# 2. 스레드 생성과 실행

#0.강의/1.자바로드맵/5.자바-고급1편



## 프로젝트 환경 구성

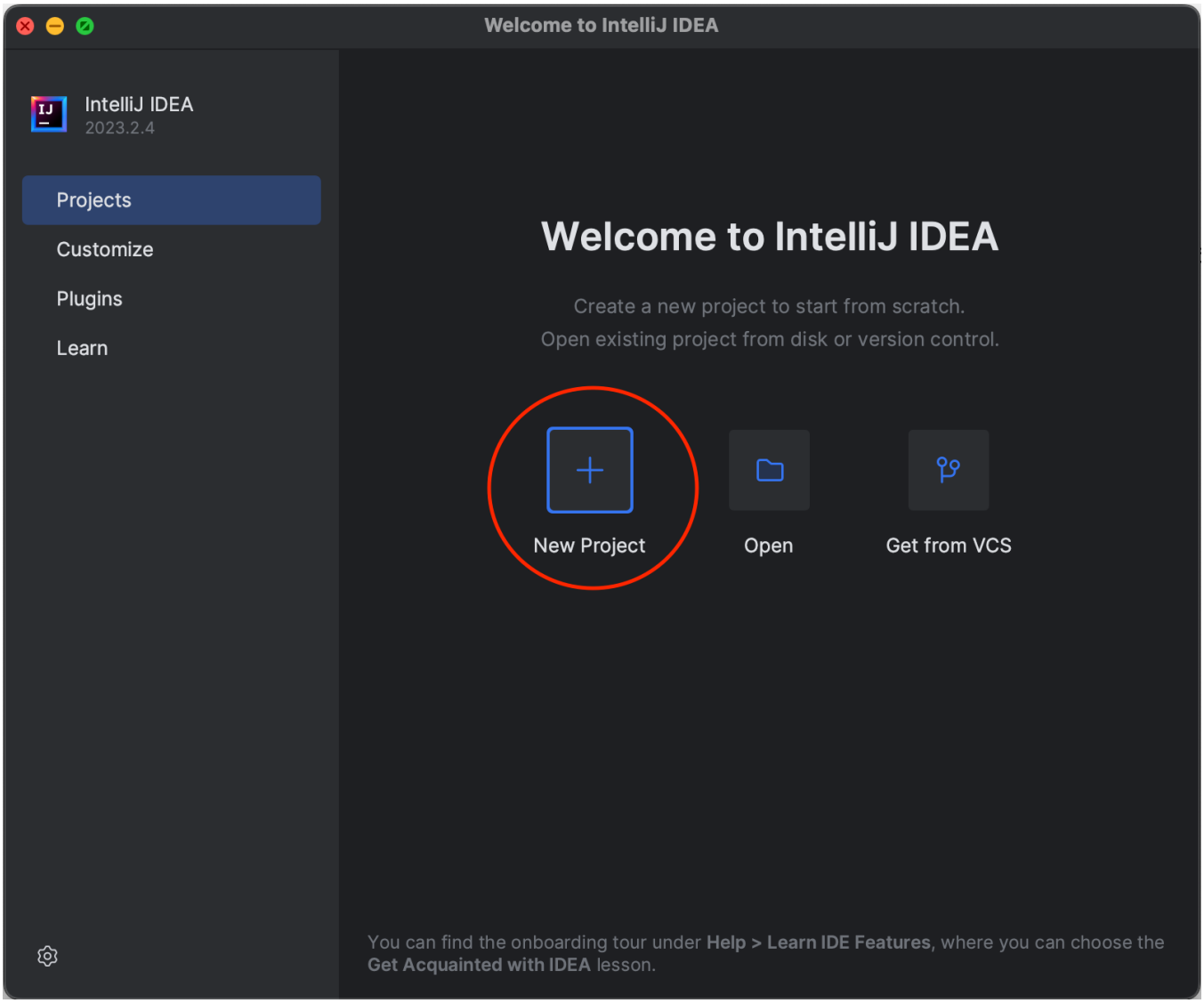
자바 입문편에서 인텔리제이 설치, 선택 이유 설명

프로젝트 환경 구성에 대한 자세한 내용은 자바 입문편 참고

여기서는 입문편을 들었다는 가정하에 설정 진행

### 인텔리제이 실행하기

**New Project**



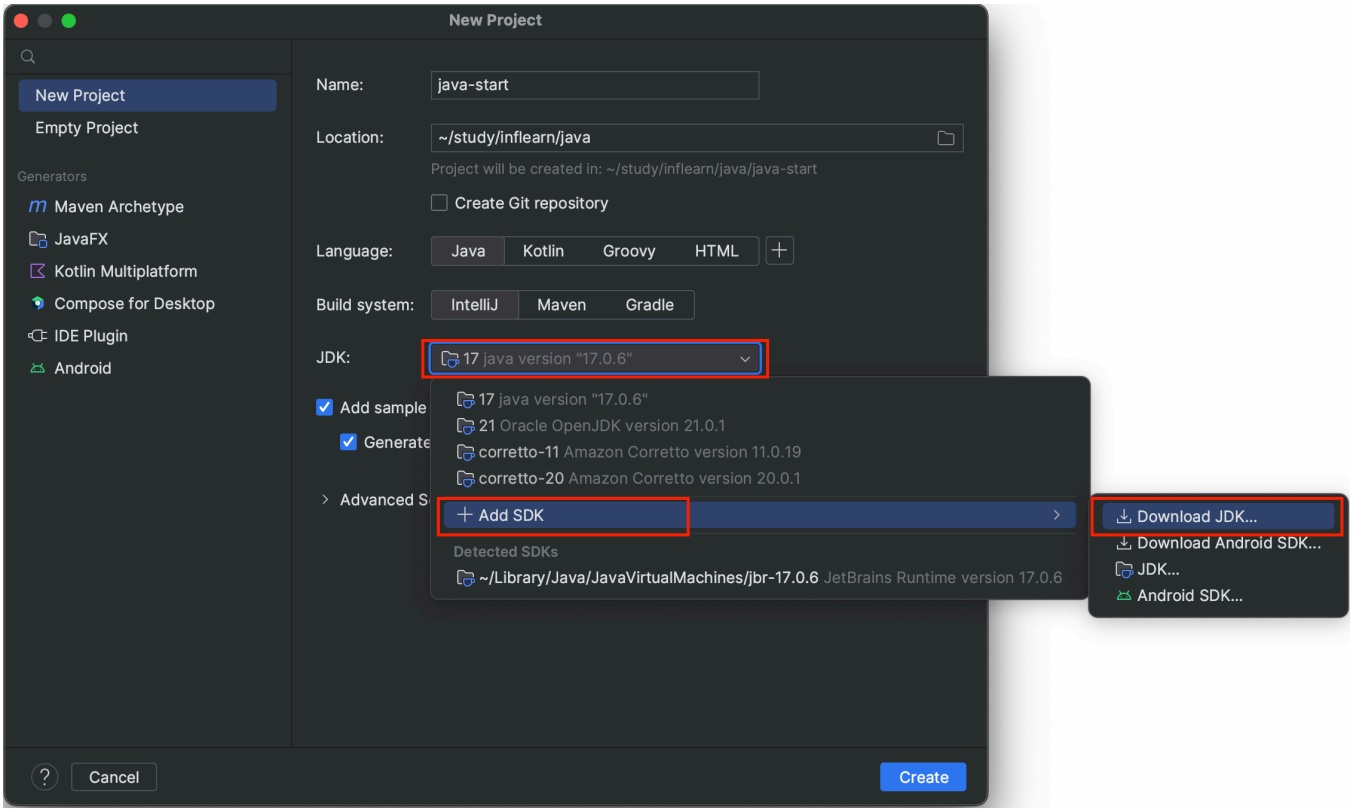

- New Project를 선택해서 새로운 프로젝트를 만들자

### New Project 화면

- Name:
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: java-mid1
  - 자바 중급2편 강의: **java-mid2**
  - 자바 고급1편 강의: **java-adv1**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: **자바 버전 21 이상(주의!)**
- `Add sample code` 선택

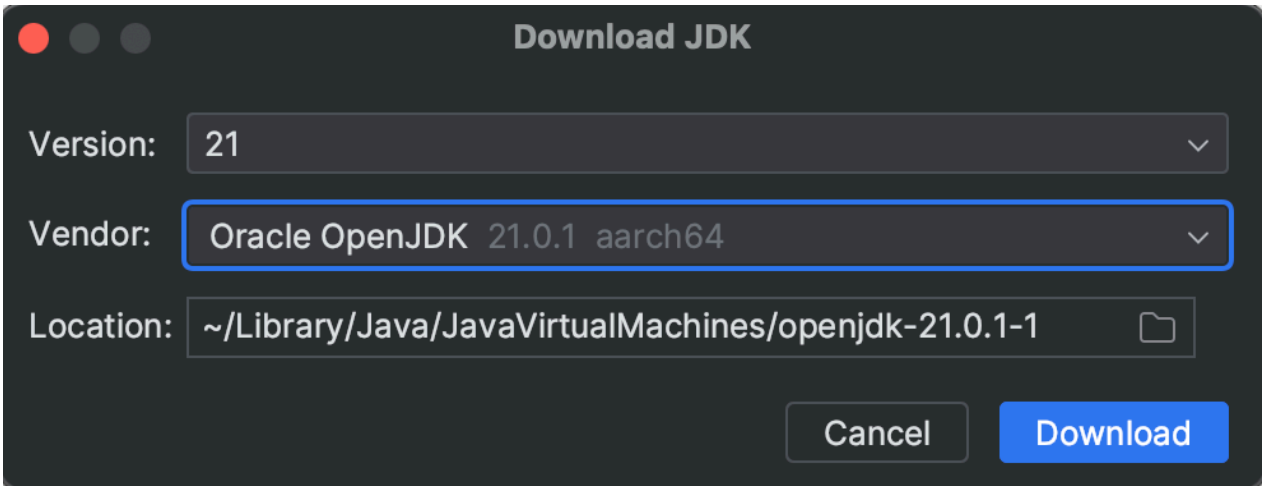
### JDK 다운로드 화면 이동 방법

자바로 개발하기 위해서는 JDK가 필요하다. JDK는 자바 프로그래머를 위한 도구 + 자바 실행 프로그램의 묶음이다.




- **Name:**
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: java-mid1
  - 자바 중급2편 강의: **java-mid2**
  - 자바 고급1편 강의: **java-adv1**

### JDK 다운로드 화면

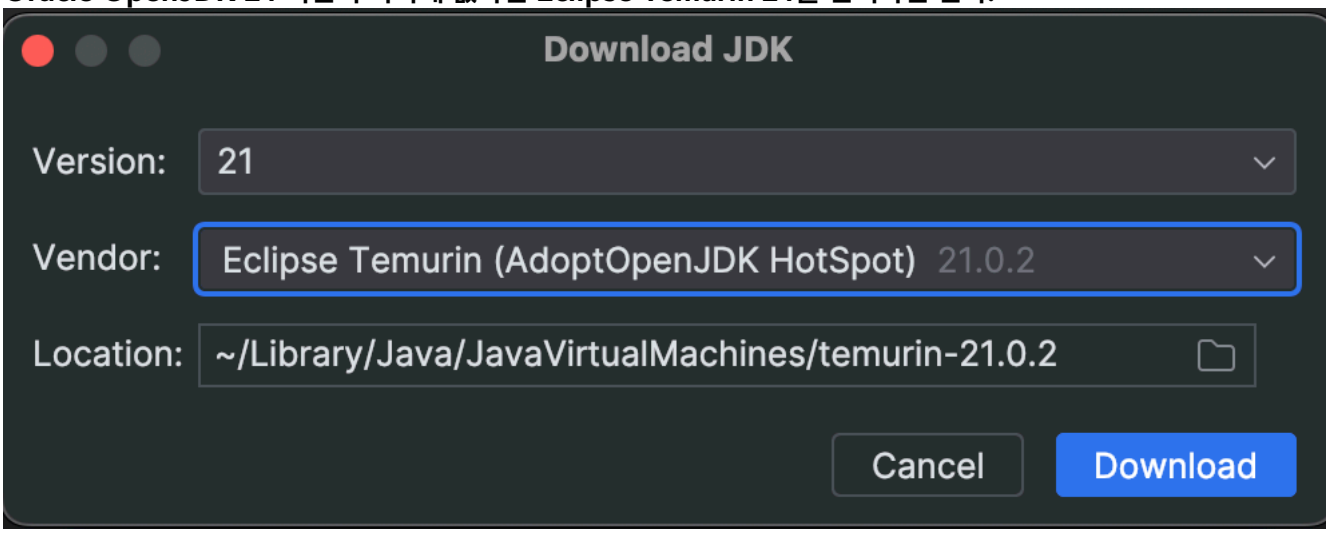



- Version: 21을 선택하자.

