

# 1

# 자바 시작

# 컴퓨터와 소프트웨어





- 컴퓨터
  - 메인프레임, PC, 태블릿, 스마트폰, 원칩 컴퓨터
- 소프트웨어



# 프로그래밍 언어



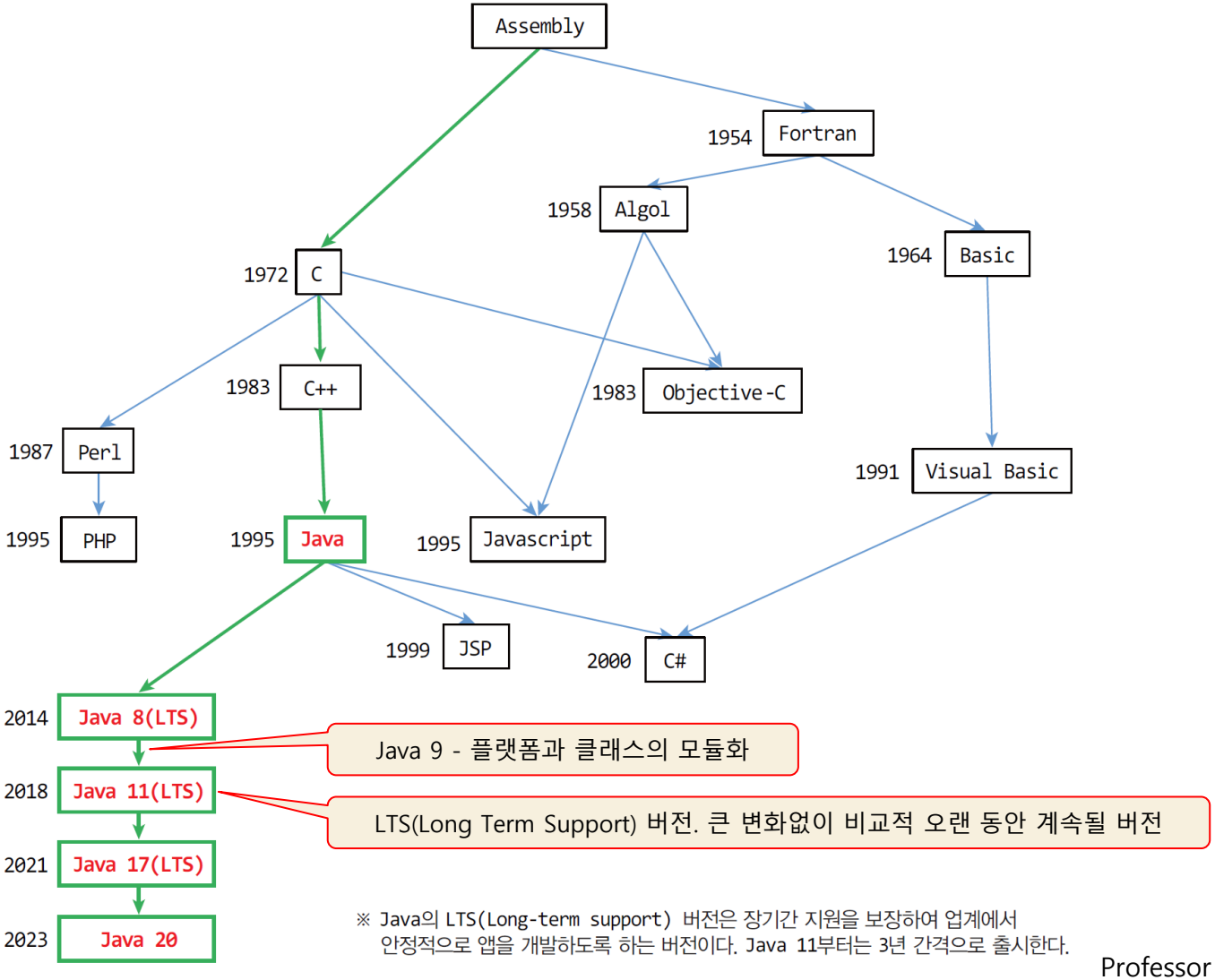


- 프로그래밍 언어
  - 프로그램 작성 언어
  - 기계어(machine language)
    - 0, 1의 이진수로 구성된 언어
    - 컴퓨터의 CPU는 기계어만 이해하고 처리가능
  - 어셈블리어
    - 기계어 명령을 ADD, SUB, MOVE 등과 같은 표현하기 쉬운 상징적인 단어인 니모닉 기호(mnemonic symbol)로 일대일 대응시킨 언어
  - 고급언어
    - 사람이 이해하기 쉽고, 복잡한 작업, 자료 구조,알고리즘을 표현하기 위해 고안된 언어
    - Pascal, Basic, C/C++, Java, C#
    - 절차 지향 언어와 객체 지향 언어

