

2025년 상반기 K-디지털 트레이닝

멀티 스레드

[KB] IT's Your Life

1 멀티 스레드 개념

• 멀티 프로세스와 멀티 스레드

- 프로세스: 운영체제는 실행 중인 프로그램을 관리
- 멀티 태스킹: 두 가지 이상의 작업을 동시에 처리하는 것
- 스레드: 코드의 실행 흐름
- 멀티 스레드: 두 개의 코드 실행 흐름. 두 가지 이상의 작업을 처리
- 멀티 프로세스 = 프로그램 단위의 멀티 태스킹 / 멀티 스레드 = 프로그램 내부에서의 멀티 태스킹



• 메인 스레드

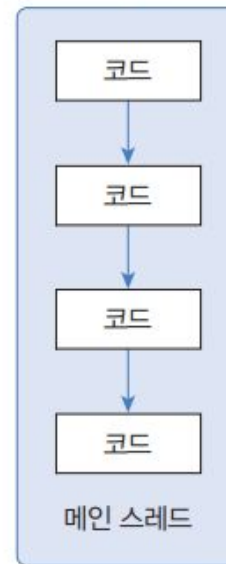
- 메인 스레드는 `main()` 메소드의 첫 코드부터 순차적으로 실행
- `main()` 메소드의 마지막 코드를 실행하거나 `return` 문을 만나면 실행을 종료
- 메인 스레드는 추가 작업 스레드들을 만들어서 실행시킬 수 있음
- 메인 스레드가 작업 스레드보다 먼저 종료되더라도 작업 스레드가 계속 실행 중이라면 프로세스는 종료되지 않음

```
public static void main(String[] args) {  
    String data = null;  
    if(...) {  
    }  
    while(...) {  
    }  
    System.out.println("...");  
}
```

코드의 실행 흐름 → 메인 스레드

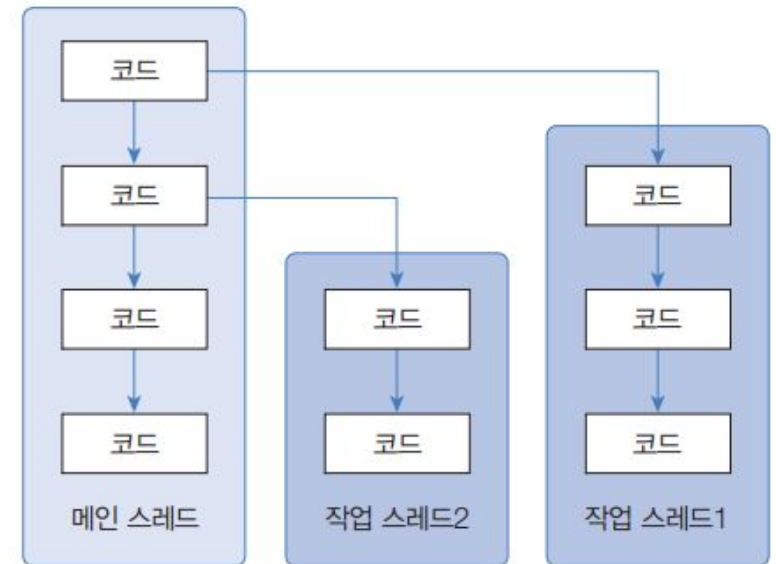
싱글 스레드 애플리케이션

프로세스



멀티 스레드 애플리케이션

프로세스



- 작업 스레드

- 멀티 스레드 프로그램을 개발 시 먼저 몇 개의 작업을 병렬로 실행할지 결정하고 각 작업별로 스레드를 생성



- Thread 클래스로 직접 생성
 - Runnable 구현 객체를 매개값으로 갖는 생성자를 호출


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

```
class Task implements Runnable {  
    @Override  
    public void run() {  
        //스레드가 실행할 코드  
    }  
}
```

- Thread 클래스로 직접 생성

- Runnable 구현 객체를 매개값으로 갖는 생성자를 호출

```
Runnable task= new Task();  
Thread thread = new Thread(task);
```

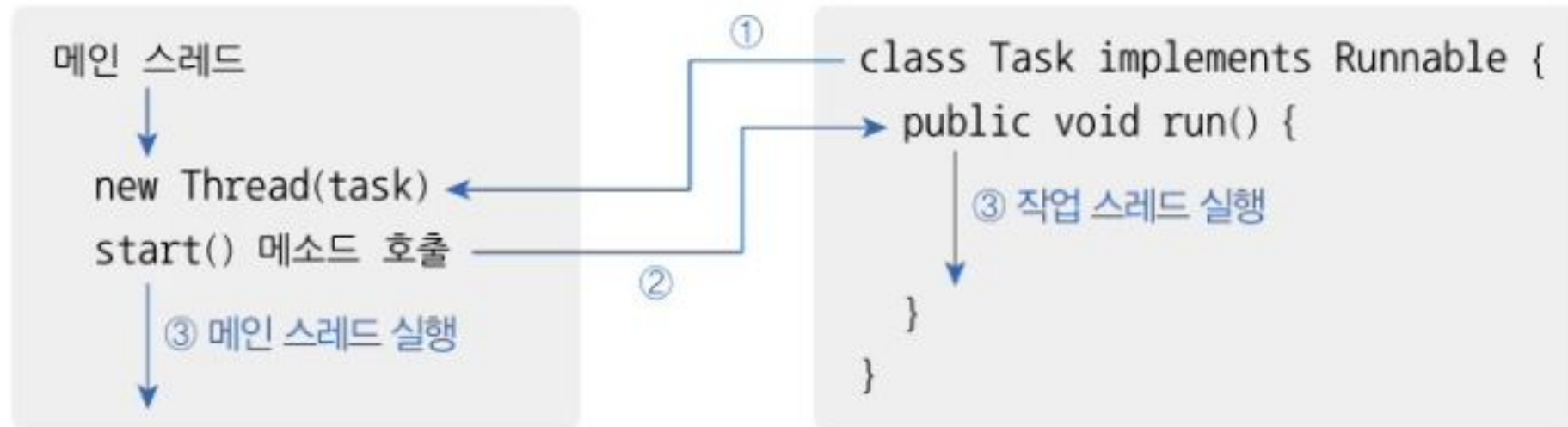


```
Thread thread = new Thread( new Runnable() {  
    @Override  
    public void run() {  
        //스레드가 실행할 코드  
    }  
});
```

```
thread.start();
```

