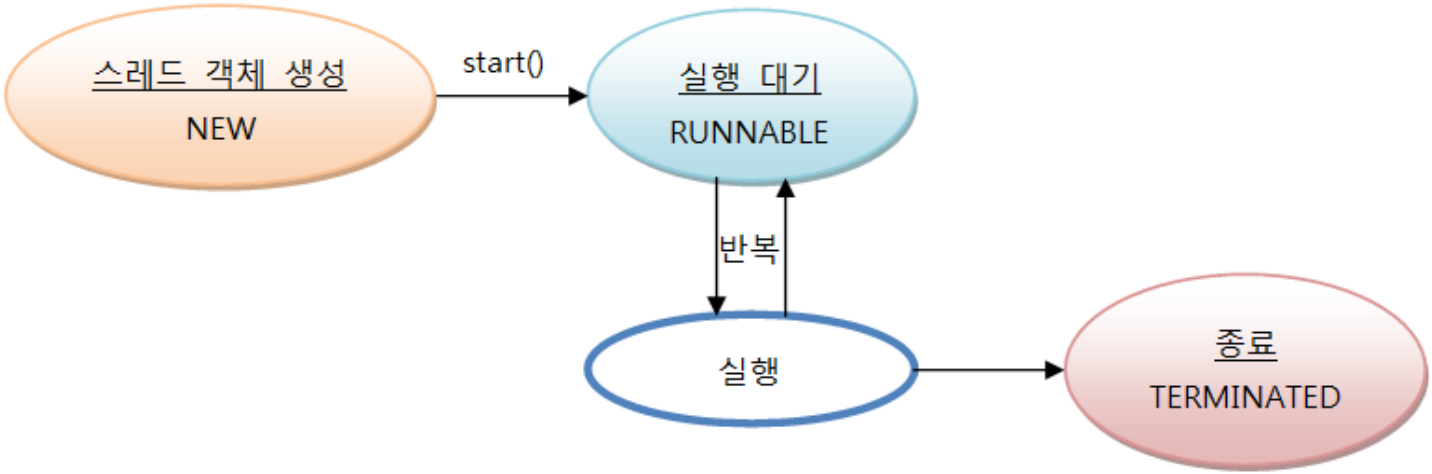
1~10까지 값을 가질 수 있으며 기본은 5

순환 할당 방식 (코드로 제어할 수 없음)

