

1. 문자 인코딩

#1.인강/0.자바/6.자바-고급2편

- /프로젝트 환경 구성
- /컴퓨터와 데이터
- /컴퓨터와 문자 인코딩1
- /컴퓨터와 문자 인코딩2
- /문자 집합 조회
- /문자 인코딩 예제1
- /문자 인코딩 예제2
- /정리

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



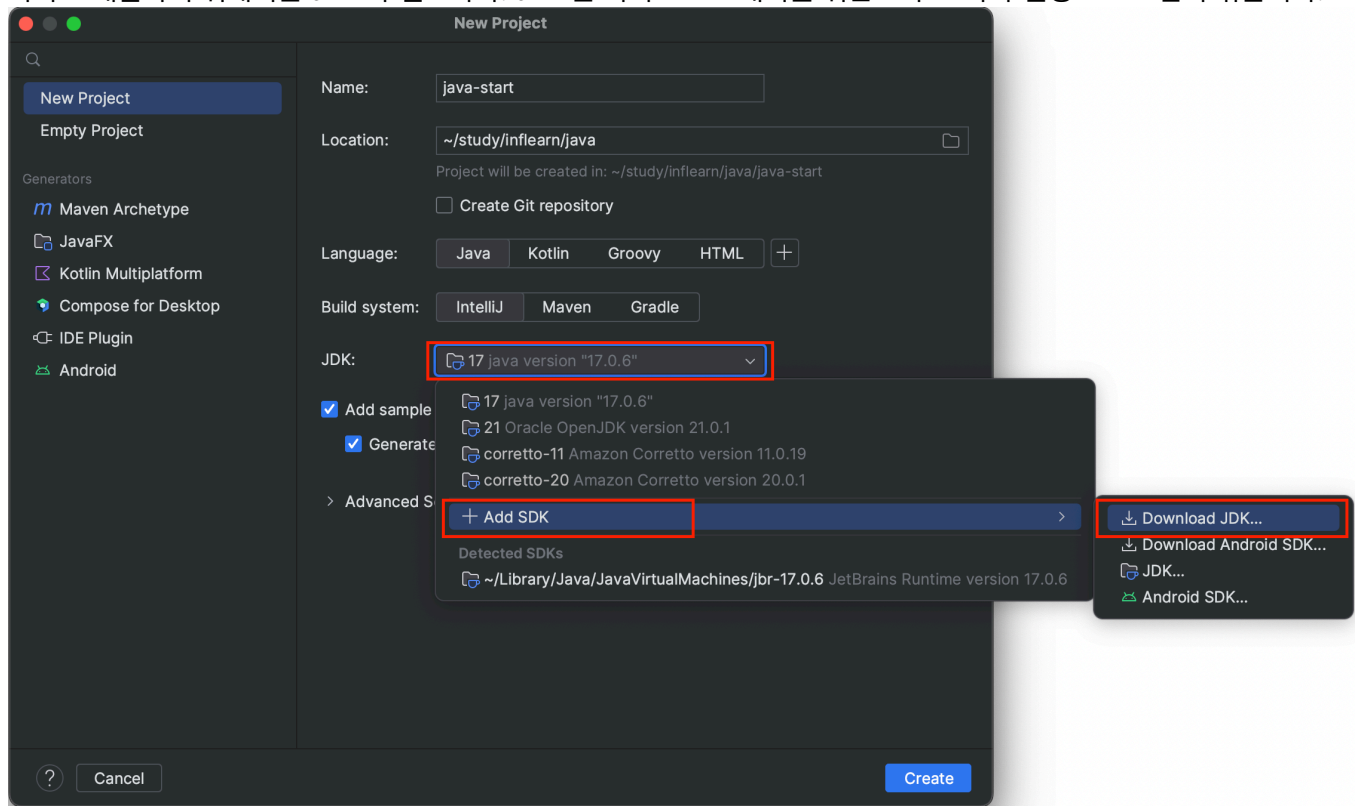
- New Project를 선택해서 새로운 프로젝트를 만들자

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
 - 자바 고급2편 강의: **java-adv2**
- 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

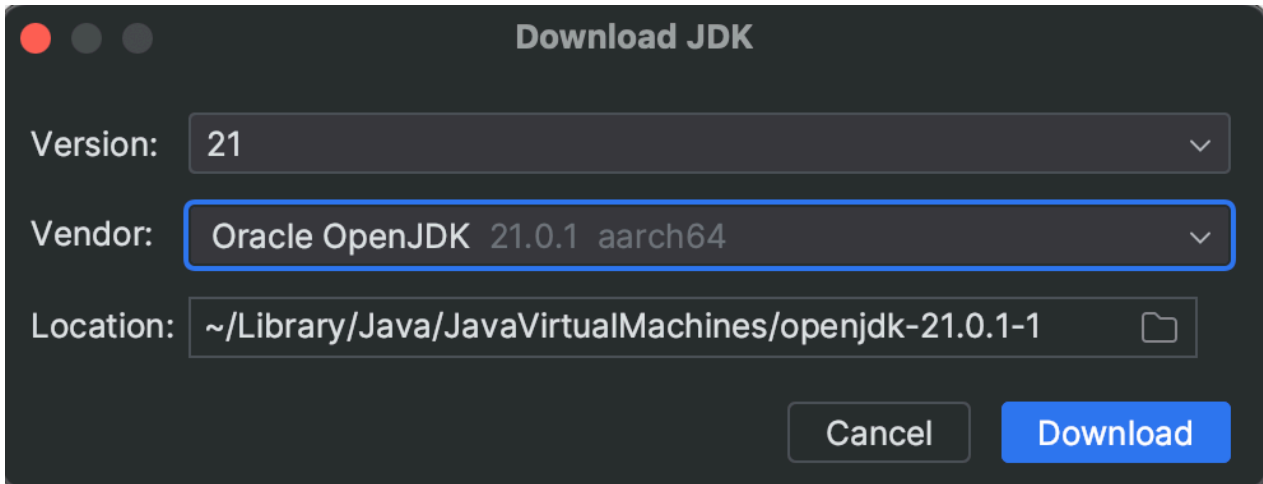
자바로 개발하기 위해서는 JDK가 필요하다. JDK는 자바 프로그래머를 위한 도구 + 자바 실행 프로그램의 묶음이다.



- **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
- 자바 고급2편 강의: **java-adv2**

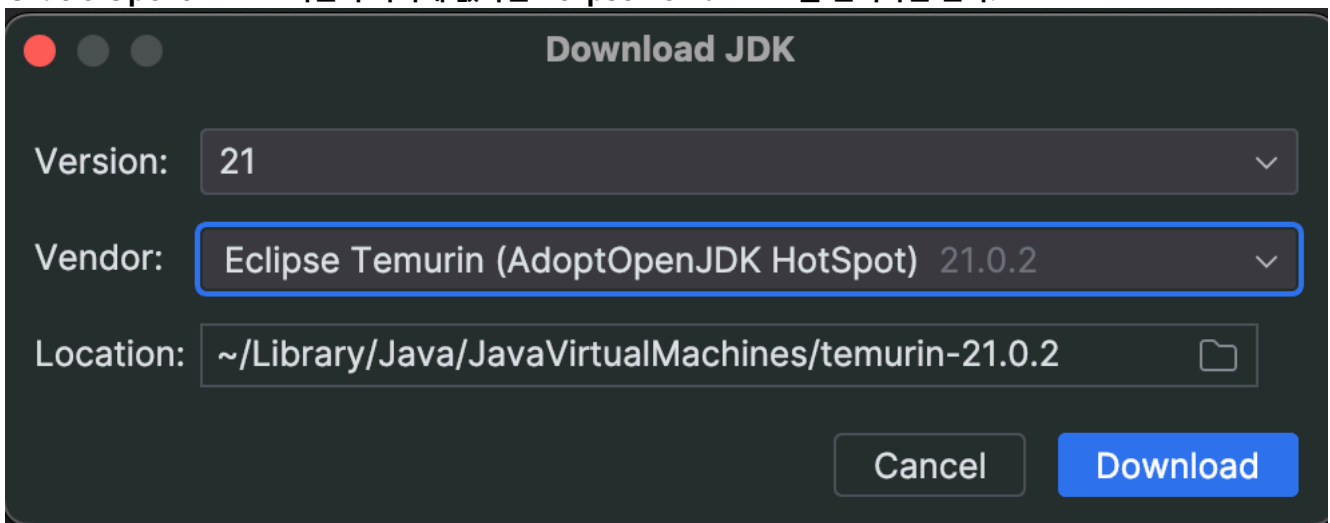
JDK 다운로드 화면



- 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

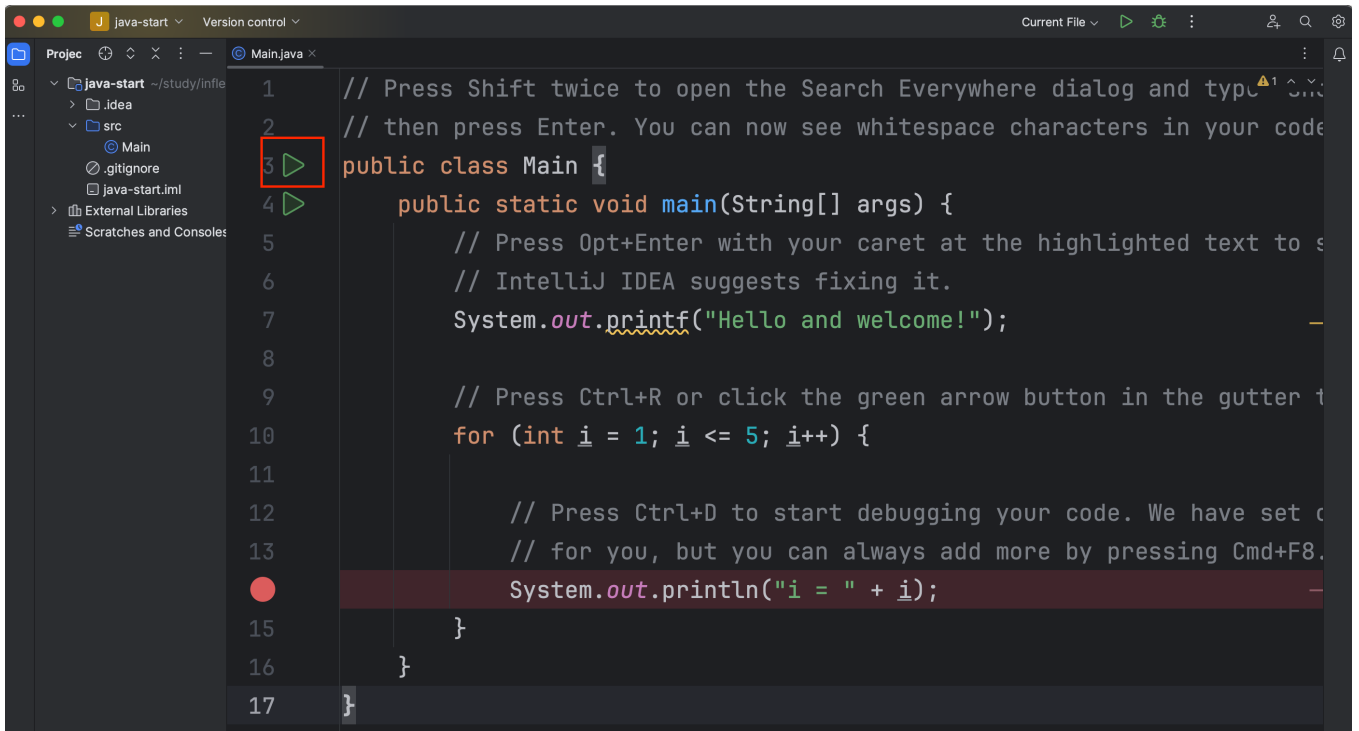
Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.



Download 버튼을 통해서 다운로드 JDK를 다운로드 받는다.

다운로드가 완료 되고 이전 화면으로 돌아가면 Create 버튼 선택하자. 그러면 다음 IntelliJ 메인 화면으로 넘어간다.