- Vendor: Oracle OpenJDK를 선택하자. 없다면 다른 것을 선택해도 된다.
  - aarch64: 애플 M1, M2, M3 CPU 사용시 선택, 나머지는 뒤에 이런 코드가 붙지 않은 JDK를 선택하면 된다.
- Location: JDK 설치 위치, 기본값을 사용하자.

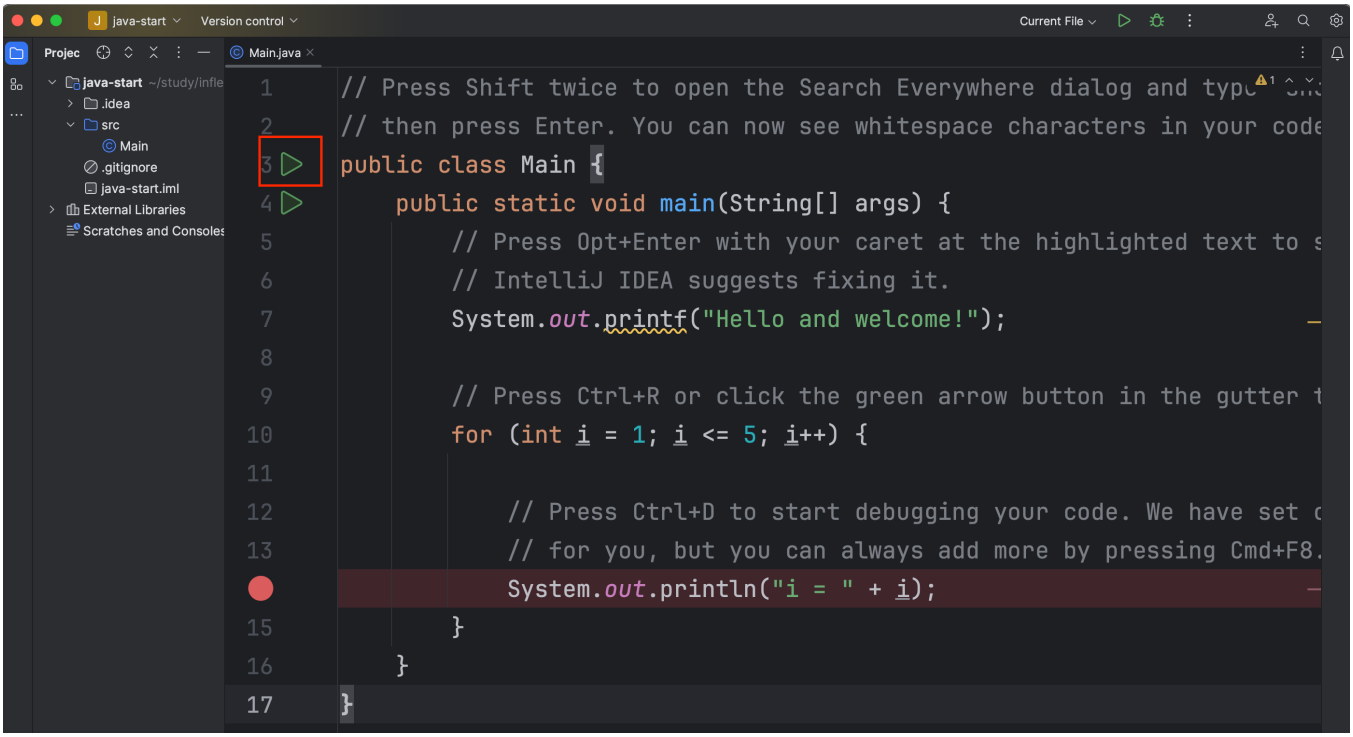
**주의 - 변경 사항**

**Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.**




`Download` 버튼을 통해서 다운로드 JDK를 다운로드 받는다.
다운로드가 완료 되고 이전 화면으로 돌아가면 `Create` 버튼 선택하자. 그러면 다음 IntelliJ 메인 화면으로 넘어간다.

**IntelliJ 메인 화면**

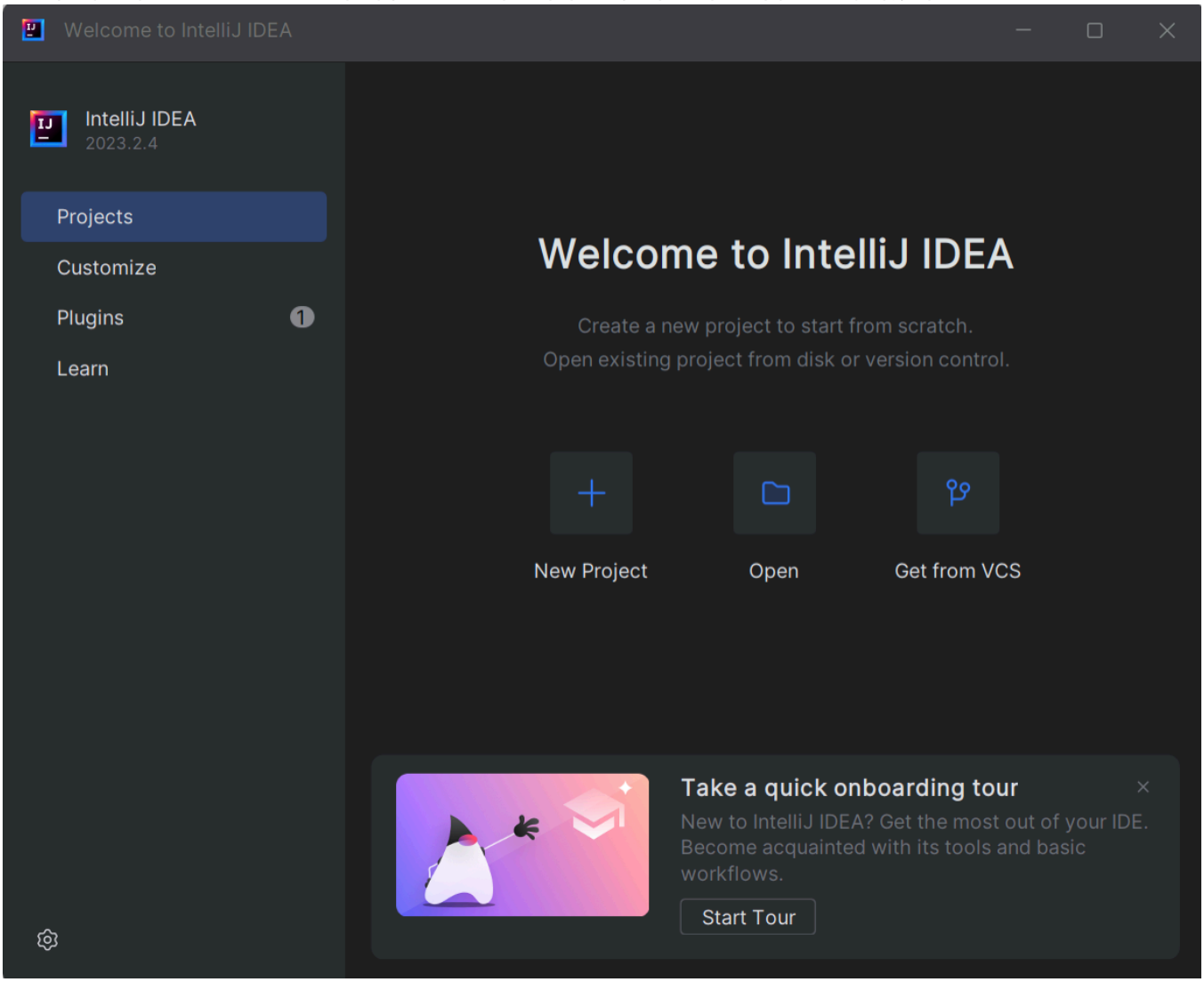



- 앞서 `Add sample code` 선택해서 샘플 코드가 만들어져 있다.

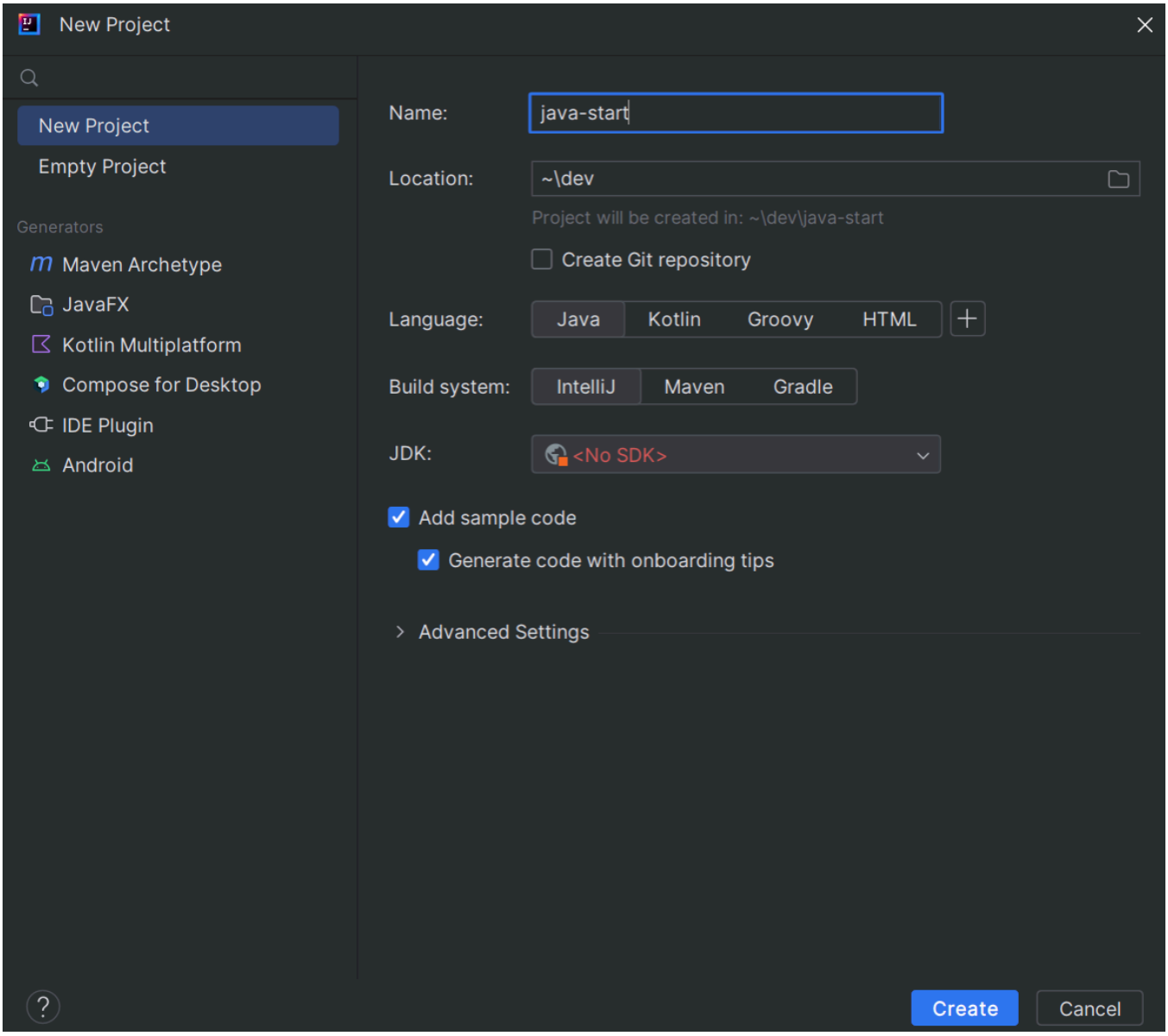
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 `Run 'Main.main()` 버튼을 선택하면 프로그램이 실행된다.

## 윈도우 사용자 추가 설명서

윈도우 사용자도 Mac용 IntelliJ와 대부분 같은 화면이다. 일부 다른 화면 위주로 설명하겠다.