# 프로그래밍 언어의 진화







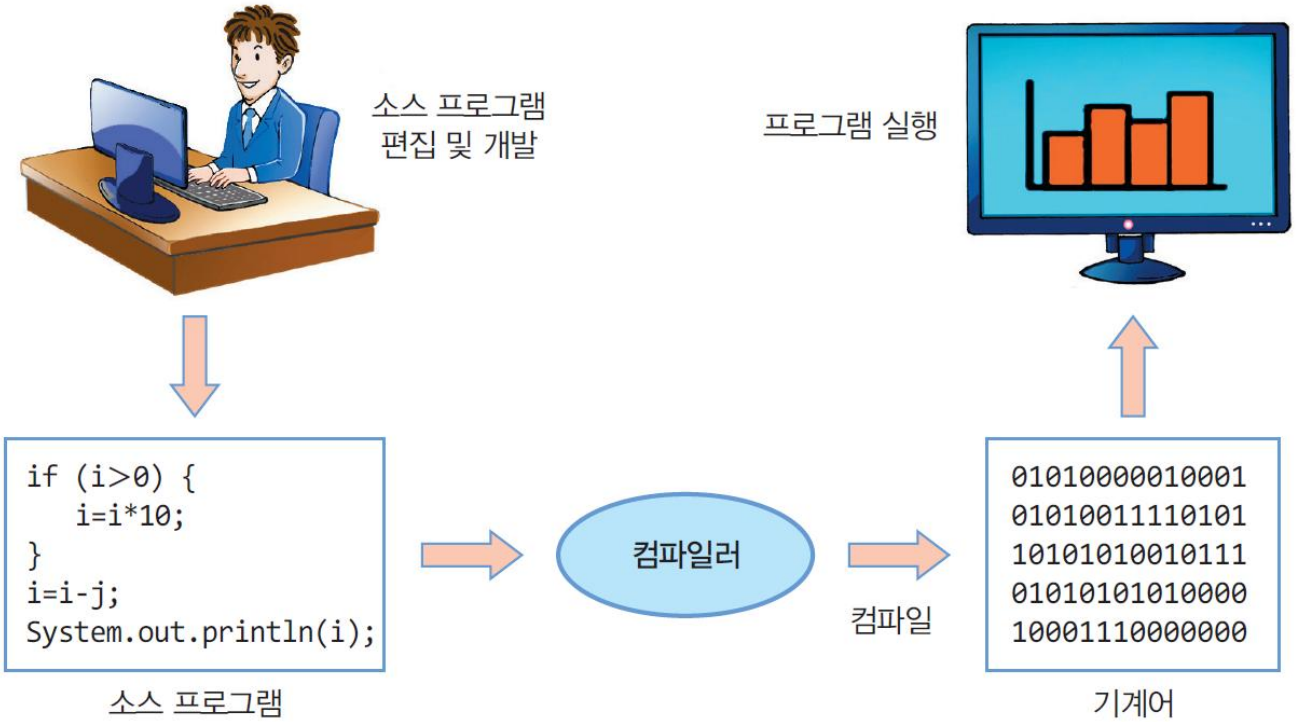

※ Java의 LTS(Long-term support) 버전은 장기간 지원을 보장하여 업계에서 안정적으로 앱을 개발하도록 하는 버전이다. Java 11부터는 3년 간격으로 출시한다.