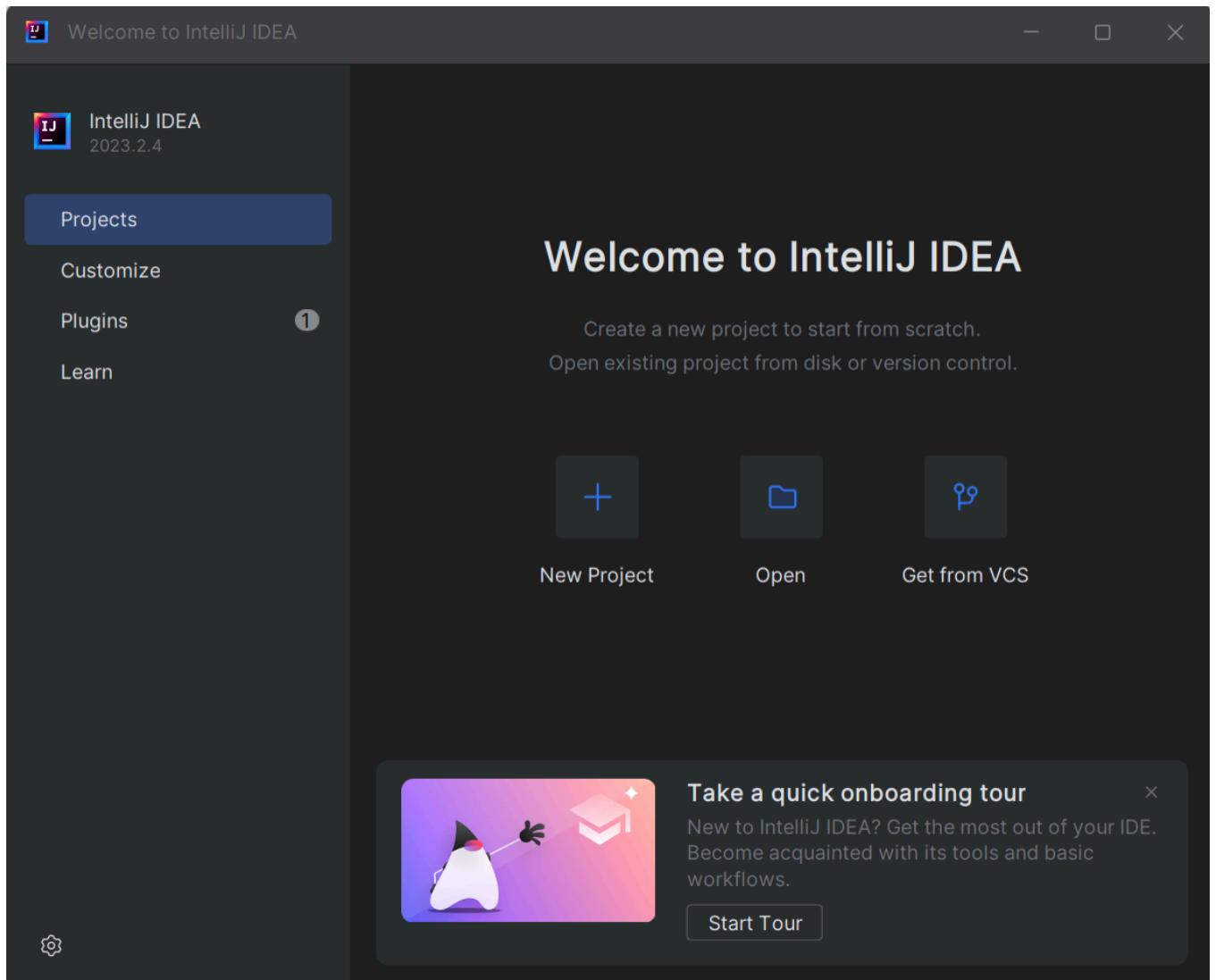
IntelliJ 메인 화면



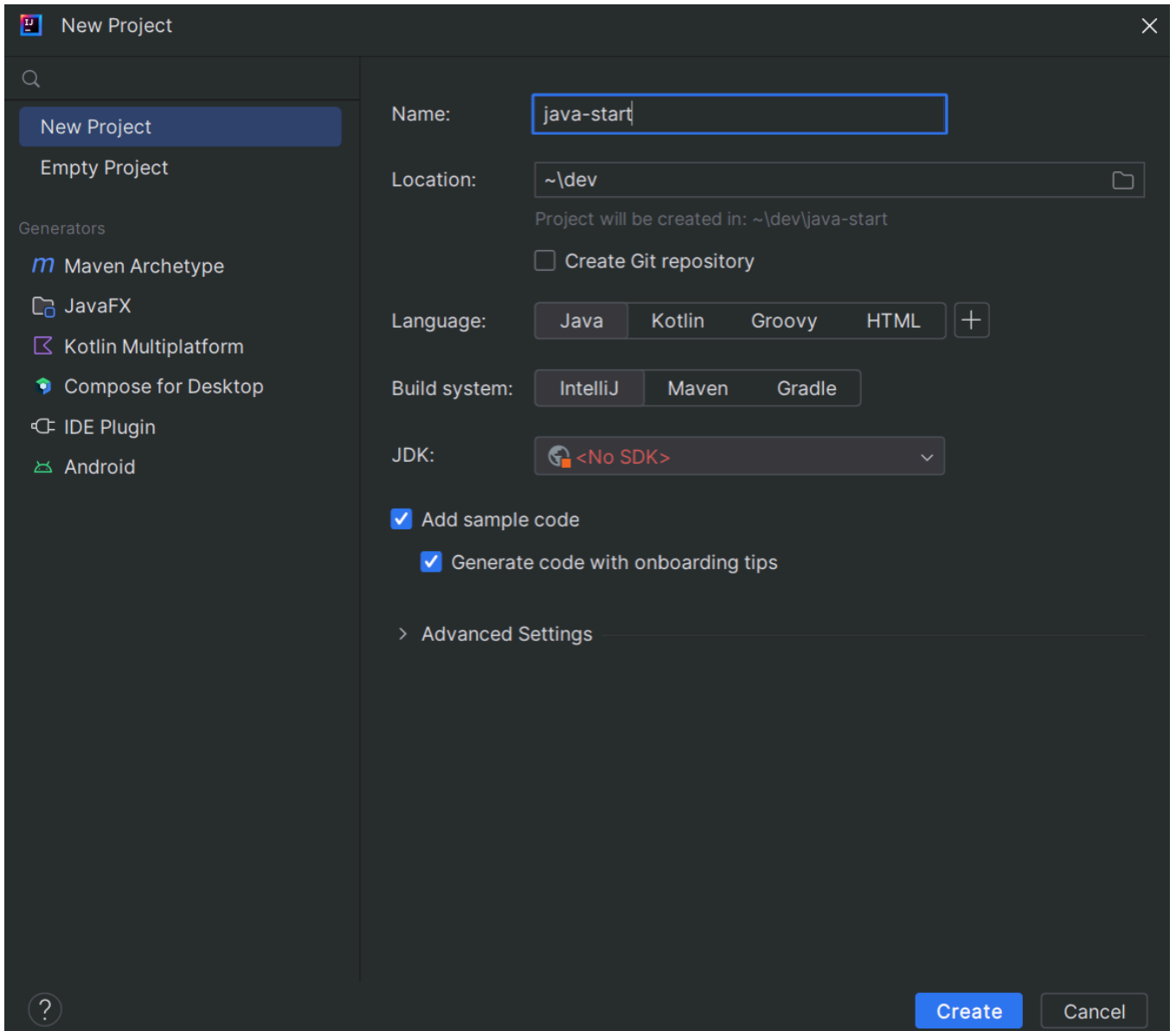
- 앞서 Add sample code 선택해서 샘플 코드가 만들어져 있다.
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 Run 'Main.main()' 버튼을 선택하면 프로그램이 실행된다.