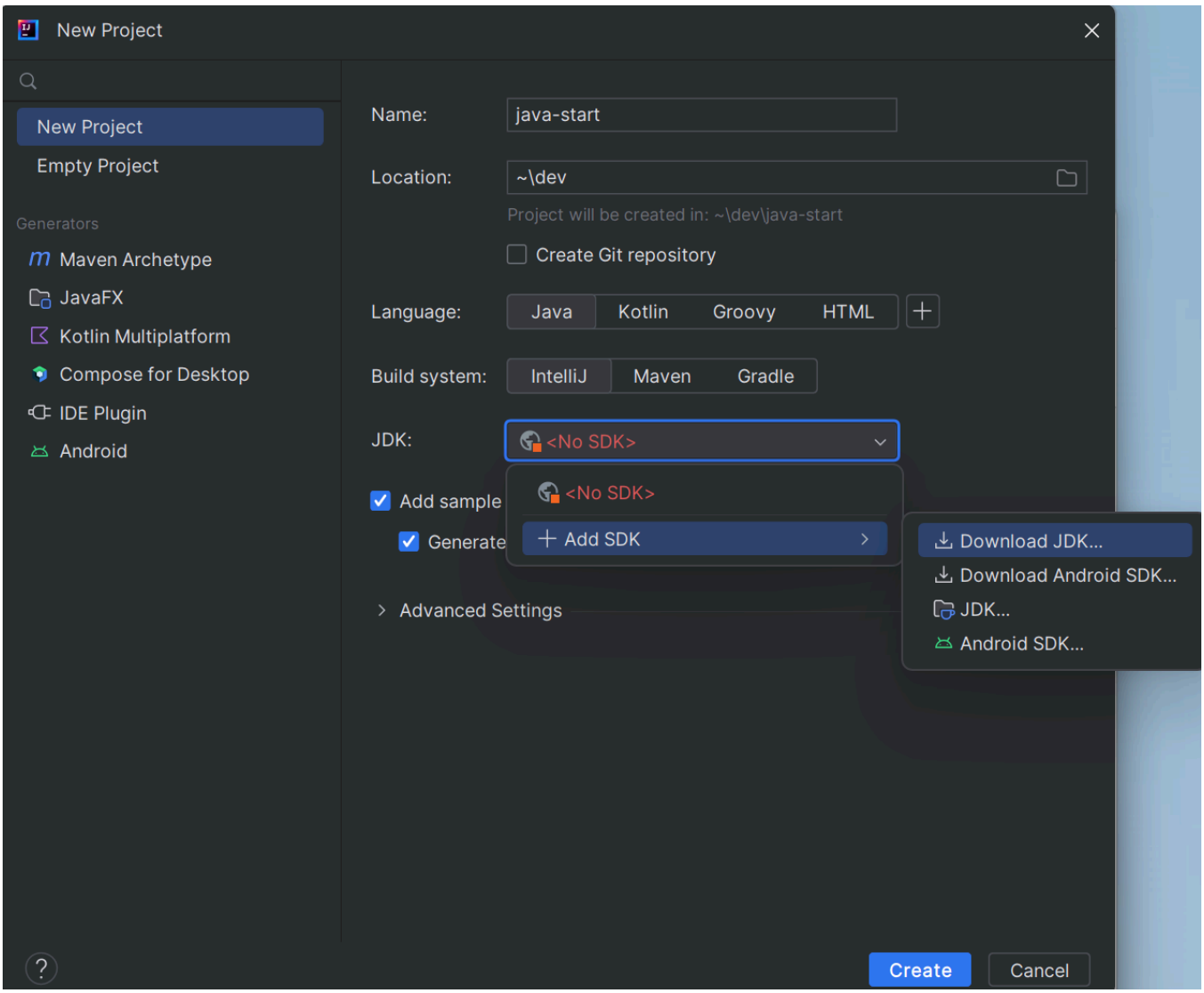



- 프로그램 시작 화면
- New Project 선택

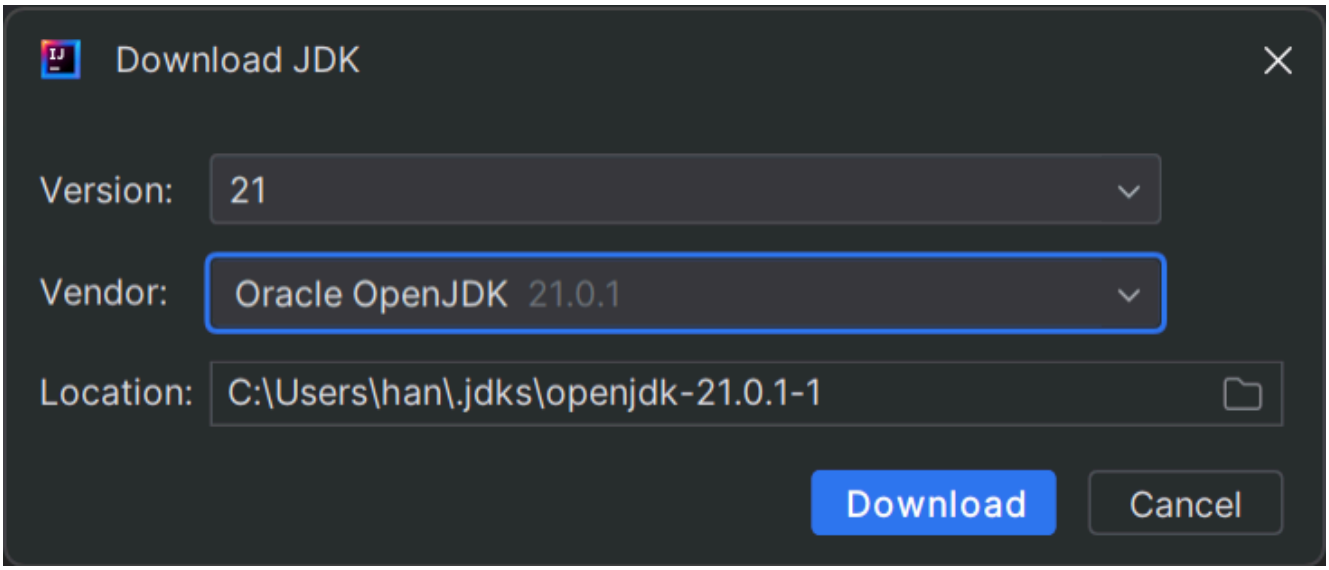



**New Project 화면**

- **Name:**
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: java-mid1
  - 자바 중급2편 강의: **java-mid2**
  - 자바 고급1편 강의: **java-adv1**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: **자바 버전 21 이상**
- `Add sample code` 선택




JDK 설치는 Mac과 동일하다.

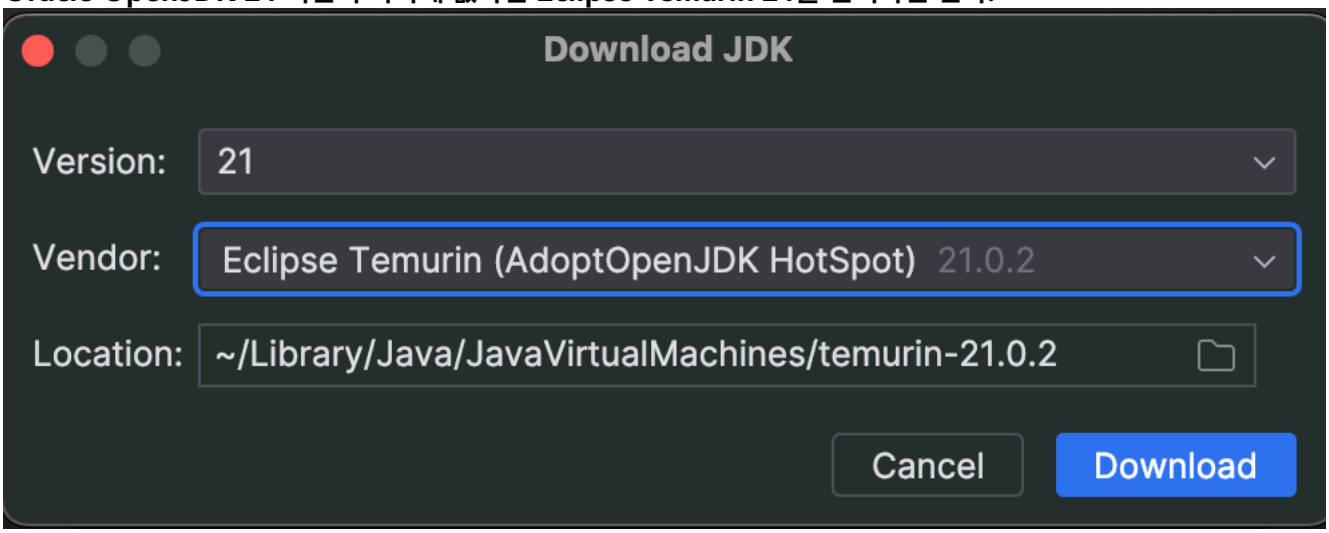


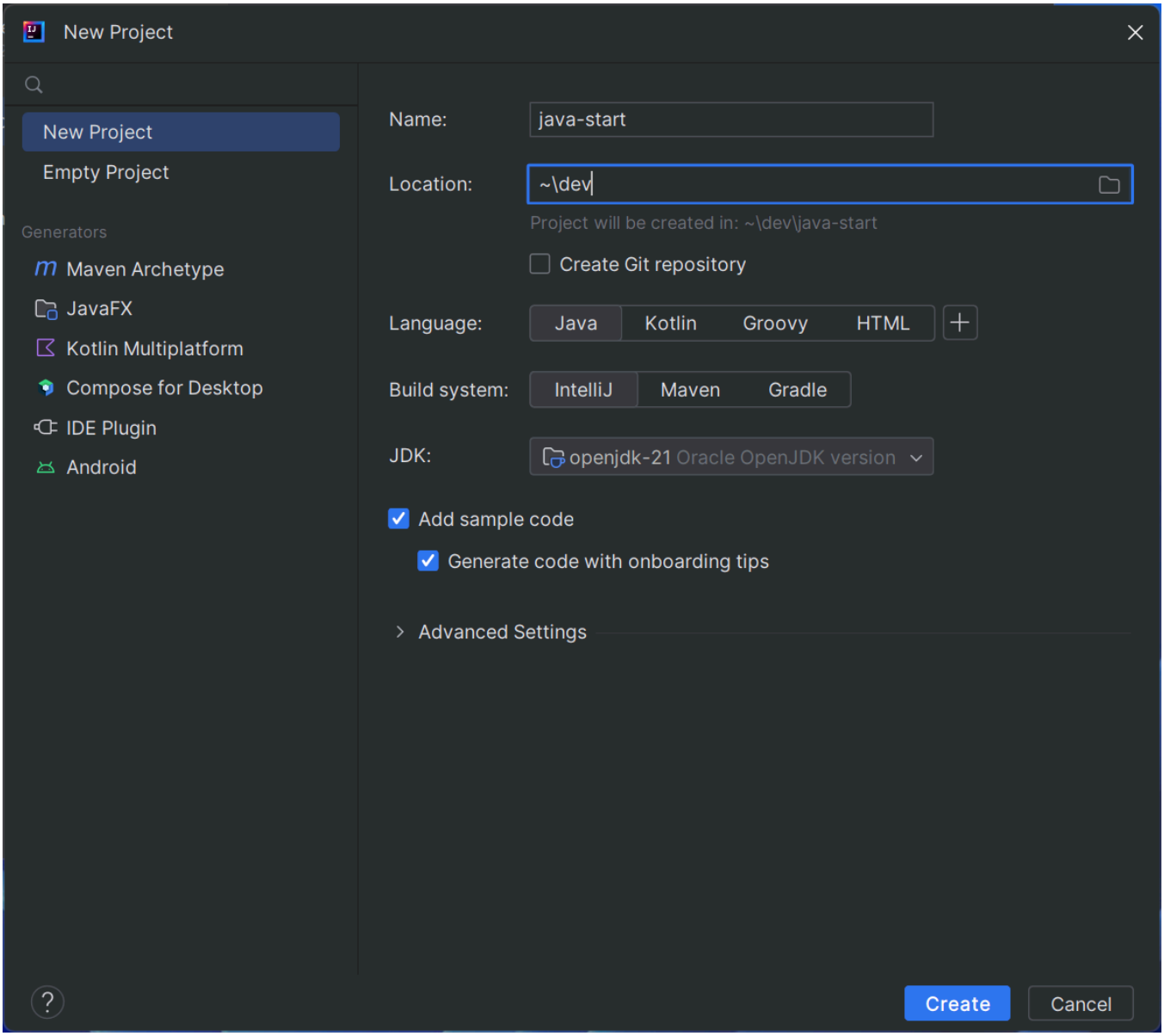

- Version: 21
- Vendor: Oracle OpenJDK

- Location은 가급적 변경하지 말자.