# 컴파일





- 소스 : 프로그래밍 언어로 작성된 텍스트 파일
- 컴파일 : 소스 파일을 컴퓨터가 이해할 수 있는 기계어로 만드는 과정
  - 소스 파일 확장자와 컴파일 된 파일의 확장자
    - 자바 : **.java** -> **.class**
    - C : **.c** -> **.obj**-> **.exe**
    - C++ : **.cpp** -> **.obj** -> **.exe**



# 자바의 태동





- 1991년 그린 프로젝트(Green Project)
  - 선마이크로시스템즈의 제임스 고슬링(James Gosling)에 의해 시작
    - 가전 제품에 들어갈 소프트웨어를 위해 개발
  - 1995년에 자바 발표
- 목적
  - 플랫폼 호환성 문제 해결
    - 기존 언어로 작성된 프로그램은 PC, 유닉스, 메인 프레임 등 플랫폼 간에 호환성 없음
    - 소스를 다시 컴파일하거나 프로그램을 재 작성해야 하는 단점
  - 플랫폼 독립적인 언어 개발
    - 모든 플랫폼에서 호환성을 갖는 프로그래밍 언어 필요
    - 네트워크, 특히 웹에 최적화된 프로그래밍 언어의 필요성 대두
  - 메모리 사용량이 적고 다양한 플랫폼을 가지는 가전 제품에 적용
    - 가전 제품 : 작은 량의 메모리를 가지는 제어 장치
    - 내장형 시스템 요구 충족
- 초기 이름 : 오크(OAK)
  - 인터넷과 웹의 엄청난 발전에 힘입어 퍼지게 됨
  - 웹 브라우저 Netscape에서 실행
- 2009년에 선마이크로시스템즈를 오라클에서 인수

# WORA





- WORA(Write Once Run Anywhere)
  - 한번 작성된 코드는 모든 플랫폼에서 바로 실행
  - C/C++ 등 기존 언어가 가진 플랫폼 종속성 극복
    - OS, H/W에 상관없이 자바 프로그램이 동일하게 실행
  - 네트워크에 연결된 어느 클라이언트에서나 실행
    - 웹 브라우저, 분산 환경 지원
- WORA를 가능하게 하는 자바의 특징
  - 바이트 코드(byte code)
    - 자바 소스를 컴파일한 목적 코드
    - CPU에 종속적이지 않은 중립적인 코드
    - JVM에 의해 해석되고 실행됨
  - JVM(Java Virtual Machine)
    - 자바 바이트 코드를 실행하는 자바 가상 기계(소프트웨어)

# 플랫폼 종속성(platform dependency)





플랫폼 = 하드웨어 플랫폼(CPU 등) + 운영체제 플랫폼

C/C++ 프로그램이 플랫폼 종속적인 이유
- 기계어가 CPU마다 다름
- 운영체제마다 API 다름
- 운영체제마다 실행파일 형식 다름



인텔 CPU를 가진 리눅스 환경에서 개발

C/C++ 프로그램 → 컴파일 → 기계어

실행

실행되지 않음

실행되지 않음



인텔 CPU + 리눅스



Apple 사의 MAC PC



인텔 CPU + 윈도우 노트북



임베디드 컴퓨터 + 리눅스

# 자바의 플랫폼 독립성, WORA







***Write Once !!***

자바 응용 프로그램

바이트 코드

자바 가상 기계가 설치된 모든 컴퓨터에서 실행됨

***Run Anywhere!!***

실행

실행

실행

실행



**자바 가상 기계**

인텔 CPU + 리눅스



**자바 가상 기계**

Apple 사의 MAC PC



**자바 가상 기계**

인텔 CPU + 윈도우 노트북



**자바 가상 기계**

임베디드 컴퓨터 + 리눅스

# 바이트 코드와 자바 가상 기계





- 바이트 코드
  - 자바 가상 기계에서 실행 가능한 바이너리 코드
    - 바이트 코드는 컴퓨터 CPU에 의해 직접 실행되지 않음
    - 자바 가상 기계가 작동 중인 플랫폼에서 실행
    - 자바 가상 기계가 인터프리터 방식으로 바이트 코드 해석
  - 클래스 파일(.class)에 저장
- 자바 가상 기계(JVM : Java Virtual Machine)
  - 동일한 자바 실행 환경 제공
    - 각기 다른 플랫폼에 설치
  - 자바 가상 기계 자체는 플랫폼에 종속적
    - 자바 가상 기계는 플랫폼마다 각각 작성됨
    - 예) 리눅스에서 작동하는 자바 가상 기계는 윈도우에서 작동하지 않음
  - 자바 가상 기계 개발 및 공급
    - 자바 개발사인 오라클, IBM 등
- 자바 응용프로그램 실행
  - 자바 가상 기계가 응용프로그램을 구성하는 클래스 파일(.class)의 바이트 코드 실행