시간 할당량(Time Slice) 정해서 하나의 스레드를 정해진 시간만큼 실행

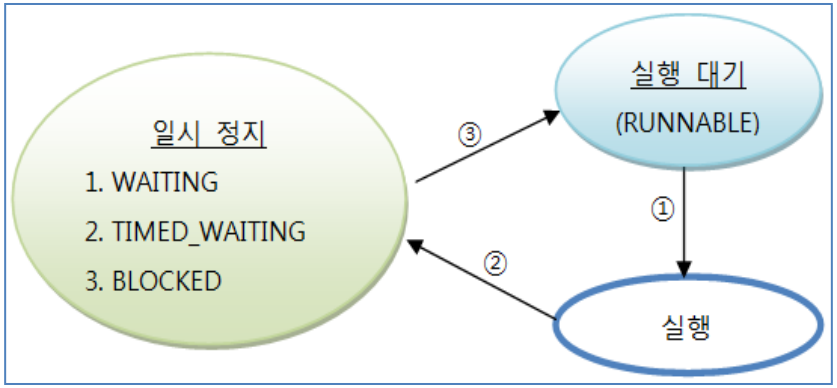
스레드의 상태

스레드의 일반적인 상태



스레드의 상태

스레드에 일시 정지 상태 도입한 경우

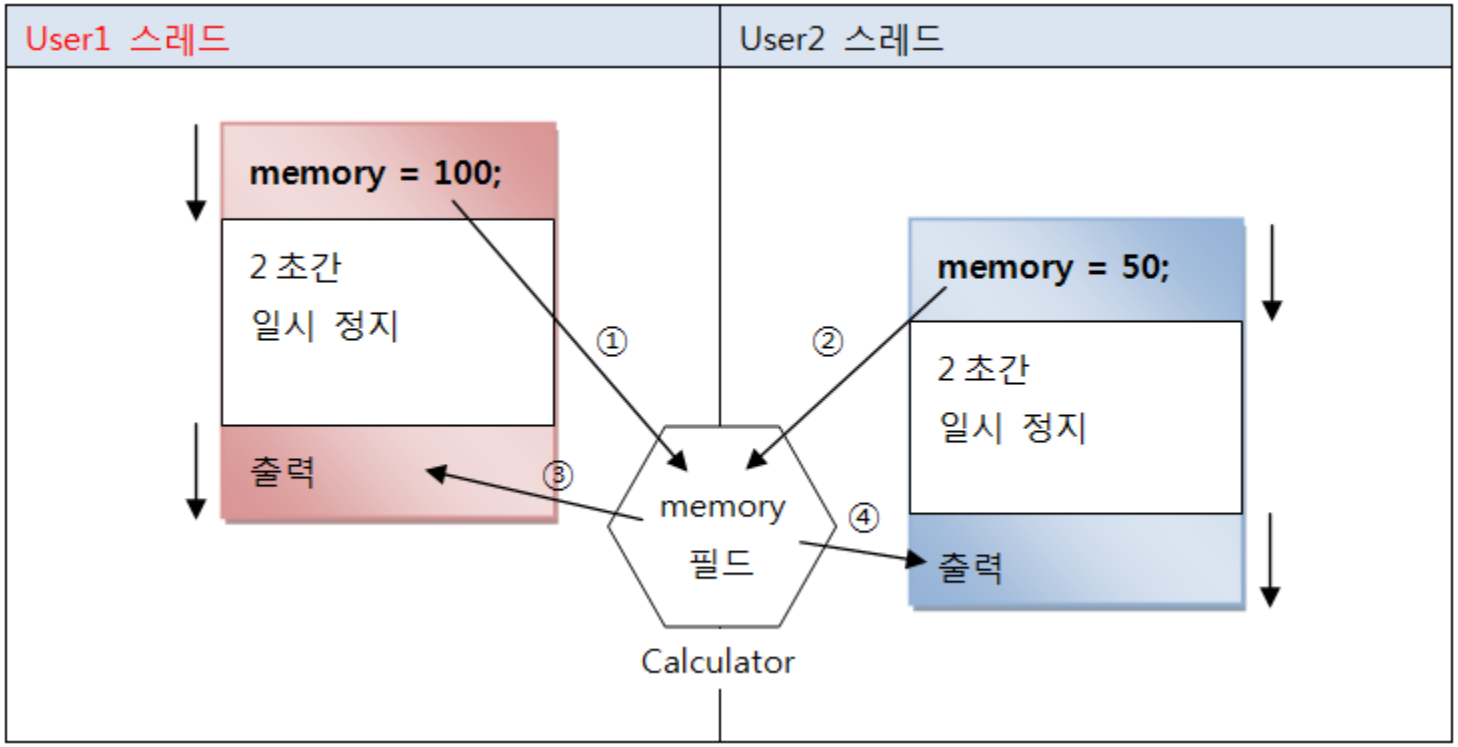


상태	열거 상수	설명
객체 생성	NEW	스레드 객체가 생성, 아직 start() 메소드가 호출되지 않은 상태
실행 대기	RUNNABLE	실행 상태로 언제든지 갈 수 있는 상태
일시 정지	BLOCKED	사용코저하는 객체의 락이 풀릴 때까지 기다리는 상태
	WAITING	다른 스레드가 통지할 때까지 기다리는 상태
	TIMED_WAITING	주어진 시간 동안 기다리는 상태
종료	TERMINATED	실행을 마친 상태

동기화 메소드와 동기화 블럭

공유 객체를 사용할 때의 주의할 점

멀티 스레드가 하나의 객체를 공유해서 생기는 오류



동기화 메소드와 동기화 블럭

동기화 메소드 및 동기화 블록 – synchronized

단 하나의 스레드만 실행할 수 있는 메소드 또는 블록
다른 스레드는 메소드나 블록이 실행이 끝날 때까지 대기해야

동기화 메소드

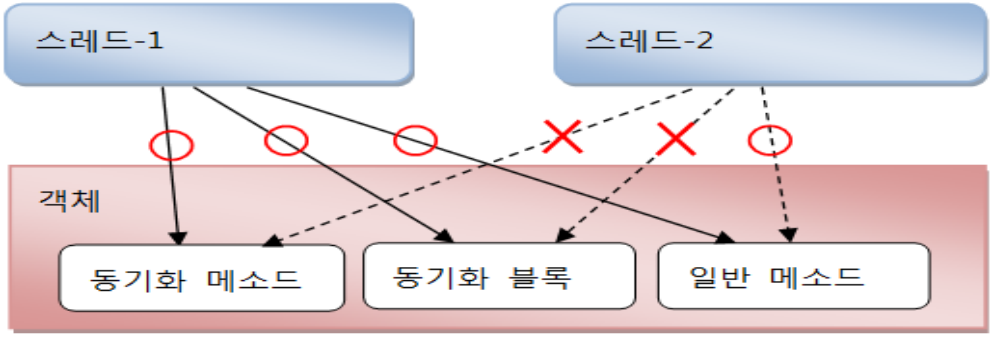
```
public synchronized void method() {
    임계 영역;  //단 하나의 스레드만 실행
}
```