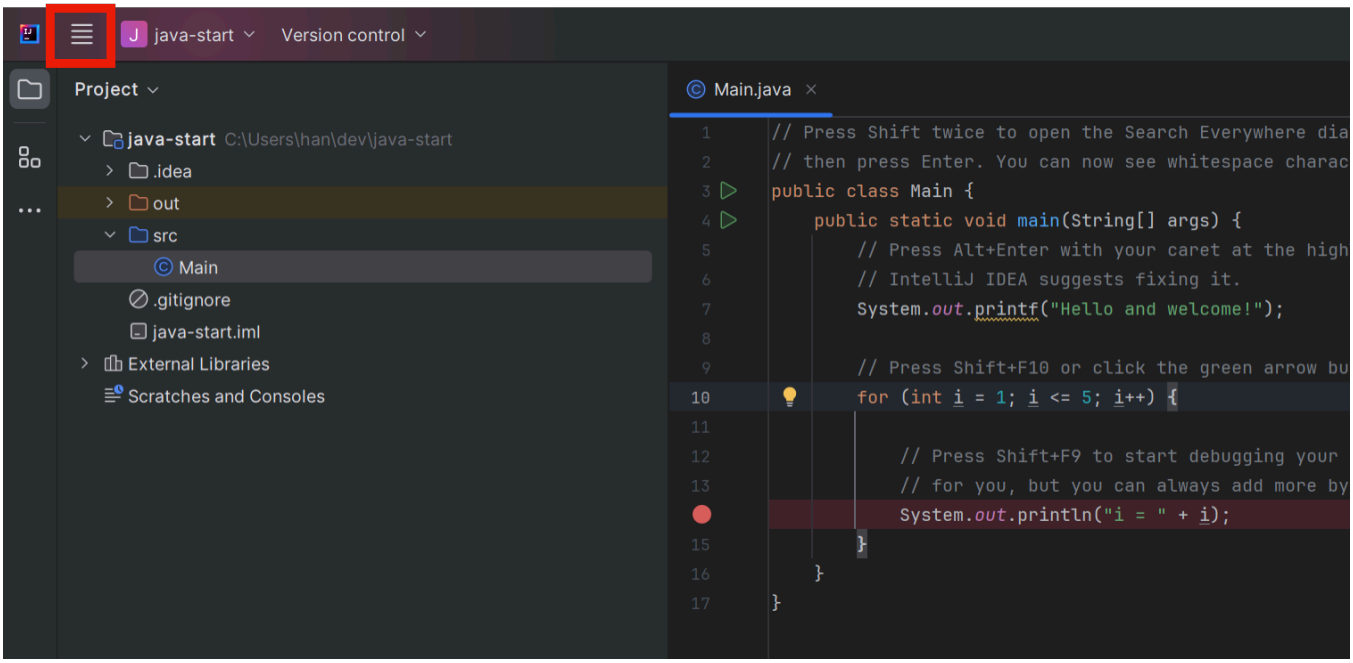
**주의 - 변경 사항**

**Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.**

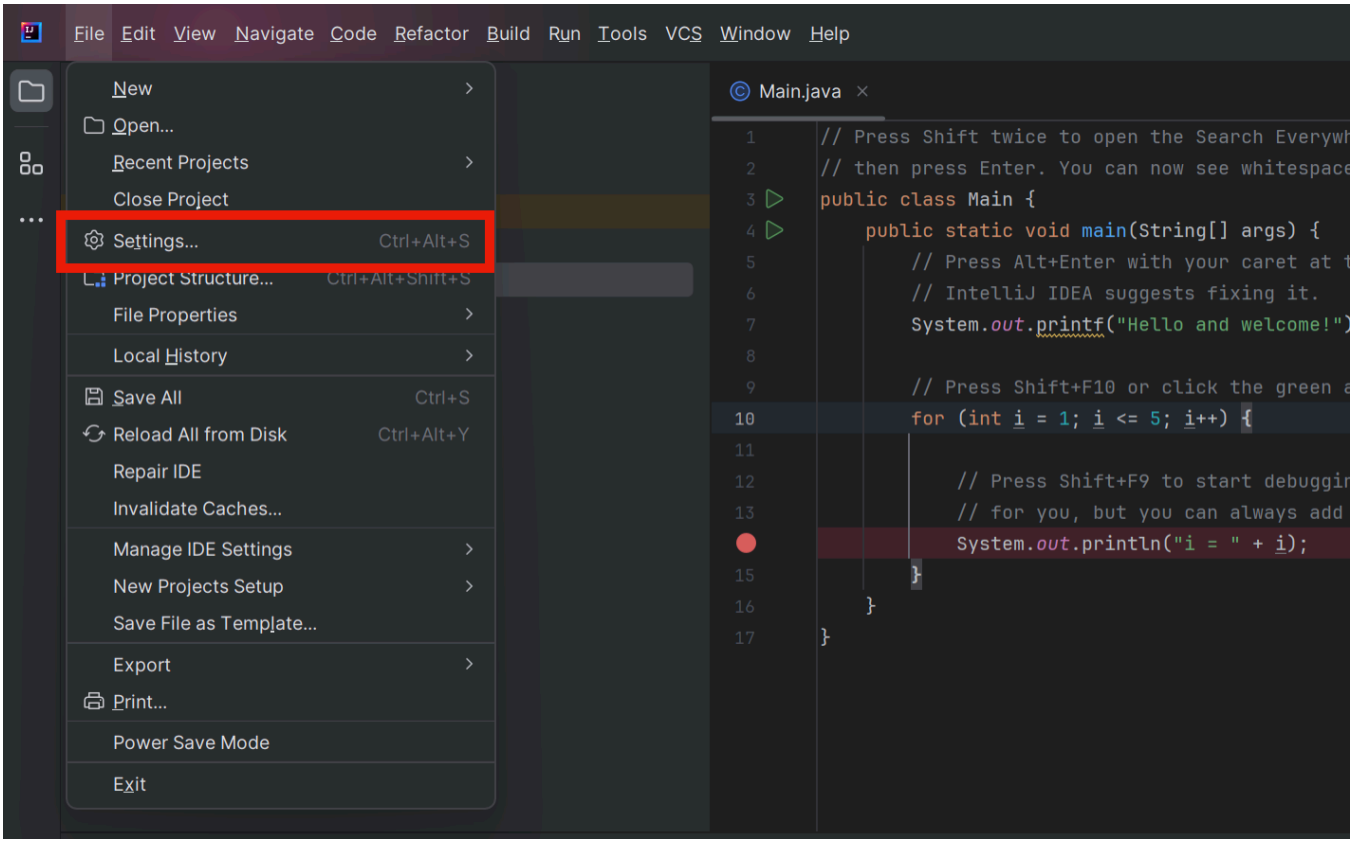





- New Project 완료 화면



- 윈도우는 메뉴를 확인하려면 왼쪽 위의 빨간색 박스 부분을 선택해야 한다.



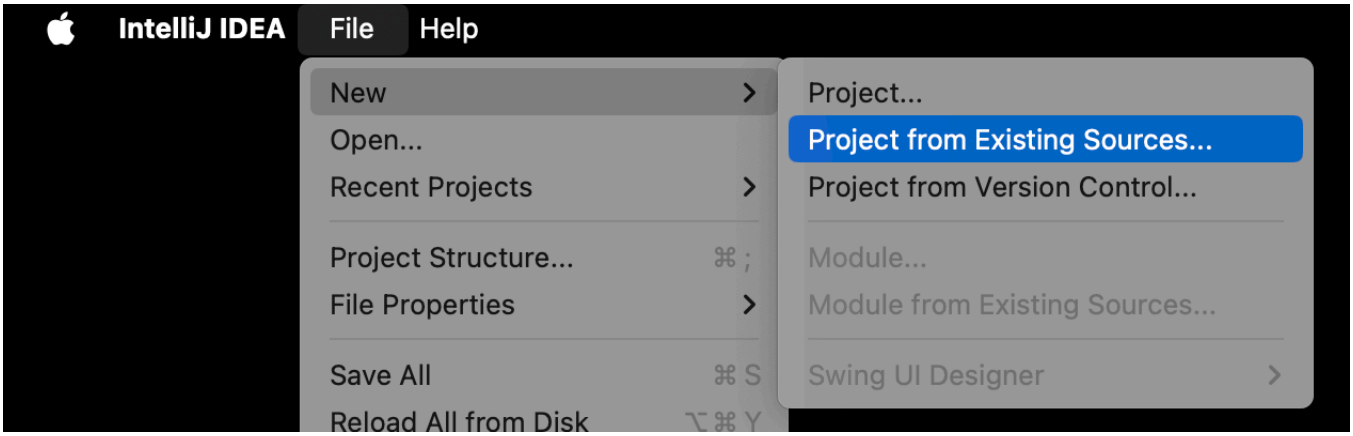

- Mac과 다르게 `Settings...` 메뉴가 `File`에 있다. 이 부분이 Mac과 다르므로 유의하자.

## 한글 언어팩 ➜ 영어로 변경

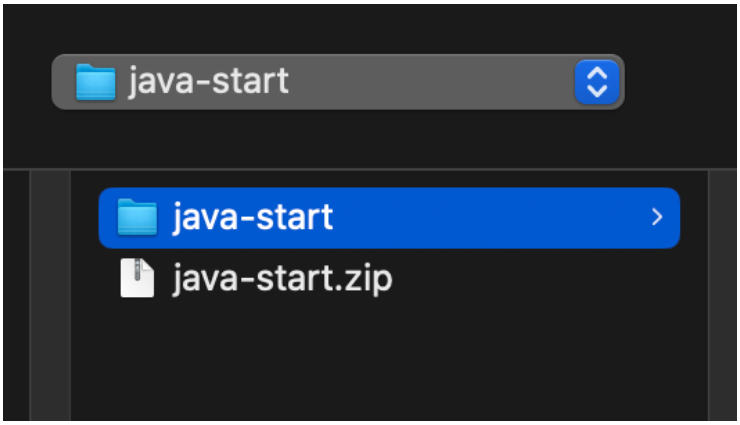
- IntelliJ는 가급적 한글 버전 대신, 영문 버전을 사용하자. 개발하면서 필요한 기능들을 검색하게 되는데, 영문으로 된 자료가 많다. 이번 강의도 영문을 기준으로 진행한다.
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- **Mac**: IntelliJ IDEA(메뉴) ➜ Settings... ➜ Plugins ➜ Installed
- **윈도우**: File ➜ Settings... ➜ Plugins ➜ Installed
  - Korean Language Pack 체크 해제
  - OK 선택후 IntelliJ 다시 시작