윈도우 사용자 추가 설명서

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



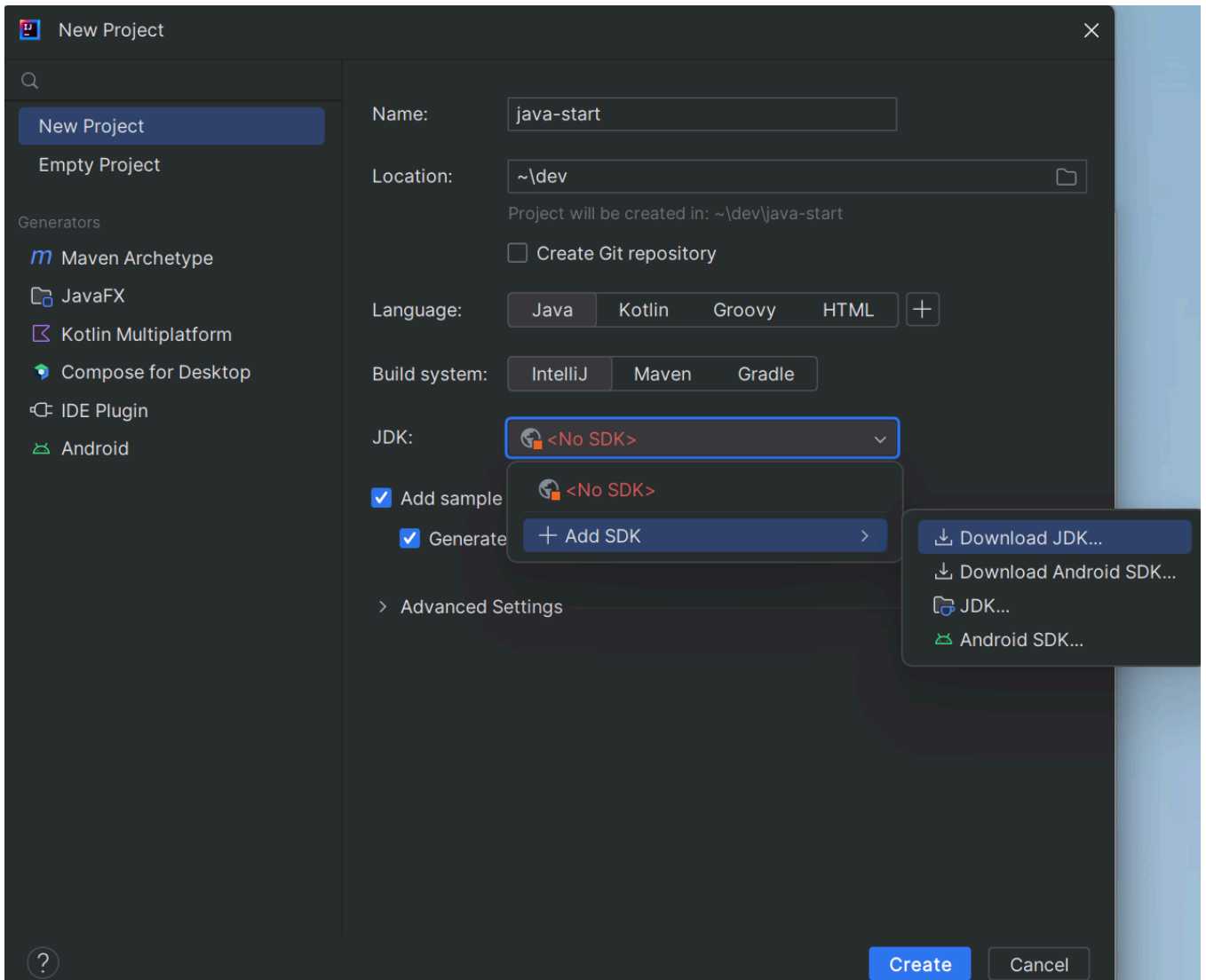
- 프로그램 시작 화면
- New Project 선택



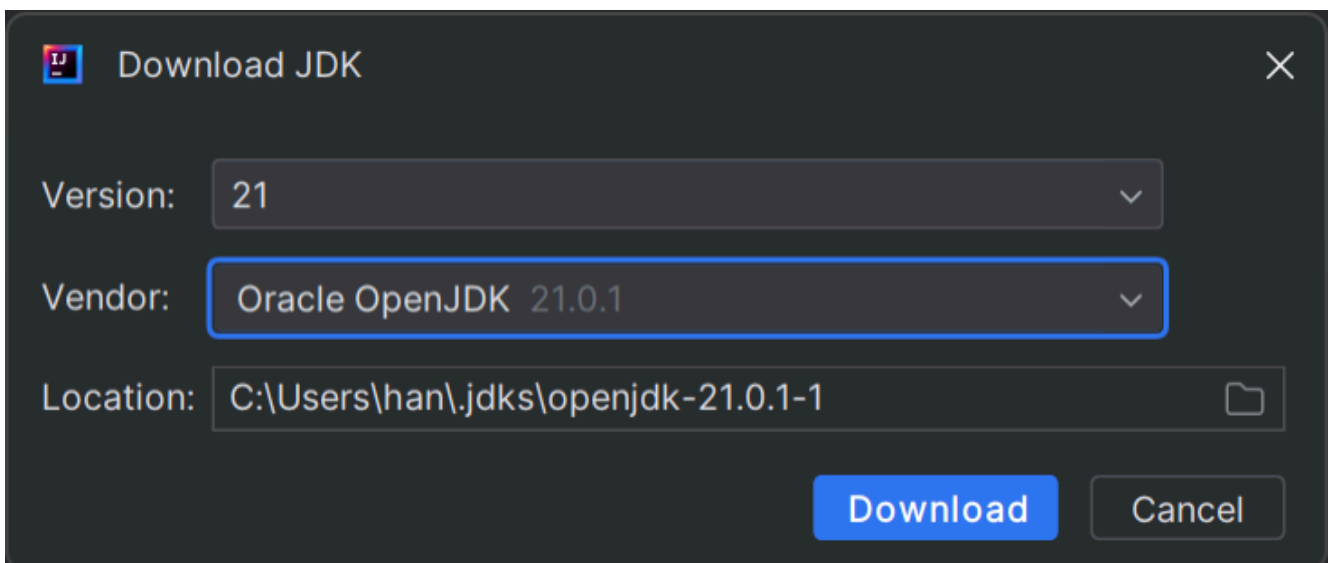
New Project 화면

- **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
 - 자바 고급2편 강의: **java-adv2**
- 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.

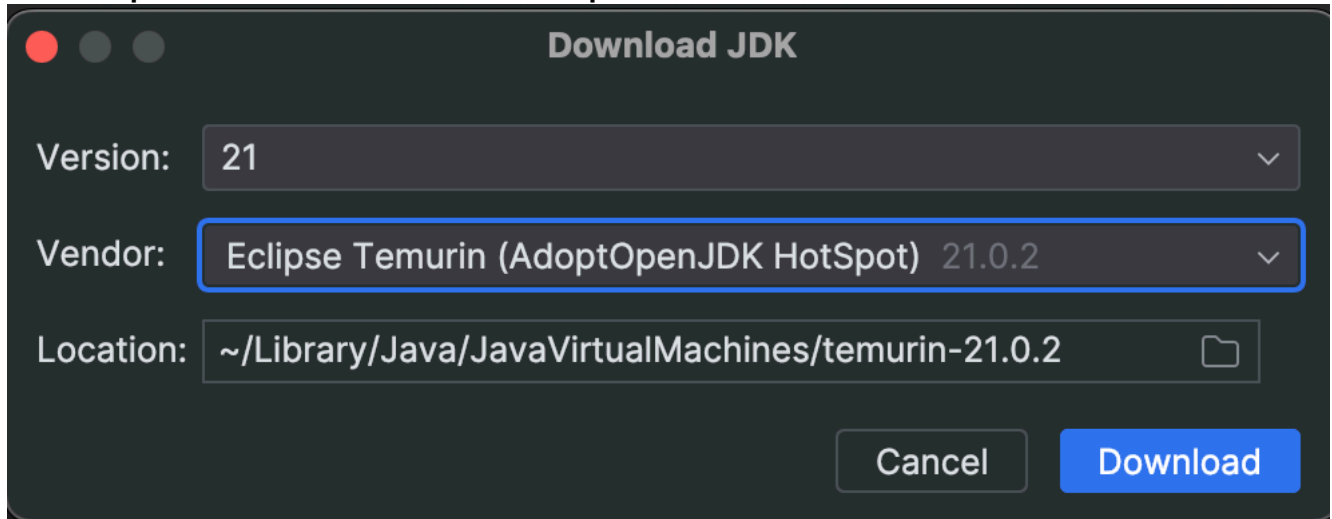


- Version: 21

- Vendor: Oracle OpenJDK
- Location은 가급적 변경하지 말자.

주의 - 변경 사항

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



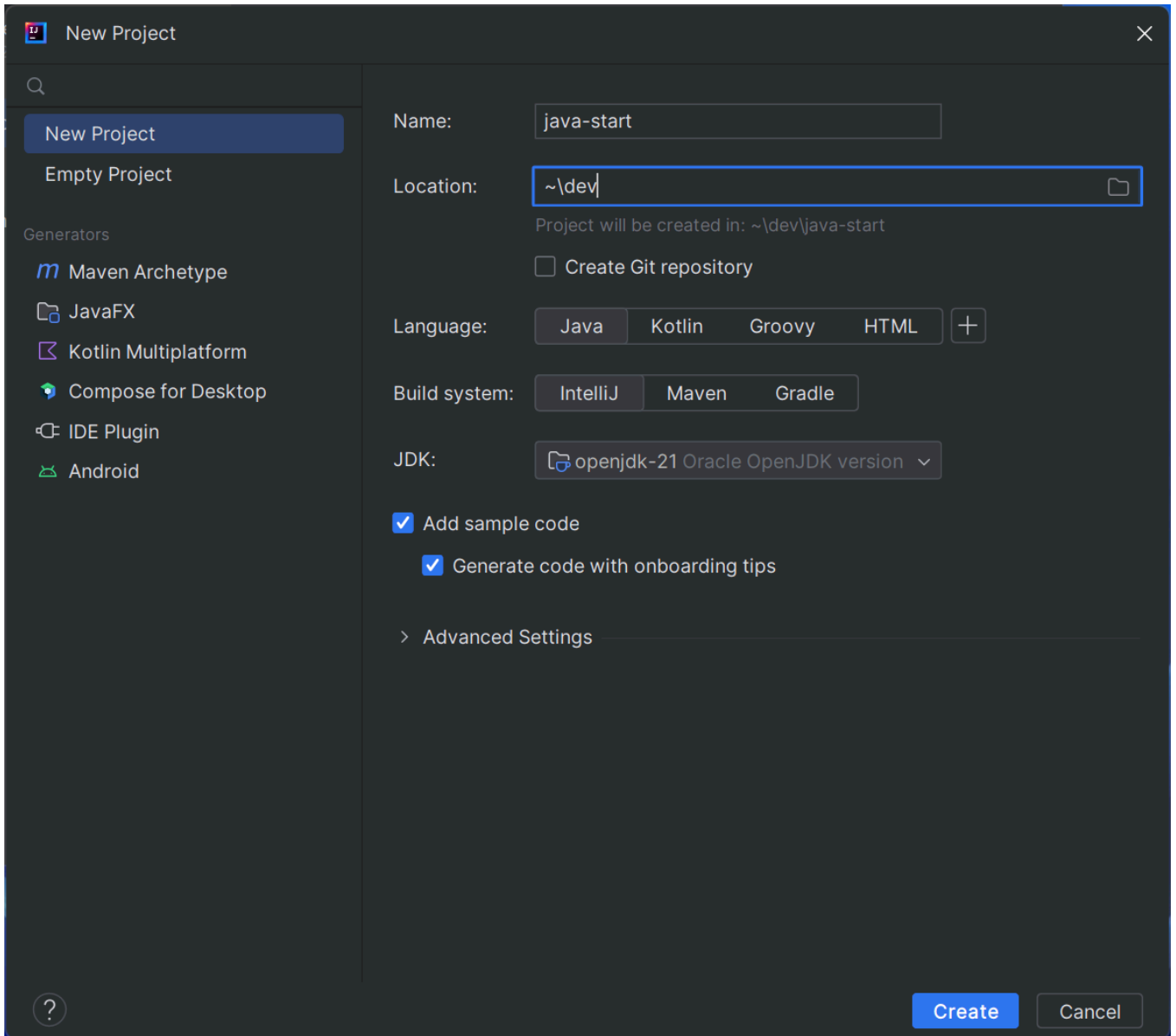
Download JDK

Version: 21

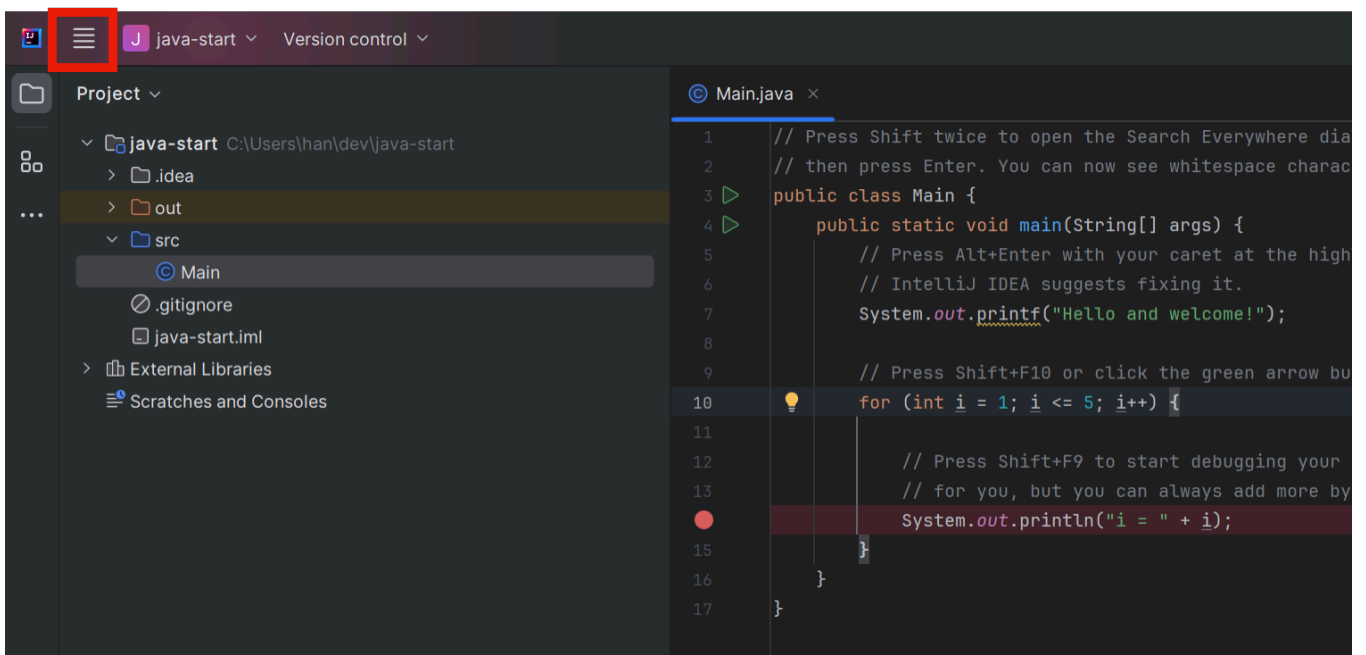
Vendor: Eclipse Temurin (AdoptOpenJDK HotSpot) 21.0.2