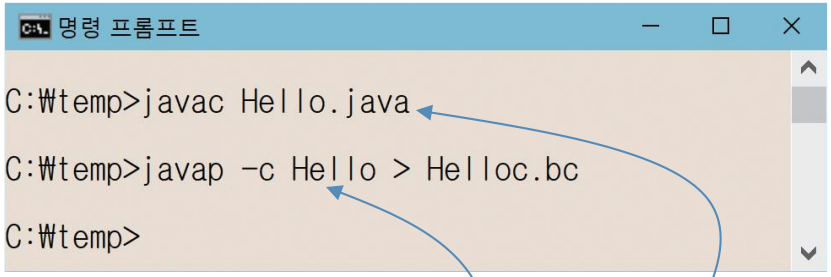
# 바이트 코드의 디어셈블(disassemble)





- 디어셈블
  - 클래스 파일에 들어 있는 바이트 코드를 텍스트로 볼 수 있게 변환하는 작업
  - JDK의 javap.exe 이용

```
public class Hello {
   public static int sum(int i, int j) {
      return i + j;// i와 j의 합을 리턴
   }
   public static void main(String[] args) {
      int i;
      int j;
      char a;
      String b;
      final int TEN = 10;
      i = 1;
      j = sum(i, TEN);
      a = '?';
      b = "Hello";
      java.lang.System.out.println(a);
      System.out.println(b);
      System.out.println(TEN);
      System.out.println(j);
   }
}
```




