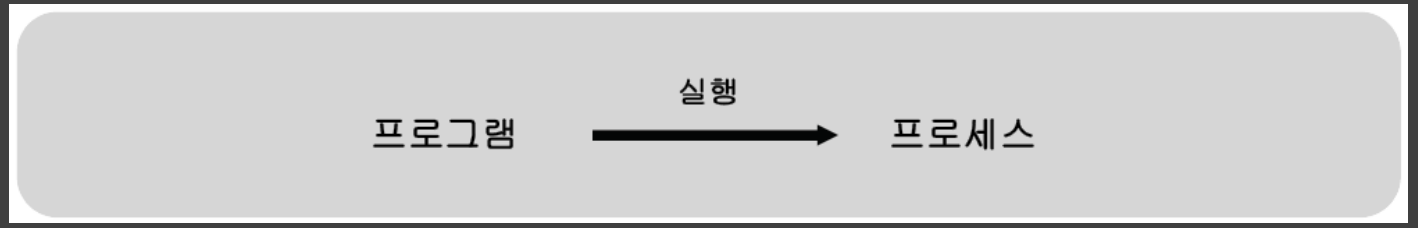
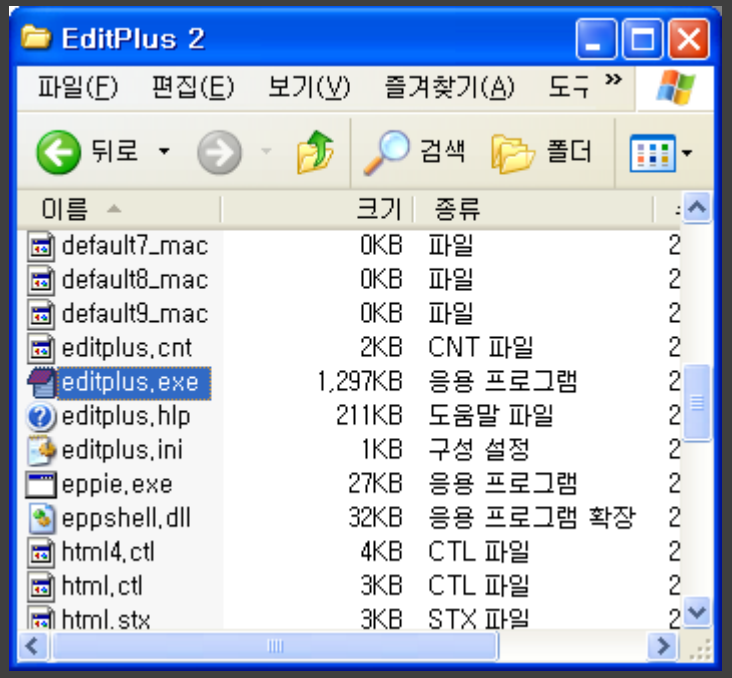
Java 기초

스레드(Thread)

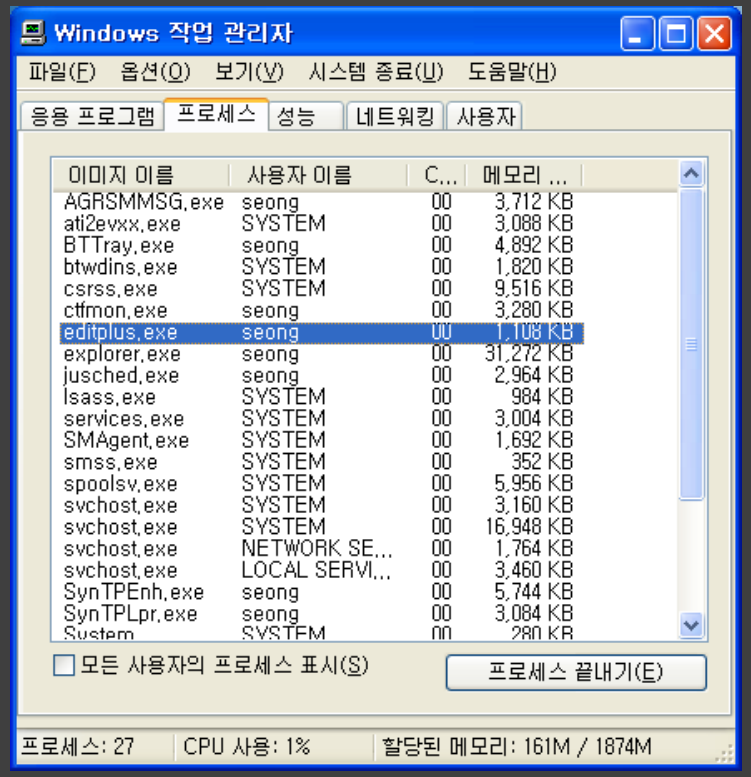
1. 프로세스와 쓰레드(process & thread) (1)



▶ 프로그램 : 실행 가능한 파일(HDD)



▶ 프로세스 : 실행 중인 프로그램(메모리)



1. 프로세스와 쓰레드(process & thread) (2)

- 프로세스 : 실행 중인 프로그램, 자원(resources)과 쓰레드로 구성
- 쓰레드 : 프로세스 내에서 실제 작업을 수행하는 실행 단위

  모든 프로세스는 하나 이상의 쓰레드를 가지고 있다.

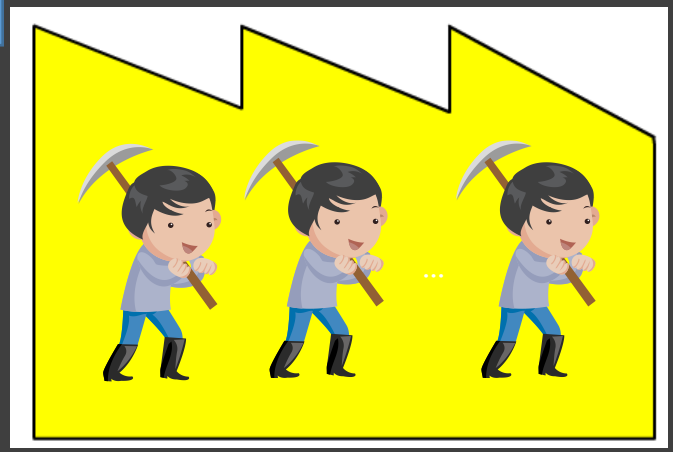
프로세스 : 쓰레드 = 공장 : 일꾼

- 싱글 쓰레드 프로세스

  = 자원+쓰레드

- 멀티 쓰레드 프로세스

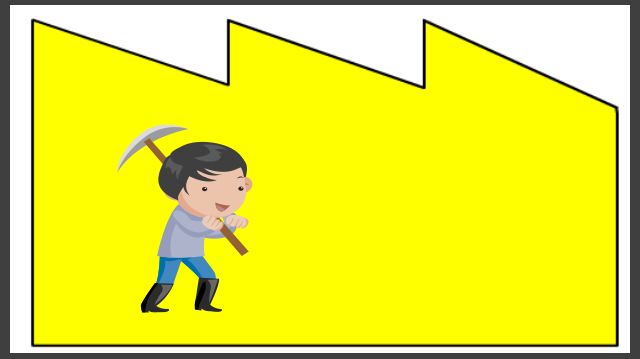
  = 자원+쓰레드+쓰레드+…+쓰레드

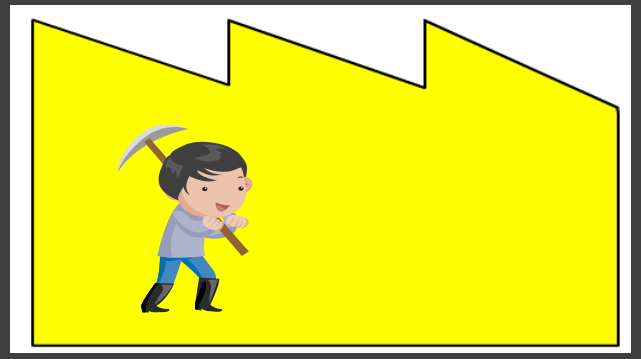


2. 멀티프로세스 vs. 멀티쓰레드

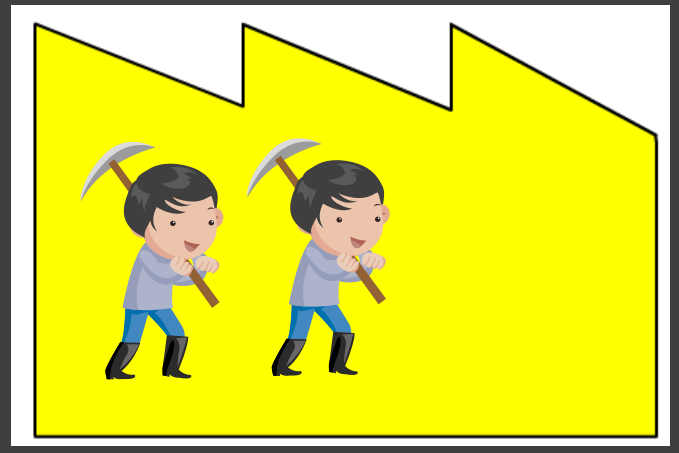
"하나의 새로운 프로세스를 생성하는 것보다
하나의 새로운 쓰레드를 생성하는 것이 더 적은 비용이 든다."