- Thread 클래스로 직접 생성

- Runnable 구현 객체를 매개값으로 갖는 생성자를 호출



- **BeepPrintExample.java**

```
package ch14.sec03.exam01;

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();          // 비프음 발생
            try { Thread.sleep(500); } catch(Exception e) {}           // 0.5초간 일시 정지
        }

        for(int i=0; i<5; i++) {
            System.out.println("띵");
            try { Thread.sleep(500); } catch(Exception e) {}           // 0.5초간 일시 정지
        }
    }
}
```

띵 띵 띵 띵 띵


- **BeepPrintExample.java**

```
package ch14.sec03.exam02;

import java.awt.Toolkit;

public class BeepPrintExample {
    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() { // 작업 스레드 생성
            @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(); // 작업 스레드 실행
```



- BeepPrintExample.java

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

0F 0F 0F 0F 0F 0F

- Thread 자식 클래스로 생성

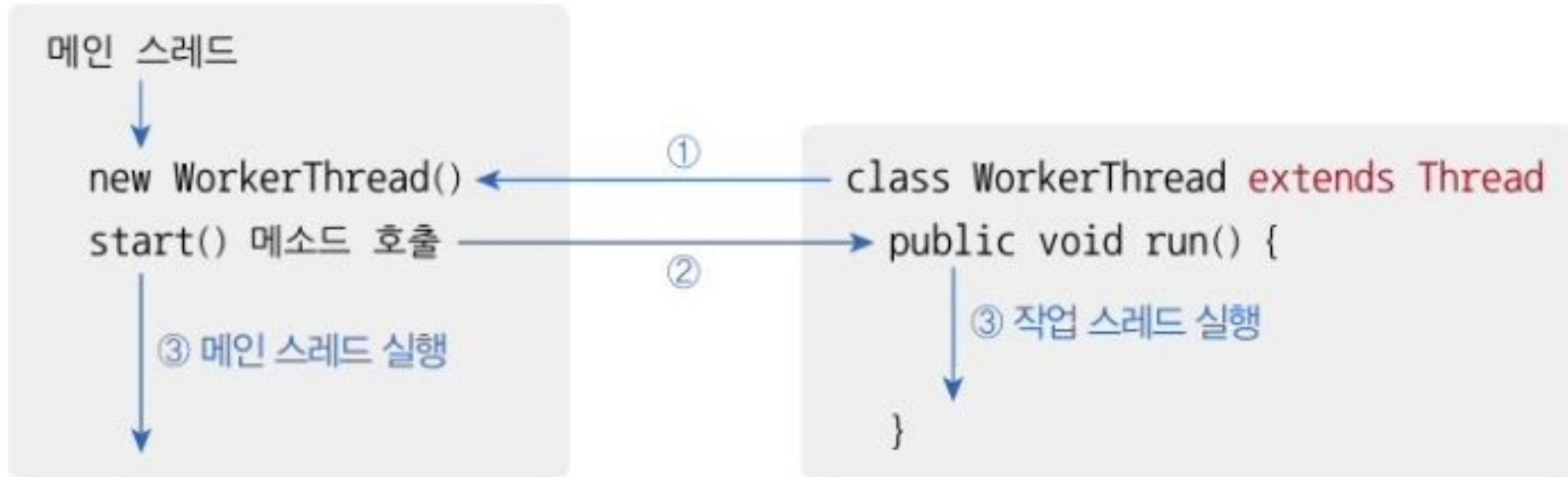
- Thread 클래스를 상속한 다음 run() 메소드를 재정의
 - 스레드가 실행할 코드를 작성
- 객체를 생성 혹은 Thread 익명 자식 객체를 사용 가능

```
public class WorkerThread extends Thread {  
    @Override  
    public void run() {  
        //스레드가 실행할 코드  
    }  
}
```

```
//스레드 객체 생성  
Thread thread = new WorkerThread();  
thread.start();
```

```
Thread thread = new Thread() {  
    @Override  
    public void run() {  
        //스레드가 실행할 코드  
    }  
};  
thread.start();
```

- Thread 자식 클래스로 생성



```
Thread thread = new Thread() {
    @Override
    public void run() {
        //스레드가 실행할 코드
    }
};
thread.start();
```