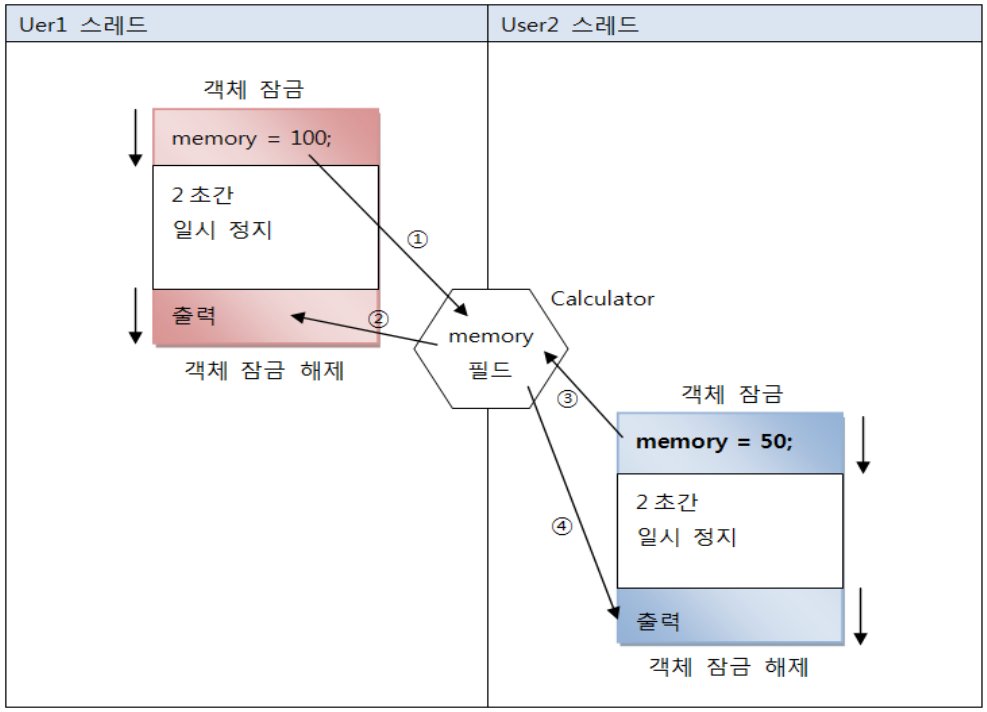
동기화 블록

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

동기화 메소드와 동기화 블럭

동기화 메소드 및 동기화 블록

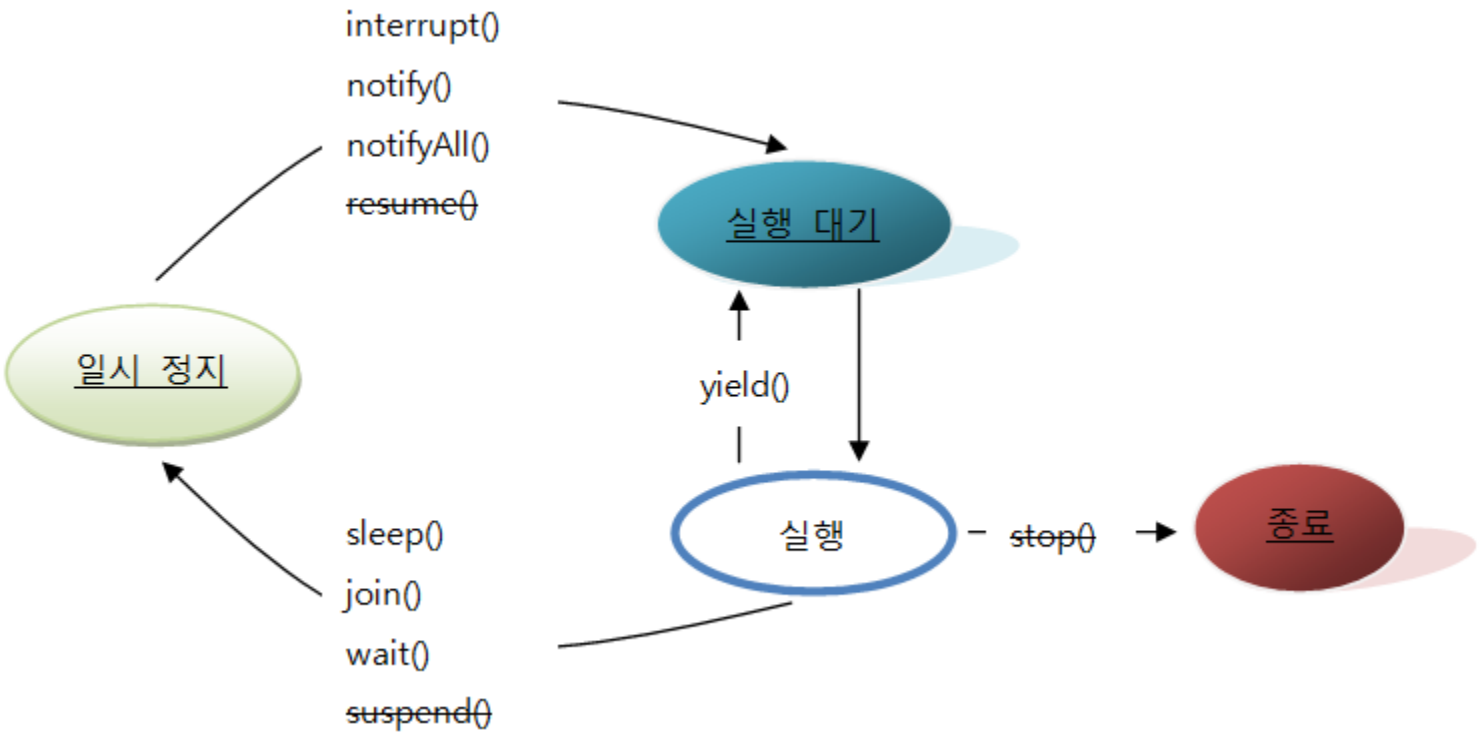




스레드의 상태 제어

상태 제어

실행 중인 스레드의 상태를 변경하는 것
상태 변화를 가져오는 메소드의 종류 (취소선 가진 메소드는 사용 X)



취소선을 가진 메소드는 스레드의 안정성을 헤친다는 이유로 더 이상 사용하지 않도록 권장하고 있는 Deprecated 메소드 들임.

스레드의 상태 제어

상태 제어 메소드

메소드	설명
interrupt()	일시 정지 상태의 스레드에서 InterruptedException 예외를 발생시켜, 예외 처리 코드(catch)에서 실행대기 상태로 가거나 종료 상태로 갈 수 있도록 한다.
notify() notifyAll()	Object객체가 가진 메소드로 동기화 블록 내에서 wait() 메소드에 의해 일시 정지 상태에 있는 스레드를 실행 대기 상태로 만든다.
~~resume()~~	suspend() 메소드에 의해 일시 정지 상태에 있는 스레드를 실행 대기 상태로 만든다. -Deprecated(대신 notify(), notifyAll() 사용)
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 naanos)	join() 메소드를 호출한 스레드는 일시정지 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.
~~suspend()~~	스레드를 일시 정지 상태로 만든다. resume() 메소드를 호출하면 다시 실행 대기 상태가 된다. – Deprecated(대신 wait() 사용)
yield()	실행 중에 우선순위가 동일한 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.
~~stop()~~	스레드를 즉시 종료시킨다. – Deprecated