- 2 프로세스 1 쓰레드 vs. 1 프로세스 2 쓰레드





VS.



3. 멀티쓰레드의 장단점

“많은 프로그램들이 멀티쓰레드로 작성되어 있다.
그러나, 멀티쓰레드 프로그래밍이 장점만 있는 것은 아니다.”

장점	- 자원을 보다 **효율적**으로 사용할 수 있다. - 사용자에 대한 **응답성**(responseness)이 **향상**된다. - 작업이 분리되어 **코드가 간결**해 진다. **“여러 모로 좋다.”**
단점	- 동기화(synchronization)에 주의해야 한다. - 교착상태(dead-lock)가 발생하지 않도록 주의해야 한다. - 각 쓰레드가 효율적으로 고르게 실행될 수 있게 해야 한다. **“프로그래밍할 때 고려해야 할 사항들이 많다.”**

4. 쓰레드의 구현과 실행

```java
public class ThreadEx1 extends Thread {
    public void run() {
        System.out.println("Thread를 상속받은 형태");
    }
}
```

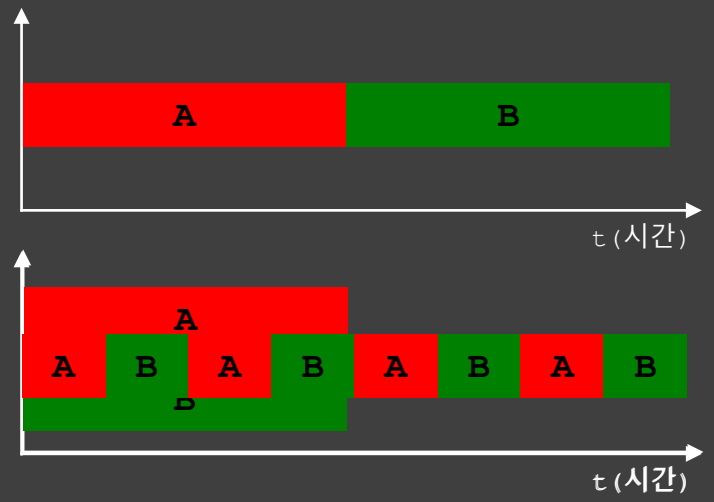
```java
public class ThreadEx2 implements Runnable {
    public void run() {
        System.out.println("Runnable을 상속받은 형태");
    }
}
```

```java
package com.hb.operator;

public class ThreadMain {
    public static void main(String[] args) {
        Thread t1 = new ThreadEx1();
        t1.start();
        Thread t2 = new Thread(new ThreadEx2());
        t2.start();
    }
}
```

5. 싱글쓰레드 vs. 멀티쓰레드(1)

▶ 싱글쓰레드

```
class ThreadTest {
    public static void main(String args[]){
        for(int i=0;i<300;i++) {
            System.out.println("-");
        }

        for(int i=0;i<300;i++) {
            System.out.println("|");
        }
    } // main
}
```



▶ 멀티쓰레드

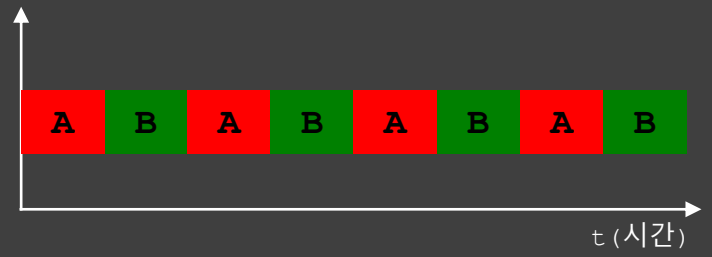
```
class ThreadTest {
    public static void main(String args[]){
        MyThread1 th1 = new MyThread1();
        MyThread2 th2 = new MyThread2();
        th1.start();
        th2.start();
    }
}

class MyThread1 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("-");
        }
    } // run()
}

class MyThread2 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("|");
        }
    } // run()
}
```

6 쓰레드의 우선순위(priority of thread)

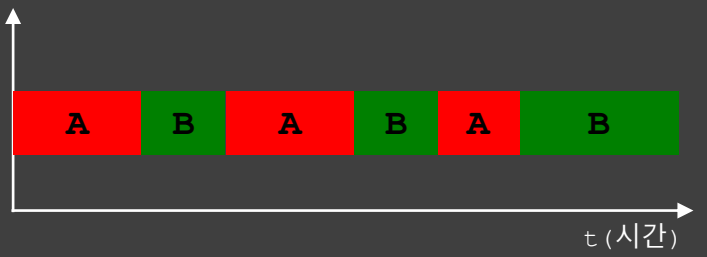
"작업의 중요도에 따라 쓰레드의 우선순위를 다르게 하여
특정 쓰레드가 더 많은 작업시간을 갖도록 할 수 있다."

```
void setPriority(int newPriority) : 쓰레드의 우선순위를 지정한 값으로 변경한다.
int getPriority() : 쓰레드의 우선순위를 반환한다.

public static final int MAX_PRIORITY = 10 // 최대우선순위
public static final int MIN_PRIORITY = 1  // 최소우선순위
public static final int NORM_PRIORITY = 5 // 보통우선순위
```

