제네릭 타입

결정되지 않은 타입을 파라미터로 가지는 클래스와 인터페이스이다.

선언부에 '< >' 부호가 붙고 그 사이에 타입 파라미터들이 위치한다.

```
public class 클래스명<A, B, ···> { ... }
public interface 인터페이스명<A, B, ···> { ... }
```

타입 파라미터는 일반적으로 대문자 알파벳 한 글자로 표현한다.

외부에서 제네릭 타입을 사용하려면 타입 파라미터에 구체적인 타입을 지정한다.

지정하지 않으면 Object 타입이 암묵적으로 사용된다.

와일드카드 타입 파라미터

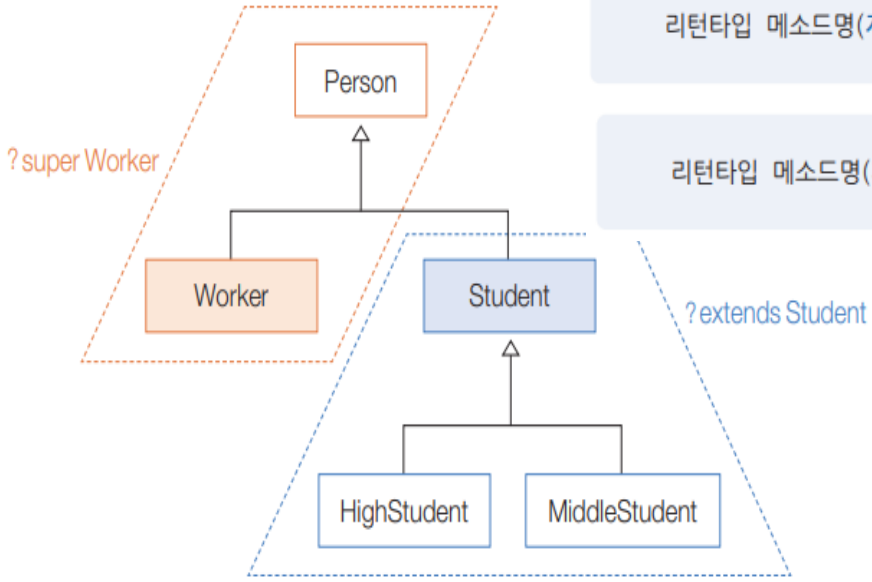
제네릭 타입을 매개값이나 리턴 타입으로 사용할 때 범위에 있는 모든 타입으로 대체할 수 있는 타입 파라미터이다.

? 기호로 표시한다.

```
리턴타입 메소드명(제네릭타입<? extends Student> 변수) { ··· }
```

```
리턴타입 메소드명(제네릭타입<? super Worker> 변수) { ··· }
```

```
리턴타입 메소드명(제네릭타입<?> 변수) { ··· }
```



제네릭 메서드

타입 피라미터를 가지고 있는 메서드다.

타입 파라미터가 메서드 선언부에 정의된다.

리턴 타입 앞에 < > 기호와 타입 파라미터를 정의한 후에 리턴 타입과 매개변수 타입에서 사용한다.

```
public  <A, B, ···>  리턴타입  메소드명(매개변수, ...) { ... }
```

타입 파라미터 정의

타입 파라미터 T는 메서드 호출시 전달되는 아규먼트의 타입에 따라 컴파일 과정에서 구체적인 타입으로 대체된다.

```
public <T> Box<T> boxing(T t) { ··· }
```

```
① Box<Integer> box1 = boxing(100);
② Box<String> box2 = boxing("안녕하세요");
```

제한된 타입 파라미터

모든 타입으로 대체할 수는 없고, 특정 타입과 자식 또는 구현 관계에 있는 타입만 대체할 수 있는 타입 파라미터로서 extends 절을 사용한다.(상위 타입은 클래스뿐만 아니라 인터페이스도 가능)

```
public <T extends 상위타입> 리턴타입 메소드(매개변수, ...) { ... }
```

```
public <T extends Number> boolean compare(T t1, T t2) {
  double v1 = t1.doubleValue();   //Number의 doubleValue() 메소드 사용
  double v2 = t2.doubleValue();   //Number의 doubleValue() 메소드 사용
  return (v1 == v2);
}
```