## 다운로드 소스 코드 실행 방법

**영상 참고**

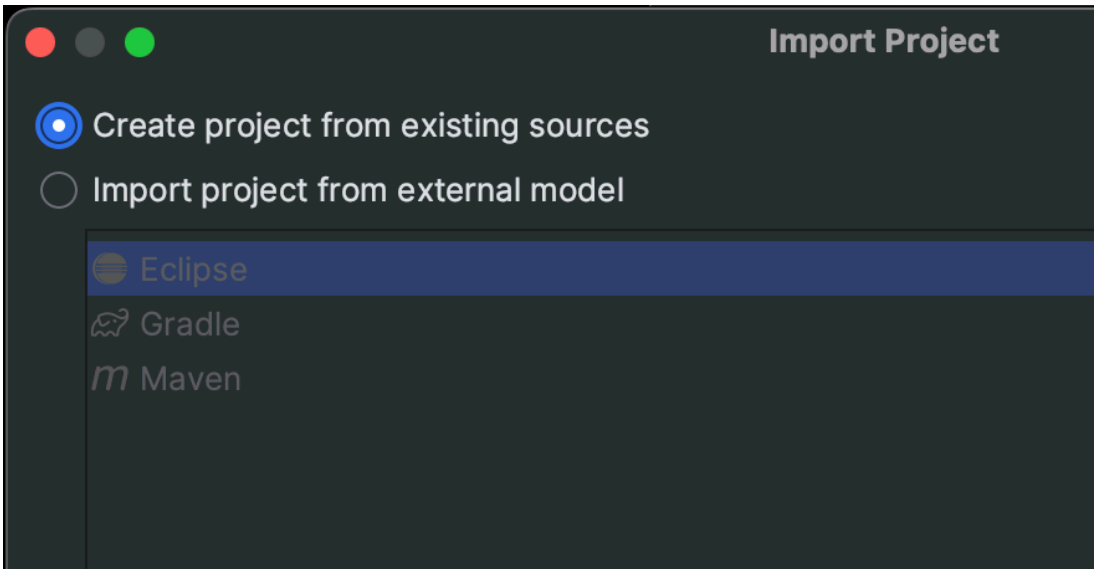



`File -> New -> Project from Existing Sources...` 선택



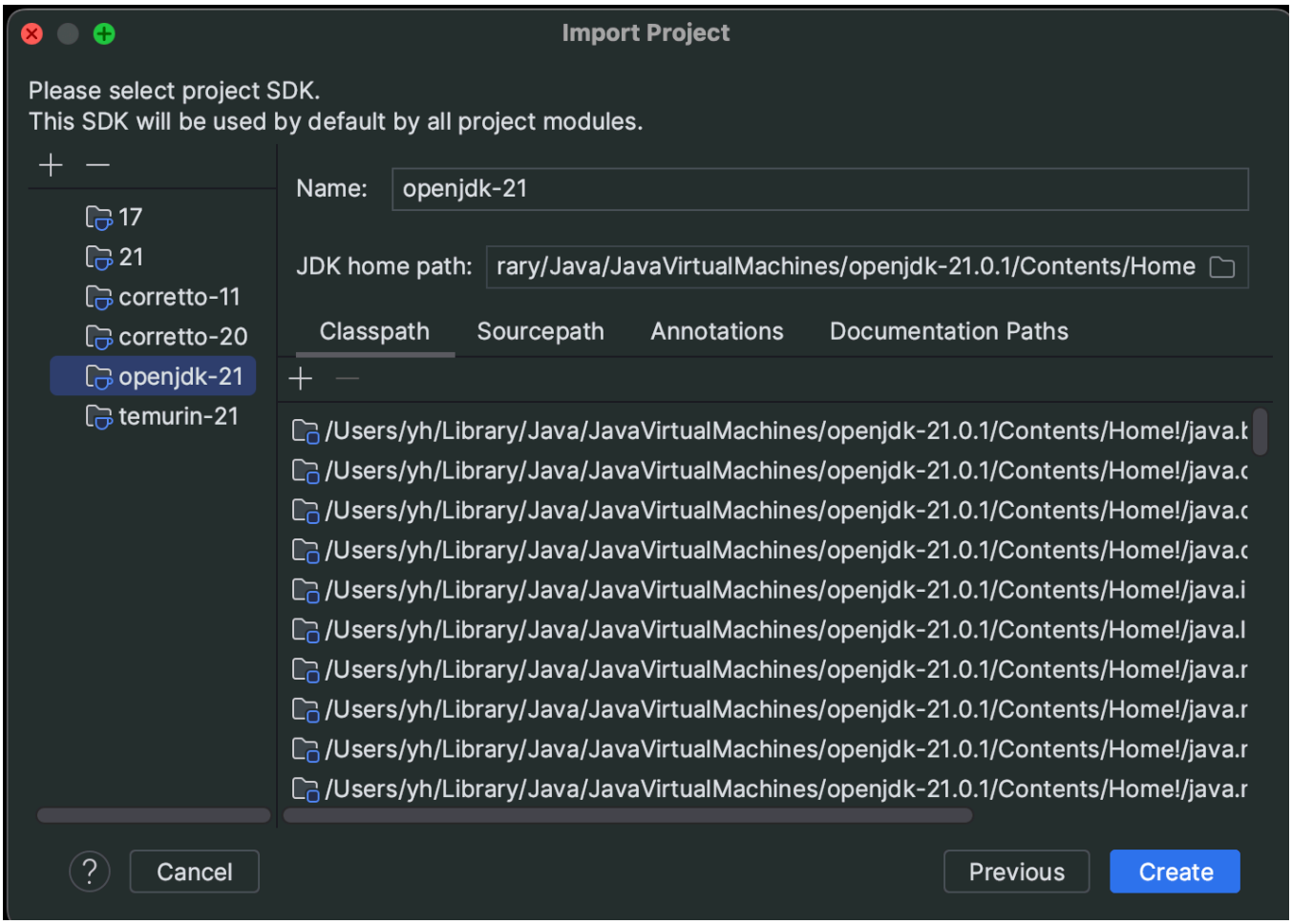

압축을 푼 프로젝트 폴더 선택

- 자바 입문 강의 폴더: java-start
- 자바 기본 강의 폴더: **java-basic**
- 자바 중급1편 강의 폴더: **java-mid1**
- 자바 중급2편 강의 폴더: **java-mid2**
- 자바 고급1편 강의: **java-adv1**

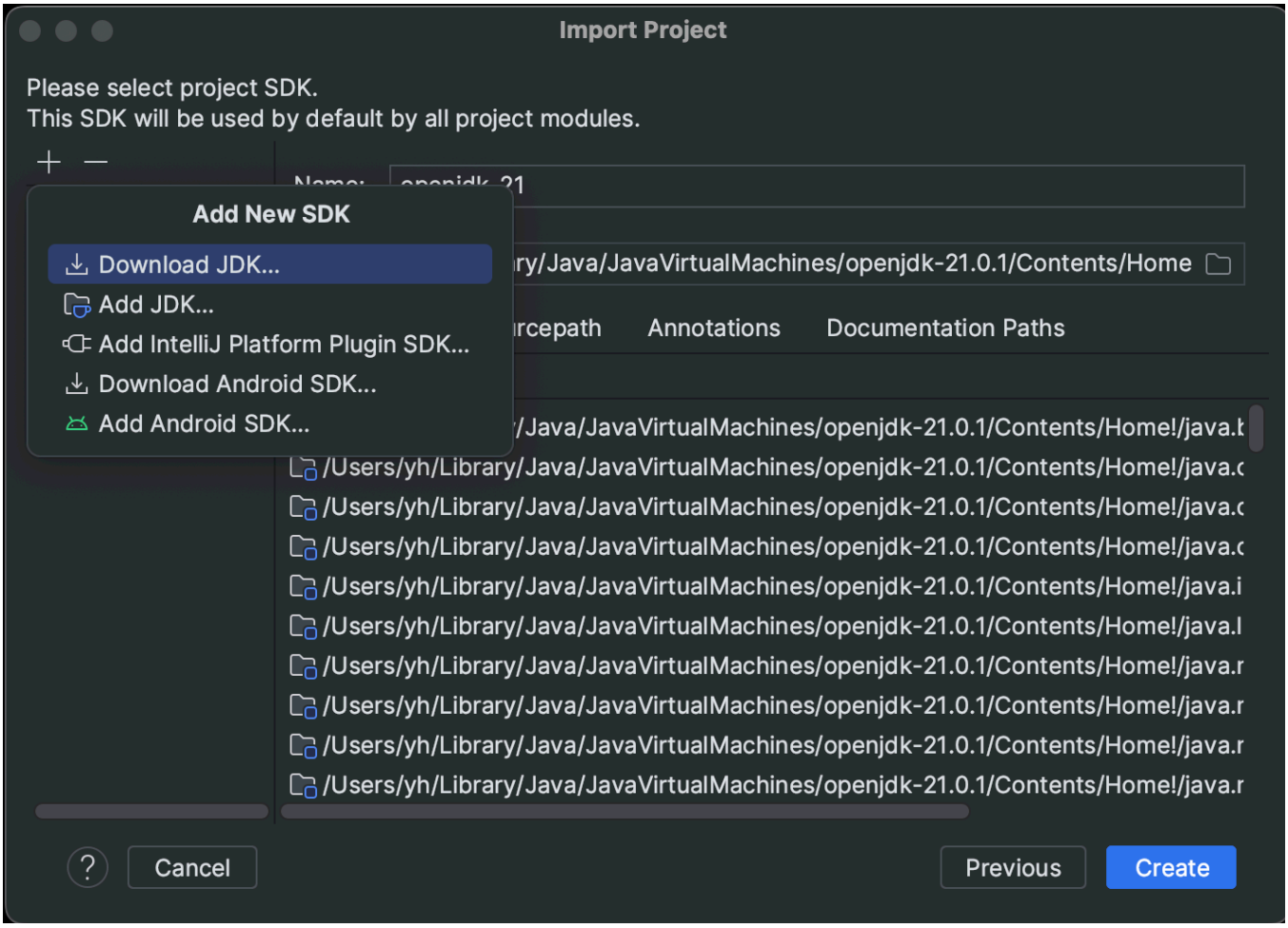



`Create project from existing sources` 선택

이후 계속 `Next` 선택




**openjdk-21 선택**




만약 JDK가 없다면 왼쪽 상단의 + 버튼을 눌러서 openjdk 21 다운로드 후 선택

**Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.**

이후 `Create` 버튼 선택

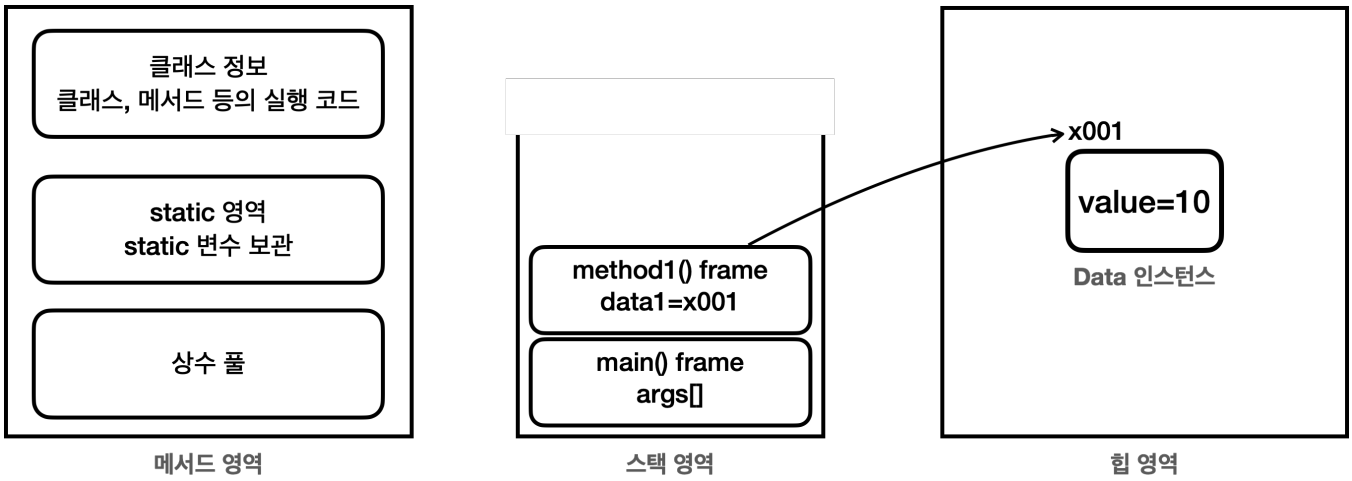
# 스레드 시작1

스레드를 제대로 이해하려면 자바 메모리 구조를 확실히 이해하고 있어야 한다.

스레드를 시작하기 전에 잠깐 자바 메모리 구조를 복습하자.

## 자바 메모리 구조 복습

### 자바 메모리 구조




- **메서드 영역(Method Area)**: 메서드 영역은 프로그램을 실행하는데 필요한 공통 데이터를 관리한다. 이 영역은 프로그램의 모든 영역에서 공유한다.
  - 클래스 정보: 클래스의 실행 코드(바이트 코드), 필드, 메서드와 생성자 코드등 모든 실행 코드가 존재한다.
  - static 영역: `static` 변수들을 보관한다.
  - 런타임 상수 풀: 프로그램을 실행하는데 필요한 공통 리터럴 상수를 보관한다.
- **스택 영역(Stack Area)**: 자바 실행 시, 하나의 실행 스택이 생성된다. 각 스택 프레임은 지역 변수, 중간 연산 결과, 메서드 호출 정보 등을 포함한다.
  - 스택 프레임: 스택 영역에 쌓이는 네모 박스가 하나의 스택 프레임이다. 메서드를 호출할 때 마다 하나의 스택 프레임이 쌓이고, 메서드가 종료되면 해당 스택 프레임이 제거된다.
- **힙 영역(Heap Area)**: 객체(인스턴스)와 배열이 생성되는 영역이다. 가비지 컬렉션(GC)이 이루어지는 주요 영역이며, 더 이상 참조되지 않는 객체는 GC에 의해 제거된다.

> **참고**: 스택 영역은 더 정확히는 각 스레드별로 하나의 실행 스택이 생성된다. 따라서 스레드 수 만큼 스택이 생성된다. 지금은 스레드를 1개만 사용하므로 스택도 하나이다. 이후 스레드를 추가할 것인데, 그러면 스택도 스레드 수 만큼 증가한다.

## 스레드 생성

스레드를 직접 만들어보자. 그래서 해당 스레드에서 별도의 로직을 수행해보자.
스레드를 만들 때는 `Thread` 클래스를 상속 받는 방법과 `Runnable` 인터페이스를 구현하는 방법이 있다.
먼저 `Thread` 클래스를 상속 받아서 스레드를 생성해보자.