▶ 우선순위가 같은 경우



▶ A의 우선순위가 높은 경우



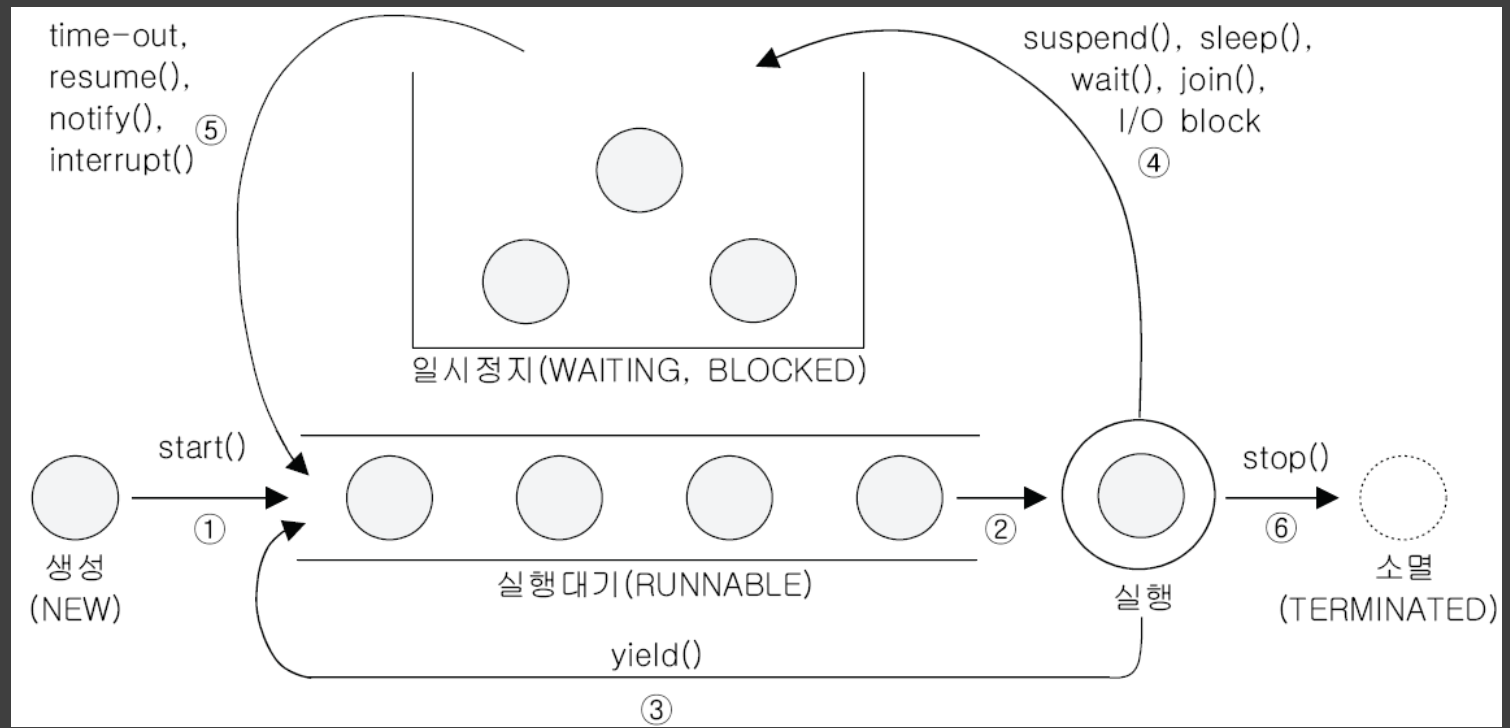
6 쓰레드의 우선순위(priority of thread)

```java
public class CalcThread extends Thread {
    public CalcThread(String name){
        setName(name);
    }
    @Override
    public void run() {
        for(int i=0;i<20000;i++){};
        System.out.println(getName());
    }
}
```

```java
public class PriorityExample {
    public static void main(String[] args) {
        for(int i=1;i<=10;i++){
            Thread thread = new CalcThread("thread"+i);
            if(i != 10){
                thread.setPriority(Thread.MIN_PRIORITY);
            }else{
                thread.setPriority(Thread.MAX_PRIORITY);
            }
            thread.start();
        }
    }
```

7 쓰레드의 상태(state of thread)

상태	설명
NEW	쓰레드가 생성되고 아직 start()가 호출되지 않은 상태
RUNNABLE	실행 중 또는 실행 가능한 상태
BLOCKED	동기화블럭에 의해서 일시정지된 상태(lock이 풀릴 때까지 기다리는 상태)
WAITING, TIMED_WAITING	쓰레드의 작업이 종료되지는 않았지만 실행가능하지 않은(unrunnable) 일시정지상태. TIMED_WAITING은 일시정지시간이 지정된 경우를 의미한다.
TERMINATED	쓰레드의 작업이 종료된 상태



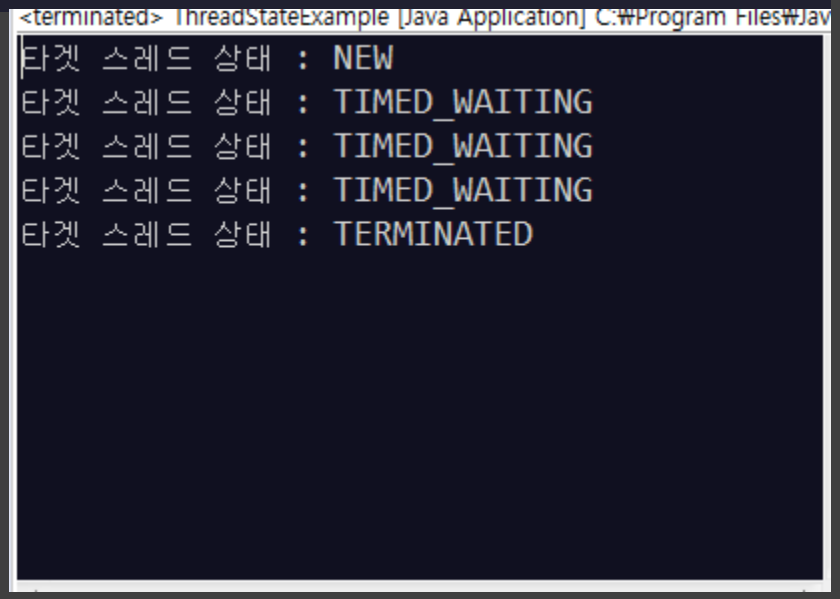
7 쓰레드의 상태(state of thread)

```java
public class StatePrintThread extends Thread {
    private Thread targetThread;
    public StatePrintThread(Thread targetThread){
        this.targetThread = targetThread;
    }
    @Override
    public void run() {
        while(true){
            Thread.State state = targetThread.getState();
            System.out.println("타겟 스레드 상태 : "+state);

            if(state==Thread.State.NEW)targetThread.start();
            if(state==Thread.State.TERMINATED)break;
            try {
                Thread.sleep(500);
            } catch (Exception e) {}
        }
```

```java
public class TargetThread extends Thread{
    @Override
    public void run() {
        for(long i = 0;i<100000000;i++){}
        try{
            Thread.sleep(1500);
        }catch(Exception e){}
        for(long i = 0;i<100000000;i++){}
    }
```

7 쓰레드의 상태(state of thread)

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



8 쓰레드의 실행제어

메소드	설명
interrupt()	일지정지 상태의 스레드에서 InterruptedException 예외를 발생시켜, 예외 처리 코드(catch)에서 실행 디기 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.
resume()	suspend()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 맴버로 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.
yield()	실행 중에 우선순위가 동일한 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.
stop()	스레드가 즉시 종료된다.

* resume(), stop(), suspend()는 쓰레드를 교착상태로 만들기 쉽기 때문에 deprecated되었다.

8.1 쓰레드의 실행제어

- sleep : 주어진 시간동안 일시 정지한다.

```
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) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

8.2 쓰레드의 실행제어

- yeild : 다른 쓰레드에게 실행을 양보한다.

```
public class ThreadA extends Thread{
    private boolean stop = false;
    private boolean work = true;

    public boolean isStop() {return stop;}
    public void setStop(boolean stop) {this.stop = stop;}
    public boolean isWork() {return work;}
    public void setWork(boolean work) {this.work = work;}

    @Override
    public void run() {
        while(!stop){
            try {Thread.sleep(500);} catch (InterruptedException e) {}
            if(work){
                System.out.println("ThreadA 작업 내용");
            }else{
                Thread.yield();
            }
        }
        System.out.println("ThreadA 종료");
    }
}
```

8.2 쓰레드의 실행제어

- yeild : 다른 쓰레드에게 실행을 양보한다.

```
public class ThreadB extends Thread{
    private boolean stop = false;
    private boolean work = true;

    public boolean isStop() {return stop;}
    public void setStop(boolean stop) {this.stop = stop;}
    public boolean isWork() {return work;}
    public void setWork(boolean work) {this.work = work;}

    @Override
    public void run() {
        while(!stop){
            try {Thread.sleep(500);} catch (InterruptedException e) {}
            if(work){
                System.out.println("ThreadB 작업 내용");
            }else{
                Thread.yield();
            }
        }
        System.out.println("ThreadB 종료");
    }
```

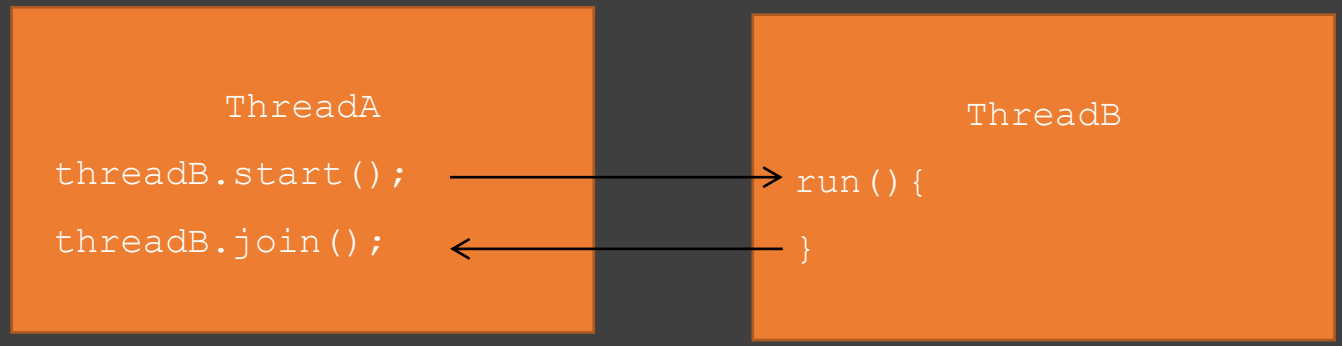
8.2 쓰레드의 실행제어

- yeild : 다른 쓰레드에게 실행을 양보한다.