- BeepPrintExample.java

```
package ch14.sec03.exam03;

import java.awt.Toolkit;

public class BeepPrintExample {
    public static void main(String[] args) {
        Thread thread = new Thread() {
            @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();
    }
}
```

- BeepPrintExample.java

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

0F 0F 0F 0F 0F 0F

- 작업 스레드의 이름

- 작업 스레드 이름을 Thread-n 대신 다른 이름으로 설정하려면 Thread 클래스의 setName() 메소드 사용

```
thread.setName("스레드 이름");
```

- 디버깅할 때 어떤 스레드가 작업을 하는지 조사하기 위해 주로 사용
- 어떤 스레드가 실행하고 있는지 확인하려면 정적 메소드인 `currentThread()`로 스레드 객체의 참조를 얻은 다음 `getName()` 메소드로 이름을 출력

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

- ThreadNameExample.java

```
package ch14.sec04;

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

        for(int i=0; i<3; i++) {
            Thread threadA = new Thread() {
                @Override
                public void run() {
                    System.out.println(getName() + " 실행"); // 스레드 이름 리턴
                }
            };
            threadA.start();
        }
    }
}
```


- ThreadNameExample.java

```
Thread chatThread = new Thread() {  
    @Override  
    public void run() {  
        System.out.println(getName() + " 실행");    // 스레드 이름 리턴  
    }  
};  
chatThread.setName("chat-thread"); // 작업 스레드 이름 변경  
chatThread.start();  
}  
}
```

```
main 실행  
Thread-1 실행  
Thread-0 실행  
chat-thread 실행  
Thread-2 실행
```

- 스레드 상태

- 실행 대기 상태

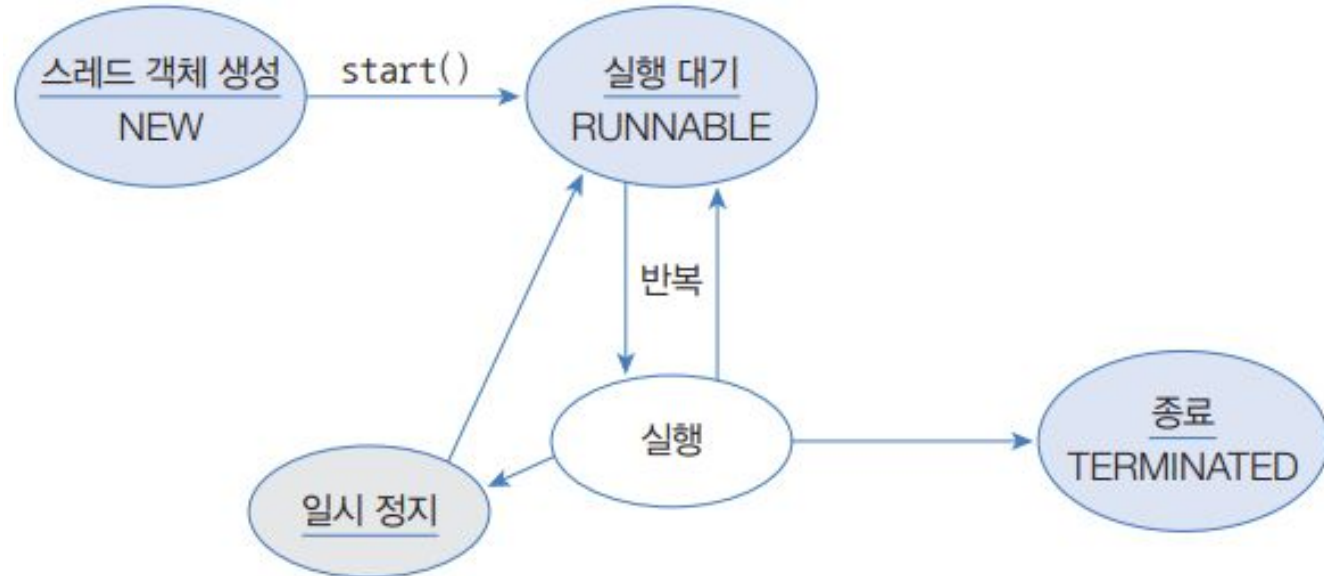
- 실행을 기다리고 있는 상태

- 실행 상태

- CPU 스케줄링에 따라 CPU를 점유하고 `run()` 메소드를 실행.
 - 스케줄링에 의해 다시 실행 대기 상태로 돌아갔다가 다른 스레드가 실행 상태 반복

- 종료 상태:

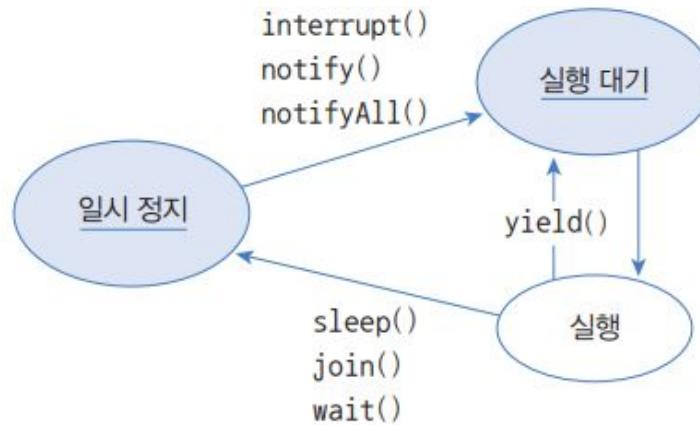
- 실행 상태에서 `run()` 메소드가 종료
 - 실행할 코드 없이 스레드의 실행을 멈춘 상태