## 스레드 생성 - Thread 상속

자바는 많은 것을 객체로 다룬다. 자바가 예외를 객체로 다루듯이, 스레드도 객체로 다룬다.
스레드가 필요하면, 스레드 객체를 생성해서 사용하면 된다.

```
package thread.start;
```

```
public class HelloThread extends Thread {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run()");
    }
}
```

- `Thread` 클래스를 상속하고, 스레드가 실행할 코드를 `run()` 메서드에 재정의한다.
- `Thread.currentThread()`를 호출하면 해당 코드를 실행하는 스레드 객체를 조회할 수 있다.
- `Thread.currentThread().getName()`: 실행 중인 스레드의 이름을 조회한다.

```
package thread.start;

public class HelloThreadMain {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main()
start");

        HelloThread helloThread = new HelloThread();
        System.out.println(Thread.currentThread().getName() + ": start() 호출
전");
        helloThread.start();
        System.out.println(Thread.currentThread().getName() + ": start() 호출
후");

        System.out.println(Thread.currentThread().getName() + ": main() end");
    }

}
```

- 앞서 만든 `HelloThread` 스레드 객체를 생성하고 `start()` 메서드를 호출한다.
- `start()` 메서드는 스레드를 실행하는 아주 특별한 메서드이다.
- `start()`를 호출하면 `HelloThread` 스레드가 `run()` 메서드를 실행한다.

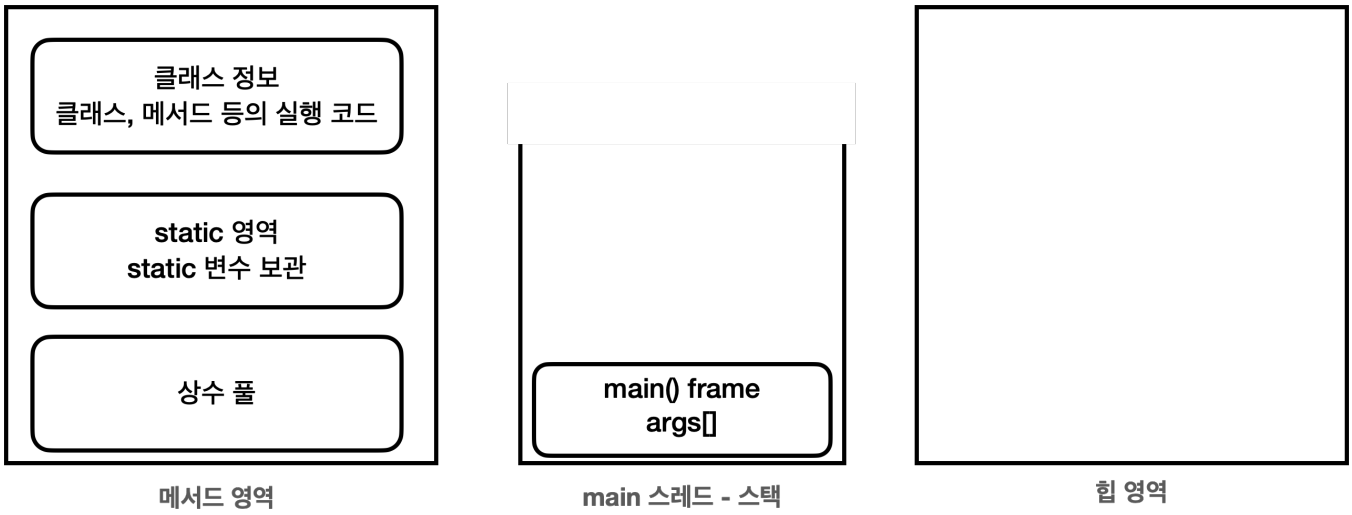
**주의!** `run()` 메서드가 아니라 반드시 `start()` 메서드를 호출해야 한다. 그래야 별도의 스레드에서 `run()` 코드가 실행된다.

**실행 결과**

```
main: main() start
main: start() 호출 전
main: start() 호출 후
Thread-0: run()
main: main() end
```

- 참고로 실행 결과는 스레드의 실행 순서에 따라 약간 다를 수 있다. 이 부분은 바로 뒤에서 설명한다.

**스레드 생성 전**




실행 결과를 보면 `main()` 메서드는 `main` 이라는 이름의 스레드가 실행하는 것을 확인할 수 있다. 프로세스가 작동하려면 스레드가 최소한 하나는 있어야 한다. 그래야 코드를 실행할 수 있다. 자바는 실행 시점에 `main` 이라는 이름의 스레드를 만들고 프로그램의 시작점인 `main()` 메서드를 실행한다.

**스레드 생성 후**


- `HelloThread` 스레드 객체를 생성한 다음에 `start()` 메서드를 호출하면 자바는 스레드를 위한 별도의 스택 공간을 할당한다.
- 스레드 객체를 생성하고, 반드시 `start()` 를 호출해야 스택 공간을 할당 받고 스레드가 작동한다.

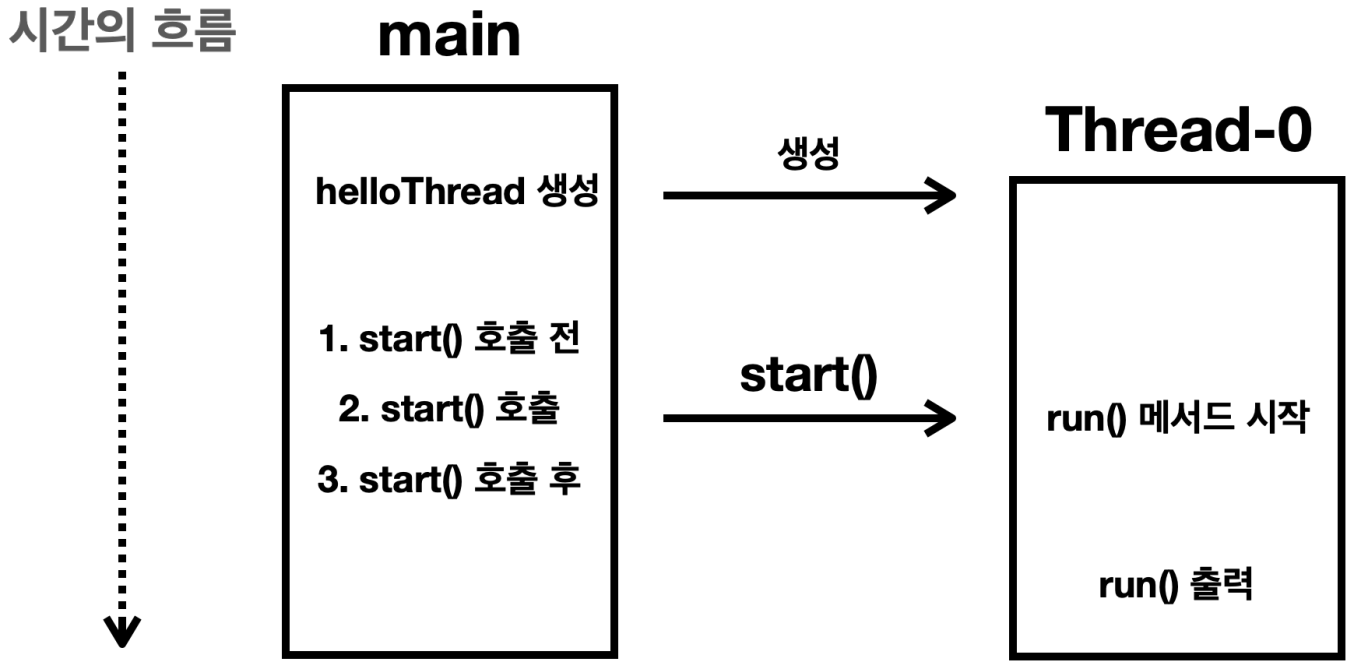
- 스레드에 이름을 주지 않으면 자바는 스레드에 `Thread-0`, `Thread-1`과 같은 임의의 이름을 부여한다.
- 새로운 `Thread-0` 스레드가 사용할 전용 스택 공간이 마련되었다.
- `Thread-0` 스레드는 `run()` 메서드의 스택 프레임을 스택에 올리면서 `run()` 메서드를 시작한다.

**메서드를 실행하면 스택 위에 스택 프레임이 쌓인다**

- `main` 스레드는 `main()` 메서드의 스택 프레임을 스택에 올리면서 시작한다.
- 직접 만드는 스레드는 `run()` 메서드의 스택 프레임을 스택에 올리면서 시작한다.

실행 결과를 보면 `Thread-0` 스레드가 `run()` 메서드를 실행한 것을 확인할 수 있다.

**시간의 흐름으로 분석**




- `main` 스레드가 `HelloThread` 인스턴스를 생성한다. 이때 스레드에 이름을 부여하지 않으면 자바가 `Thread-0`, `Thread-1`과 같은 임의의 이름을 부여한다.
- `start()` 메서드를 호출하면, `Thread-0` 스레드가 시작되면서 `Thread-0` 스레드가 `run()` 메서드를 호출한다.
- 여기서 핵심은 `main` 스레드가 `run()` 메서드를 실행하는게 아니라 `Thread-0` 스레드가 `run()` 메서드를 실행한다는 점이다.
- `main` 스레드는 단지 `start()` 메서드를 통해 `Thread-0` 스레드에게 실행을 지시할 뿐이다. 다시 강조하지만 `main` 스레드가 `run()`을 호출하는 것이 아니다! `main` 스레드는 다른 스레드에게 일을 시작하라고 지시만 하고, 바로 `start()` 메서드를 빠져나온다.
- 이제 `main` 스레드와 `Thread-0` 스레드는 동시에 실행된다.
- `main` 스레드 입장에서 보면 그림의 1, 2, 3번 코드를 멈추지 않고 계속 수행한다. 그리고 `run()` 메서드는 `main`이 아닌 별도의 스레드에서 실행된다.

**스레드 간 실행 순서는 보장하지 않는다.**

스레드는 동시에 실행되기 때문에 스레드 간에 실행 순서는 얼마든지 달라질 수 있다. 따라서 다음과 같이 다양한 실행 결과가 나올 수 있다.

`main` 스레드가 빨리 실행된 경우

```
main: main() start
main: start() 호출 전
main: start() 호출 후
main: main() end
Thread-0: run()
```

- `main` 스레드가 모든 로직을 다 수행한 다음에 `Thread-0`가 수행된다.

`Thread-0` 스레드가 빨리 실행된 경우

```
main: main() start
main: start() 호출 전
Thread-0: run()
main: start() 호출 후
main: main() end
```

- `Thread-0` `start()` 이후 `Thread-0`가 먼저 수행되고, `main`이 수행된다.

`main` 스레드 실행 중간에 `Thread-0` 스레드가 실행된 경우

```
main: main() thread.start
main: start() 호출 전
main: start() 호출 후
Thread-0: run()
main: main() end
```

- `main` 스레드가 "start() 호출 후"를 출력한 다음에 `Thread-0` 스레드 `run()`을 출력한다.

스레드 간의 실행 순서는 얼마든지 달라질 수 있다.
CPU 코어가 2개여서 물리적으로 정말 동시에 실행될 수도 있고, 하나의 CPU 코어에 시간을 나누어 실행될 수도 있다. 그리고 한 스레드가 얼마나 오랜기간 실행되는지도 보장하지 않는다. 한 스레드가 먼저 다 수행된 다음에 다른 스레드가 수행될 수도 있고, 둘이 완전히 번갈아 가면서 수행되는 경우도 있다.

스레드는 순서와 실행 기간을 모두 보장하지 않는다. 이것이 바로 멀티스레드다!