```
public class YeildExample {

    public static void main(String[] args) {
        ThreadA threadA = new ThreadA();
        ThreadB threadB = new ThreadB();

        threadA.start();
        threadB.start();

        try {Thread.sleep(3000);} catch (Exception e) { }
        threadA.setWork(false);

        try {Thread.sleep(3000);} catch (Exception e) { }
        threadA.setWork(true);

        try {Thread.sleep(3000);} catch (Exception e) { }
        threadA.setStop(true);
        threadB.setStop(true);
    }
```

8.3 쓰레드의 실행제어

- join : 다른 스레드의 종료를 기다린다..



8.3 쓰레드의 실행제어

- join : 다른 스레드의 종료를 기다린다..

```java
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());
    }
```

```java
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;}
    }
```

9. 쓰레드의 동기화 - synchronized

- 한 번에 하나의 쓰레드만 객체에 접근할 수 있도록 객체에 락(lock)을 걸어서 데이터의 일관성을 유지하는 것.

1. 특정한 객체에 lock을 걸고자 할 때

```
synchronized(객체의 참조변수) {
    //...
}
```

2. 메서드에 lock을 걸고자할 때

```
public synchronized void calcSum() {
    //...
}
```

```
public synchronized void withdraw(int money) {
    if(balance >= money) {
        try {
            Thread.sleep(1000);
        } catch(Exception e) {}

        balance -= money;
    }
}
```

```
public void withdraw(int money) {
    synchronized(this) {
        if(balance >= money) {
            try {
                Thread.sleep(1000);
            } catch(Exception e) {}

            balance -= money;
        }
    } // synchronized(this)
}
```

9.1. 쓰레드의 동기화 - Example

```java
public class Calculator {
    private int memory;

    public int getMemory(){
        return memory;
    }

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

```java
public class User1 extends Thread {
    private Calculator calculator;

    public void setCalculator(Calculator calculator){
        this.setName("User1");
        this.calculator = calculator;
    }

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

```java
public class User2 extends Thread {
    private Calculator calculator;

    public void setCalculator(Calculator calculator){
        this.setName("User2");
        this.calculator = calculator;
    }

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

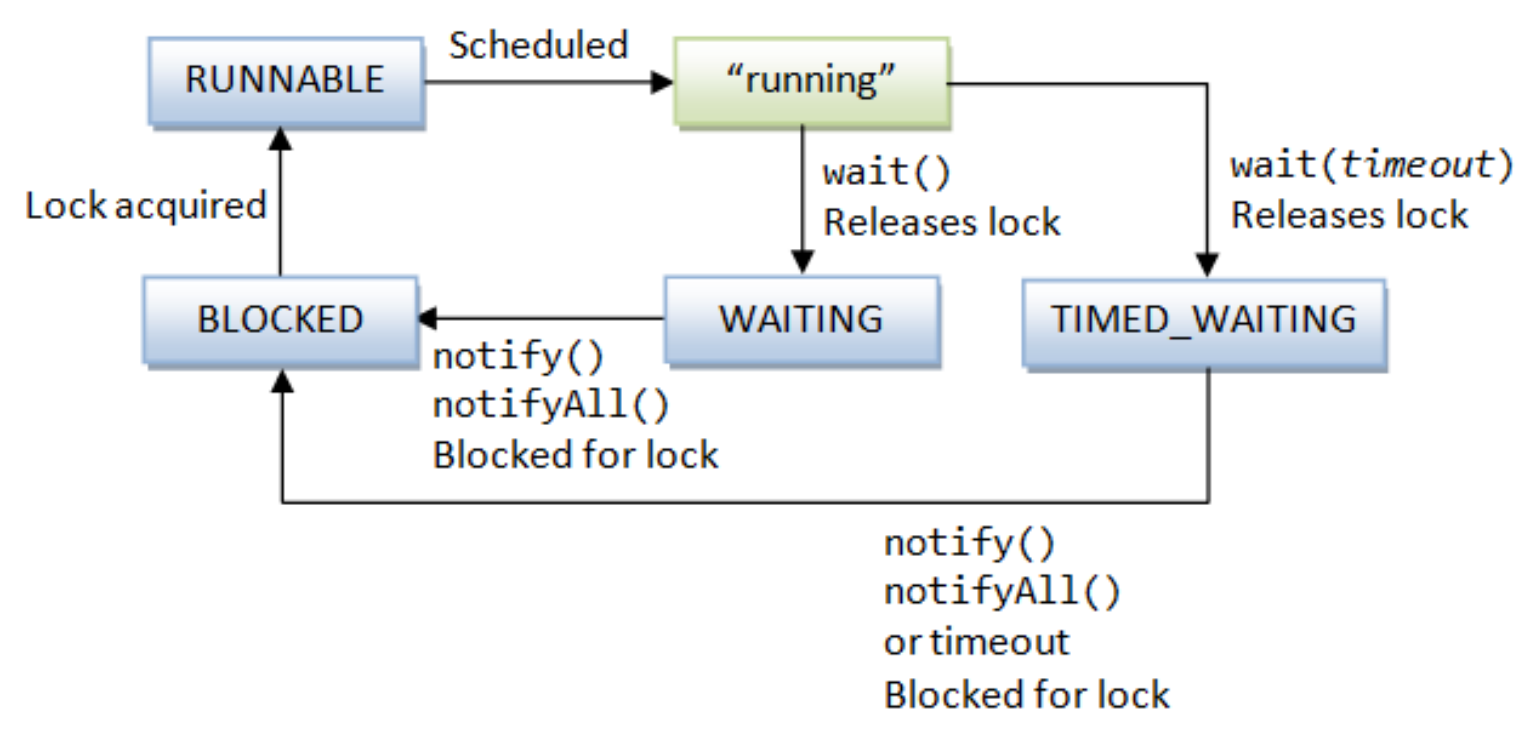
9.1. 쓰레드의 동기화 - Example

```
public class MainThreadExample {

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

        User1 user1 = new User1();
        user1.setCalculator(calculator);
        user1.start();

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

```
<terminated> MainThr
User1 : 100
User2 : 50
```

8.4 쓰레드의 실행제어

- notify : 일시 정지 상태의 다른 메서드를 실행 대기 상태로 만든다.
- wait : 스레드를 일시 정지 상태로 만든다.
- notifyAll : wait로 정지된 모든 스레드를 실행 대기 상태로 돌린다.



8.4 쓰레드의 실행제어

- notify,wait,notifyAll 예제(1)

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

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

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

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

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

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

8.4 쓰레드의 실행제어

- notify,wait,notifyAll 예제(1)

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

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

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

```java
public class WorkObject {
    public synchronized void methodA() {
        System.out.println("ThreadA의 methodA() 작업 실행");
        notify();
        try {wait();} catch (InterruptedException e) {}
    }

    public synchronized void methodB() {
        System.out.println("ThreadB의 methodB() 작업 실행");
        notify();
        try {wait();} catch (InterruptedException e) {}
    }
}
```

8.4 쓰레드의 실행제어

- notify,wait,notifyAll 예제(2)