• 일시 정지 상태

- 스레드가 실행할 수 없는 상태
- 스레드가 다시 실행 상태로 가기 위해서는 일시 정지 상태에서 실행 대기 상태로 가야 함
- Thread 클래스의 **sleep()** 메소드:
 - 실행 중인 스레드를 일정 시간 멈추게 함
 - 매개값 단위는 밀리세컨드(1/1000)

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



구분	메소드	설명
일시 정지로 보냄	sleep(long millis)	주어진 시간 동안 스레드를 일시 정지 상태로 만든다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다.
	join()	join() 메소드를 호출한 스레드는 일시 정지 상태가 된다. 실행 대기 상태가 되려면, join() 메소드를 가진 스레드가 종료되어야 한다.
	wait()	동기화 블록 내에서 스레드를 일시 정지 상태로 만든다.
일시 정지에서 벗어남	interrupt()	일시 정지 상태일 경우, InterruptedException을 발생시켜 실행 대기 상태 또는 종료 상태로 만든다.
	notify() notifyAll()	wait() 메소드로 인해 일시 정지 상태인 스레드를 실행 대기 상태로 만든다.
실행 대기로 보냄	yield()	실행 상태에서 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.

- **SleepExample.java**

```
package ch14.sec05.exam01;

import java.awt.Toolkit;

public class SleepExample {
    public static void main(String[] args) {
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        for(int i=0; i<10; i++) {
            toolkit.beep();
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
            }
        }
    }
}
```

- 다른 스레드의 종료를 기다림

- 어떤 스레드 **A**는 특정 스레드 **B**의 작업 결과를 바탕으로 동작
- 스레드 **A**는 스레드 **B**의 작업이 끝날 때까지 하는 일 없이 기다려야 함
→ 어떻게 스레드 **B**의 작업 종료를 인지할 수 있는가

- `join()` 메서드

- 해당 스레드가 종료할 때까지 대기 하다가 스레드가 종료하면 실행을 재개



- **SumThread.java**

```
package ch14.sec05.exam02;

public class SumThread extends Thread {
    private long sum;

    public long getSum() {
        return sum;
    }

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

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

- **SumThread.java**

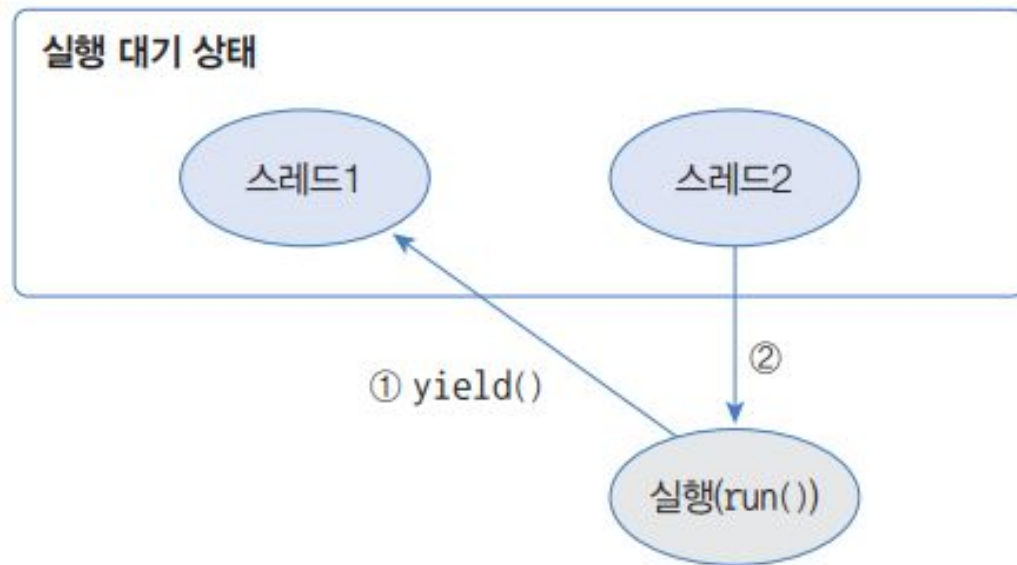
```
package ch14.sec05.exam02;