스레드의 상태 제어

주어진 시간 동안 일시 정지 - sleep()

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

얼마 동안 일시 정지 상태로 있을 것인지 밀리 세컨드(1/1000) 단위로 지정

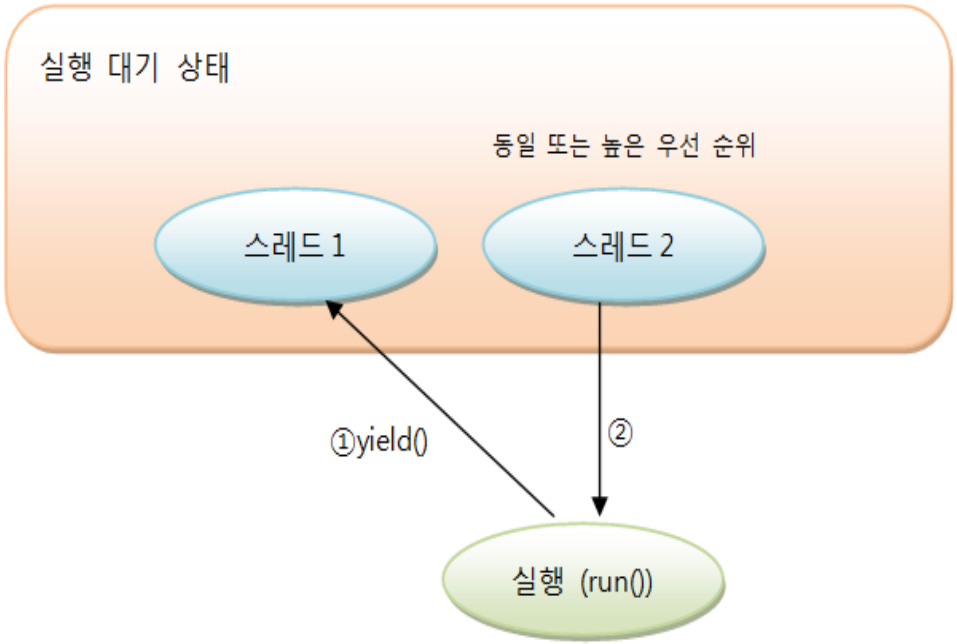
일시 정지 상태에서 interrupt() 메소드 호출

InterruptedException 발생

스레드의 상태 제어

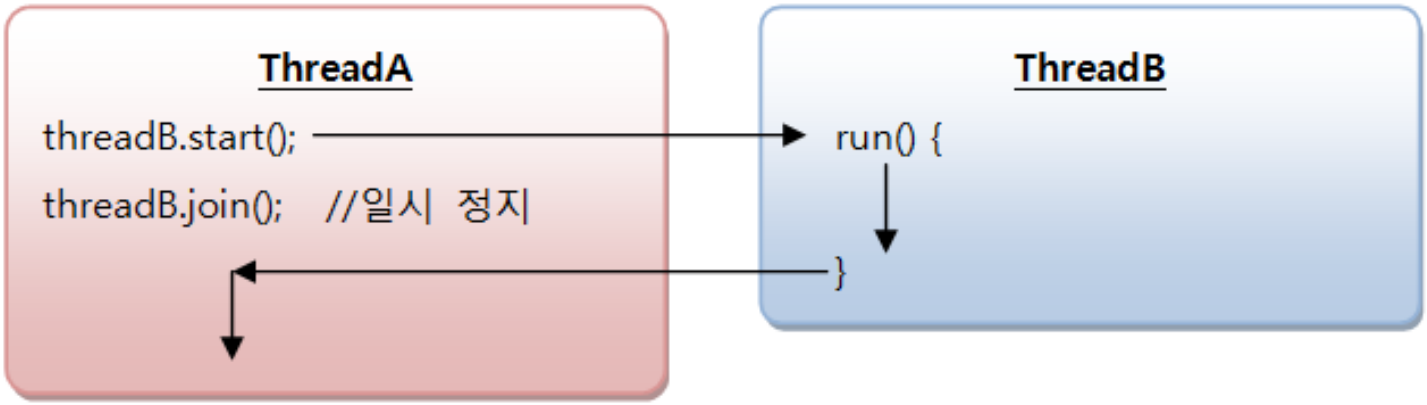
다른 스레드에게 실행 양보 - yield()

Ex) 무의미한 반복하는 스레드일 경우



스레드의 상태 제어

다른 스레드의 종료를 기다림 - join()