- Hello.java를 컴파일하는 명령
- 컴파일되면 Hello.class 생성

- Hello.class 파일을 디어셈블하는 명령
- 디어셈블된 결과 Hello.bc 파일 생성

# 디어셈블하여 바이트 코드 보기

Helloc.bc - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static int sum(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: iconst_1
       1: istore_1
       2: iload_1
       3: bipush        10
       5: invokestatic  #2                  // Method sum:(II)I
       8: istore_2
       9: bipush        63
      11: istore_3
      12: ldc           #3                  // String Hello
      14: astore        4
      16: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      19: iload_3
      20: invokevirtual #5                  // Method java/io/PrintStream.println:(C)V
      23: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      26: aload         4
      28: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      31: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      34: bipush        10
      36: invokevirtual #7                  // Method java/io/PrintStream.println:(I)V
      39: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      42: iload_2
      43: invokevirtual #7                  // Method java/io/PrintStream.println:(I)V
      46: return
}
```

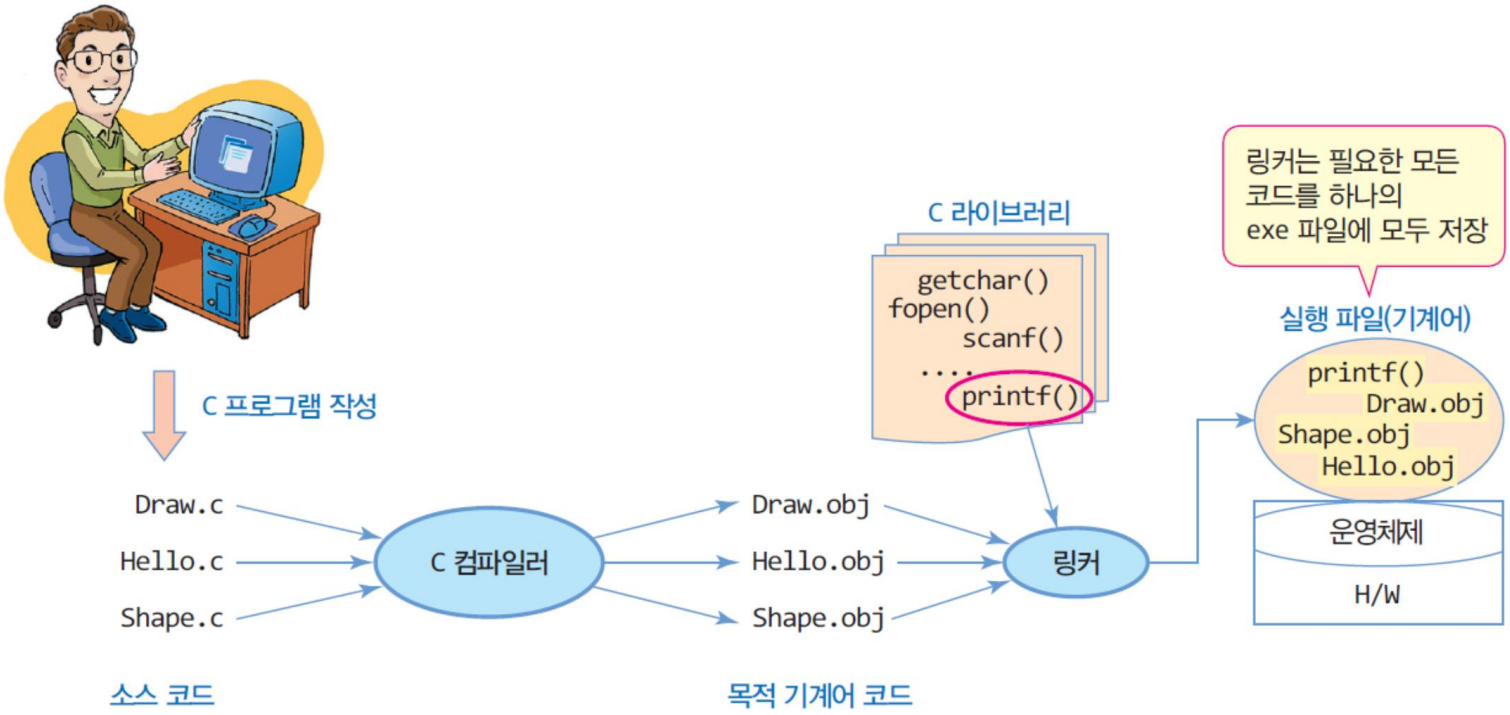
sum() 메소드를 컴파일한 바이트 코드를 디어셈블한 결과(자바의 어셈블리 코드로 출력)

# C/C++ 프로그램의 개발 및 실행 환경





- C/C++ 프로그램의 개발
  - 여러 소스(.c) 파일로 나누어 개발
  - 링크 과정을 통해, 실행에 필요한 모든 코드(컴파일된 코드)를 하나의 실행 파일(.exe)로 생성
- 실행
  - 실행 파일(exe)이 큰 경우, 메모리가 적은 임베디드 컴퓨터(가전)의 경우 낭패

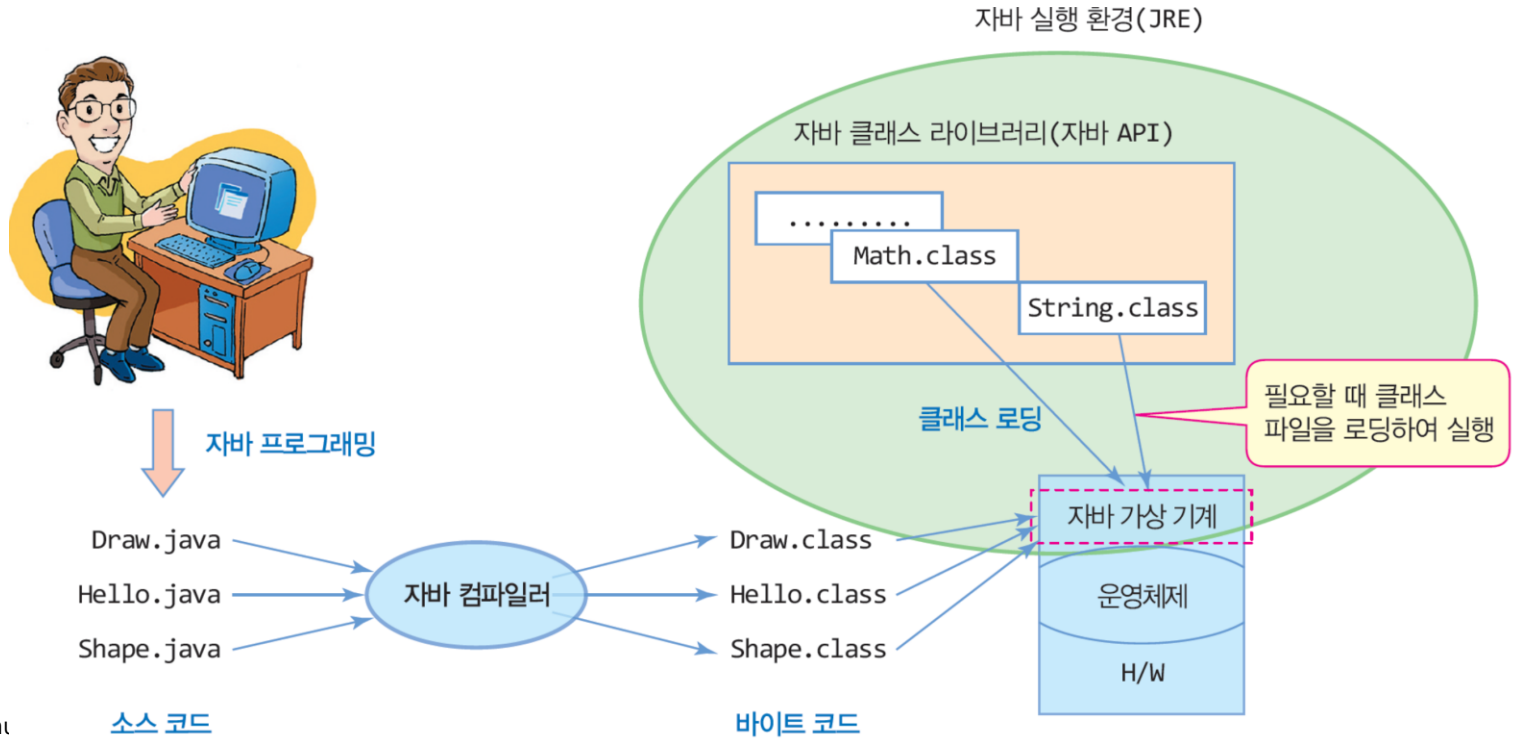


# 자바의 개발 및 실행 환경





- 자바 프로그램의 개발