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

1~100 합: 5050

- 다른 스레드에게 실행 양보

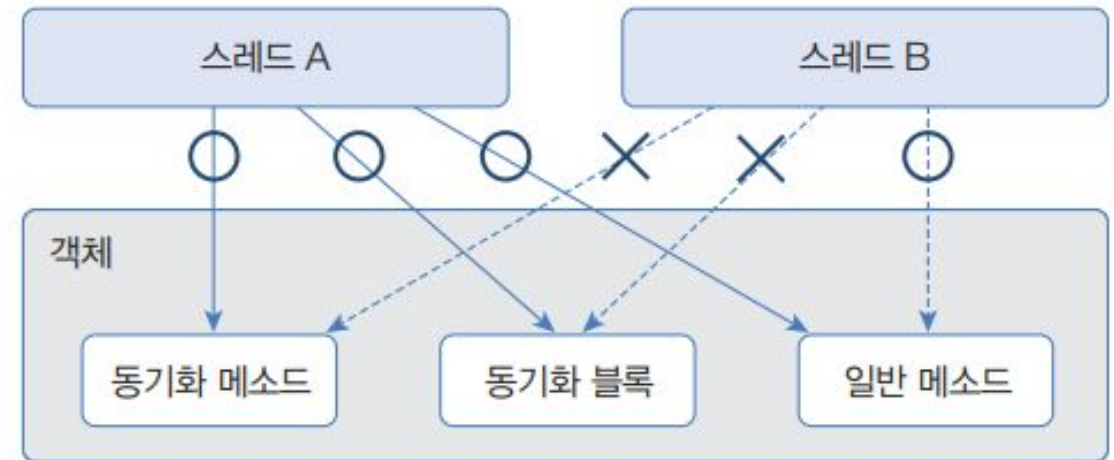
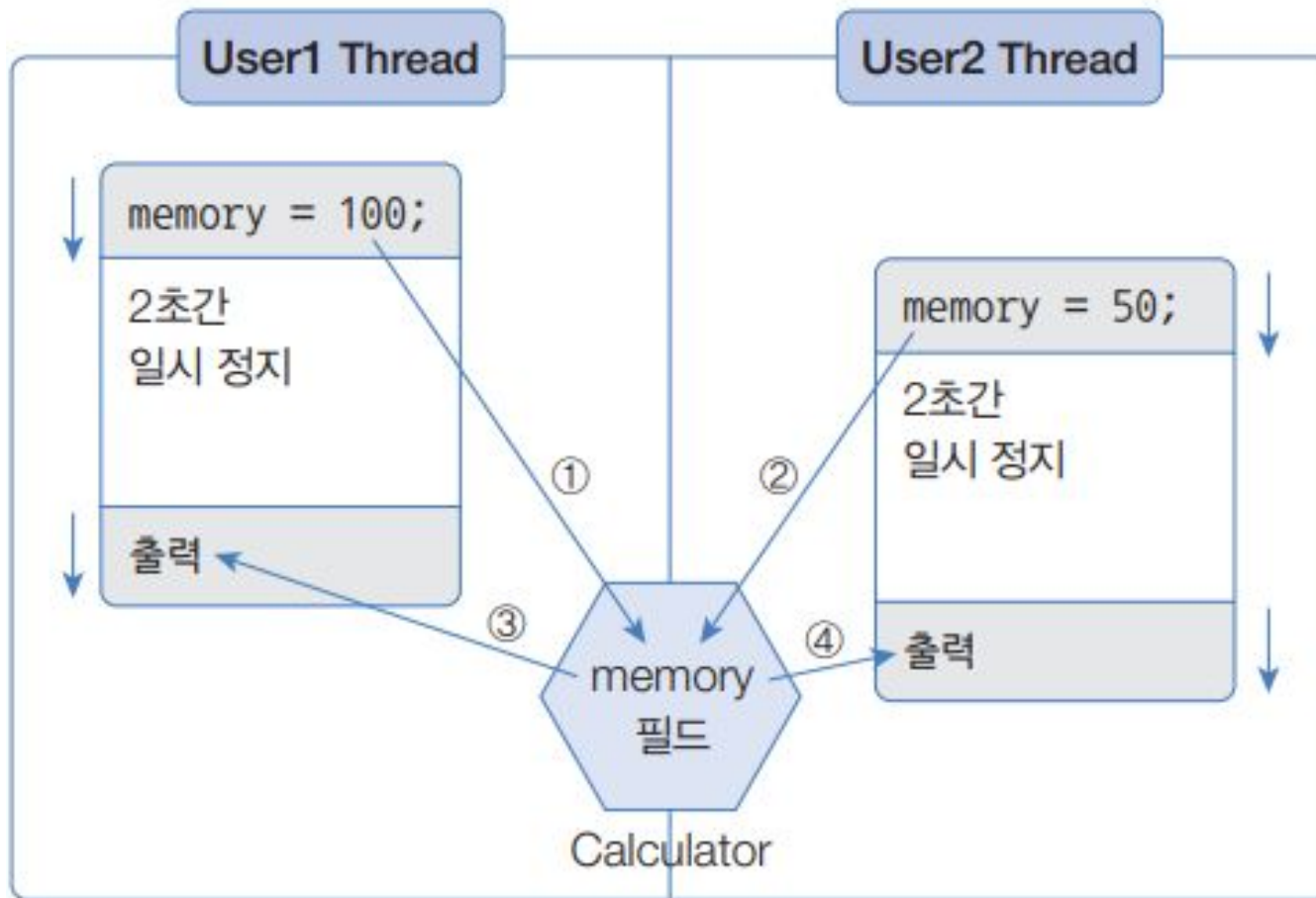
- `yield()` 메소드: 실행되는 스레드는 실행 대기 상태로 돌아가고, 다른 스레드가 실행되도록 양보
- 무의미한 반복을 막아 프로그램 성능 향상



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


• 동기화 메소드와 블록

- 스레드 작업이 끝날 때까지 객체에 잠금을 걸어 스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없게 함