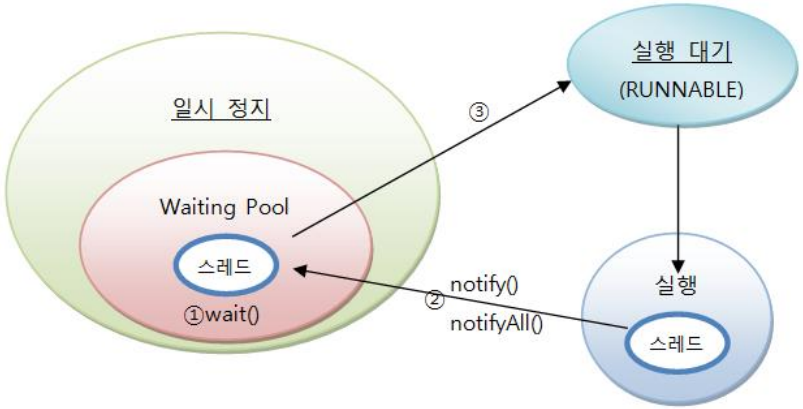


계산 작업을 하는 스레드가 모든 계산 작업 마쳤을 때, 결과값을 받아 이용하는 경우 주로 사용

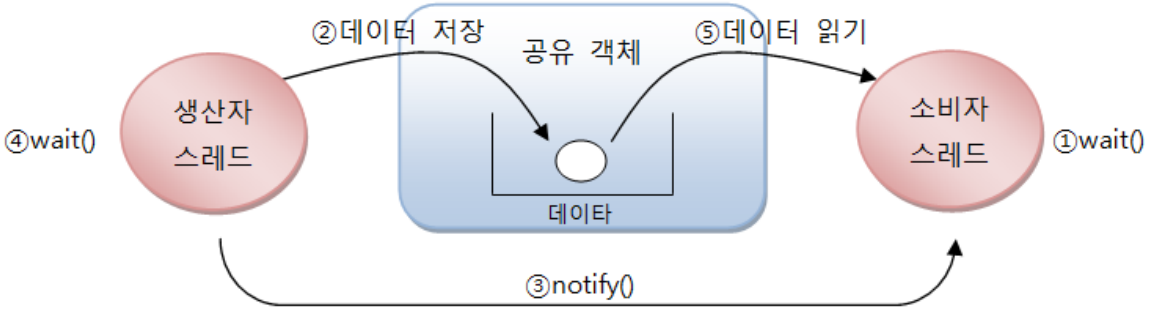
스레드의 상태 제어

스레드간 협업 – wait(), notify(), notifyAll()

동기화 메소드 또는 블록에서만 호출 가능한 Object의 메소드



두 개의 스레드가 교대로 번갈아 가며 실행해야 할 경우 주로 사용



스레드의 상태 제어

스레드의 안전한 종료 – stop 플래그, interrupt()

경우에 따라 실행 중인 스레드 즉시 종료해야 할 필요 있을 때 사용

stop() 메소드 사용시

스레드 즉시 종료 되는 편리함

Deprecated - 사용 중이던 자원들이 불안전한 상태로 남겨짐

안전한 종료 위해 stop 플래그 이용하는 방법

stop 플래그로 메소드의 정상 종료 유도

```
public class XXXThread extends Thread {
  private boolean stop;  //stop 플래그 필드

  public void run() {
    while( !stop ) {  ●---------- stop 이 true 가 되면 run() 이 종료된다.
      스레드가 반복 실행하는 코드;
    }
    //스레드가 사용한 자원 정리
  }
}
```

스레드의 상태 제어

스레드의 안전한 종료

- interrupt() 메소드를 이용하는 방법
  - 스레드가 일시 정지 상태일 경우 InterruptedException 발생 시킴
  - 실행대기 또는 실행상태에서는 InterruptedException 발생하지 않음
  - 일시 정지 상태로 만들지 않고 while문 빠져 나오는 방법으로도 쓰임

데몬 스레드

데몬(daemon) 스레드

주 스레드의 작업 돕는 보조적인 역할 수행하는 스레드

주 스레드가 종료되면 데몬 스레드는 강제적으로 자동 종료

워드프로세서의 자동저장, 미디어플레이어의 동영상 및 음악 재생, GC

스레드를 데몬 스레드로 만들기

주 스레드가 데몬이 될 스레드의 setDaemon(true) 호출

반드시 start() 메소드 호출 전에 setDaemon(true) 호출

그렇지 않으면 IllegalThreadStateException이 발생

현재 실행중인 스레드가 데몬 스레드인지 구별법

isDaemon() 메소드의 리턴값 조사 – true면 데몬 스레드

스레드 그룹

스레드 그룹

관련된 스레드 묶어 관리 목적으로 이용
스레드 그룹은 계층적으로 하위 스레드 그룹 가질 수 있음

자동 생성되는 스레드 그룹
- system 그룹: JVM 운영에 필요한 스레드들 포함
- system/main 그룹: 메인 스레드 포함

스레드는 반드시 하나의 스레드 그룹에 포함
- 기본적으로 자신을 생성한 스레드와 같은 스레드 그룹
- 스레드 그룹에 포함시키지 않으면 기본적으로 system/main 그룹

스레드 그룹 이름 얻기

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

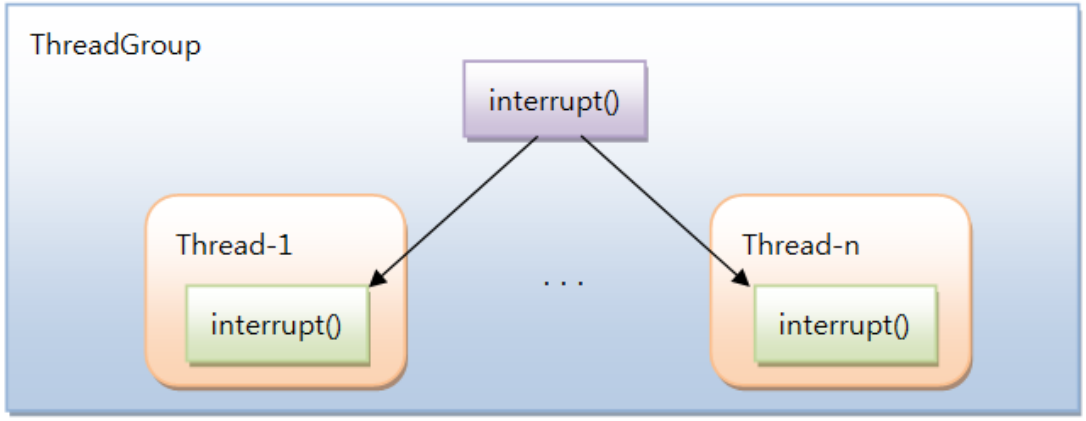
스레드 그룹 생성

```
ThreadGroup tg = new ThreadGroup(String name);
ThreadGroup tg = new ThreadGroup(ThreadGroup parent, String name);
```