Location: ~/.Library/Java/JavaVirtualMachines/temurin-21.0.2

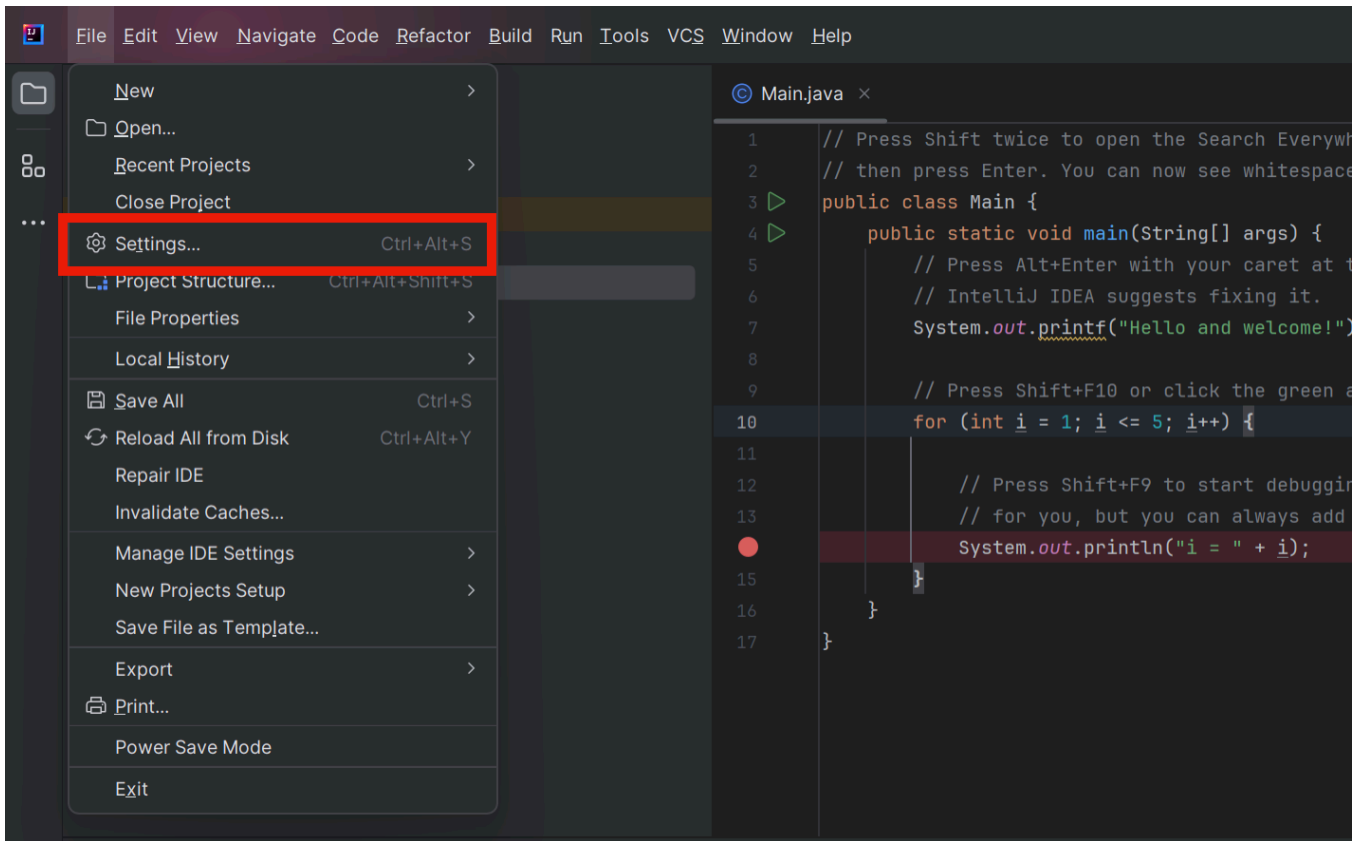
Cancel Download



- New Project 완료 화면



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



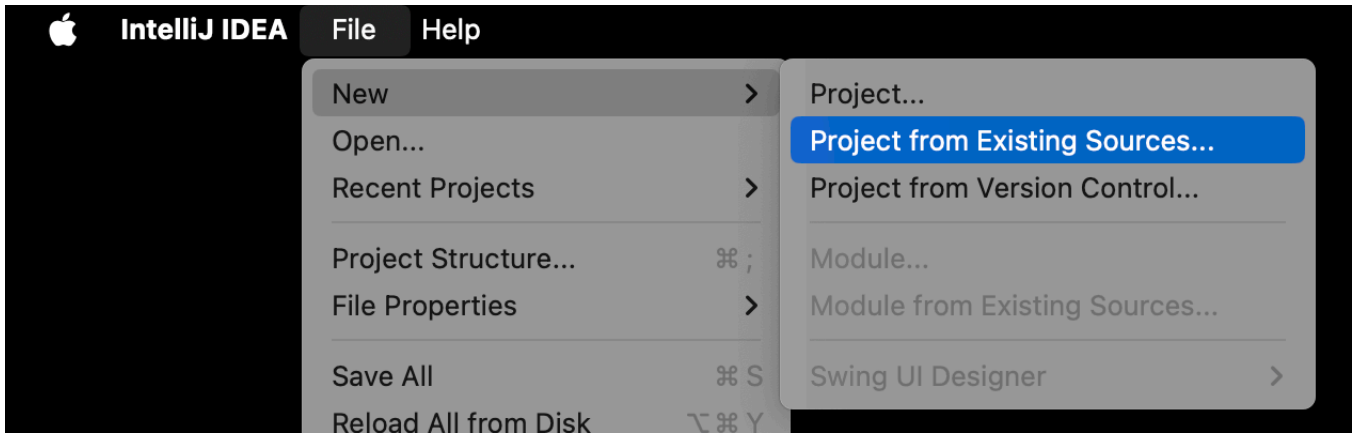
- Mac과 다르게 Settings... 메뉴가 File에 있다. 이 부분이 Mac과 다르므로 유의하자.

한글 언어팩 → 영어로 변경

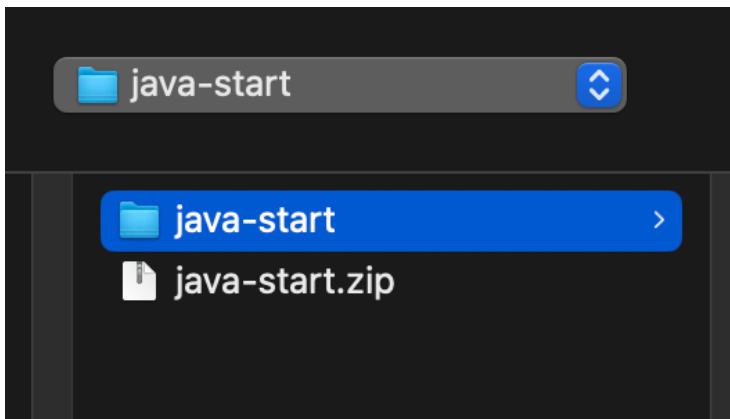
- IntelliJ는 가급적 한글 버전 대신, 영문 버전을 사용하자. 개발하면서 필요한 기능들을 검색하게 되는데, 영문으로 된 자료가 많다. 이번 강의도 영문을 기준으로 진행한다.
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- **Mac:** IntelliJ IDEA(메뉴) → Settings... → Plugins → Installed
- **윈도우:** File → Settings... → Plugins → Installed
 - Korean Language Pack 체크 해제
 - OK 선택후 IntelliJ 다시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

다운로드 소스 코드 실행 방법

영상 참고

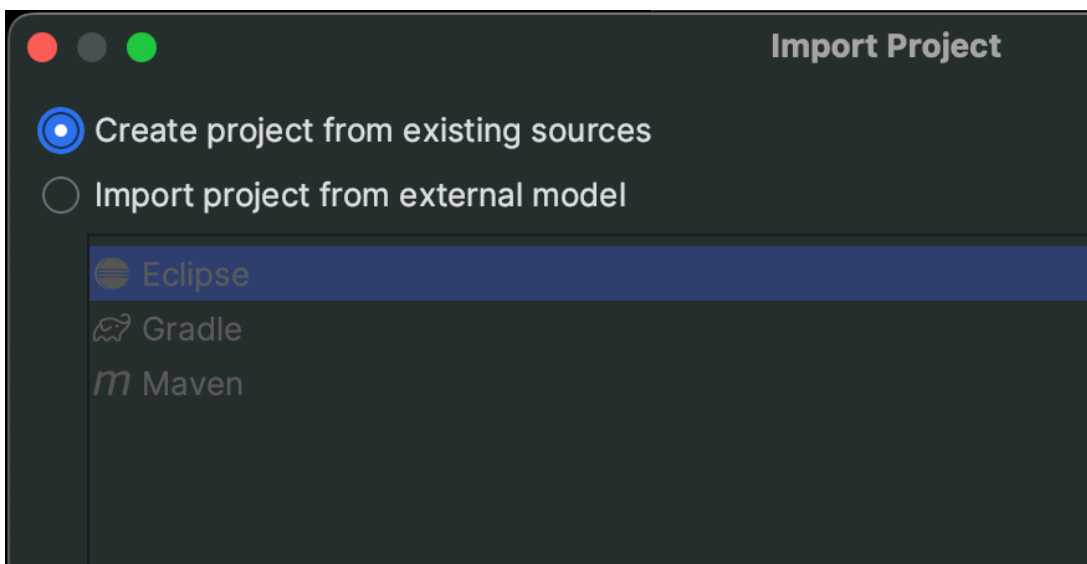


File -> New -> Project from Existing Sources... 선택



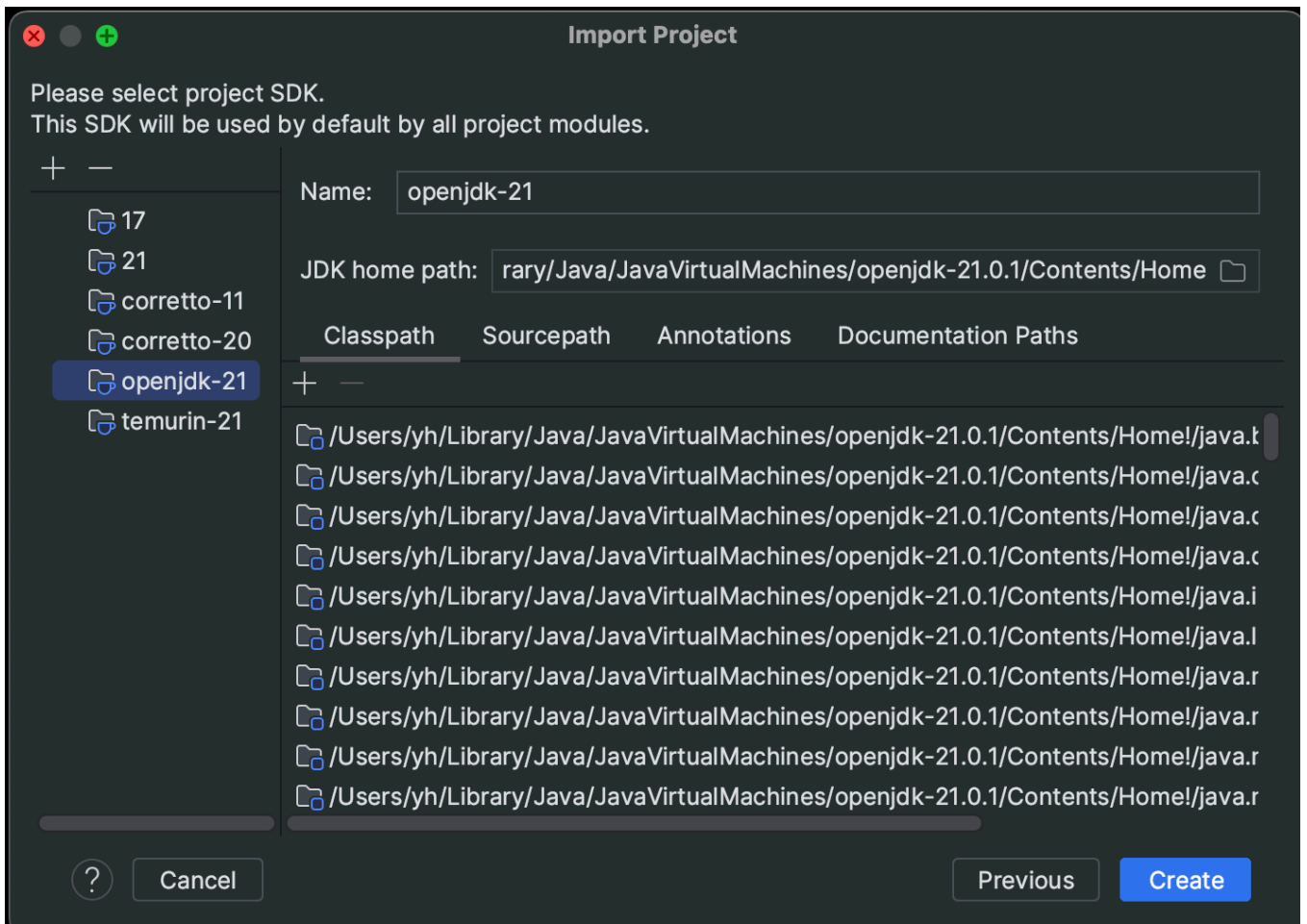
압축을 푼 프로젝트 폴더 선택

- 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
- 자바 고급2편 강의: **java-adv2**