```
public class DataBox {
    private String data;

    public synchronized String getData() {
        if(this.data == null) {
            try {wait();} catch(InterruptedException e) {}
        }
        String returnValue = data;
        System.out.println("ConsummerThread가 읽은 데이터: " + returnValue);
        data = null;
        notify();
        return returnValue;
    }

    public synchronized void setData(String data) {
        if(this.data != null) {
            try {wait();} catch(InterruptedException e) {}
        }
        this.data = data;
        System.out.println("ProducerThread가 생성한 데이터: " + data);
        notify();
    }
}
```

8.4 쓰레드의 실행제어

- notify,wait,notifyAll 예제(2)

```
public class ConsumerThread extends Thread {
    private DataBox dataBox;
    public ConsumerThread(DataBox dataBox) {this.dataBox = dataBox;}

    @Override
    public void run() {
        for(int i=1; i<=3; i++) {String data = dataBox.getData();}
    }
}
public class ProducerThread extends Thread {
    private DataBox dataBox;
    public ProducerThread(DataBox dataBox) {this.dataBox = dataBox;}

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

8.4 쓰레드의 실행제어

- notify,wait,notifyAll 예제(2)

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

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

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

8.5 쓰레드의 실행제어

- stop() : 스레드를 즉시 종료시킨다. 종료시키면서 사용 중이던 자원들이 불안전한 상태로 남겨지기 때문에 deprecated 되었다.

```
public class StopFlagExample {
    public static void main(String[] args)  {
        PrintThread1 printThread = new PrintThread1();
        printThread.start();
        try {Thread.sleep(1000);} catch (Exception e) {}
        printThread.setStop(true);
    }
public class PrintThread1 extends Thread {
    private boolean stop;
    public void setStop(boolean stop) {this.stop = stop;}

    public void run() {
        while(!stop) System.out.println("실행 중");
        System.out.println("자원 정리");
        System.out.println("실행 종료");
    }
```

8.5 쓰레드의 실행제어

- interrupt() : 스레드가 일시정지 상태에 있을 때 InterruptedException을 발생시켜 run메소드를 정상 종료 시킨다.

```
public class InterruptExample {
    public static void main(String[] args)  {
        Thread thread = new PrintThread2();
        thread.start();
        try {Thread.sleep(1000);} catch (InterruptedException e) {}
        thread.interrupt();
    }
}
```

8.5 쓰레드의 실행제어

- interrupt() : 스레드가 일시정지 상태에 있을 때 InterruptedException을 발생시켜 run메소드를 정상 종료 시킨다.

```
public class PrintThread2 extends Thread {
    public void run() {
        //how1 : sleep을 통해 일시정지 상태를 주는 방법
        /*try {
            while(true) {
                System.out.println("실행 중");
                Thread.sleep(1);
            }
        } catch(Exception e) {}*/

        //how2 : interrupted() 메소드를 이용하는 방법
        while(true) {
            System.out.println("실행 중");
            if(Thread.interrupted()) break;
        }

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

11. 데몬 스레드

- 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드
- 주 스레드가 종료되면 데몬 스레드는 강제적으로 종료된다.
- 스레드를 데몬으로 만들기 위해서는 주 스레드가 데몬이 될 스레드의 setDaemon(true)를 호출해주면 된다.
- setDaemon은 start()를 호출하기 전에 선언한다.

11. 데몬 스레드

- DaemonThread 예제

```
public class DaemonExample {
    public static void main(String[] args) {
        AutoSaveThread autoSaveThread = new AutoSaveThread();
        autoSaveThread.setDaemon(true);
        autoSaveThread.start();
        try {Thread.sleep(3000);} catch (InterruptedException e) {}
        System.out.println("메인 스레드 종료");
    }
public class AutoSaveThread extends Thread {
    public void save() {System.out.println("작업 내용을 저장함.");}
    @Override
    public void run() {
        while(true) {
            try {Thread.sleep(1000);} catch (InterruptedException e) {break;}
            save();
        }
    }
```

12.1 스레드 그룹 – 이름 얻기

- 스레드 그룹은 관련 스레드를 묶어서 관리할 목적으로 이용한다.
- 스레드는 반드시 하나의 스레드 그룹에 포함되는데, 명시적으로는 스레드 그룹에 포함시키지 않으면 기본적으로 자신을 생성한 스레드와 같은 스레드의 그룹에 속하게 된다.
- 현재 스레드가 속한 스레드 그룹의 이름을 얻기 위해선 아래와 같은 코드를 사용할 수 있다.

```
ThreadGroup group = Thread.currentThread().getThreadGroup();

String groupName = group.getName();
```

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

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

- 위에서 키는 스레드 객체, 값은 스레드의 상태 기록들을 갖고 있는 StackTraceElement[] 배열이다.

12.1 스레드 그룹 – 이름 얻기

- 예제

```
public class ThreadInfoExample {
    public static void main(String[] args) {
        AutoSaveThread autoSaveThread = new AutoSaveThread();
        autoSaveThread.setName("AutoSaveThread");
        autoSaveThread.setDaemon(true);
        autoSaveThread.start();

        Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
        Set<Thread> threads = map.keySet();
        for(Thread thread : threads) {
            System.out.println("Name: " + thread.getName() + ((thread.isDaemon())?"(데몬)":
            System.out.println("\t" + "소속그룹: " + thread.getThreadGroup().getName());
            System.out.println();
        }
    }
public class AutoSaveThread extends Thread {
    public void save() {System.out.println("작업 내용을 저장함.");}
    @Override
    public void run() {
        while(true) {
            try {Thread.sleep(1000);} catch (InterruptedException e) {break;}
            save();
        }
    }
```

12.1 스레드 그룹 – 이름 얻기

- 예제

```
Name: Signal Dispatcher(데몬)
        소속그룹: system

Name: AutoSaveThread(데몬)
        소속그룹: main

Name: main(주)
        소속그룹: main

Name: Attach Listener(데몬)
        소속그룹: system

Name: Finalizer(데몬)
        소속그룹: system

Name: Reference Handler(데몬)
        소속그룹: system
```

12.2 스레드 그룹 – 스레드 그룹 생성

- 명시적으로 스레드 그룹을 만들고 싶다면 아래 생성자 중 하나를 사용해 ThreadGroup 객체를 만들면 된다..

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

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.
- 새로운 스레드 그룹을 생성한 후, 이 그룹에 스레드를 포함시키려면 Thread 객체를 생성할 때 생성자 매개값으로 스레드 그룹을 지정하면 된다. 스레드 그룹을 매개값으로 갖는 Thread 생성자는 아래 4가지가 있다.

```
Thread t = new Thread(ThreadGroup group, Runnable target);

Thread t = new Thraed(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);
```

- ta

12.3 스레드 그룹 – 스레드 그룹 일괄 interrupt()

- ThreadGroup에서 제공하는 interrupt() 메소드를 이용하면 그룹 내에 포함된 모든 스레드들을 일괄 inturrupt 할 수 있다. 스레드 그룹이 interrupt() 메소드는 포함된 모든 스레드 interrupt() 메소드를 내부적으로 호출해 주기 때문이다.
- 스레드 그룹의 interrupt() 메소드는 소속된 스레드의 interrupt() 메소드를 호출만 할 뿐 개별 스레드에서 발생하는 interruptedException에 대한 예외 처리를 하지 않는다. 따라서 안전한 종료를 위해 개별 스레드가 예외 처리를 해야 한다.
- 아래는 ThreadGroup이 가지고 있는 주요 메소드 들이다.

메소드		설명
int	activeCount()	현재 그룹 및 하위 그룹에서 활동 중인 모든 스레드의 수를 리턴한다.
int	activeGroupCount()	현재 그룹에서 활동중인 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 종료상태가 되어야 한다.
boolean	isDestroyed()	현재 그룹이 삭제되었는지 여부를 리턴한다.
int	getMaxPriority()	현재 그룹에 포함된 스레드가 가질 수 있는 최대 우선순위를 리턴한다.

12.3 스레드 그룹 – 스레드 그룹 일괄 interrupt()

- 메서드 목록(계속).

메소드		설명
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 이름을 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 interrupt한다.

12.3 스레드 그룹 – 스레드 그룹 일괄 interrupt()

- 예제

```java
public class WorkThread extends Thread {
    public WorkThread(ThreadGroup threadGroup, String threadName) {
        super(threadGroup, threadName);
    }