부모(parent) 그룹 지정하지 않으면?
현재 스레드 속한 그룹의 하위 그룹으로 생성

스레드 그룹의 일괄 interrupt()

스레드그룹의 interrupt() 호출 시 소속된 모든 스레드의 interrupt()호출



스레드 풀

스레드 폭증으로 일어나는 현상

- 병렬 작업 처리가 많아지면 스레드 개수 증가
- 스레드 생성과 스케줄링으로 인해 CPU가 바빠짐
- 메모리 사용량이 늘어남
- 애플리케이션의 성능 급격히 저하

스레드 풀(Thread Pool)

- 작업 처리에 사용되는 스레드를 제한된 개수만큼 미리 생성
- 작업 큐(Queue)에 들어오는 작업들을 하나씩 스레드가 맡아 처리
- 작업 처리가 끝난 스레드는 작업 결과를 애플리케이션으로 전달
- 스레드는 다시 작업 큐에서 새로운 작업을 가져와 처리

스레드 풀

ExecutorService 인터페이스와 Executors 클래스

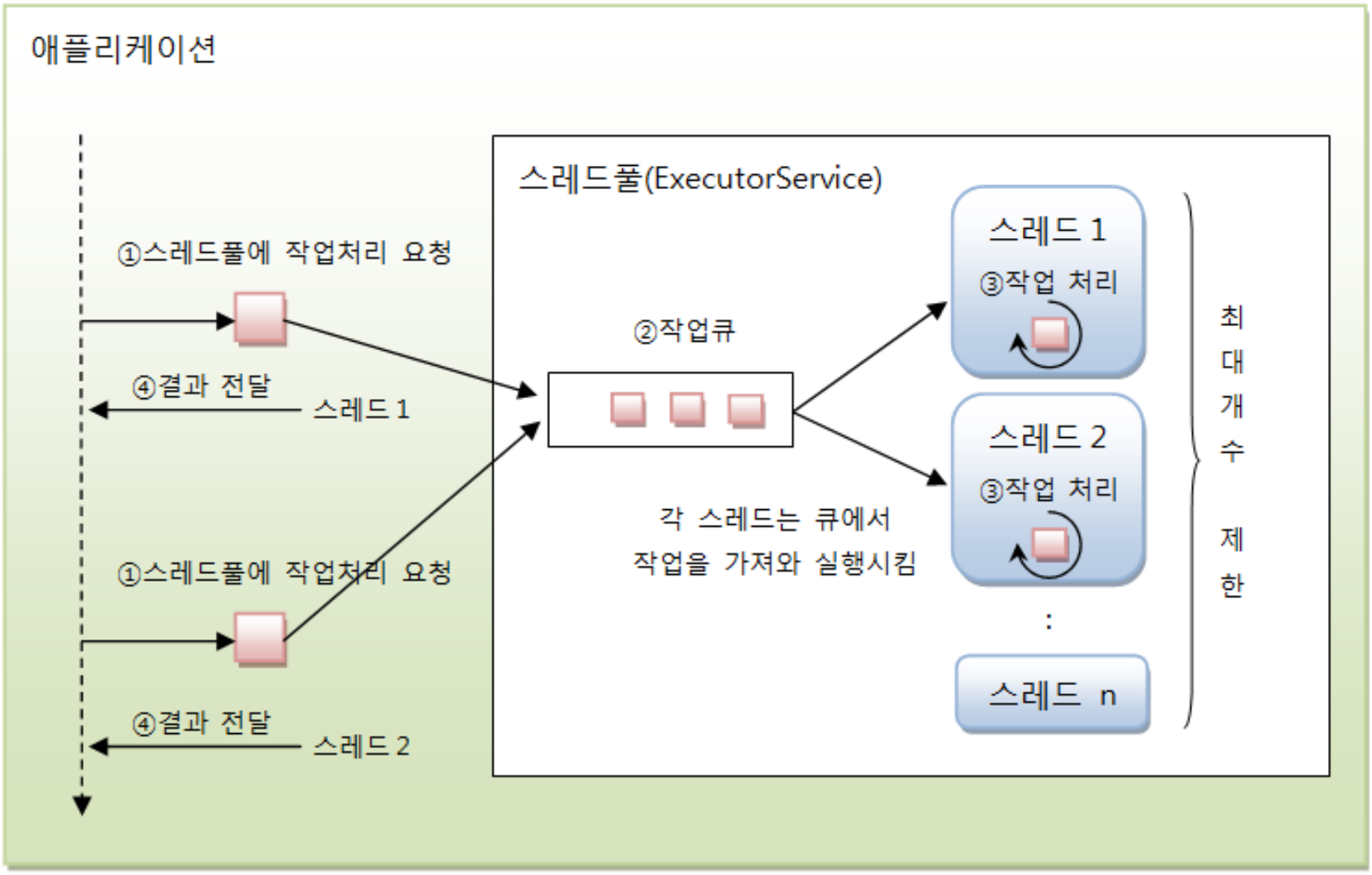
스레드 풀 생성, 사용 - java.util.concurrent 패키지에서 제공