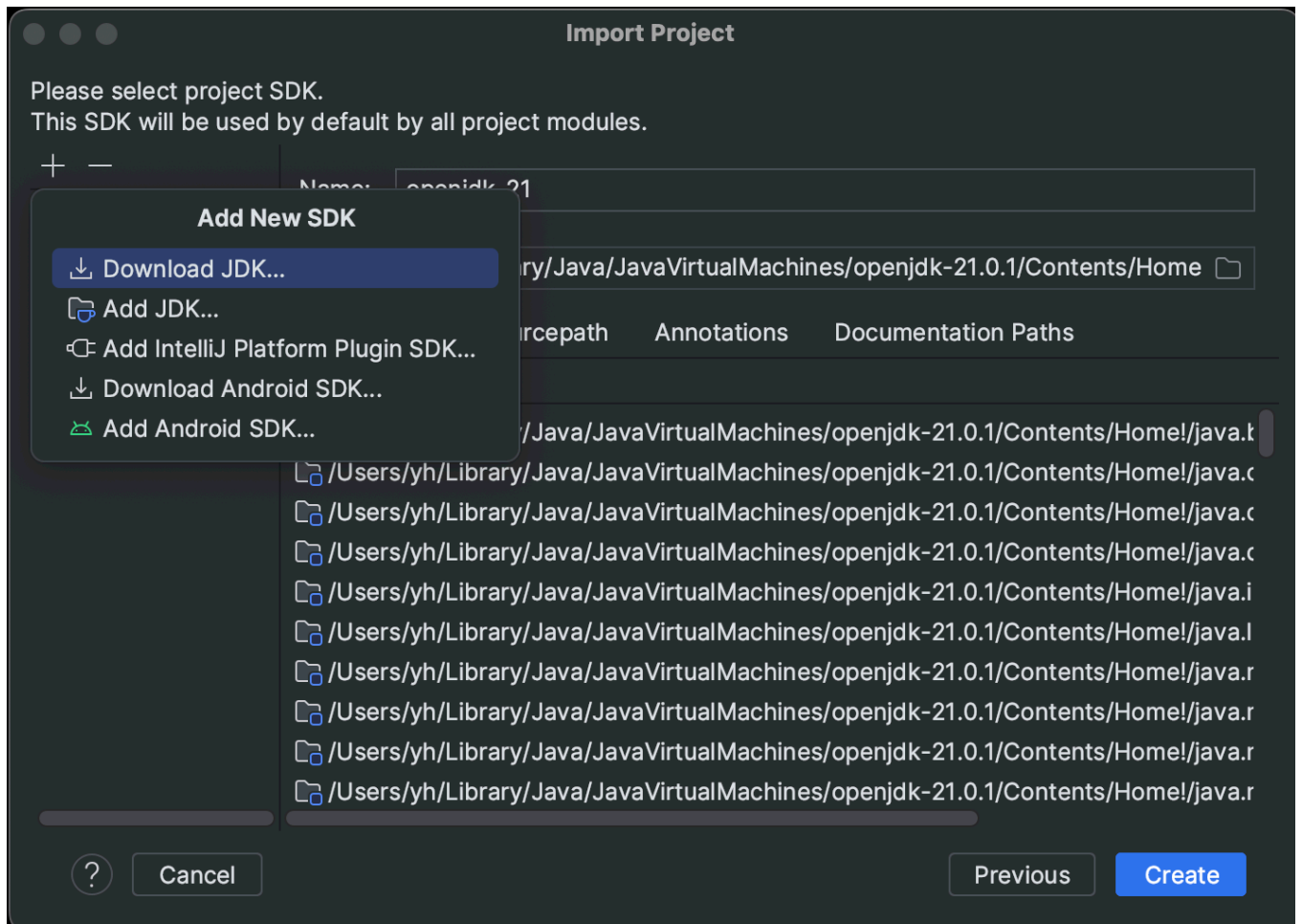


Create project from existing sources 선택

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

컴퓨터와 데이터

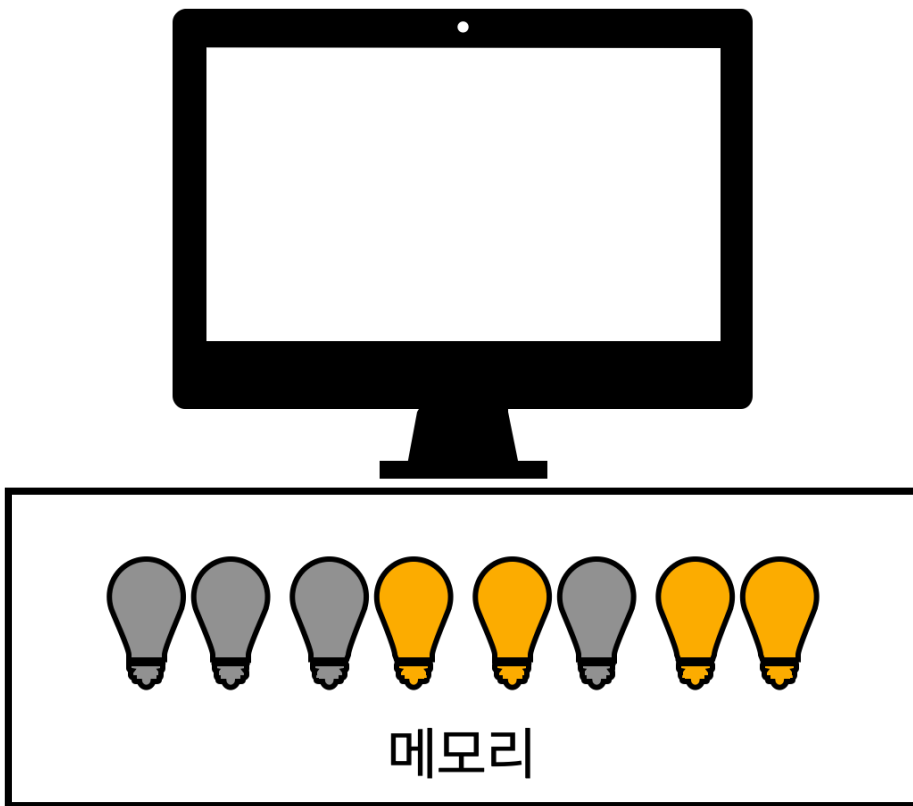
개발자가 개발하며 다루는 데이터는 크게 010101로 되어 있는 바이너리 데이터(또는 byte 기반의 데이터)와 "ABC", "가나

다"와 같은 문자로 되어 있는 텍스트 데이터 두 가지다.

텍스트 데이터가 어떤 원리를 사용해서 만들어지는지 제대로 이해하지 못하면, 실무에서 한글 글자가 이상하게 깨져서 나올 때, 근본적인 원인을 찾아서 해결하기 어렵다.

실무에서 보면 생각보다 많은 개발자들이 텍스트 데이터와 문자 인코딩의 원리를 정확히 알고 있지 않아서, 이런 문제를 만났을 때 많이 고생하는 모습을 보았다.

이제 본격적으로 입출력을 다루기 전에, 가장 기본적인 컴퓨터가 데이터를 저장하는 원리부터 시작해서, 실무에 꼭 필요한 문자 인코딩까지 기본 이론을 확실히 이해하고 넘어가자



컴퓨터의 메모리는 반도체로 만들어져 있는데, 이것은 쉽게 이야기해서 수 많은 전구들이 모여있는 것이다. 이 '전구들'은 사실 트랜지스터라고 불리는 아주 작은 전자 스위치이다. 각 트랜지스터는 전기가 흐르거나 흐르지 않는 두 가지 상태를 가질 수 있어서, 이를 통해 0과 1이라는 이진수를 표현한다.

이 트랜지스터들이 모여 메모리를 구성한다. 우리가 흔히 말하는 RAM(Random Access Memory)은 이런 방식으로 만들어진 메모리의 한 종류이다.

컴퓨터가 정보를 저장하거나 처리할 때, 이 '전구들'을 켜고 끄는 방식으로 데이터를 기록하고 읽어들인다. 이 과정은 매우 빠르게 일어나며, 현대의 컴퓨터 메모리는 초당 수십억 번의 데이터 접근을 처리할 수 있다.

여기서 핵심은 메모리라는 것은 단순히 전구를 켜고 끄는 방식으로 작동한다는 점이다. 그렇다면 여기에 우리가 사용하는 10진수 숫자 데이터를 어떻게 메모리에 저장할 수 있을까?

2진수

전구를 켜고 끈다는 것은 0과 1만 나타낼 수 있는 2진수로 표현할 수 있다.

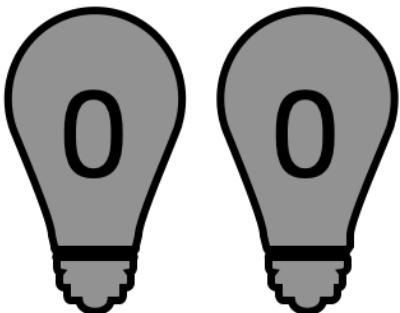


- 전구를 끈다: 숫자 0

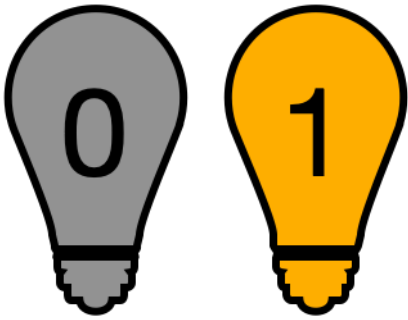


- 전구를 켜다: 숫자 1

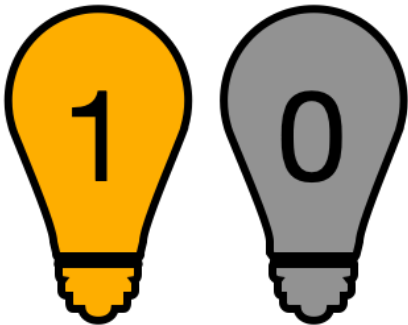
숫자 0을 메모리에 저장한다면 메모리의 전구를 하나 끄면 되고, 숫자 1을 저장한다면 전구를 하나 켜면 된다.
그렇다면 숫자 2나 3은 어떻게 표현할 수 있을까? 숫자 2나 3을 표현하려면 전구를 하나 더 사용하면 된다.



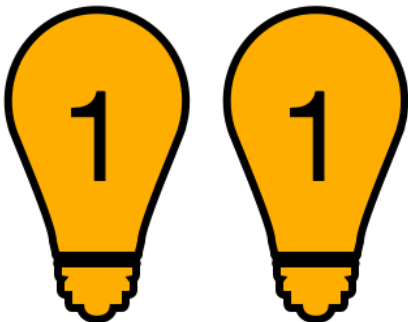
- 숫자 0



- 숫자 1



- 숫자 2



- 숫자 3

전구 1개는 단지 0과 1이라는 2가지를 표현할 수 있지만, 전구 2개를 함께 묶어서 사용하면 총 4가지를 표현할 수 있다.

예를 들어서 숫자 3을 메모리에 저장한다면 컴퓨터는 메모리의 전구 2개를 모두 켜고.