• 동기화 메소드 및 블록 선언

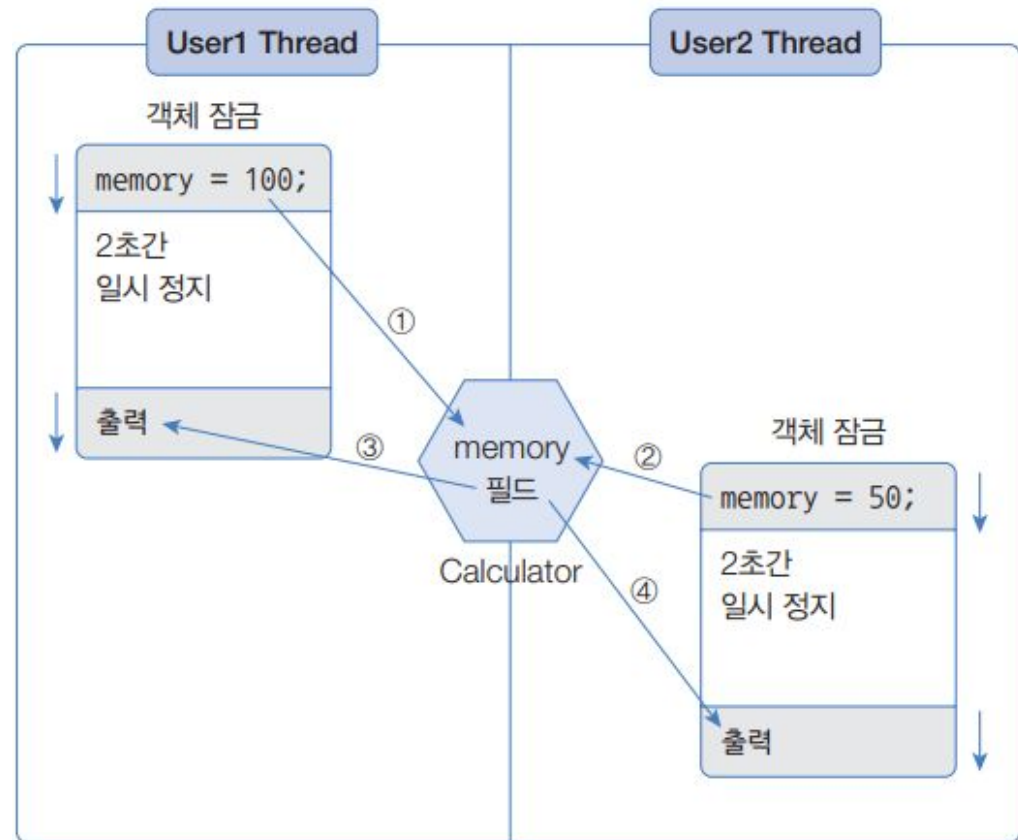
- 인스턴스와 정적 메소드에 **synchronized** 키워드 붙임
- 동기화 메소드를 실행 즉시 객체는 잠금이 일어나고, 메소드 실행이 끝나면 잠금 풀림
- 메소드 일부 영역 실행 시 객체 잠금을 걸고 싶다면 동기화 블록을 만듦

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

```
public void method () {
    //여러 스레드가 실행할 수 있는 영역

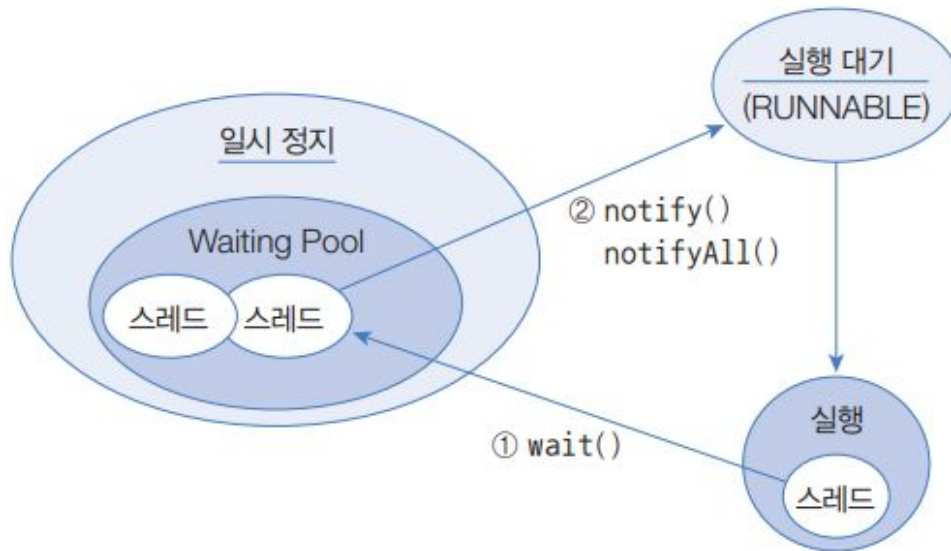
    synchronized(공유객체) {
        //단 하나의 스레드만 실행하는 영역
    }

    //여러 스레드가 실행할 수 있는 영역
}
```



- **wait()**과 **notify()**를 이용한 스레드 제어

- 두 스레드 교대 실행 시 공유 객체는 두 스레드가 작업할 내용을 각각 동기화 메소드로 정함
- 한 스레드 작업 완료 시 **notify()** 메소드를 호출해 일시 정지 상태에 있는 다른 스레드를 실행 대기 상태로 만들고, **wait()** 메소드를 호출하여 자신은 일시 정지 상태로 만듦