  - 여러 소스(.java)로 나누어 개발
  - 바이트 코드(.class)를 하나의 실행 파일(exe)로 만드는 링크 과정 없음
- 실행
  - main() 메소드를 가진 클래스에서 부터 실행 시작
  - 자바 가상 기계는 필요할 때, 클래스 파일 로딩, 적은 메모리로 실행 가능





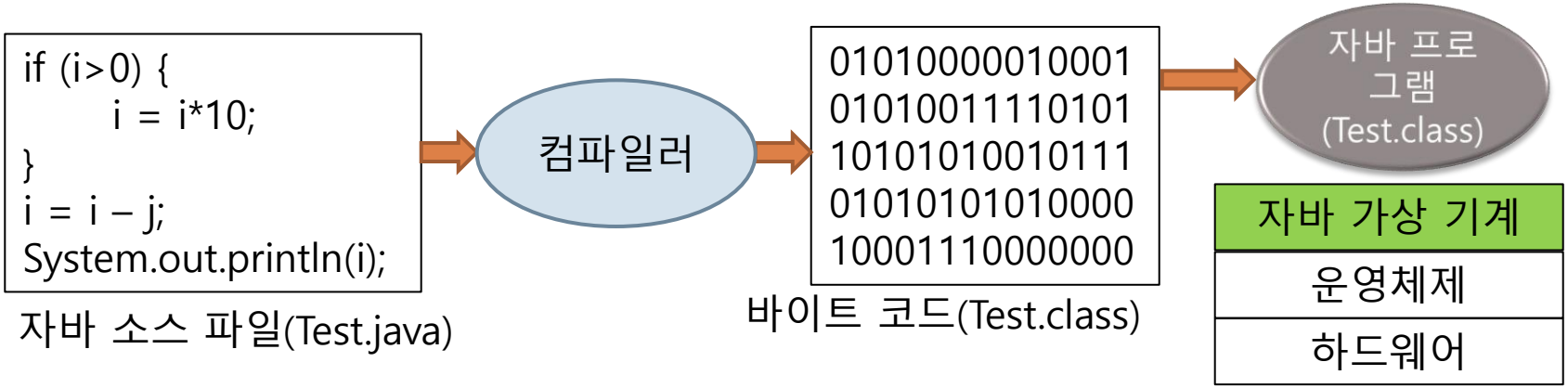

* 자바는 하나의 실행 파일(exe)로 만드는 링크 과정 없음

# 자바와 C/C++의 실행 환경 차이

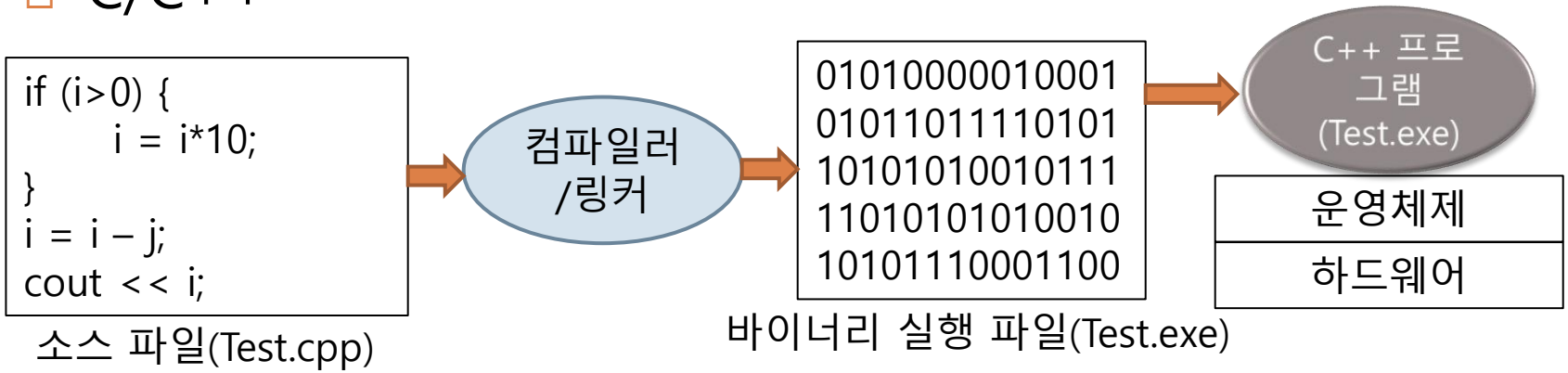




## 자바




## C/C++



# Tip: 자바와 C/C++실행 환경 및 과정





- 자바
  - 컴파일러가 바로 바이트 코드한 후 링크 과정 없음
  - 바이트 코드는 JVM에서만 실행 가능
  - 자바는 필요한 클래스들을 프로그램 실행 중에 동적으로 로딩
    - 동적 로딩은 JVM에 포함된 클래스 로더에 의해 이루어짐
    - ClassLoader 클래스를 이용하여 개발자가 직접 클래스 로딩가능
- C/C++
  - 컴파일
    - C/C++에서는 컴파일러가 중간 단계인 목적 코드를 생성
  - 링크
    - 링커가 목적 코드와 라이브러리 연결, 실행 가능한 최종 실행 파일 생성
    - 정적 라이브러리는 실행 파일에 포함
      - 실행 파일 크기가 커짐
    - 동적 라이브러리의 경우는 실행 중에 동적 링크
  - 목적 코드 및 실행 파일은 플랫폼에 따라 다름
    - 플랫폼이 바뀌거나 다른 플랫폼에서 실행시키려면 다시 컴파일 및 링크