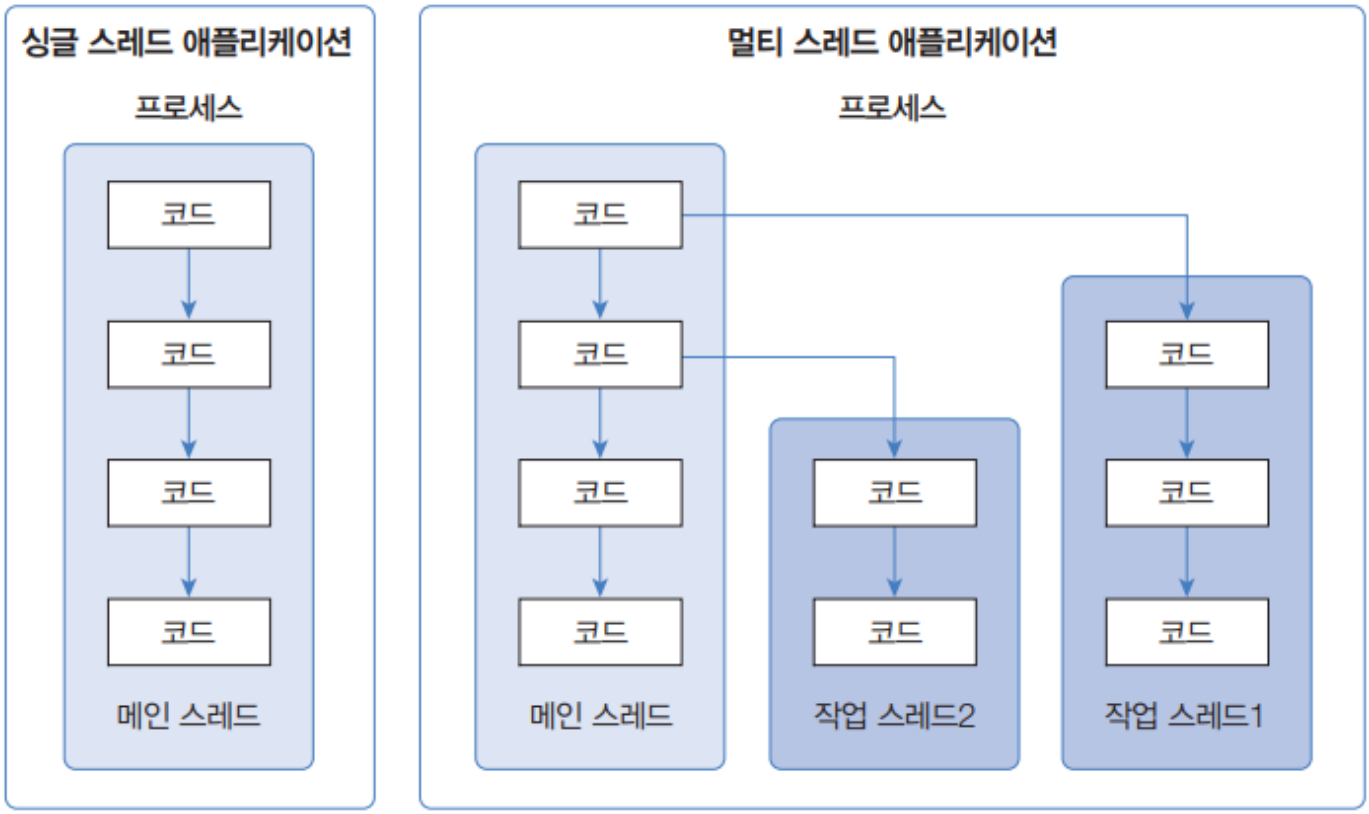
멀티스레드 프로그래밍

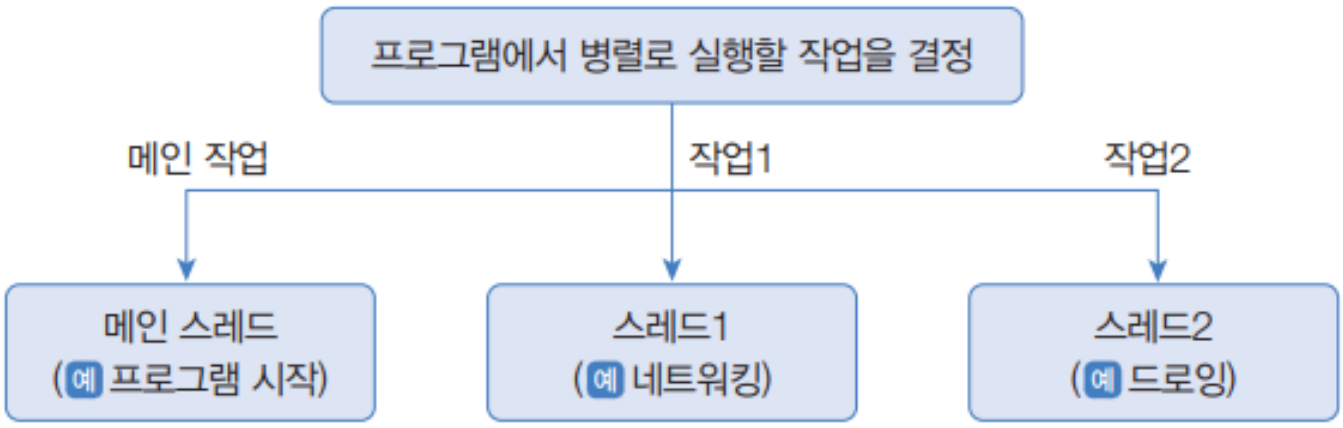
멀티 프로세스와 멀티 스레드

- 프로세스: 실행 중인 프로그램
- 멀티 태스킹: 두 가지 이상의 작업을 동시에 처리하는 것
- 스레드: 프로세스 내에서 코드의 실행 흐름
- 멀티 스레드: 두 개의 코드 실행 흐름. 두 가지 이상의 작업을 처리
- 멀티 프로세스 = 실행 중인 프로그램이 2개 이상

프로그램 단위의 멀티 태스킹 - 멀티 프로세스

프로그램 내부에서의 멀티 태스킹 - 멀티 스레드


