Java 프로그래밍

<싱글 톤>
클래스의 인스턴스를 오직 하나만 생성하고 이에 접근할 수 있는 전역적인 접근점을 제공하는 패턴
어디서든지 동일한 인스턴스에 접근할 수 있으므로, 리소스를 공유하거나 상태를 유지해야 하는
경우 유용합니다. 또한 Singleton 패턴은 전역 상태를 관리할 때도 사용될 수 있습니다.

```
package com.day11;

import java.util.HashMap;

// p.447 - HashMap 사용

class Car{
	String name;
	public Car(){}
	public Car(String name) {
		this.name=name;
	}
}

class CarFactory{
	private static CarFactory instance = new CarFactory();
	HashMap<String, Car> carMap = new HashMap<>();
	public static CarFactory getInstance() {
		if(instance == null) {
			instance = new CarFactory();
		}
		return instance;
	}

	public Car createCar(String name) {
			if(carMap.containsKey(name)) {
				return carMap.get(name);
			}
			Car car = new Car();
			carMap.put(name, car);
			return car;
	}

}

public class CarTest {

	public static void main(String[] args) {
		CarFactory factory = CarFactory.getInstance();
		// CarFactory factory = new CarFactory(); - 기존에 만들었던 방식(싱글 톤)
		Car sonata1 = factory.createCar("연수 차");
		Car sonata2 = factory.createCar("연수 차");
		System.out.println(sonata1 == sonata2); // true
```

```
                Car avante1 = factory.createCar("승연 차");
                Car avante2 = factory.createCar("승연 차");
                System.out.println(avante1 == avante2); // true
                System.out.println(sonata1 == avante1); // false
        }

}
```

<직렬화 - Serialize>
시스템 내부에서 사용되는 객체 또는 데이터를 외부의 자바 시스템에서도 사용할 수 있도록 바이트(byte) 형태로 데이터 변환하는 것이 직렬화이다.

```
package com.day11;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

// p.551 직렬화(Serialize)

class Person implements Serializable{ // 직렬화로 Object를 String형태로 만들어 줌.
        private String name;
        private String job;

        public Person(String name, String job) {
                this.name = name;
                this.job = job;
        }

        public Person() {

        }

        @Override
        public String toString() {
                return "name=" + name + ", job=" + job;
        }
}

public class SerializeTest {

        public static void main(String[] args) {
                Person personAhn = new Person("안재용", "대표이사");
                Person personKim = new Person("김철수", "상무이사");
                try(FileOutputStream fos = new FileOutputStream("serial.out");
```

```
                ObjectOutputStream oos = new ObjectOutputStream(fos)){
                oos.writeObject(personAhn);
                oos.writeObject(personKim);

        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }

        ////
        try(FileInputStream fis = new FileInputStream("serial.out");
                ObjectInputStream ois = new ObjectInputStream(fis)){
                Person p1 = (Person) ois.readObject();
                Person p2 = (Person) ois.readObject();
                System.out.println(p1);
                System.out.println(p2);

        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        }


    }

}

package com.day11;

import java.io.File;
import java.io.IOException;
// p.557
public class FileTest {

    public static void main(String[] args) throws IOException {
        File file = new File("src\\com\\day11\\FileTest.txt");
        file.createNewFile();
        System.out.println(file.isFile()); // 파일인가?
        System.out.println(file.isDirectory()); // 디렉토리인가?
        System.out.println(file.getName()); // 파일의 이름
        System.out.println(file.getAbsolutePath()); // 파일의 절대경로
        System.out.println(file.getPath()); // 파일의 경로
        System.out.println(file.canRead()); // 읽을 수 있는가?
        System.out.println(file.canWrite()); // 쓸수 있는가?
        System.out.println(file.delete()); // 파일 삭제
    }
```

}

<Thread>
Thread는 프로그램 또는 프로세스 안에서 실행되는 독립적인 실행 흐름을 말합니다. 즉, 하나의 프로세스 안에서 여러 개의 스레드가 동시에 실행될 수 있습니다. 각각의 스레드는 독립적으로 실행되며, 서로 다른 작업을 수행할 수 있습니다.

스레드를 사용하면 여러 작업을 동시에 처리할 수 있어 프로그램의 성능을 향상시킬 수 있습니다. 예를 들어, UI 스레드에서 사용자 인터페이스를 업데이트하고, 백그라운드 스레드에서 네트워크 요청을 처리하는 등의 작업을 분리하여 동시에 처리할 수 있습니다.

```
package com.day11;

public class ThreadGugu01 extends Thread {
	private int dan;
	public ThreadGugu01(int dan) {
		this.dan=dan;
	}

	public void run() {
		int i;
		for (i = 1; i < 10; i++) {
			System.out.println(dan + "*" + i + " = " + dan * i);
			System.out.println();
			try {
				Thread.sleep(1000);
			}catch (InterruptedException e) {
				e.printStackTrace();
			}

		}
	}

	public static void main(String[] args) {
		ThreadGugu01 tg1 = new ThreadGugu01(5);
		ThreadGugu01 tg2 = new ThreadGugu01(7);
		ThreadGugu01 tg3 = new ThreadGugu01(9);

		tg1.start();
		tg2.start();
		tg3.start();

	}

}
```

가장 작은 실행단위가 Thread
무엇이 먼저 실행될지는 JVM에서 준비해서 실행한다.
start로 run이 호출이 되고, run이 자동 실행되어서 동작이 된다.