Executors의 정적 메소드 이용 -ExecutorService 구현 객체 생성

스레드 풀 = ExecutorService 객체

스레드 풀

ExecutorService 동작 원리



스레드 풀 생성

다음 두 가지 메소드 중 하나로 간편 생성

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

newCachedThreadPool()

- int 값이 가질 수 있는 최대 값만큼 스레드 추가, 운영체제의 상황에 따라 달라짐
- 1개 이상의 스레드가 추가되었을 경우
- 60초 동안 추가된 스레드가 아무 작업을 하지 않으면
- 추가된 스레드를 종료하고 풀에서 제거

newFixedThreadPool(int nThreads)

- 코어 스레드 개수와 최대 스레드 개수가 매개값으로 준 nThread
- 스레드가 작업 처리하지 않고 놀고 있더라도 스레드 개수가 줄지 않음

스레드 풀 종료

스레드 풀의 스레드는 기본적으로 데몬 스레드가 아님

main 스레드 종료되더라도 스레드 풀 스레드는 작업 처리 위해 계속 실행

애플리케이션은 종료되지 않음

스레드 풀 종료해 모든 스레드 종료시켜야

스레드 풀 종료 메소드

리턴타입	메소드명(매개변수)	설명
void	shutdown()	현재 처리 중인 작업뿐만 아니라 작업큐에 대기하고 있는 모든 작업을 처리한 뒤에 스레드풀을 종료시킨다.
List<Runnable>	shutdownNow()	현재 작업 처리 중인 스레드를 interrupt 해서 작업 중지를 시도하고 스레드풀을 종료시킨다. 리턴값은 작업큐에있는 미처리된 작업(Runnable)의 목록이다.
boolean	awaitTermination(long timeout, TimeUnit unit)	shutdown() 메소드 호출 이후, 모든 작업 처리를 timeout 시간 내에 완료하면 true 를 리턴하고, 완료하지 못하면 작업 처리 중인 스레드를 interrupt 하고 false 를 리턴한다.

스레드 풀

작업 생성

하나의 작업은 Runnable 또는 Callable 객체로 표현

Runnable과 Callable의 차이점

작업 처리 완료 후 리턴값이 있느냐 없느냐

Runnable 구현 클래스	Callable 구현 클래스

Runnable task = 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;
	  }
	}

스레드 풀

작업 처리 요청

ExecutorService의 작업 큐에 Runnable 나 Callable 객체 넣음
작업 처리 요청 위해 ExecutorService는 두 가지 메소드 제공

리턴타입	메소드명(매개변수)	설명
void	execute(Runnable command)	- Runnable 을 작업큐에 저장 - 작업 처리 결과를 받지 못함
Future<?>	submit(Runnable task)	- Runnable 또는 Callable 을 작업큐에 저장
Future<V>	submit(Runnable task, V result)	- 리턴된 Future 를 통해 작업 처리 결과 얻을
Future<V>	submit(Callable<V> task)	수 있음

작업 처리 도중 예외 발생할 경우

execute()

스레드 종료 후 해당 스레드 제거
스레드 풀은 다른 작업 처리를 위해 새로운 스레드 생성

submit()

스레드가 종료되지 않고 다음 작업 위해 재사용

블로킹 방식의 작업 완료 통보 받기

작업이 완료될 때까지 기다렸다가 (지연 되었다가) 메소드 실행

리턴타입	메소드명(매개변수)	설명
Future<?>	submit(Runnable task)	- Runnable 또는 Callable을 작업큐에 저장 - 리턴된 Future를 통해 작업 처리 결과 얻음
Future<V>	submit(Runnable task, V result)	
Future<V>	submit(Callable<V> task)	

Future 객체

작업 결과가 아니라 지연 완료(pending completion) 객체

작업이 완료될까지 기다렸다가 최종 결과를 얻기 위해 get() 메소드 사용

UI 변경과 같은 스레드에 사용 불가

리턴타입	메소드명(매개변수)	설명
V	get()	작업이 완료될 때까지 블로킹되었다가 처리 결과 V를 리턴
V	get(long timeout, TimeUnit unit)	timeout 시간동안 작업이 완료되면 결과 V를 리턴하지만, 작업이 완료되지 않으면 TimeoutException을 발생시킴

Future 객체에 속한 다른 메소드

리턴타입	메소드명(매개변수)	설명
boolean	cancel(boolean mayInterruptIfRunning)	작업 처리가 진행중일 경우 취소 시킴
boolean	isCancelled()	작업이 취소되었는지 여부
boolean	isDone()	작업 처리가 완료되었는지 여부