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

스레드 그룹과 스레드 이름을 설정

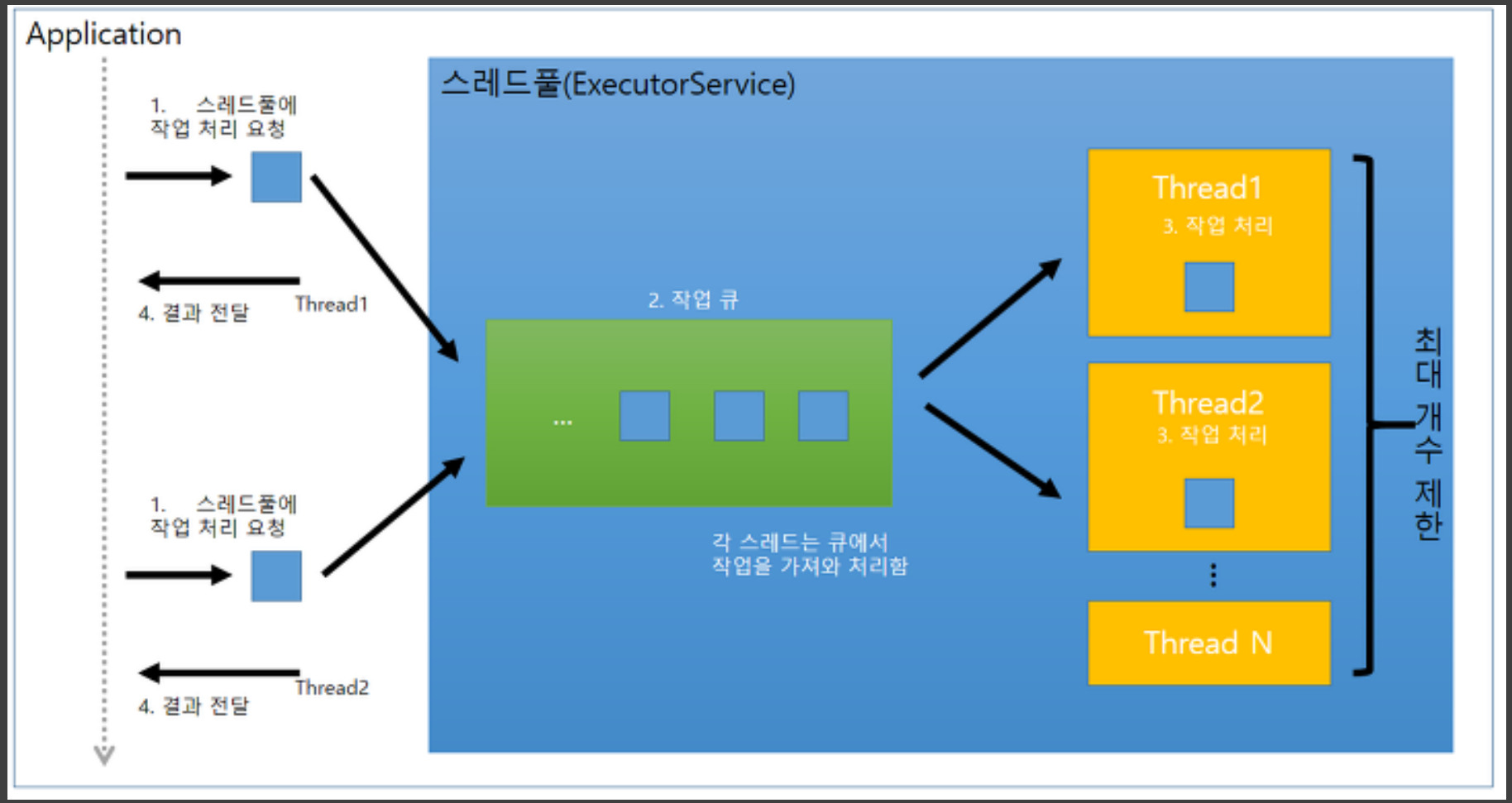
InterruptedException 발생 시 스레드 종료

12.3 스레드 그룹 – 스레드 그룹 일괄 interrupt()

- 예제

```
public class ThreadGroupExample {
    public static void main(String[] args) {
        ThreadGroup myGroup = new ThreadGroup("myGroup");
        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 ]");
        ThreadGroup mainGroup = Thread.currentThread().getThreadGroup();
        mainGroup.list();
        System.out.println();

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

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

13. 스레드 풀


Application
1. 스레드풀에 작업 처리 요청
4. 결과 전달
Thread1
1. 스레드풀에 작업 처리 요청
4. 결과 전달
Thread2
스레드풀(ExecutorService)
2. 작업 큐
...
각 스레드는 큐에서 작업을 가져와 처리함
Thread1
3. 작업 처리
Thread2
3. 작업 처리
Thread N
최대 개수 제한

13. 스레드 풀

- 병렬작업의 처리가 많아지면 스레드 개수가 증가되고 그에 따른 스레드 생성과 스케줄링으로 인해 CPU가 바빠져 메모리 사용량이 늘어난다. 전체적으로 어플리케이션 성능이 저하되는 효과를 가져온다.
- 갑작스런 스레드 폭증을 막으려면 스레드 폴(Thread Pool)을 사용해야 한다. 스레드 풀은 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐(Queue)에 들어오는 작업들을 하나씩 스레드가 맡아 처리한다.
- 자바는 스레드 풀을 생성하고 사용할 수 있도록 java.util.concurrent 패키지에서 ExecutorService 인터페이스와 Executors 클래스를 제공하고 있다
- Executors의 다양한 정적 메소드를 이용해서 ExecutorService 구현 객체를 만드는데, 이것이 바로 스레드풀이다.

13.1 스레드 풀 – 생성 및 종료

- ExecutorService 구현 객체 생성 방법

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

- 두개의 메서드를 통해 ExcutorService 구현 객체를 얻는 방식은 아래와 같다.

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

13.1 스레드 풀 – 생성 및 종료

- newCachedThreadPool()과 newFixedThreadPool() 메소드를 사용하지 않고 코어 스레드 개수와 최대 스레드 개수를 설정하고 싶다면 직접 ThreadPoolExecutor 객체를 생성해서 리턴한다..

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

13.1 스레드 풀 – 생성 및 종료

- 스레드 폴의 스레드는 데몬 스레드가 아니기 때문에 main 스레드가 종료되더라도 작업을 처리하기 위해 계속 실행 상태로 남아있다.
- 애플리케이션을 종료하려면 스레드폴을 종료시켜 스레드들이 종료 상태가 되도록 처리해주어야 한다.

리턴타입	메소드명 (매개변수)	설명
void	shutdown()	현재 처리 중인 작업뿐만 아니라 작업 큐에 대기하고 있는 모든 작업을 처리한 후에 스레드풀을 종료시킨다.
List(Runnable)	shutdownNow()	현재 작업 처리 중인 스레드를 interrupt해서 작업 중지를 시도하고 스레드풀을 종료시킨다. 리턴값은 작업 큐에 있는 미처리된 작업(Runnable)의 목록이다.
boolean	awaitTermination(long timeout, TimeUnit unit)	shutdown() 메소드 호출 이후, 모든 작업 처리를 timieout 시간 내에 완료하면 true를 리턴하고, 못하면 작업 처리 중인 스레드를 interrupt하고 false를 리턴한다.

- 남아있는 작업을 무시하고 스레드 풀을 종료할 때에는 shutdown()을 일반적으로 호출하고, 남아있는 작업과는 상관없이 강제로 호출할 때에는 shutdownNow()를 호출한다.

13.2 스레드 풀 – 작업 생성과 처리 요청

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

Runnable 구현 클래스	Callable 구현 클래스

Runnable task = new Runnable( ) {
  @Override
  public void run ( ) {
  . . .
  }
}
	Callable<T> task = new Callable<T> ( ) {
	  @Override
	  public T call( ) throws Exception {
	  . . .
	  return 0;
	  }
	}

13.2 스레드 풀 – 작업 생성과 처리 요청

- 작업 처리 요청이란 ExecutorService의 작업 큐에 Runnable 또는 Callable 객체를 넣는 행위를 말한다.
- ExecutorService는 작업 처리 요청을 위해 다음 두 가지 종류의 메소드를 제공한다.

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

13.2 스레드 풀 – 작업 생성과 처리 요청

- 예제.