값을 읽을 때도 마찬가지다. 메모리에서 전구 2개를 읽고, 만약 둘다 켜져있다면 숫자 3을 화면에 출력한다.

여기서 핵심은 컴퓨터는 사람과 같이 10진수 숫자를 이해하고 숫자를 메모리에 저장하거나 불러오는 것이 아니라는 점이다. 단지 전구의 상태만 변경하거나 확인할 뿐이다.

앞으로 0은 전구가 꺼진 상태, 1은 전구가 켜진 상태라 하겠다.

전구 1개와 같이 2가지만 표현할 수 있는 것을 **1비트(bit)**라고 한다.

- 1bit: 2가지 표현

- 0
- 1
- 2bit: 4가지 표현
 - 00, 01
 - 10, 11
- 3bit: 8가지 표현
 - 000, 001, 010, 011
 - 100, 101, 110, 111
- 4bit: 16가지 표현
 - 0000, 0001, 0010, 0011
 - 0100, 0101, 0110, 0111
 - 1000, 1001, 1010, 1011
 - 1100, 1101, 1110, 1111

1bit를 추가할 때 마다 표현할 수 있는 숫자는 2배씩 늘어난다.

- 1bit → 2 (0 ~ 1)
- 2bit → 4 (0 ~ 3)
- 3bit → 8 (0 ~ 7)
- 4bit → 16 (0 ~ 15)
- 5bit → 32 (0 ~ 31)
- 6bit → 64 (0 ~ 63)
- 7bit → 128 (0 ~ 127)
- 8bit → 256 (0 ~ 255)

참고: 8bit = 1byte

예를 들어 전구 8개를 함께 묶어서 사용하면 총 256가지 상태를 표현할 수 있고, 숫자 0 ~ 255(0포함 256가지)를 표현할 수 있다.

숫자 저장 예시

그렇다면 우리가 일반적으로 사용하는 10진수 100을 컴퓨터에 저장한다면 어떻게 될까? 컴퓨터는 10진수를 이해하지 못한다.

10진수 100을 메모리에 저장한다면 컴퓨터는 10진수 100을 2진수로 1100100 변경해서 저장한다.

참고로 간단한 수학 공식으로 10진수는 2진수로 쉽게 변경할 수 있고 그 반대도 가능하다.

bit를 다룰 때 사용하는 2진수는 사람이 직관적으로 이해하기 어렵다.

2진수는 10진수로 쉽게 변환할 수 있으므로 앞으로는 이해하기 쉽게 2진수 대신 10진수로 설명하겠다.

참고: 음수 표현

음수를 표현해야 한다면 처음 1bit를 음수, 양수를 표현하는데 사용한다.

8bit → 256가지 표현

- 0과 양수만 표현하는 경우:
 - 8bit 모두 숫자 표현에 사용
 - 0 ~ 255
- 음수 표현이 필요한 경우:
 - 1bit는 음수와 양수를 구분하는데 사용, 나머지 7bit로 숫자 범위 사용
 - 0 ~ 127 (양수 표현시 첫 비트를 0으로 사용, 나머지 7bit로 128가지 0과 양수 숫자 표현)
 - -128 ~ -1(음수 표현시 첫 비트를 1로 사용, 나머지 7bit로 128가지 음수 숫자 표현)

컴퓨터와 문자 인코딩1

간단한 수학 공식을 사용하면 사람이 사용하는 10진수를 컴퓨터가 사용하는 2진수로 쉽게 변경할 수 있다. 따라서 컴퓨터는 10진수를 2진수로 변경해서 메모리(전구)에 저장할 수 있다.

그렇다면 숫자가 아닌 문자는 어떻게 메모리에 저장할 수 있을까? 컴퓨터는 전구를 켜고 끄는 2진수만 알고 있다. 10진수는 정해진 수학 공식을 사용하면 쉽게 2진수로 변환할 수 있지만, 문자 'A', 'B'를 2진수로 변경하는 수학 공식 같은 것은 세상에 없다.

이런 문제를 해결하기 위해 초창기 컴퓨터 과학자들은 문자 집합을 만들고, 각 문자에 숫자를 연결시키는 방법을 생각해 냈다.

문자 집합(Character Set)

문자 저장	문자 집합(Character Set)		숫자 조회
	문자	숫자	
	A	65	
	B	66	
	a	97	
	b	98	

예를 들어 우리가 문자 'A'를 저장하면, 컴퓨터는 문자 집합을 통해 'A'의 숫자 값 65를 찾는다. 그리고 65를 메모리에 저장한다. (물론 2진수로 변환해서 저장한다.)

메모리에 저장된 문자를 불러올 때는 반대로 작동한다. 메모리에 저장된 숫자 값 65를 불러온다. 그리고 문자 집합을 통해 문자 'A'를 찾아서 화면에 출력한다.

- **문자 인코딩:** 문자 집합을 통해 문자를 숫자로 변환하는 것
- **문자 디코딩:** 문자 집합을 통해 숫자를 문자로 변환하는 것

ASCII 문자 집합

각 컴퓨터 회사가 독자적인 문자 집합을 사용한다면, 서로 다른 컴퓨터 간에 문자가 올바르게 표시되지 않는 문제가 발생할 수 있다. 이러한 호환성 문제를 해결하기 위해 ASCII(American Standard Code for Information Interchange)라는 표준 문자 집합이 1960년도에 개발되었다.

초기 컴퓨터에서는 주로 영문 알파벳, 숫자, 키보드의 특수문자, 스페이스, 엔터와 같은 기본적인 문자만 표현하면 충분했다. 따라서 7비트를 사용하여 총 128가지 문자를 표현할 수 있는 ASCII 공식 문자 집합이 만들어졌다.

제어 문자 (0-31, 127)

십진수	문자	설명
0	NUL	Null 문자
1	SOH	헤더 시작
2	STX	텍스트 시작
3	ETX	텍스트 끝
4	EOT	전송 끝
...		
27	ESC	Escape
127	DEL	삭제

출력 가능한 문자 (32-126)

십진수	문자	설명
32		공백

33	!	느낌표
34	"	큰따옴표
...	# \$ % & ' () * + , - . /	특수 문자
48-57	0-9	숫자
58-62	: ; < = >	특수문자
63	?	물음표
64	@	앳 기호
65-90	A-Z	대문자
...		
97-122	a-z	소문자
...		
126	~	틸드

- ASCII의 숫자는 10진수 숫자가 아니라 문자로 표현된 숫자이다. 예를 들어 컴퓨터 입장에서 문자는 그림과 같은 것이다. 여기서 설명하는 ASCII의 숫자는 컴퓨터 입장에서는 그림으로 된 숫자이다. 쉽게 이야기해서 String 타입에 들어있는 "123"으로 이해하면 된다.

ISO_8859_1

서유럽을 중심으로 컴퓨터 사용 인구가 늘어나면서, 서유럽 문자를 표현하는 문자 집합이 필요해졌다.

ISO_8859_1

- 1980년도
- 기존 ASCII에 서유럽 문자의 추가 필요
- 국제 표준화 기구에서 서유럽 문자를 추가한 새로운 문자 규격을 만들
- ISO_8859_1, LATIN1, ISO-LATIN-1 등으로 불림
 - 8bit(1byte) 문자 집합 → 총 256가지 표현 가능
 - 기존 7비트 ASCII(0-127)를 그대로 유지
 - ASCII에 128가지 문자를 추가함(주로 서유럽 문자, 추가 특수 문자들) 예) À, Á, Â, Ã, Ä, Å
- 기존 ASCII 문자 집합과 호환 가능

한글 문자 집합

한국에도 컴퓨터 사용 인구가 늘어나면서, 한글을 표현할 수 있는 문자 집합이 필요해졌다.

EUC-KR

- 1980년도
- 초창기 등장한 한글 문자 집합(더 이전에 KS5601이 있었음)
- 모든 한글을 담는 것 보다는 자주 사용하는 한글 2350개만 포함해서 만들었다.
- 한글의 글자는 아주 많기 때문에 256가지만 표현할 수 있는 1byte로 표현하는 것은 불가능하다.
- 2byte(16bit)를 사용하면 총 65536가지 표현 가능
- ASCII + 자주 사용하는 한글 2350개 + 한국에서 자주 사용하는 기타 글자
 - 한국에서 자주 사용하는 한자 4,888개
 - 일본어 가타카나등도 함께 포함
- ASCII는 1byte, 한글은 2byte를 사용한다.
 - 영어를 사용하면 1byte를, 한글을 사용하면 2byte를 메모리에 저장한다.
- 기존 ASCII 문자 집합과 호환 가능

MS949

- 1990년도
- 마이크로소프트가 EUC-KR을 확장하여 만든 인코딩
- 한글 초성, 중성 종성 모두 조합하면 가능한 한글의 수는 총 11,172자
- EUC-KR은 "뽕", "뽕"과 같이 드물게 사용하는 음절을 표현하지 못함
- 기존 EUC-KR과 호환을 이루면서 한글 11,172자를 모두 수용하도록 만든 것이 MS949
- EUC-KR과 마찬가지로 ASCII는 1byte, 한글은 2byte를 사용함
- 기존 ASCII 문자 집합과 호환 가능
- 윈도우 시스템에서 계속 사용됨

컴퓨터와 문자 인코딩2

전세계 문자 집합

전세계적으로 컴퓨터 인구가 늘어나면서, 전세계 문자를 대부분 다 표현할 수 있는 문자 집합이 필요해졌다.

문제

- EUC-KR이나 MS949 같은 한글 문자표를 PC에 설치하지 않으면 다른 나라 사람들은 한글로 작성된 문서를 열 어볼 수 없다.
- 우리도 마찬가지다. 히브리어, 아랍어를 보려면 각 나라의 문자표가 필요하다.

- 한 문서 안에 영어, 한글, 중국어, 일본어, 히브리어, 아랍어를 함께 저장해야 한다면?
- 1980년대 말, 다양한 문자 인코딩 표준이 존재했지만, 이들은 모두 특정 언어 또는 문자 세트를 대상으로 했기 때문에 국제적으로 호환성 문제가 많았다.

유니코드의 등장

- 이를 해결하기 위해 전 세계의 모든 문자들을 단일 문자 세트로 표현할 수 있는 유니코드(Unicode) 표준이 1990년대에 도입되었다.
- 하나의 문자 세트에 전 세계 대부분의 언어를 넣어보자! 이름하여 유니코드 → Uni(Universal) → 전 세계적인 코드!
- 전 세계의 모든 문자와 기호를 하나의 표준으로 통합하여 표현할 수 있는 문자 집합을 만드는 것
- UTF-16, UTF-8의 시작
- 두 표준이 비슷하게 등장, 초반에는 UTF-16이 인기

UTF-16

- 1990년도
- 16 bit(2byte) 기반
- 자주 사용하는 기본 다국어들은 2byte로 표현, 2byte는 65536가지를 표현할 수 있다.
 - 영어, 유럽 언어, 한국어, 중국어, 일본어등이 2byte를 사용한다.
- 그 외는 4byte로 표현 4byte는 42억 가지를 표현할 수 있다.
 - 고대 문자, 이모지, 중국어 확장 한자등
- **단점: ASCII 영문도 2byte를 사용한다. ASCII와 호환되지 않음**
 - UTF-16을 사용한다면 영문의 경우 다른 문자 집합 보다 2배의 메모리를 더 사용한다.
 - 웹에 있는 문서의 80% 이상은 영문 문서이다.
 - ASCII와 호환되지 않는다는 점도 큰 단점 중 하나이다.
- 초반에는 UTF-16이 인기, 이 시기에 등장한 자바도 언어 내부적으로 문자를 표현할 때 UTF-16을 사용함, 그래서 자바의 char 타입이 2byte를 사용함
- 대부분의 문자를 2byte로 처리하기 때문에 계산이 편리함

UTF-8

- 1990년도
- 8bit(1byte) 기반, 가변길이 인코딩
- 1byte ~ 4byte를 사용해서 문자를 인코딩
 - 1byte: ASCII, 영문, 기본 라틴 문자
 - 2byte: 그리스어, 히브리어 라틴 확장 문자
 - 3byte: 한글, 한자, 일본어
 - 4byte: 이모지, 고대문자등
- 단점: 상대적으로 사용이 복잡함