작업완료 통보 방식에 따른 구분

리턴값이 없는 작업 완료 통보

Runnable 객체로 생성해 처리

리턴값이 있는 작업 완료 통보

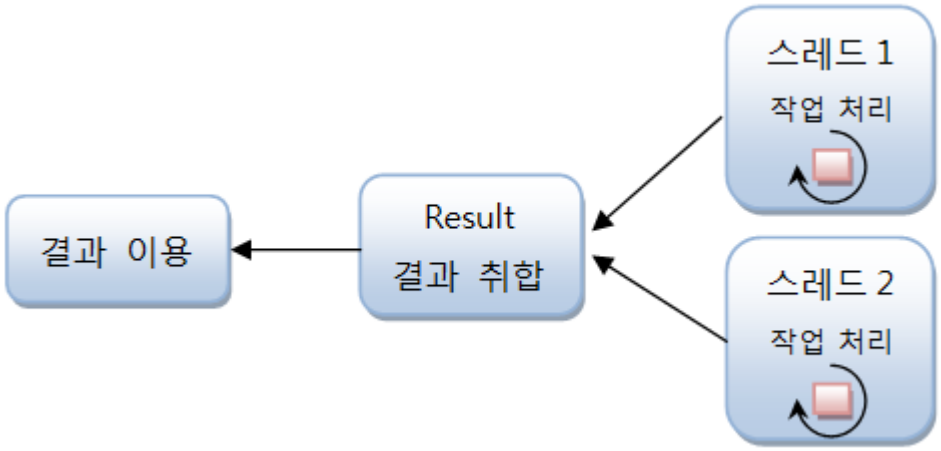
작업 객체를 Callable 로 생성

스레드 풀

작업완료 통보 방식에 따른 구분

작업 처리 결과를 외부 객체에 저장

보통은 두 개 이상의 스레드 작업을 취합할 목적으로 사용

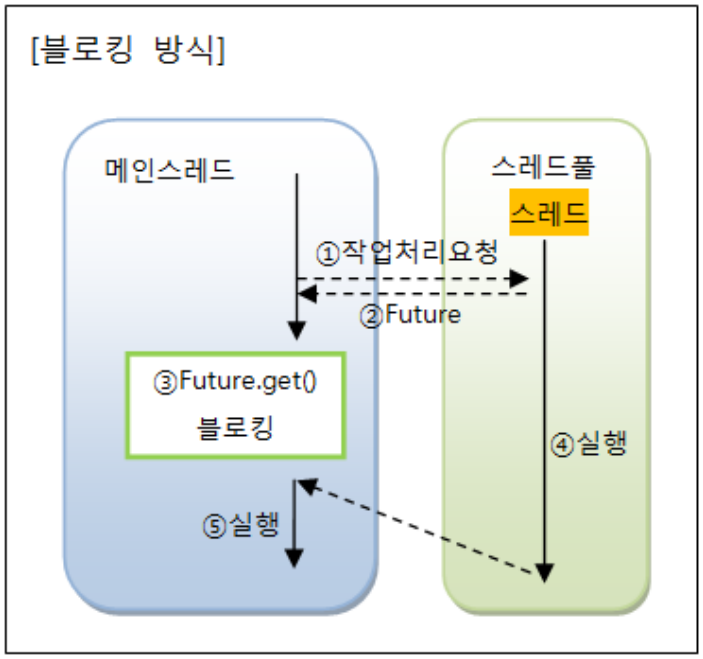


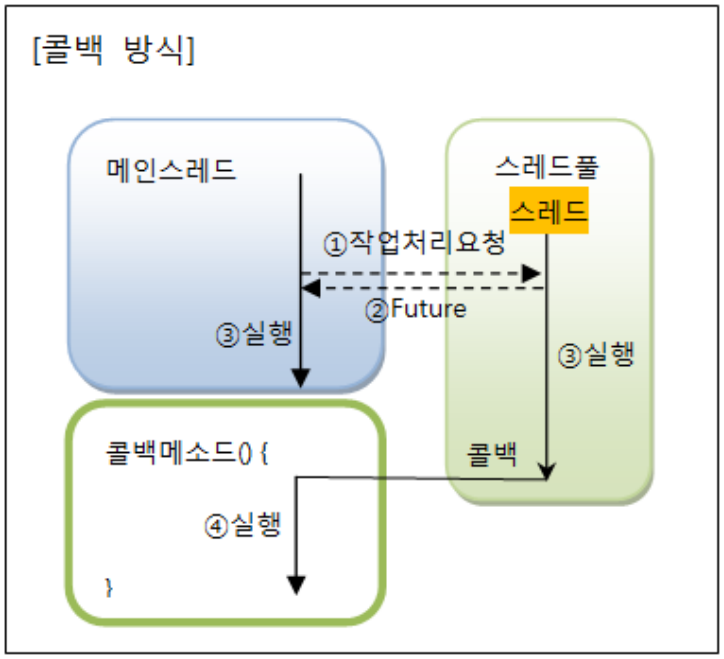
콜 백 방식의 작업 완료 통보 받기

콜 백의 개념

애플리케이션이 스레드에게 작업 처리를 요청한 후, 다른 기능 수행할 동안 스레드가 작업을 완료하면 애플리케이션의 메소드를 자동 실행하는 기법 이때 자동 실행되는 메소드를 콜백 메소드

작업 완료 통보 얻기: 블로킹 vs 콜 백 이해





스레드 풀

작업완료 통보 방식에 따른 구분

작업 완료 순으로 통보 받기

작업 요청 순서대로 작업 처리가 완료되는 것은 아님

작업의 양과 스레드 스케줄링에 따라 먼저 요청한 작업이 나중에 완료되는 경우도 발생

여러 개의 작업들이 순차적으로 처리될 필요성이 없고,

처리 결과도 순차적으로 이용할 필요가 없다면

→ 작업 처리가 완료된 것부터 결과를 얻어 이용하는 것이 좋음

리턴타입	메소드명(매개변수)	설명
Future<V>	poll()	완료된 작업의 Future 를 가져옴. 완료된 작업이 없다면 즉시 null 을 리턴함
Future<V>	poll(long timeout, TimeUnit unit)	완료된 작업의 Future 를 가져옴. 완료된 작업이 없다면 timeout 까지 블로킹됨.
Future<V>	take()	완료된 작업의 Future 를 가져옴. 완료된 작업이 없다면 있을 때까지 블로킹됨.
Future<V>	submit(Callable<V> task)	스레드풀에 Callable 작업 처리 요청
Future<V>	submit(Runnable task, V result)	스레드풀에 Runnable 작업 처리 요청