```java
public class ExecuteExample {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        for(int i=0; i<10 ; i++){
            Runnable runnable = new Runnable(){
                @Override
                public void run() {
                    ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor) executorService;
                    int poolSize = threadPoolExecutor.getPoolSize();
                    String threadName = Thread.currentThread().getName();
                    System.out.println("[총 스레드 개수:"+poolSize+"] 작업 스레드 이름: "+threadName);
                    int value = Integer.parseInt("삼");
                }
            };
            executorService.execute(runnable);
            //executorService.submit(runnable);

            try {Thread.sleep(10);} catch (InterruptedException e) {}
        }
        executorService.shutdown();
    }
}
```

13.2 스레드 풀 – 작업 생성과 처리 요청

- 예제(execute로 실행 시)

```
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-4
Exception in thread "pool-1-thread-4" java.lang.NumberFormatException: For input string: "삼"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at sec08.exam01_threadgroup.ExecuteExample$1.run(ExecuteExample.java:20)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-5
Exception in thread "pool-1-thread-5" java.lang.NumberFormatException: For input string: "삼"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at sec08.exam01_threadgroup.ExecuteExample$1.run(ExecuteExample.java:20)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
```

13.2 스레드 풀 – 작업 생성과 처리 요청

- 예제(submit으로 실행 시)

```
[총 스레드 개수:1] 작업 스레드 이름: pool-1-thread-1
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-2
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-1
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-2
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-1
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-2
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-1
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-2
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-1
[총 스레드 개수:2] 작업 스레드 이름: pool-1-thread-2
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- ExecutorService의 submit() 메소드는 매개값으로 준 Runnable 또는 Callable 작업을 스레드 풀의 작업 큐에 저장하고 즉시 Future 객체를 리턴한다.

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

- Future 객체는 작업 결과가 아니라 작업이 완료될 때까지 기다렸다가(지연했다가=블로킹되었다가) 최종 결과를 얻는데 사용된다. 그래서 Future를 지연 완료 객체라고 한다.
- Future의 get() 메소드를 호출하면 스레드가 작업을 완료할 때 까지 블로킹 되었다가 작업을 완료하면 처리 결과를 리턴한다.

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

- 리턴 타입인 V는 sumbit(Runnable task, V result)의 두 번째 매개값인 V타입이거나 submit(Callable<V> task)의 Callable 타입 파라미터 V 타입이다.

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 다음은 세 가지 submit() 메소드별로 Future의 get() 메소드가 리턴하는 값이 무엇인지 보여준다.

메소드	작업 처리 완료 후 리턴 타입	작업처리 도중 예외 발생
submit(Runnable task)	future.get()->null;	future.get()->예외 발생
submit(Runnable task, Integer result)	future.get()->int 타입 값;	future.get()->예외 발생
submit(Callable<String> task)	future.get()->String 타입 값;	future.get()->예외 발생

- Future를 이용한 블로킹 방식의 작업 완료 통보에서 주의할 점은 작업을 처리하는 스레드가 작업을 완료하기 전까지는 get() 메소드가 블로킹 되므로 다른 코드를 실행할 수 없다.
- 만약 UI를 변경하고 이벤트를 처리하는 스레드가 get() 메소드를 호출하면 작업을 완료하기 전까지 UI를 변경할 수도 없고 이벤트를 처리할 수도 없게 된다.
- 그렇기 때문에 get() 메소드를 호출하는 스레드는 새로운 스레드이거나 스레드 풀의 또 다른 스레드가 되어야 한다.

새로운 스레드 생성해서 호출	스레드풀의 스레드가 호출

Runnable task = new Runnable( ) {
  @Override
  public void run ( ) {
  . . .
  }
}
	Callable<T> task = new Callable<T> ( ) {
	  @Override
	  public T call( ) throws Exception {
	  . . .
	  return 0;
	  }
	}

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

새로운 스레드 생성해서 호출	스레드풀의 스레드가 호출

new Thread(new Runnable( ) {
@Runnable
public void run ( ) {
try {
future.get( );
} catch (Exception e) {}
}
}).start();
	executorService.submit(new Runnable( ) {
	@Runnable
	public void run ( ) {
	try {
	future.get( );
	} catch (Exception e) {}
	}
	});

- Future 객체는 작업 결과를 얻기 위한 get() 메소드 이외에도 다음과 같은 메소드를 제공한다.

리턴타입	메소드명(매개변수)	설명
boolean	cancel(boolean mayInterruptIfRunning)	작업 처리가 진행 중일 경우 취소시킴
boolean	isCanceled()	작업이 취소되었는지 여부
boolean	isDone()	작업 처리가 완료되었는지 여부

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 리턴값이 없는 작업 완료 통보.

```
public class NoResultExample {
    public static void main(String[] args) {
        ExecutorService  executorService = Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors()
        );

        System.out.println("[작업 처리 요청]");
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                int sum = 0;
                for(int i=1; i<=10; i++) {
                    sum += i;
                }
                System.out.println("[처리 결과] " + sum);
            }
        };
        Future future = executorService.submit(runnable);
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 리턴값이 없는 작업 완료 통보.

```
        try {
            future.get();
            System.out.println("[작업 처리 완료]");
        } catch(Exception e) {
            System.out.println("[실행 예외 발생함] " + e.getMessage());
        }

        executorService.shutdown();
    }
}
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 리턴값이 있는 작업 완료 통보.

```java
public class ResultByCallableExample {
    public static void main(String[] args) {
        ExecutorService  executorService = Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors()
        );

        System.out.println("[작업 처리 요청]");
        Callable<Integer> task = new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                int sum = 0;
                for(int i=1; i<=10; i++) {
                    sum += i;
                }
                return sum;
            }
        };
        Future<Integer> future = executorService.submit(task);
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 리턴값이 있는 작업 완료 통보.

```
        try {
            int sum = future.get();
            System.out.println("[처리 결과] " + sum);
            System.out.println("[작업 처리 완료]");
        } catch (Exception e) {
            System.out.println("[실행 예외 발생함] " + e.getMessage());
        }

        executorService.shutdown();
    }
}
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 작업 처리 결과를 외부 객체에 저장.

```
public class ResultByRunnableExample {
    public static void main(String[] args) {
        ExecutorService  executorService = Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors()
        );

        System.out.println("[작업 처리 요청]");
        class Task implements Runnable {
            Result result;
            Task(Result result) {
                this.result = result;
            }
            @Override
            public void run() {
                int sum = 0;
                for(int i=1; i<=10; i++) {
                    sum += i;
                }
                result.addValue(sum);
            }
        }
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 작업 처리 결과를 외부 객체에 저장.

```
        Result result = new Result();
        Runnable task1 = new Task(result);
        Runnable task2 = new Task(result);
        Future<Result> future1 = executorService.submit(task1, result);
        Future<Result> future2 = executorService.submit(task2, result);

        try {
            result = future1.get();
            result = future2.get();
            System.out.println("[처리 결과] " + result.accumValue);
            System.out.println("[작업 처리 완료]");
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("[실행 예외 발생함] " + e.getMessage());
        }

        executorService.shutdown();
    }
}

class Result {
    int accumValue;
    synchronized void addValue(int value) {
        accumValue += value;
    }
}
```

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 스레드 폴에서 작업 처리가 완료된 것만 통보받는 방법이 있다.
- CompletionService는 처리 완료된 작업을 가져오는 poll과 take 메소드를 제공한다.

리턴타입	메소드명(매개변수)	설명
Future<V>	poll()	완료된 작업의 Future를 가져옴. 완료된 작업이 없다면 즉시 null을 리턴
Future<V>	poll(long timeout, TimeUnit unit)	완료된 작업의 Future를 가져옴. 완료된 작업이 없다면 timeout까지 블로킹됨
Future<V>	take()	완료된 작업의 Future를 가져옴. 완료된 작업이 없다면 있을때까지 블로킹됨
Future<V>	submit(Callable<V> task)	스레드풀에 Callable 작업 처리 요청
Future<V>	submit(Runnable task, V result)	스레드풀에 Runnable 작업 처리 요청

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 작업 완료 순으로 통보받기.