- UTF-16은 대부분의 기본 문자들이 2바이트로 표현되기 때문에, 문자열의 특정 문자에 접근하거나 문자 수를 세는 작업이 상대적으로 간단함. 반면, UTF-8에서는 각 문자가 가변 길이로 인코딩되므로 이런 작업이 더 복잡함
- 단점: ASCII를 제외한 일부 언어에서 더 많은 용량 사용
 - UTF-8은 ASCII 문자를 1바이트로, 비ASCII 문자를 2~4바이트로 인코딩한다.
 - 한글, 한자, 아랍어, 히브리어와 같은 문자들은 UTF-8에서 3바이트 또는 4바이트를 차지한다. 반면, UTF-16에서는 이들 문자가 대부분 2바이트로 인코딩된다.
- 장점: **ASCII 문자는 1바이트로 표현, ASCII 호환**
- 현대의 사실상 표준 인코딩 기술
 - 1990년도 후반 ~ 2000년도 초반에 인터넷과 웹이 빠르게 성장하면서 저변 확대
 - 2008년 W3C 웹 표준에 UTF-8 채택
 - 현재 대부분의 웹사이트와 애플리케이션에서 기본 인코딩으로 사용

정리

UTF-8이 현대의 사실상 표준 인코딩 기술이 된 이유

- **저장 공간 절약과 네트워크 효율성:** UTF-8은 ASCII 문자를 포함한 많은 서양 언어의 문자에 대해 1바이트를 사용한다. 반면에 UTF-16은 최소 2바이트를 사용하므로, 주로 ASCII 문자로 이루어진 영문 텍스트에서는 UTF-8이 2배 더 효율적이다. 특히 데이터를 네트워크로 전달할 때는 매우 큰 효율의 차이를 보인다. 참고로 웹에 있는 문서의 80% 이상은 영문 문서이다.
- **ASCII와의 호환성:** UTF-8은 ASCII와 호환된다. UTF-8로 인코딩된 텍스트에서 ASCII 범위에 있는 문자는 기존 ASCII와 동일한 방식으로 처리된다. 예를 들어 문자 "A"는 65로 인코딩 된다. 많은 레거시 시스템은 ASCII 기반으로 구축되어 있다. UTF-8은 이러한 시스템과의 호환성을 유지하면서도 전 세계의 모든 문자를 표현할 수 있다.

결론: UTF-8을 사용하자

참고: 한글 윈도우의 경우 기존 윈도우와 호환성 때문에 기본 인코딩을 MS949로 유지한다. 한글 윈도우도 기본 인코딩을 UTF-8로 변경하려고 노력중이다.

문자 집합 조회

문자 집합을 사용해서 문자 인코딩을 어떻게 하는지 코드로 알아보자. 먼저 사용 가능한 문자 집합을 조회해보자.

사용 가능한 문자 집합 조회

```
package charset;

import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.util.Set;
import java.util.SortedMap;

public class AvailableCharsetsMain {

    public static void main(String[] args) {
        // 이용 가능한 모든 Charset 자바 + OS
        SortedMap<String, Charset> charsets = Charset.availableCharsets();
        for (String charsetName : charsets.keySet()) {
            System.out.println("charsetName = " + charsetName);
        }

        System.out.println("====");
        // 문자로 조회(대소문자 구분X), MS949, ms949, x-windows-949
        Charset charset1 = Charset.forName("MS949");
        System.out.println("charset1 = " + charset1);

        // 별칭 조회
        Set<String> aliases = charset1.aliases();
        for (String alias : aliases) {
            System.out.println("alias = " + alias);
        }

        // UTF-8 문자로 조회
        Charset charset2 = Charset.forName("UTF-8");
        System.out.println("charset2 = " + charset2);

        // UTF-8 상수로 조회
        Charset charset3 = StandardCharsets.UTF_8;
        System.out.println("charset3 = " + charset3);

        // 시스템의 기본 Charset 조회
        Charset defaultCharset = Charset.defaultCharset();
        System.out.println("defaultCharset = " + defaultCharset);
    }
}
```

실행 결과

```
charsetName = EUC-KR
charsetName = ISO-8859-1
charsetName = US-ASCII
charsetName = UTF-16
charsetName = UTF-16BE
charsetName = UTF-16LE
charsetName = UTF-8
charsetName = x-windows-949
...
=====
charset1 = x-windows-949
alias = ms949
alias = ms_949
alias = windows-949
alias = windows949
charset2 = UTF-8
charset3 = UTF-8
defaultCharset = UTF-8
```

이용가능한 모든 문자 집합 조회

`Charset.availableCharsets()` 를 사용하면 이용가능한 모든 문자 집합을 조회할 수 있다.

여기에는 자바가 기본으로 제공하는 문자 집합과 OS가 제공하는 문자 집합을 포함한다.

Charset.forName()

특정 문자 집합을 지정해서 찾을 때는 `Charset.forName(...)` 을 사용하면 된다. 인자로 문자 집합의 이름이나 별칭을 사용하면 되는데, 대소문자는 구분하지 않는다.

별칭은 `aliases()` 메서드를 사용하면 구할 수 있다.

예를 들어서 MS949는 MS949, ms949, ms_949, windows-949, windows949, x-windows-949 등으로 찾을 수 있다.

StandardCharsets.UTF_8

자주 사용하는 문자 집합은 `StandardCharsets` 에 상수로 지정되어 있다.

```
public final class StandardCharsets {
    public static final Charset US_ASCII = sun.nio.cs.US_ASCII.INSTANCE;
    public static final Charset ISO_8859_1 = sun.nio.cs.ISO_8859_1.INSTANCE;
```

```

public static final Charset UTF_8 = sun.nio.cs.UTF_8.INSTANCE;
public static final Charset UTF_16BE = new sun.nio.cs.UTF_16BE();
public static final Charset UTF_16LE = new sun.nio.cs.UTF_16LE();
public static final Charset UTF_16 = new sun.nio.cs.UTF_16();
}

```

Charset.defaultCharset()

현재 시스템에서 사용하는 기본 문자 집합을 반환한다.

문자 인코딩 예제1

Charset (문자 집합)을 사용해서 실제 문자 인코딩을 해보자.

```

package charset;

import java.nio.charset.Charset;
import java.util.Arrays;

import static java.nio.charset.StandardCharsets.*;

public class EncodingMain1 {

    private static final Charset EUC_KR = Charset.forName("EUC-KR");
    private static final Charset MS_949 = Charset.forName("MS949");

    public static void main(String[] args) {
        System.out.println("== ASCII 영문 처리 ==");
        encoding("A", US_ASCII);
        encoding("A", ISO_8859_1);
        encoding("A", EUC_KR);
        encoding("A", MS_949);
        encoding("A", UTF_8);
        encoding("A", UTF_16BE);

        System.out.println("== 한글 지원 ==");
        encoding("가", EUC_KR);
    }
}

```

```

        encoding("가", MS_949);
        encoding("가", UTF_8);
        encoding("가", UTF_16BE);
    }

    private static void encoding(String text, Charset charset) {
        byte[] bytes = text.getBytes(charset);
        System.out.printf("%s -> [%s] 인코딩 -> %s %sbyte\n", text, charset,
Arrays.toString(bytes), bytes.length);
    }
}

```

- 문자를 컴퓨터가 이해할 수 있는 숫자(byte)로 변경하는 것을 문자 인코딩이라 한다.
- `String.getBytes(Charset)` 메서드를 사용하면 `String` 문자를 byte 배열로 변경할 수 있다.
- 이때 중요한 점이 있는데, 문자를 byte로 변경하려면 문자 집합이 필요하다는 점이다. 따라서 어떤 문자 집합을 참고해서 byte로 변경할지 정해야 한다. `String.getBytes()`의 인자로 `Charset` 객체를 전달하면 된다.
 - 문자 집합을 지정하지 않으면 현재 시스템에서 사용하는 기본 문자 집합을 인코딩에 사용한다.

실행 결과

```

== ASCII 영문 처리 ==
A -> [US-ASCII] 인코딩 -> [65] 1byte
A -> [ISO-8859-1] 인코딩 -> [65] 1byte
A -> [EUC-KR] 인코딩 -> [65] 1byte
A -> [x-windows-949] 인코딩 -> [65] 1byte
A -> [UTF-8] 인코딩 -> [65] 1byte
A -> [UTF-16BE] 인코딩 -> [0, 65] 2byte

== 한글 지원 ==
가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte
가 -> [x-windows-949] 인코딩 -> [-80, -95] 2byte
가 -> [UTF-8] 인코딩 -> [-22, -80, -128] 3byte
가 -> [UTF-16BE] 인코딩 -> [-84, 0] 2byte

```

영문

- US-ASCII, ISO-8859-1, EUC-KR, MS949, UTF-8은 모두 ASCII와 호환된다.
 - 영문 A는 1byte만 사용하고, 숫자 65로 인코딩된다.
- UTF-16은 ASCII와 호환되지 않는다.
 - 영문 A는 2byte를 사용하고, 숫자 [0, 65]로 인코딩된다.

한글

- EUC-KR, MS949는 한글 인코딩에 2byte를 사용하고 같은 값으로 인코딩한다.
 - EUC-KR을 확장해서 만든 것이 MS949이다.
- UTF-8은 한글 인코딩에 3byte를 사용한다.
- UTF-16은 한글 인코딩에 2byte를 사용한다.

참고: UTF_16, UTF_16BE, UTF_16LE가 있는데, 우리는 UTF_16BE를 사용하면 된다. BE, LE는 byte의 순서의 차이이다.

- UTF_16BE: [-84, 0]
- UTF_16LE: [0, -84]
- UTF_16: 인코딩한 문자가 BE, LE중에 어떤 것인지 알려주는 2byte가 앞에 추가로 붙는다.
- 이제 UTF_16을 잘 사용하지 않고, UTF-8은 이런 이슈가 없으므로 참고만 하고 넘어가자.

참고: byte 출력에 마이너스 숫자가 보이는 이유

이 내용은 크게 중요한 내용은 아니므로 이런 것이 있구나 하고 대략 듣고 넘어가면 충분하다!

'가'를 EUC-KR로 인코딩하는 예시

가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte

- **byte의 기본 개념**
 - 1byte는 8개의 bit로 구성된다. (예: 00000000)
 - 1byte로 256가지 경우를 표현할 수 있다.
- **한글 '가'의 EUC-KR 인코딩**
 - '가'는 EUC-KR에서 2byte로 표현된다.
 - 첫 번째 byte: 10110000 (10진수로 176)
 - 두 번째 byte: 10100001 (10진수로 161)
 - 따라서 '가'는 10진수로 표현하면 [176, 161]로 표현되어야 한다.
- **자바에서의 byte 표현**
 - 자바의 byte 타입은 양수와 음수를 모두 표현할 수 있다.
 - 자바의 byte는 첫 번째 비트(bit)가 0이면 양수, 1이면 음수로 간주된다.
 - ◆ 0 ~ 127: 첫 비트가 0이면 양수로 간주되며, 나머지 7bit로 0부터 127까지의 128가지 숫자를 표현한다.
 - ◆ -128 ~ -1: 첫 비트가 1이면 음수로 간주되며, 나머지 7bit로 -128부터 -1까지의 128가지 숫자를 표현한다.

- 결국 자바의 byte는 256가지 값을 표현하지만, 표현 가능한 숫자의 범위는 -128 ~127 이다.
- 가의 EUC-KR 인코딩을 자바 byte로 표현
 - 첫 번째 byte (10110000) (10진수로 176):
 - ◆ 첫 bit가 1이므로 음수로 해석
 - ◆ 자바에서는 -80으로 표현됨
 - ◆ 참고로 자바에서 음수를 표현할 때는 2의 보수라는 계산 공식이 사용된다.
 - 두 번째 byte (10100001) (10진수로 161):
 - ◆ 첫 bit가 1이므로 음수로 해석
 - ◆ 자바에서는 -95로 표현됨
- 정리
 - 자바의 byte를 사용해도 실제 메모리에 저장되는 값은 동일하다 (10110000, 10100001).
 - 자바의 byte 타입이 첫 비트로 음수를 표현하기 때문에, 화면에 보여지는 10진수 숫자만 다를 뿐이다.
 - 실제 개발 단계에서 이런 문제를 가지고 고민할 일은 거의 없다. 참고만 하고 넘어가자
 - ◆ 중간에 byte의 값을 더하거나 빼면서 변경한다면 이런 부분들을 잘 알아야겠지만, 실무에서 문자 인코딩된 byte의 값을 더하거나 빼는 계산은 거의 하지 않는다.