싱글 스레드 애플리케이션
프로세스
코드
코드
코드
코드
메인 스레드
멀티 스레드 애플리케이션
프로세스
코드
코드
코드
코드
메인 스레드
코드
코드
작업 스레드2
코드
코드
코드
작업 스레드1


프로그램에서 병렬로 실행할 작업을 결정
메인 작업
작업1
작업2
메인 스레드
(예 프로그램 시작)
스레드1
(예 네트워킹)
스레드2
(예 드로잉)

프로세스 : 실행 중인 프로그램, 자원(resources)과 쓰레드로 구성

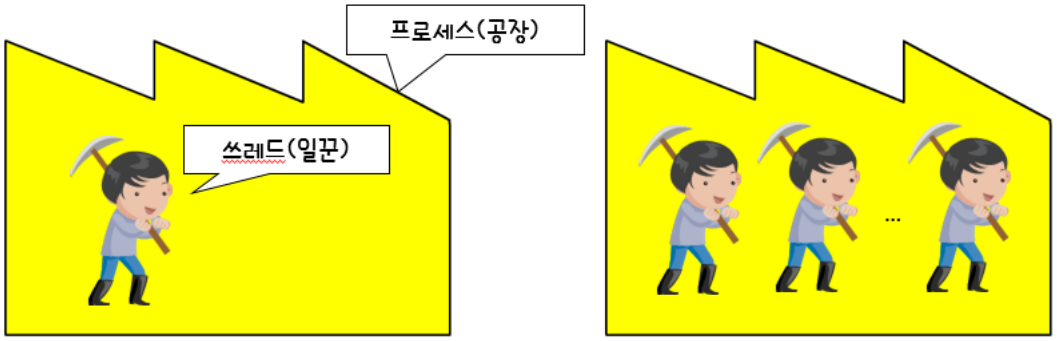
쓰레드 : 프로세스 내에서 실제 작업을 수행.

모든 프로세스는 하나 이상의 쓰레드를 가지고 있다.