<Runnable 구현하여 Thread 생성>

```
package com.day11;

public class ThreadGugu02 implements Runnable{
	// Thread를 Interface화 한 것 - Runnable
	private int dan;
	public ThreadGugu02(int dan) {
		this.dan=dan;
	}

	@Override
	public void run() {
		int i;
		for (i = 1; i < 10; i++) {
			System.out.println(dan + "*" + i + " = " + dan * i);
		}

	}
	public static void main(String[] args) {
	ThreadGugu02 th1 = new ThreadGugu02(1);
	ThreadGugu02 th2 = new ThreadGugu02(2);
	ThreadGugu02 th3 = new ThreadGugu02(3);
	Thread t1 = new Thread(th1);
	t1.start();
	Thread t2 = new Thread(th2);
	t2.start();
	Thread t3 = new Thread(th3);
	t3.start();

	}
}

package com.day11;

// Thread 상속 받아 구현하기
class Saram1 extends Thread{
	private String name;
	public Saram1(String name) {
		this.name=name;
	}
	public void say() {
		for (int i = 1; i < 6; i++) {
			System.out.println(name + "이 " + i + "번째 말한다.");
		}
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
```

```
	}
	@Override
	public void run() {
		say();
	}

}



public class ThreadSpeak01{
	public static void main(String[] args) {
		Saram1 s1 = new Saram1("홍길동");
		Saram1 s2 = new Saram1("이순신");
		Saram1 s3 = new Saram1("강감찬");

		s1.start();
		s2.start();
		s3.start();

	}



}

package com.day11;

// Runnable 사용
class Saram2 implements Runnable {
	private String name;

	public Saram2(String name) {
		this.name = name;
	}

	@Override
	public void run() {
		for (int i = 1; i < 6; i++) {
			System.out.println(name + "이 " + i + "번째 말한다.");
		}
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

```
public class ThreadSaram02 {

	public static void main(String[] args) {
		Saram2 s1 = new Saram2("홍길동");
		Saram2 s2 = new Saram2("이순신");
		Saram2 s3 = new Saram2("강감찬");

		new Thread(s1).start();
		new Thread(s2).start();
		new Thread(s3).start();
		// Thread t3 = new Thread(s3);
		// t3.start(); 와 같음
	}

}
```

<Synchronized - 스레드 동기화>
스레드 동기화는 여러 개의 스레드가 공유된 리소스에 동시에 접근할 때 발생할 수 있는 문제를 해결하기 위한 메커니즘입니다. 동기화를 통해 스레드 간의 순서와 일관성을 유지하고, 경쟁 조건과 데이터 불일치 문제를 방지할 수 있습니다.

```
package com.day11;

// 학생
class Student extends Thread {
	private String name;
	private SharedBoard board;

	public Student(String name, SharedBoard board) {
		this.name = name;
		this.board = board;
	}

	@Override
	public void run() {
		for (int i = 0; i < 10; i++) { // board 10번 접근 카운팅
			board.add();
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}

// 공유 게시판
class SharedBoard {
	private int sum;
	synchronized public void add() { // 10씩 증가
```

```
                int n = sum;
                n += 10;
                sum = n;
                System.out.println(Thread.currentThread().getName() + ":" + sum);
        }

}

public class SynchronizedEx {

        public static void main(String[] args) {
                SharedBoard board = new SharedBoard(); // 공유 자원
                Thread th1 = new Student("홍길동", board);
                Thread th2 = new Student("이순신", board);
                th1.start();
                th2.start();
        }

}
```

<Lamda식>
람다 표현식(Lambda Expression)은 메서드를 간결하게 표현하는 방식
익명 함수(Anonymous Function)를 이용하여 사용한다.

```
package com.day11;

public class LamdaThreadTest {

        public static void main(String[] args) {
                //Runnable runnable = new Runnable();//오류발생
                Runnable  runnable =  new Runnable() {
                        @Override
                        public void run() {
                                System.out.println(Thread.currentThread().getName());
                        }
                };
                new Thread(runnable).start();
                //람다표현식
                Runnable runnable2 =
()->System.out.println(Thread.currentThread().getName());
                 Thread th2 = new Thread(runnable2);
                 th2.start();

                 Runnable  runnable3 =
()->System.out.println(Thread.currentThread().getName());
                 new Thread(runnable3).start();
        }

}
```