문자 인코딩 예제2

문자 인코딩, 디코딩

이번에는 문자를 인코딩은 물론이고, 디코딩까지 해보자. 그리고 좀 더 다양한 예시를 알아보자.

```
package charset;

import java.nio.charset.Charset;
import java.util.Arrays;

import static java.nio.charset.StandardCharsets.*;

public class EncodingMain2 {
    private static final Charset EUC_KR = Charset.forName("EUC-KR");
    private static final Charset MS_949 = Charset.forName("MS949");

    public static void main(String[] args) {
        System.out.println("== 영문 ASCII 인코딩 ==");
        test("A", US_ASCII, US_ASCII);
        test("A", US_ASCII, ISO_8859_1); // ASCII 확장(LATIN-1)
```

```

test("A", US_ASCII, EUC_KR); // ASCII 포함
test("A", US_ASCII, MS_949); // ASCII 포함
test("A", US_ASCII, UTF_8); // ASCII 포함
test("A", US_ASCII, UTF_16BE); // UTF_16 디코딩 실패

```

```

System.out.println("== 한글 인코딩 - 기본 ==");
test("가", US_ASCII, US_ASCII); // X
test("가", ISO_8859_1, ISO_8859_1); // X
test("가", EUC_KR, EUC_KR);
test("가", MS_949, MS_949);
test("가", UTF_8, UTF_8);
test("가", UTF_16BE, UTF_16BE);

```

```

System.out.println("== 한글 인코딩 - 복잡한 문자 ==");
test("뽕", EUC_KR, EUC_KR); // X
test("뽕", MS_949, MS_949);
test("뽕", UTF_8, UTF_8);
test("뽕", UTF_16BE, UTF_16BE);

```

```

System.out.println("== 한글 인코딩 - 디코딩이 다른 경우 ==");
test("가", EUC_KR, MS_949);
test("뽕", MS_949, EUC_KR); // 인코딩 가능, 디코딩 X
test("가", EUC_KR, UTF_8); // X
test("가", MS_949, UTF_8); // X
test("가", UTF_8, MS_949); // X

```

```

System.out.println("== 영문 인코딩 - 디코딩이 다른 경우 ==");
test("A", EUC_KR, UTF_8);
test("A", MS_949, UTF_8);
test("A", UTF_8, MS_949);
test("A", UTF_8, UTF_16BE); // X

```

```

}

```

```

private static void test(String text, Charset encodingCharset, Charset
decodingCharset) {
    byte[] encoded = text.getBytes(encodingCharset);
    String decoded = new String(encoded, decodingCharset);
    System.out.printf("%s -> [%s] 인코딩 -> %s %sbyte -> [%s] 디코딩 -> %s\n",
        text, encodingCharset, Arrays.toString(encoded),
encoded.length,
        decodingCharset, decoded);
    }
}

```

실행 결과

```
== 영문 ASCII 인코딩 ==
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [US-ASCII] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [ISO-8859-1] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [EUC-KR] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [x-windows-949] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [UTF-8] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [UTF-16BE] 디코딩 -> 💎

== 한글 인코딩 - 기본 ==
가 -> [US-ASCII] 인코딩 -> [63] 1byte -> [US-ASCII] 디코딩 -> ?
가 -> [ISO-8859-1] 인코딩 -> [63] 1byte -> [ISO-8859-1] 디코딩 -> ?
가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte -> [EUC-KR] 디코딩 -> 가
가 -> [x-windows-949] 인코딩 -> [-80, -95] 2byte -> [x-windows-949] 디코딩 -> 가
가 -> [UTF-8] 인코딩 -> [-22, -80, -128] 3byte -> [UTF-8] 디코딩 -> 가
가 -> [UTF-16BE] 인코딩 -> [-84, 0] 2byte -> [UTF-16BE] 디코딩 -> 가

== 한글 인코딩 - 복잡한 문자 ==
뵐 -> [EUC-KR] 인코딩 -> [63] 1byte -> [EUC-KR] 디코딩 -> ?
뵐 -> [x-windows-949] 인코딩 -> [-108, -18] 2byte -> [x-windows-949] 디코딩 -> 뵐
뵐 -> [UTF-8] 인코딩 -> [-21, -73, -127] 3byte -> [UTF-8] 디코딩 -> 뵐
뵐 -> [UTF-16BE] 인코딩 -> [-67, -63] 2byte -> [UTF-16BE] 디코딩 -> 뵐

== 한글 인코딩 - 디코딩이 다른 경우 ==
가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte -> [x-windows-949] 디코딩 -> 가
뵐 -> [x-windows-949] 인코딩 -> [-108, -18] 2byte -> [EUC-KR] 디코딩 -> 💎💎
가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte -> [UTF-8] 디코딩 -> 💎💎
가 -> [x-windows-949] 인코딩 -> [-80, -95] 2byte -> [UTF-8] 디코딩 -> 💎💎
가 -> [UTF-8] 인코딩 -> [-22, -80, -128] 3byte -> [x-windows-949] 디코딩 -> 媛💎

== 영문 인코딩 - 디코딩이 다른 경우 ==
A -> [EUC-KR] 인코딩 -> [65] 1byte -> [UTF-8] 디코딩 -> A
A -> [x-windows-949] 인코딩 -> [65] 1byte -> [UTF-8] 디코딩 -> A
A -> [UTF-8] 인코딩 -> [65] 1byte -> [x-windows-949] 디코딩 -> A
A -> [UTF-8] 인코딩 -> [65] 1byte -> [UTF-16BE] 디코딩 -> 💎
```

영문 ASCII 인코딩

```
== 영문 ASCII 인코딩 ==
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [US-ASCII] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [ISO-8859-1] 디코딩 -> A
```



```
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [EUC-KR] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [x-windows-949] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [UTF-8] 디코딩 -> A
A -> [US-ASCII] 인코딩 -> [65] 1byte -> [UTF-16BE] 디코딩 -> ⚡
```

- 영문 'A'를 인코딩하면 1byte를 사용하고 숫자 65가 된다.
- 숫자 65를 디코딩하면 UTF-16을 제외하고 모두 디코딩이 가능하다.
 - UTF-16의 경우 디코딩에 실패해서 ⚡라는 특수문자가 출력되었다.
- ASCII는 UTF-16을 제외한 대부분의 문자 집합에 호환된다.

한글 인코딩 - 기본

== 한글 인코딩 - 기본 ==

```
가 -> [US-ASCII] 인코딩 -> [63] 1byte -> [US-ASCII] 디코딩 -> ?
가 -> [ISO-8859-1] 인코딩 -> [63] 1byte -> [ISO-8859-1] 디코딩 -> ?
가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte -> [EUC-KR] 디코딩 -> 가
가 -> [x-windows-949] 인코딩 -> [-80, -95] 2byte -> [x-windows-949] 디코딩 -> 가
가 -> [UTF-8] 인코딩 -> [-22, -80, -128] 3byte -> [UTF-8] 디코딩 -> 가
가 -> [UTF-16BE] 인코딩 -> [-84, 0] 2byte -> [UTF-16BE] 디코딩 -> 가
```

- 한글 '가'는 ASCII, ISO-8859-1로 인코딩 할 수 없다.
 - 이 경우 흥미롭게도 숫자 63이 되는데 63은 ASCII로 ?라는 뜻이다. 한마디로 모르는 이상한 문자가 인코딩 되었다는 뜻이다.
- EUC-KR, MS949, UTF-8, UTF-16은 한글 인코딩 디코딩이 잘 수행되는 것을 확인할 수 있다.
 - 2byte: EUC-KR, MS949, UTF-16
 - 3byte: UTF-8
- 한글 '가'는 EUC-KR, MS949 모두 같은 값을 반환한다 따라서 서로 호환된다.

한글 인코딩 - 복잡한 문자

== 한글 인코딩 - 복잡한 문자 ==

```
뵘 -> [EUC-KR] 인코딩 -> [63] 1byte -> [EUC-KR] 디코딩 -> ?
뵘 -> [x-windows-949] 인코딩 -> [-108, -18] 2byte -> [x-windows-949] 디코딩 -> 뵘
뵘 -> [UTF-8] 인코딩 -> [-21, -73, -127] 3byte -> [UTF-8] 디코딩 -> 뵘
뵘 -> [UTF-16BE] 인코딩 -> [-67, -63] 2byte -> [UTF-16BE] 디코딩 -> 뵘
```

- EUC-KR은 자주 사용하는 한글 2350개만 표현할 수 있다. 따라서 '뵘'과 문자는 문자 집합에 없으므로 인코딩할 수 없다.

- MS949, UTF-8, UTF-16은 모든 한글을 표현할 수 있다. 따라서 '뽕'과 같은 문자도 인코딩 할 수 있다.

한글 인코딩 - 디코딩이 다른 경우

== 한글 인코딩 - 디코딩이 다른 경우 ==

가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte -> [x-windows-949] 디코딩 -> 가
 뽕 -> [x-windows-949] 인코딩 -> [-108, -18] 2byte -> [EUC-KR] 디코딩 -> 뽕
 가 -> [EUC-KR] 인코딩 -> [-80, -95] 2byte -> [UTF-8] 디코딩 -> 가
 가 -> [x-windows-949] 인코딩 -> [-80, -95] 2byte -> [UTF-8] 디코딩 -> 가
 가 -> [UTF-8] 인코딩 -> [-22, -80, -128] 3byte -> [x-windows-949] 디코딩 -> 가

- '가'와 같이 자주 사용하는 한글은 EUC-KR, MS949 서로 호환된다. 따라서 EUC-KR로 인코딩해도 MS949로 디코딩 할 수 있다. MS949는 EUC-KR을 포함한다.
- '뽕'과 같이 특수한 한글은 MS949로 인코딩 할 수 있지만, EUC-KR의 문자 집합에 없으므로 EUC-KR로 디코딩 할 수 없다.
- 한글을 인코딩할 때 UTF-8과 EUC-KR(MS949)는 서로 호환되지 않는다.

영문 인코딩 - 디코딩이 다른 경우

== 영문 인코딩 - 디코딩이 다른 경우 ==

A -> [EUC-KR] 인코딩 -> [65] 1byte -> [UTF-8] 디코딩 -> A
 A -> [x-windows-949] 인코딩 -> [65] 1byte -> [UTF-8] 디코딩 -> A
 A -> [UTF-8] 인코딩 -> [65] 1byte -> [x-windows-949] 디코딩 -> A
 A -> [UTF-8] 인코딩 -> [65] 1byte -> [UTF-16BE] 디코딩 -> A

- ASCII에 포함되는 영문은 UTF-16을 제외한 대부분의 문자 집합에서 호환된다.

정리

- ASCII 영문 인코딩: UTF-16을 제외하고 모두 호환
- 사실상 표준인 UTF-8을 사용하자.

한글이 깨지는 가장 큰 2가지 이유

- EUC-KR(MS949), UTF-8이 서로 호환되지 않음
 - 한글이 깨지는 대부분의 문제는 UTF-8로 인코딩한 한글을 EUC-KR(MS949)로 디코딩하거나 또는 EUC-KR(MS949)로 인코딩한 한글을 UTF-8로 디코딩할 때 발생한다.
- EUC-KR(MS949) 또는 UTF-8로 인코딩한 한글을 ISO-8859-1로 디코딩 할 때
 - EUC-KR(MS949) 또는 UTF-8로 인코딩한 한글을 한글을 지원하지 않는 ISO-8859-1로 디코딩 할 때

발생한다.

정리