프로세스 : 쓰레드 = 공장 : 일꾼

싱글 쓰레드 프로세스
= 자원+쓰레드

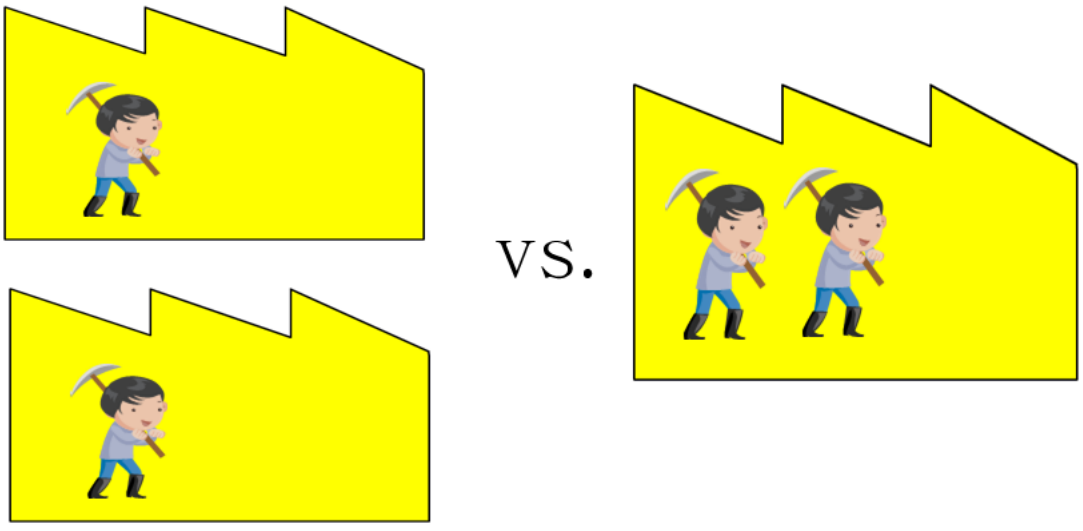
멀티 쓰레드 프로세스
= 자원+쓰레드+쓰레드+…+쓰레드



"하나의 새로운 프로세스를 생성하는 것보다

하나의 새로운 쓰레드를 생성하는 것이 더 적은 비용이 든다."

2 프로세스 1 쓰레드 vs. 1 프로세스 2 쓰레드



1. Thread클래스를 상속

```
class MyThread extends Thread {
    public void run() { /* 작업내용 */ } // Thread클래스의 run()을 오버라이딩
}
```

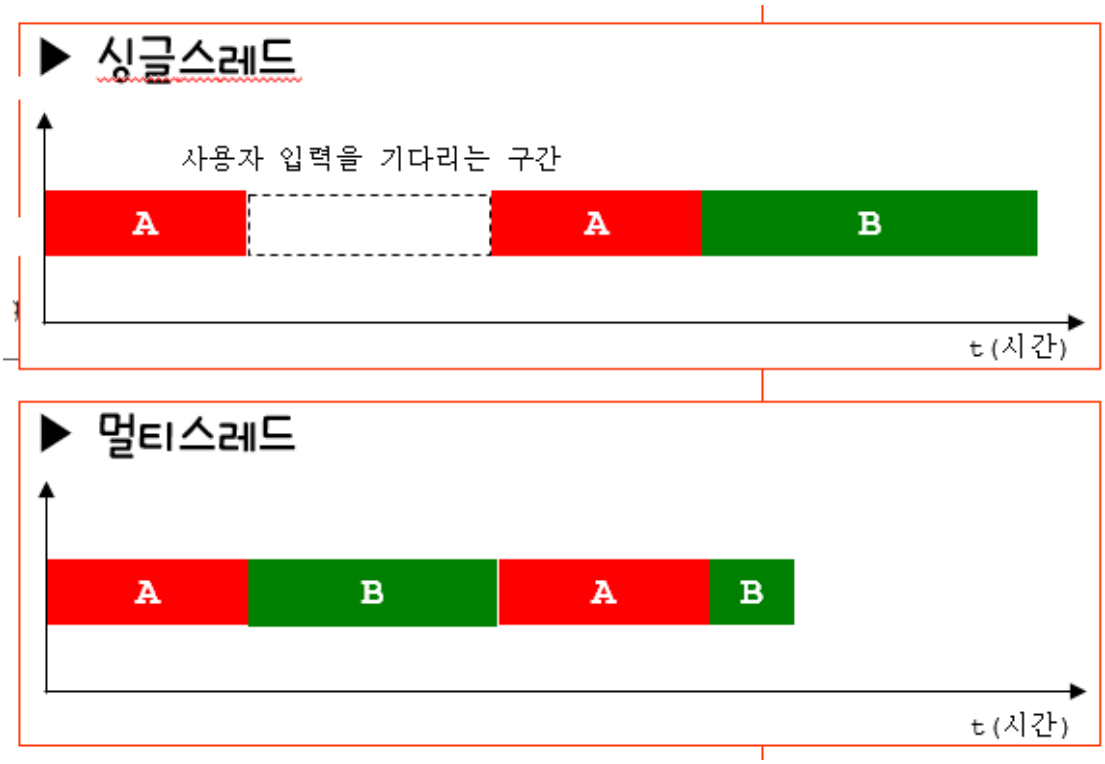
2. Runnable인터페이스를 구현

```
class MyThread implements Runnable {
    public void run() { /* 작업내용 */ } // Runnable인터페이스의 추상메서드 run()을 구현
}
```

```
public interface Runnable {
    public abstract void run();
}
```

```
MyThread myt = new MyThread();
myt.start();
```

```
MyThread myt = new MyThread();
Thread t = new Thread(myt);
myt.start();
```