```java
public class CompletionServiceExample extends Thread {
    public static void main(String[] args) {
        ExecutorService  executorService = Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors()
        );

        CompletionService<Integer> completionService  =
                new ExecutorCompletionService<Integer>(executorService);

        System.out.println("[작업 처리 요청]");
        for(int i=0; i<3; i++) {
            completionService.submit(new Callable<Integer>() {
                @Override
                public Integer call() throws Exception {
                    int sum = 0;
                    for(int i=1; i<=10; i++) {
                        sum += i;
                    }
                    return sum;
                }
            });
        }
```

CompletionService 생성

스레드 풀에게 작업처리 요청

13.3 스레드 풀 – 블로킹 방식의 작업 완료 통보

- 작업 완료 순으로 통보받기.

```
        System.out.println("[처리 완료된 작업 확인]");
        executorService.submit(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    try {
                        Future<Integer> future = completionService.take();
                        int value = future.get();
                        System.out.println("[처리 결과] " + value);
                    } catch (Exception e) {
                        break;
                    }
                }
            }
        });

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

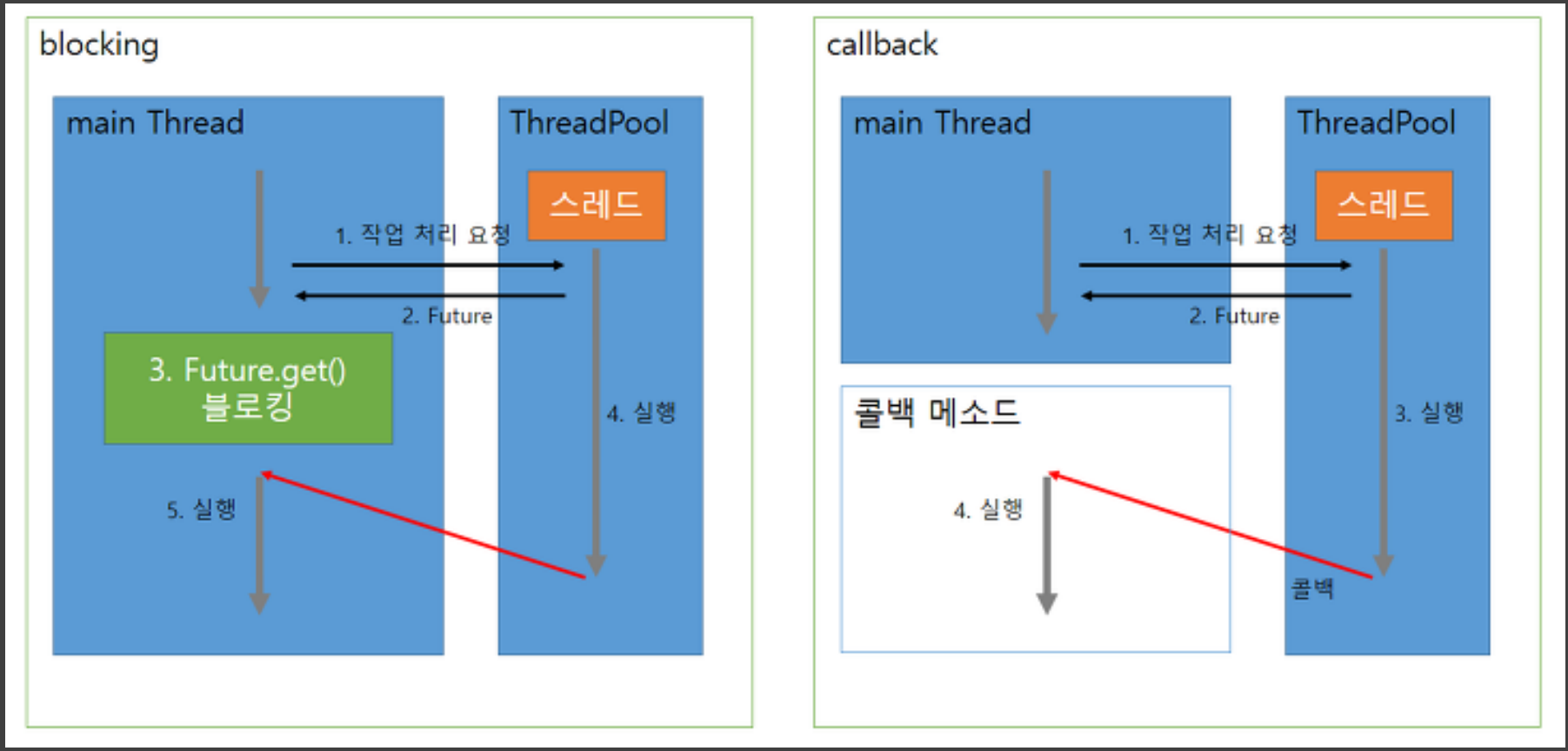
스레드 풀의 스레드에서 실행하도록 함

완료된 작업 가져오기

3초뒤 스레드 종료

```
[작업 처리 요청]
[처리 완료된 작업 확인]
[처리 결과] 55
[처리 결과] 55
[처리 결과] 55
```

13.4 스레드 풀 – 콜백 방식의 작업 완료 통보



13.4 스레드 풀 – 콜백 방식의 작업 완료 통보

- 블로킹 방식은 작업 처리를 요청한 후 작업이 완료될 때까지 블로킹되지만, 콜백 방식은 작업 처리를 요청한 후 결과를 기다릴 필요 없이 다른 기능을 수행할 수 있다.
- ExecutorService는 콜백을 위한 별도의 기능을 제공하지 않지만, Runnable 구현 클래스를 작성할 때 콜백 기능을 구현할 수 있다.
- 콜백 메서드를 가진 클래스를 정의해야 하는데 직접 만들어 정의해도 되고 java.nio.channals.CompletionHandler를 이용할 수 있다.

```
CompletionHandler<V, A> callback = new CompletionHandler<V, A>() {
    @Override
    public void completed(V result, A attachment){
        //작업 정상처리 시 호출되는 콜백 메서드 내용
    }
    @Override
    public void failed(Throwable exc, A attachment){
        //작업 처리 도중 예외가 발생했을 때 호출되는 콜백 메서드 내용
    }
}
```

13.4 스레드 풀 – 콜백 방식의 작업 완료 통보

- V : 결과값의 타입, A : 첨부값의 타입
- Runnable 객체 안에서는 다음과 같이 사용할 수 있다.

```
Runnable task = new Runnable() {
    @Override
    public void run() {
        try {
            //작업처리
            V result = ..;
            callback.completed(result, null);
        }catch(Exception e) {
            callback.failed(e, null);
        }
    }
}
```

13.4 스레드 풀 – 콜백 방식의 작업 완료 통보

- 예제.

```
public class CallbackExample {
    private ExecutorService executorService;

    public CallbackExample() {
        executorService = Executors.newFixedThreadPool(
            Runtime.getRuntime().availableProcessors()
        );
    }

    private CompletionHandler<Integer, Void> callback =
            new CompletionHandler<Integer, Void>() {
        @Override
        public void completed(Integer result, Void attachment) {
            System.out.println("completed() 실행: " + result);
        }

        @Override
        public void failed(Throwable exc, Void attachment) {
            System.out.println("failed() 실행: " + exc.toString());
        }
    };
```

콜백 메서드를 가진 CompletionHandler 생성

13.4 스레드 풀 – 콜백 방식의 작업 완료 통보

- 예제.

```
public void doWork(final String x, final String y) {
    Runnable task = new Runnable() {
        @Override
        public void run() {
            try {
                int intX = Integer.parseInt(x);
                int intY = Integer.parseInt(y);
                int result = intX + intY;
                callback.completed(result, null);
            } catch(NumberFormatException e) {
                callback.failed(e, null);
            }
        }
    };
    executorService.submit(task);
}

public void finish() {
    executorService.shutdown();
}

public static void main(String[] args) {
    CallbackExample example = new CallbackExample();
    example.doWork("3", "3");
    example.doWork("3", "삼");
    example.finish();
}
```

정상처리 호출

예외처리 호출

스레드 풀에게 작업 처리 요청

스레드풀 종료