- **notify():** `wait()`에 의해 일시 정지된 스레드 중 한 개를 실행 대기 상태로 만듦
- **notifyAll():** `wait()`에 의해 일시 정지된 모든 스레드를 실행 대기 상태로 만듦

7 스레드 안전 종료

- 안전하게 스레드 종료하기

- 스레드 강제 종료 `stop()` 메소드: **deprecated**(더 이상 사용하지 않음)
- 스레드를 안전하게 종료하려면 사용하던 리소스(파일, 네트워크 연결)를 정리하고 `run()` 메소드를 빨리 종료해야 함

- 조건 이용

- `while` 문으로 반복 실행 시 조건을 이용해 `run()` 메소드 종료를 유도

```
public class XXXThread extends Thread {  
    private boolean stop;  
  
    public void run() {  
        while( !stop ) {  
            //스레드가 반복 실행하는 코드;  
        }  
        //스레드가 사용한 리소스 정리  
    }  
}
```

stop이 필드 선언

stop이 true가 되면 while 문을 빠져나감

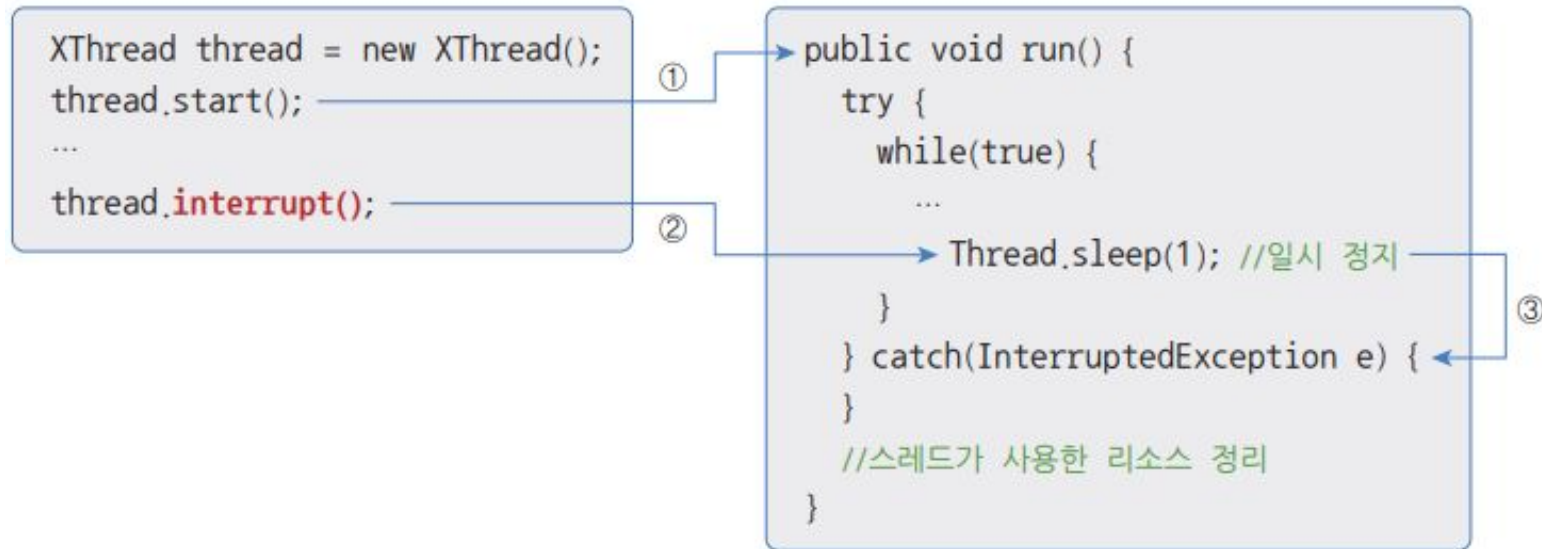
리소스 정리

스레드 종료

7 스레드 안전 종료

- **interrupt()** 메소드 이용

- 스레드가 일시 정지 상태에 있을 때 **InterruptedException** 예외 발생
- 예외 처리를 통해 **run()** 메소드를 정상 종료



- Thread의 **interrupted()**와 **isInterrupted()** 메소드는 **interrupt()** 메소드 호출 여부를 리턴

```

boolean status = Thread.interrupted();
boolean status = objThread.isInterrupted();