스레드 상태

실행 대기 상태: 실행을 기다리고 있는 상태

실행 상태: CPU 스케쥴링에 따라 CPU를 점유하고 run() 메소드를 실행.

스케줄링에 의해 다시 실행 대기 상태로 돌아갔다가 다른 스레드가 실행 상태 반복

종료 상태: 실행 상태에서 run() 메소드가 종료되어 실행할 코드 없이 스레드의 실행을 멈춘 상태

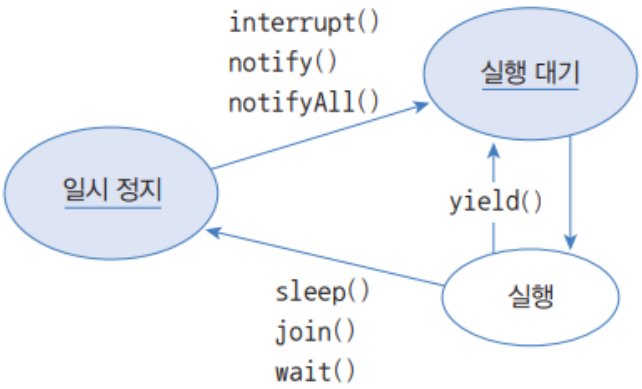
[일시 정지 상태]

스레드가 실행할 수 없는 상태

스레드가 다시 실행 상태로 가기 위해서는 일시 정지 상태에서 실행 대기 상태로 가야야 함

Thread 클래스의 sleep() 메소드: 실행 중인 스레드를 일정 시간 멈추게 함

매개값 단위는 밀리세컨드(1/1000)

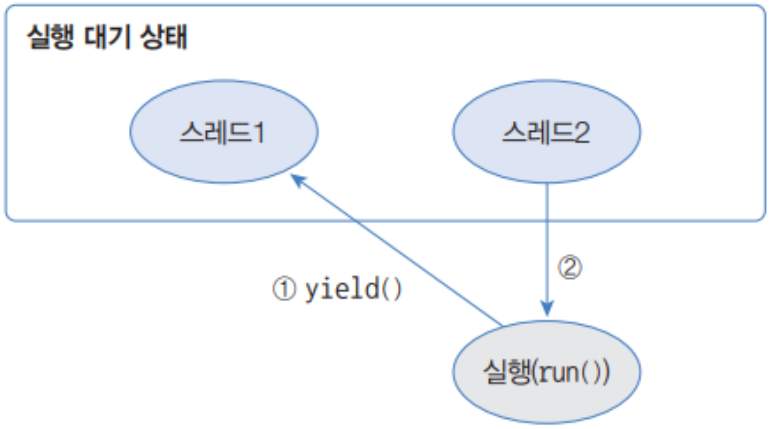


구분	메소드	설명
일시 정지로 보냄	sleep(long millis)	주어진 시간 동안 스레드를 일시 정지 상태로 만든다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다.
	join( )	join( ) 메소드를 호출한 스레드는 일시 정지 상태가 된다. 실행 대기 상태가 되려면, join( ) 메소드를 가진 스레드가 종료되어야 한다.
	wait( )	동기화 블록 내에서 스레드를 일시 정지 상태로 만든다.
일시 정지에서 벗어남	interrupt( )	일시 정지 상태일 경우, InterruptedException을 발생시켜 실행 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.