# 스레드 시작2

## start() vs run()

스레드의 `start()` 대신에 재정의한 `run()` 메서드를 직접 호출하면 어떻게 될까?

```
package thread.start;

public class BadThreadMain {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main()
start");

        HelloThread helloThread = new HelloThread();
        System.out.println(Thread.currentThread().getName() + ": run() 호출
전");
        helloThread.run(); // run() 직접 실행
        System.out.println(Thread.currentThread().getName() + ": run() 호출
후");

        System.out.println(Thread.currentThread().getName() + ": main() end");
    }

}
```

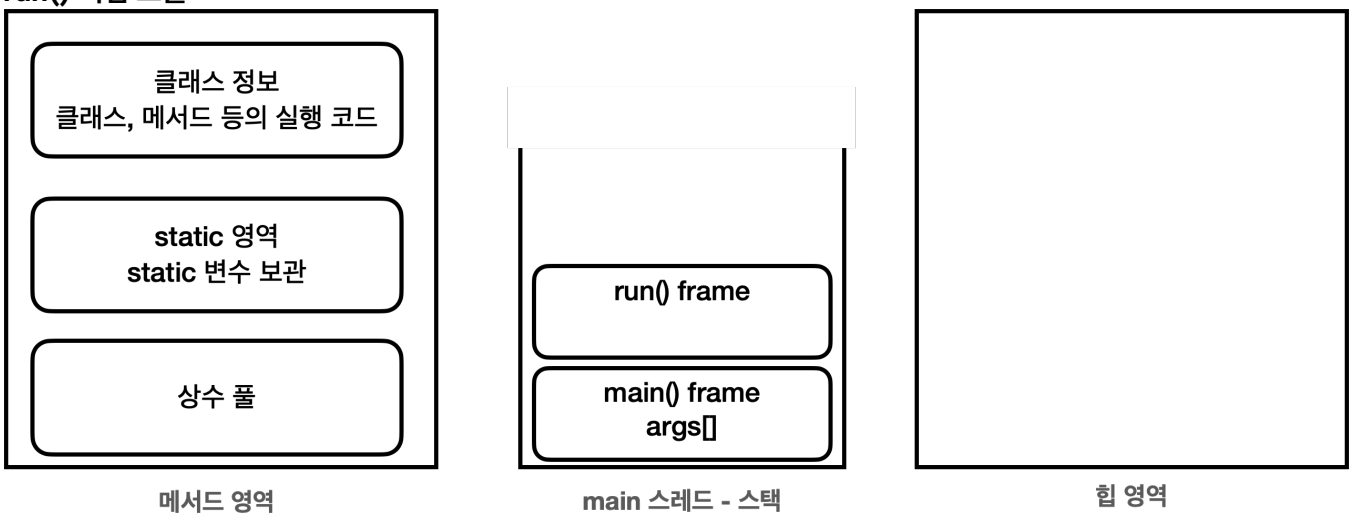
- `helloThread.run()` : `start()` 를 호출해야 하지만 여기서는 문제를 확인하기 위해 `run()` 을 직접 호출한다.

**실행 결과**

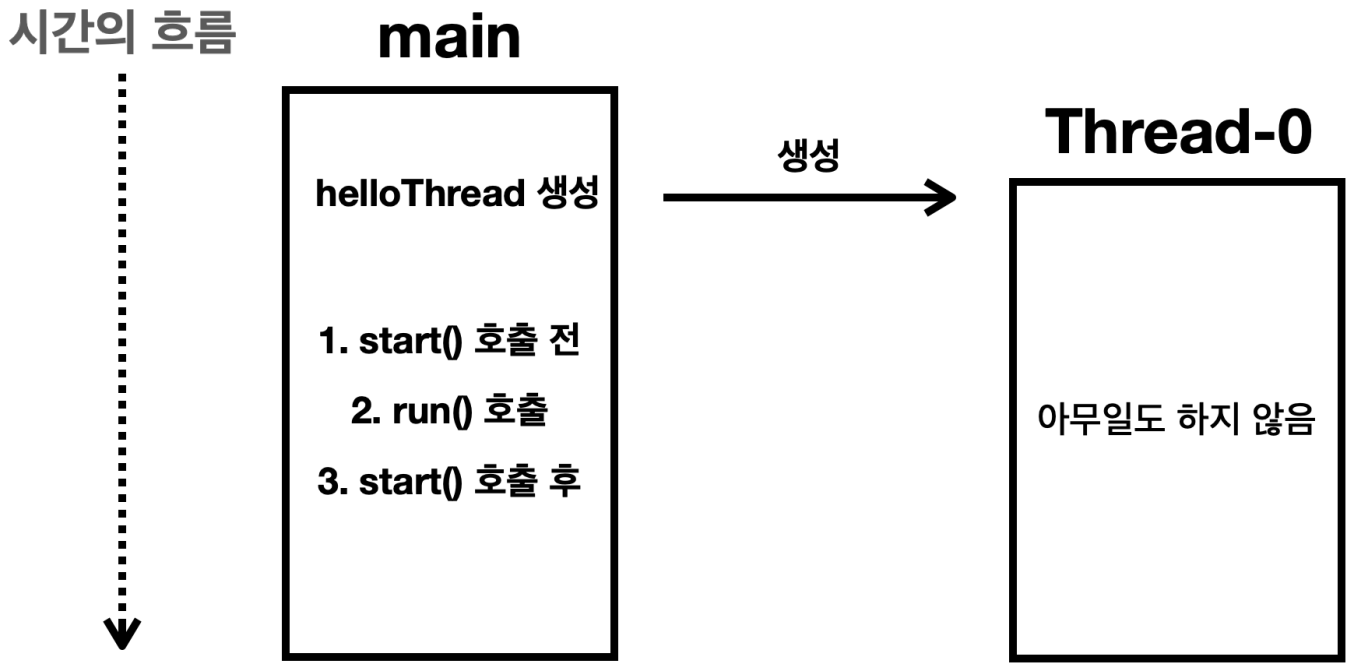
```
main: main() start
main: run() 호출 전
main: run()
```

```
main: run() 호출 후
main: main() end
```

**run() 직접 호출**




- 실행 결과를 잘 보면 별도의 스레드가 `run()`을 실행하는 것이 아니라, `main` 스레드가 `run()` 메서드를 호출할 것을 확인할 수 있다.
- 자바를 처음 실행하면 `main` 스레드가 `main()` 메서드를 호출하면서 시작한다.
- `main` 스레드는 `HelloThread` 인스턴스에 있는 `run()` 이라는 메서드를 호출한다.
- `main` 스레드가 `run()` 메서드를 실행했기 때문에 `main` 스레드가 사용하는 스택위에 `run()` 스택 프레임이 올라간다.




결과적으로 `main` 스레드에서 모든 것을 처리한 것이 된다.

스레드의 `start()` 메서드는 스레드에 스택 공간을 할당하면서 스레드를 시작하는 아주 특별한 메서드이다. 그리고

해당 스레드에서 `run()` 메서드를 실행한다. 따라서 `main` 스레드가 아닌 별도의 스레드에서 재정의한 `run()` 메서드를 실행하려면, 반드시 `start()` 메서드를 호출해야 한다.

> **참고:** 스레드와 메모리 구조에 대한 부분은 강의를 진행하면서 점점 더 자세히 설명한다.

# 데몬 스레드

## 데몬 스레드

스레드는 사용자(user) 스레드와 데몬(daemon) 스레드 2가지 종류로 구분할 수 있다.

**사용자 스레드(non-daemon 스레드)**

- 프로그램의 주요 작업을 수행한다.
- 작업이 완료될 때까지 실행된다.
- 모든 user 스레드가 종료되면 JVM도 종료된다.

**데몬 스레드**

- 백그라운드에서 보조적인 작업을 수행한다.
- 모든 user 스레드가 종료되면 데몬 스레드는 자동으로 종료된다.

JVM은 데몬 스레드의 실행 완료를 기다리지 않고 종료된다. 데몬 스레드가 아닌 모든 스레드가 종료되면, 자바 프로그램도 종료된다.

> **용어 - 데몬**: 그리스 신화에서 데몬은 신과 인간 사이의 중간적 존재로, 보이지 않게 활동하며 일상적인 일들을 도왔다. 이런 의미로 컴퓨터 과학에서는 사용자에게 직접적으로 보이지 않으면서 시스템의 백그라운드에서 작업을 수행하는 것을 데몬 스레드, 데몬 프로세스라 한다. 예를 들어서 사용하지 않는 파일이나 메모리를 정리하는 작업들이 있다.

```
package thread.start;

public class DaemonThreadMain {

    public static void main(String[] args) {
```

```
        System.out.println(Thread.currentThread().getName() + ": main()
start");
        DaemonThread daemonThread = new DaemonThread();
        daemonThread.setDaemon(true); // 데몬 스레드 여부
        daemonThread.start();
        System.out.println(Thread.currentThread().getName() + ": main() end");
    }

    static class DaemonThread extends Thread {

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + ": run()
start");
            try {
                Thread.sleep(10000); // 10초간 실행
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(Thread.currentThread().getName() + ": run()
end");
        }
    }

}
```

- `setDaemon(true)` : 데몬 스레드로 설정한다.
- 데몬 스레드 여부는 `start()` 실행 전에 결정해야 한다. 이후에는 변경되지 않는다.
- 기본 값은 `false` 이다. (user 스레드가 기본)

> **참고**: `run()` 메서드 안에서 `Thread.sleep()` 를 호출할 때 체크 예외인 `InterruptedException` 을 밖으로 던질 수 없고 반드시 잡아야 한다. `run()` 메서드는 체크 예외를 밖으로 던질 수 없는데, 이 부분은 뒤에서 설명한다.

**실행 결과 - setDaemon(true)**

```
main: main() start
main: main() end
Thread-0: run() start
```

- `setDaemon(true)`로 설정해보자.
- `Thread-0`는 데몬 스레드로 설정된다.
- 유일한 user 스레드인 `main` 스레드가 종료되면서 자바 프로그램도 종료된다.
- 따라서 `run() end`가 출력되기 전에 프로그램이 종료된다.

**실행 결과 - setDaemon(false)**

```
main: main() start
main: main() end
Thread-0: run() start
Thread-0: run() end
```

- `setDaemon(false)`로 설정해보자.
- `Thread-0`는 user 스레드로 설정된다.
- `main` 스레드가 종료되어도, user 스레드인 `Thread-0`가 종료될 때 까지 자바 프로그램을 종료되지 않는다.
- 따라서 `Thread-0: run() end`가 출력된다.
- user 스레드인 `main` 스레드와 `Thread-0` 스레드가 모두 종료되면서 자바 프로그램도 종료된다.

# 스레드 생성 - Runnable

스레드를 만들 때는 `Thread` 클래스를 상속 받는 방법과 `Runnable` 인터페이스를 구현하는 방법이 있다.
앞서 `Thread` 클래스를 상속 받아서 스레드를 생성해보았다.
이번에는 `Runnable` 인터페이스를 구현하는 방식으로 스레드를 생성해보자.

**Runnable 인터페이스**

```
package java.lang;

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

- 자바가 제공하는 스레드 실행용 인터페이스

```
package thread.start;
```

```
public class HelloRunnable implements Runnable {

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run()");
    }
}
```

```
package thread.start;

public class HelloRunnableMain {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main()
start");

        HelloRunnable runnable = new HelloRunnable();
        Thread thread = new Thread(runnable);
        thread.start();

        System.out.println(Thread.currentThread().getName() + ": main() end");
    }
}
```

**실행 결과**

```
main: main() start
main: main() end
Thread-0: run()
```

실행 결과는 기존과 같다. 차이가 있다면, 스레드와 해당 스레드가 실행할 작업이 서로 분리되어 있다는 점이다.
스레드 객체를 생성할 때, 실행할 작업을 생성자로 전달하면 된다.

## Thread 상속 vs Runnable 구현

**스레드 사용할 때는 `Thread`를 상속 받는 방법보다 `Runnable` 인터페이스를 구현하는 방식을 사용하자.**
두 방식이 서로 장단점이 있지만, 스레드를 생성할 때는 `Thread`클래스를 상속하는 방식보다 `Runnable` 인터페이스

를 구현하는 방식이 더 나은 선택이다.

**Thread 클래스 상속 방식**

**장점**

- 간단한 구현: `Thread` 클래스를 상속받아 `run()` 메서드만 재정의하면 된다.

**단점**

- 상속의 제한: 자바는 단일 상속만을 허용하므로 이미 다른 클래스를 상속받고 있는 경우 `Thread` 클래스를 상속받을 수 없다.
- 유연성 부족: 인터페이스를 사용하는 방법에 비해 유연성이 떨어진다.

**Runnable 인터페이스를 구현 방식**

**장점**

- 상속의 자유로움: `Runnable` 인터페이스 방식은 다른 클래스를 상속받아도 문제없이 구현할 수 있다.
- 코드의 분리: 스레드와 실행할 작업을 분리하여 코드의 가독성을 높일 수 있다.
- 여러 스레드가 동일한 `Runnable` 객체를 공유할 수 있어 자원 관리를 효율적으로 할 수 있다.

**단점**

- 코드가 약간 복잡해질 수 있다. `Runnable` 객체를 생성하고 이를 `Thread`에 전달하는 과정이 추가된다.

정리하자면 `Runnable` 인터페이스를 구현하는 방식을 사용하자. 스레드와 실행할 작업을 명확히 분리하고, 인터페이스를 사용하므로 `Thread` 클래스를 직접 상속하는 방식보다 더 유연하고 유지보수 하기 쉬운 코드를 만들 수 있다.

# 로거 만들기

현재 어떤 스레드가 코드를 실행하는지 출력하기 위해 다음과 같이 긴 코드를 작성하는 것은 너무 번거롭다.