```

8 데몬 스레드

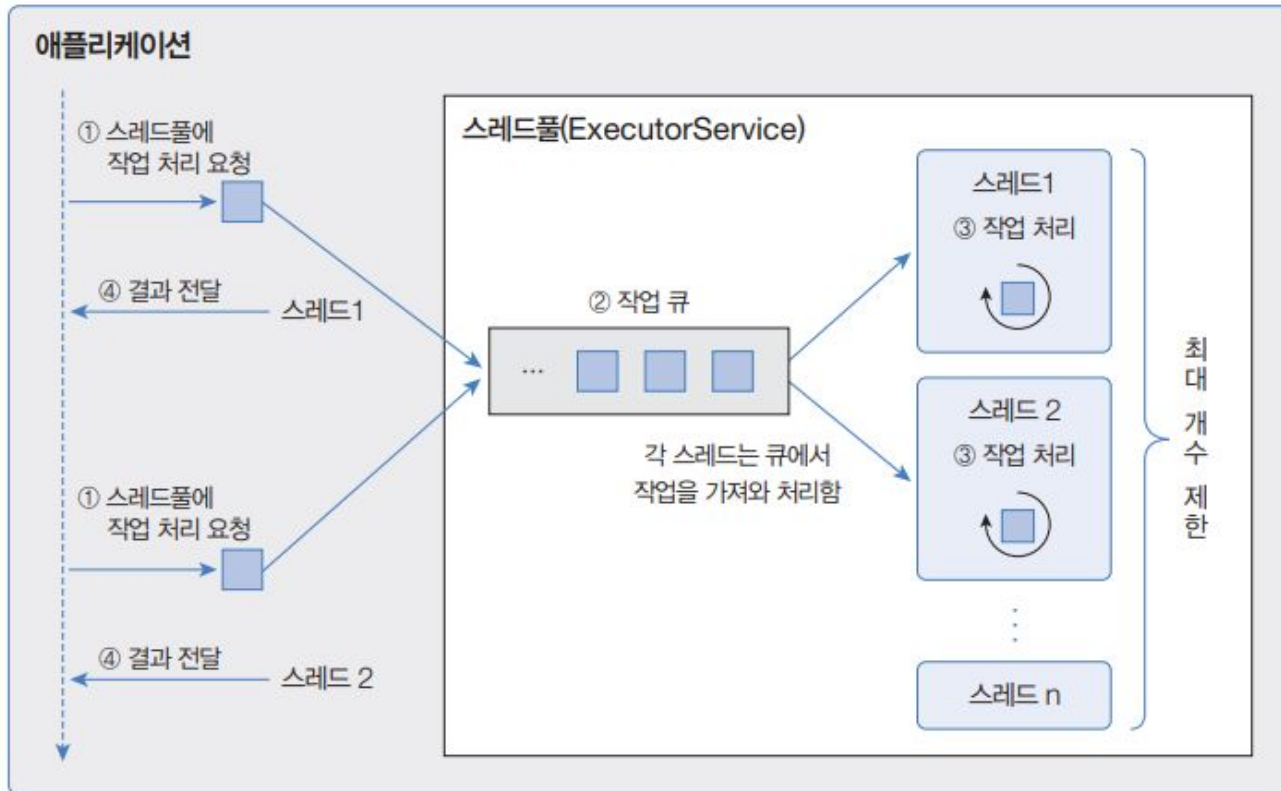
- 데몬 스레드

- 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드
- 주 스레드가 종료되면 데몬 스레드도 따라서 자동 종료
- 데몬 스레드 적용 예:
 - 워드프로세서의 자동 저장, 미디어플레이어의 동영상 및 음악 재생, 가비지 컬렉터
- 주 스레드가 데몬이 될 스레드의 `setDaemon(true)`를 호출

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

• 스레드풀로 작업 처리 제한하기

- 작업 처리에 사용되는 스레드 개수를 제한하고 작업 큐에 들어오는 작업들을 스레드가 하나씩 맡아 처리하는 방식
- 작업 처리가 끝난 스레드는 다시 작업 큐에서 새로운 작업을 가져와 처리
- 작업량이 증가해도 스레드의 개수가 늘어나지 않아 애플리케이션의 성능의 급격한 저하 방지



• 스레드풀 생성

- `java.util.concurrent` 패키지에서 `ExecutorService` 인터페이스와 `Executors` 클래스를 제공
- `Executors`의 다음 두 정적 메소드를 이용하면 스레드풀인 `ExecutorService` 구현 객체를 만들 수 있음

메소드명(매개변수)	초기 수	코어 수	최대 수
<code>newCachedThreadPool()</code>	0	0	<code>Integer.MAX_VALUE</code>
<code>newFixedThreadPool(int nThreads)</code>	0	생성된 수	<code>nThreads</code>

- 초기 수: 스레드풀이 생성될 때 기본적으로 생성되는 스레드 수
- 코어 수: 스레드가 증가된 후 사용되지 않는 스레드를 제거할 때 최소한 풀에서 유지하는 스레드 수
- 최대 수: 증가되는 스레드의 한도 수

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

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
```


- 스레드풀 종료

- 스레드풀의 스레드는 **main** 스레드가 종료되더라도 작업을 처리하기 위해 계속 실행 상태로 남음

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 interrupt해서 작업을 중지시키고 스레드풀을 종료시킨다. 리턴값은 작업 큐에 있는 미처리된 작업(Runnable)의 목록이다.

- 작업 생성과 처리 요청

- 하나의 작업은 Runnable 또는 Callable 구현 클래스로 표현

Runnable 익명 구현 클래스

```
new Runnable() {
    @Override
    public void run() {
        //스레드가 처리할 작업 내용
    }
}
```

Callable 익명 구현 클래스

```
new Callable<T> {
    @Override
    public T call() throws Exception {
        //스레드가 처리할 작업 내용
        return T;
    }
}
```

리턴 타입	메소드명(매개변수)	설명
void	execute(Runnable command)	<ul style="list-style-type: none"> - Runnable을 작업 큐에 저장 - 작업 처리 결과를 리턴하지 않음
Future<T>	submit(Callable<T> task)	<ul style="list-style-type: none"> - Callable을 작업 큐에 저장 - 작업 처리 결과를 얻을 수 있도록 Future를 리턴