[다른 스레드에게 실행 양보]

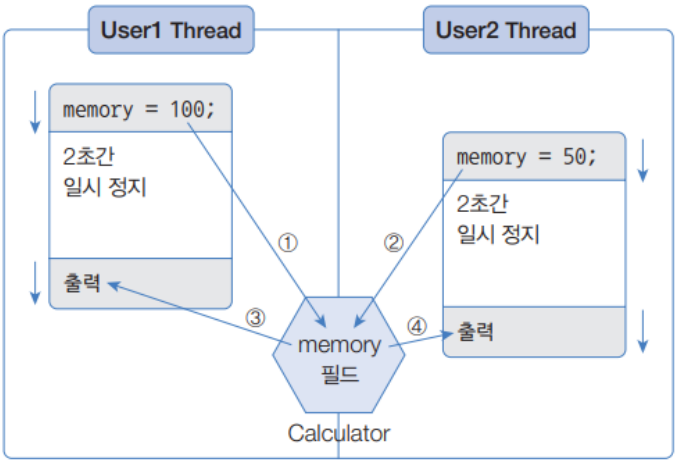
yield() 메소드: 실행되는 스레드는 실행 대기 상태로 돌아가고, 다른 스레드가 실행되도록 양보

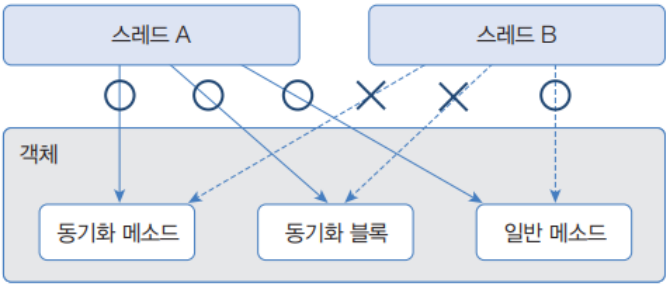
무의미한 반복을 막아 프로그램 성능 향상



[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

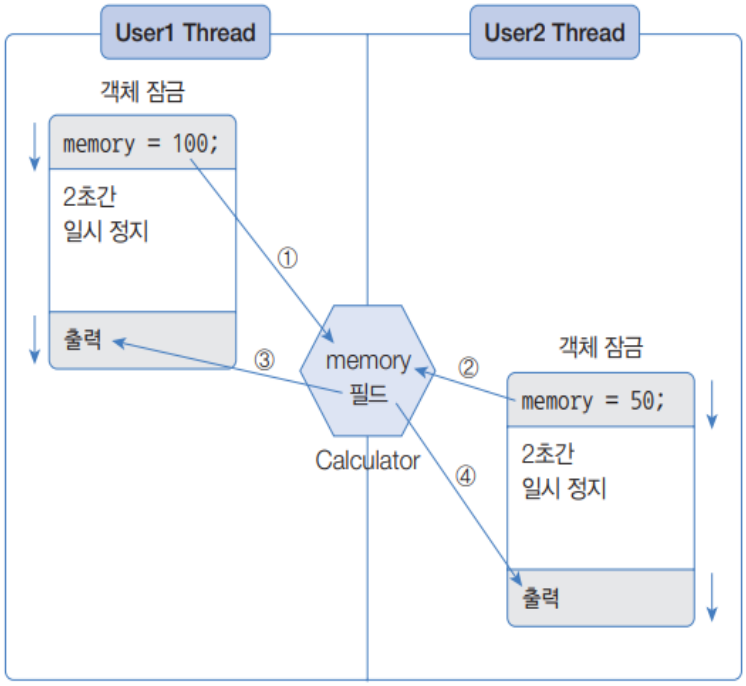
동기화 메소드를 실행 즉시 객체는 잠금이 일어나고, 메소드 실행이 끝나면 잠금 풀림