```
System.out.println(Thread.currentThread().getName() + ": run()");
```

다음 예시와 같이 실행하면, 현재 시간, 스레드 이름, 출력 내용등이 한번에 나오는 편리한 기능을 만들어보자.

```
log("hello thread");
log(123);
```

**실행 결과**

```
15:39:02.000 [     main] hello thread
15:39:02.002 [     main] 123
```

```
package util;

import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public abstract class MyLogger {

    private static final DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("HH:mm:ss.SSS");

    public static void log(Object obj) {
        String time = LocalTime.now().format(formatter);
        System.out.printf("%s [%9s] %s\n", time,
Thread.currentThread().getName(), obj);
    }

}
```

- `util` 이라는 패키지를 사용했다. 프로젝트 전반에 사용되는 유틸리티라는 뜻이다.
- 현재 시간을 원하는 포멧으로 출력하기 위해 `DateTimeFormatter` 를 사용한다.
- `printf` 에서 `%s` 는 문자열을 뜻한다. 인자를 순서대로 사용한다. 여기서는 현재 시간, 스레드 이름, 출력할 객체 순서이다.
  - 참고로 마지막의 출력할 객체는 문자열이 아니라 `Object` 타입인데, `%s` 를 사용하면 `toString()` 을 사용해서 문자열로 변환 후 출력한다. 이렇게 `Object` 타입을 사용하면 문자열 뿐만 아니라 객체도 출력할 수 있다.
- `%9s` 는 다음과 같이 문자를 출력할 때 9칸을 확보한다는 뜻이다. 9칸이 차지 않으면 왼쪽에 그 만큼 비워둔다. 이 기능은 단순히 출력시 정렬을 깔끔하게 하려고 사용한다.
  - 예)
    - `[     main]` : 앞에 5칸 공백
    - `[ Thread-0]` : 앞에 1칸 공백

```
package util;
```

```
import static util.MyLogger.log;

public class MyLoggerMain {
    public static void main(String[] args) {
        log("hello thread");
        log(123);
    }
}
```

- 사용할 때는 지금과 같이 `import static`을 사용하면 메서드 이름만으로 간단히 사용할 수 있다.

**실행 결과**

```
15:39:02.000 [     main] hello thread
15:39:02.002 [     main] 123
```

스레드를 학습할 때는 스레드 이름, 그리고 해당 스레드가 언제 실행되었는지 확인하는 것이 중요하다.
앞으로는 `System.out.println()` 대신에 스레드 이름과 실행 시간을 알려주는 `MyLogger`를 사용하겠다.

## 여러 스레드 만들기

이번에는 많은 스레드를 어떻게 한 번에 만드는지 알아보자.

단순히 스레드 3개를 생성하고 실행해보자.

```
package thread.start;

import static util.MyLogger.*;

public class ManyThreadMainV1 {
    public static void main(String[] args) {
        log("main() start");

        HelloRunnable runnable = new HelloRunnable();
        Thread thread1 = new Thread(runnable);
        thread1.start();
```

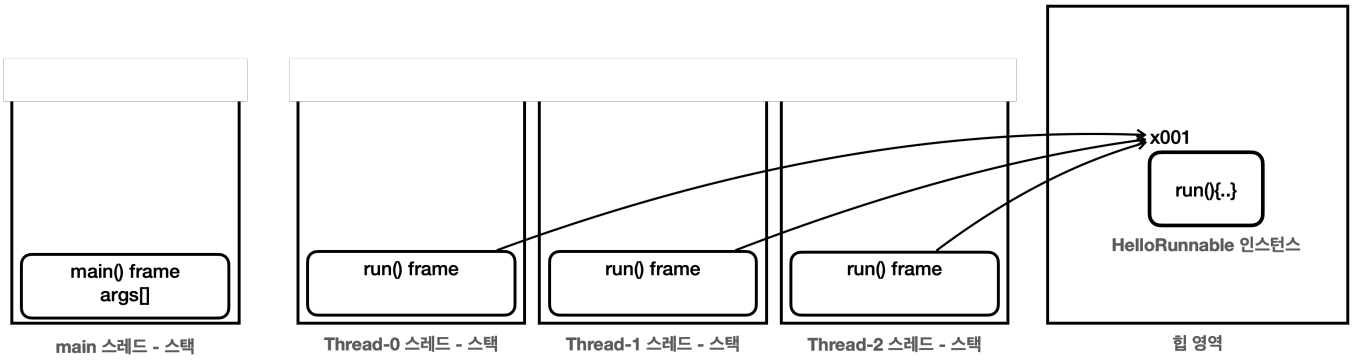
```
        Thread thread2 = new Thread(runnable);
        thread2.start();
        Thread thread3 = new Thread(runnable);
        thread3.start();

        log("main() end");
    }
}
```

**실행 결과**

```
15:52:34.602 [     main] main() start
15:52:34.604 [     main] main() end
Thread-2: run()
Thread-1: run()
Thread-0: run()
```

- 실행 결과는 다를 수 있다. 스레드의 실행 순서는 보장되지 않는다.




- 스레드3개를 생성할 때 모두 같은 `HelloRunnable` 인스턴스(`x001`)를 스레드의 실행 작업으로 전달했다.
- `Thread-0`, `Thread-1`, `Thread-2`는 모두 `HelloRunnable` 인스턴스에 있는 `run()` 메서드를 실행한다.

스레드 100개를 생성하고 실행해보자.

```
package thread.start;

import static util.MyLogger.log;

public class ManyThreadMainV2 {
    public static void main(String[] args) {
```

```
        log("main() start");

        HelloRunnable runnable = new HelloRunnable();
        for (int i = 0; i < 100; i++) {
            Thread thread = new Thread(runnable);
            thread.start();
        }

        log("main() end");
    }
}
```

반복문을 사용하면 스레드의 숫자를 유동적으로 변경하면서 실행할 수 있다.

**실행 결과**

```
16:04:35.503 [     main] main() start
Thread-9: run()
Thread-5: run()
Thread-8: run()
Thread-3: run()
Thread-0: run()
Thread-1: run()
Thread-7: run()
Thread-4: run()
Thread-2: run()
Thread-6: run()
...
16:04:35.505 [     main] main() end
```

- 실행 결과는 다를 수 있다. 스레드의 실행 순서는 보장되지 않는다.

## Runnable을 만드는 다양한 방법

중첩 클래스를 사용하면 `Runnable`을 더 편리하게 만들 수 있다.

(중첩 클래스는 자바 중급1편 참고)

참고로 모두 결과는 같다.

## 정적 중첩 클래스 사용

```
package thread.start;

import static util.MyLogger.log;

public class InnerRunnableMainV1 {

    public static void main(String[] args) {
        log("main() start");

        Runnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start();

        log("main() end");
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            log("run()");
        }
    }
}
```

- 특정 클래스 안에서만 사용되는 경우 이렇게 중첩 클래스를 사용하면 된다.

**실행 결과**

```
16:09:54.882 [     main] main() start
16:09:54.884 [     main] main() end
16:09:54.884 [ Thread-0] run()
```

## 익명 클래스 사용

```
package thread.start;
```

```
import static util.MyLogger.log;

public class InnerRunnableMainV2 {

    public static void main(String[] args) {
        log("main() start");

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                log("run()");
            }
        };
        Thread thread = new Thread(runnable);
        thread.start();

        log("main() end");
    }
}
```

- 특정 메서드 안에서만 간단히 정의하고 사용하고 싶다면 익명 클래스를 사용하면 된다.

### 익명 클래스 변수 없이 직접 전달

```
package thread.start;

import static util.MyLogger.log;

public class InnerRunnableMainV3 {

    public static void main(String[] args) {
        log("main() start");

        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                log("run()");
            }
        });
        thread.start();

        log("main() end");
```

```
    }
}
```

- 익명 클래스를 참조하는 변수를 만들지 않고 직접 전달할 수 있다.

### 람다

```
package thread.start;

import static util.MyLogger.log;

public class InnerRunnableMainV4 {

    public static void main(String[] args) {
        log("main() start");

        // 람다를 배우면 이해
        Thread thread = new Thread(() -> log("run()"));
        thread.start();

        log("main() end");
    }
}
```

- 람다를 사용하면 메서드(함수) 코드 조각을 전달할 수 있다.
- 우리는 아직 람다를 학습하지 않았기 때문에 정적 중첩 클래스나 익명 클래스를 주로 사용하겠다.
- 참고로 람다는 별도의 강의에서 다룰 예정이다.

## 문제와 풀이

### 문제1: Thread 상속

다음 요구사항에 맞게 멀티스레드 프로그램을 작성해라.

1. `Thread` 클래스를 상속받은 `CounterThread`라는 스레드 클래스를 만들자.
2. 이 스레드는 1부터 5까지의 숫자를 1초 간격으로 출력해야 한다. 앞서 우리가 만든 `log()` 기능을 사용해서 출력해라.
3. `main()` 메서드에서 `CounterThread` 스레드 클래스를 만들고 실행해라.

4. 실행 결과를 참고하자.

**실행 결과**

```
09:46:23.329 [ Thread-0] value: 1
09:46:24.332 [ Thread-0] value: 2
09:46:25.338 [ Thread-0] value: 3
09:46:26.343 [ Thread-0] value: 4
09:46:27.349 [ Thread-0] value: 5
```

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest1Main {
    // 여기에 코드 작성
}
```

**정답**

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest1Main {

    public static void main(String[] args) {
        CounterThread thread = new CounterThread();
        thread.start();
    }

    static class CounterThread extends Thread {
        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                log("value: " + i);
```

```
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}
```

## 문제 2: Runnable 구현

다음 요구사항에 맞게 멀티스레드 프로그램을 작성해라.

1. `CounterRunnable` 이라는 이름의 클래스를 만들자, 이 클래스는 `Runnable` 인터페이스를 구현해야 한다.
2. `CounterRunnable`은 1부터 5까지의 숫자를 1초 간격으로 출력해야 한다. 앞서 우리가 만든 `log()` 기능을 사용해서 출력해라.
3. `main` 메서드에서 `CounterRunnable`의 인스턴스를 이용해서 `Thread`를 생성하고 실행해라.
4. 스레드의 이름은 "counter"로 지정해야 한다.

**실행 결과**

```
09:53:36.705 [  counter] value: 1
09:53:37.713 [  counter] value: 2
09:53:38.719 [  counter] value: 3
09:53:39.725 [  counter] value: 4
09:53:40.726 [  counter] value: 5
```

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest2Main {
    // 여기에 코드 작성
}
```

**정답**

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest2Main {

    public static void main(String[] args) {
        Thread thread = new Thread(new CounterRunnable(), "counter");
        thread.start();
    }

    static class CounterRunnable implements Runnable {
        public void run() {
            for (int i = 1; i <= 5; i++) {
                log("value: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

## 문제 3: Runnable 익명 클래스 구현

방금 작성한 문제2를 익명 클래스로 구현해라.

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest3Main {
```

```
    // 여기에 코드 작성
}
```

**정답**

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest3Main {

    public static void main(String[] args) {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                for (int i = 1; i <= 5; i++) {
                    log("value: " + i);
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        };

        Thread thread = new Thread(runnable, "counter");
        thread.start();
    }

}
```

## 문제 4: 여러 스레드 사용

- `Thread-A`, `Thread-B` 두 스레드를 만들어라
- `Thread-A`는 1초에 한 번씩 "A"를 출력한다.

- `Thread-B`는 0.5초에 한 번씩 "B"를 출력한다.
- 이 프로그램은 강제 종료할 때 까지 계속 실행된다.

**실행 결과**

```
10:04:27.000 [ Thread-A] A
10:04:27.000 [ Thread-B] B
10:04:27.507 [ Thread-B] B
10:04:28.006 [ Thread-A] A
10:04:28.012 [ Thread-B] B
10:04:28.518 [ Thread-B] B
10:04:29.011 [ Thread-A] A
10:04:29.023 [ Thread-B] B
... 무한 실행
```

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest4Main {
    // 여기에 코드 작성
}
```

**정답**

```
package thread.start.test;

import static util.MyLogger.log;

public class StartTest4Main {

    public static void main(String[] args) {
        Thread threadA = new Thread(new PrintWork("A", 1000), "Thread-A");
        Thread threadB = new Thread(new PrintWork("B", 500), "Thread-B");

        threadA.start();
```

```
        threadB.start();
    }

    static class PrintWork implements Runnable {
        private String content;
        private int sleepMs;

        public PrintWork(String content, int sleepMs) {
            this.content = content;
            this.sleepMs = sleepMs;
        }

        @Override
        public void run() {
            while (true) { // 무한 루프
                log(content);
                try {
                    Thread.sleep(sleepMs);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}
```

## 정리