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) {
```



[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

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

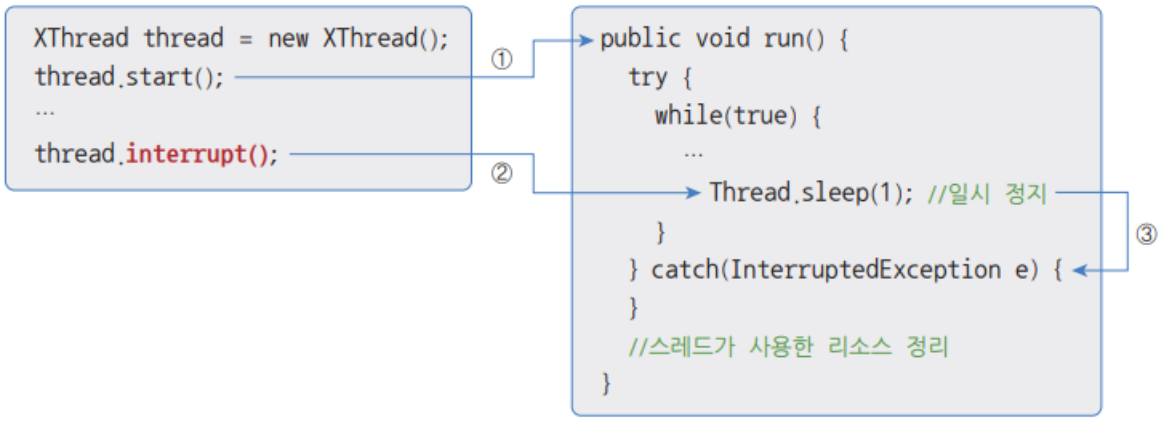
```
public class XXXThread extends Thread {
  private boolean stop;          ····· stop이 필드 선언

  public void run() {
    while( !stop ) {             ····· stop이 true가 되면 while 문을 빠져나감
      //스레드가 반복 실행하는 코드;
    }
    //스레드가 사용한 리소스 정리  ····· 리소스 정리
  }                              ····· 스레드 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();
```

데몬 스레드

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

[동기화된 컬렉션]

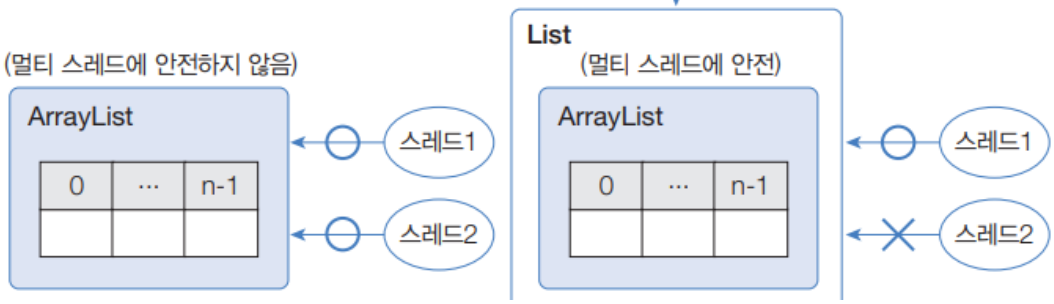
동기화된 메소드로 구성된 Vector와 Hashtable는 멀티 스레드 환경에서 안전하게 요소를 처리한다.

Collections의 synchronizedXXX() 메소드: ArrayList, HashSet, HashMap 등

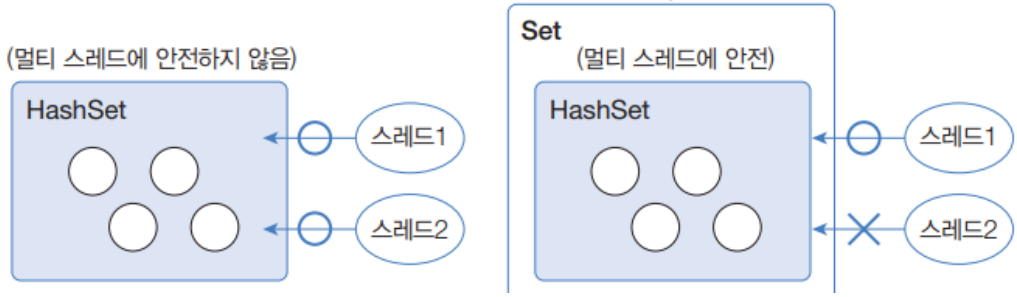
비동기화된 메소드를 동기화된 메소드로 래핑

리턴 타입	메소드(매개변수)	설명
List<T>	synchronizedList(List<T> list)	List를 동기화된 List로 리턴
Map<K,V>	synchronizedMap(Map<K,V> m)	Map을 동기화된 Map으로 리턴
Set<T>	synchronizedSet(Set<T> s)	Set을 동기화된 Set으로 리턴

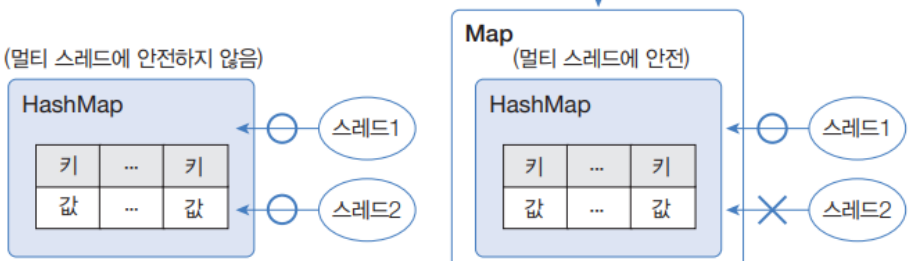
```
List<T> list = Collections.synchronizedList(new ArrayList<T>());
```



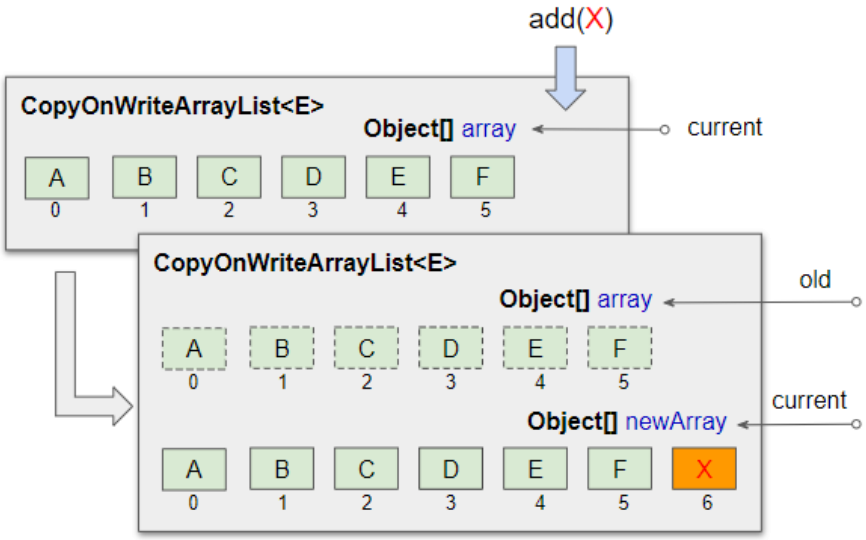

```
Set<E> set = Collections.synchronizedSet(new HashSet<E>());
```




```
Map<K, V>map = Collections.synchronizedMap(new HashMap<K, V>());
```



Collections.synchronizedList(List 객체) & CopyOnWriteArrayList<>()



read 작업량 < write 작업량 : SynchronizedList

read 작업량 > write 작업량 : CopyOnWriteList

SynchronizedList는 get 작업에서도 락이 걸리기때문에 멀티 쓰레드 환경에서 조회가 많은 작업일 경우에는 오버헤드가 더 크다. CopyOnWriteList는 set, add 과정에서 데이터를 복제후 설정하는 방식으로 사용하기때문에 추가적인 오버헤드가 발생한다. 따라서 멀티스레드 환경에서 어떤 작업이 주로 수행되는가에 따라 사용될 리스트를 선택해야 한다.