# JDK와 JRE



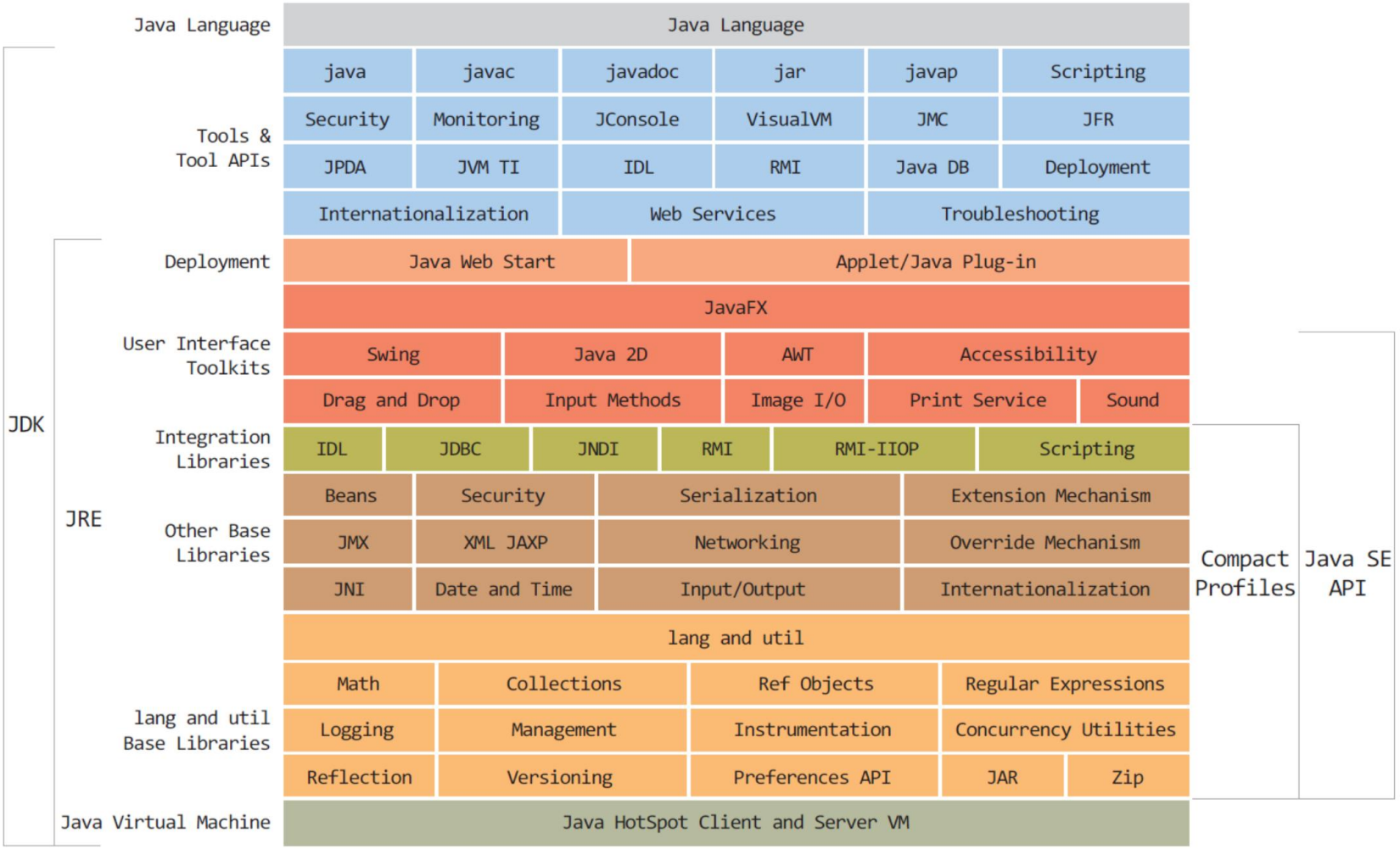


- JDK(Java Development Kit)
  - 자바 응용 개발 환경. 개발에 필요한 도구 포함
    - 컴파일러, 컴파일된 자바 API 클래스들이 들어 있는 모듈 파일들, 샘플 등 포함
- JRE(Java Runtime Environment)
  - 자바 실행 환경. JVM 포함
  - 컴파일된 자바 API 들이 들어 있는 모듈 파일
  - 개발자가 아닌 경우 JRE만 따로 다운 가능
- JDK와 JRE의 개발 및 배포
  - 오라클의 Technology Network의 자바 사이트에서 다운로드
    - https://www.oracle.com/java/technologies/
- JDK의 bin 디렉터리에 포함된 주요 개발 도구
  - javac - 자바 소스를 바이트 코드로 변환하는 컴파일러
  - java - 자바 응용프로그램 실행기. 자바 가상 기계를 작동시켜 자바프로그램 실행
  - javadoc – 자바 소스로부터 HTML 형식의 API 도큐먼트 생성
  - jar – 자바 클래스들(패키지포함)을 압축한 자바 아카이브 파일(.jar) 생성 관리
  - jmod: 자바의 모듈 파일(.jmod)을 만들거나 모듈 파일의 내용 출력
  - jlink: 응용프로그램에 맞춘 맞춤형(custom) JRE 제공
  - jdb - 자바 응용프로그램의 실행 중 오류를 찾는 데 사용하는 디버거
  - javap - 클래스 파일의 바이트 코드를 소스와 함께 보여주는 디어셈블러

# Java SE 구성





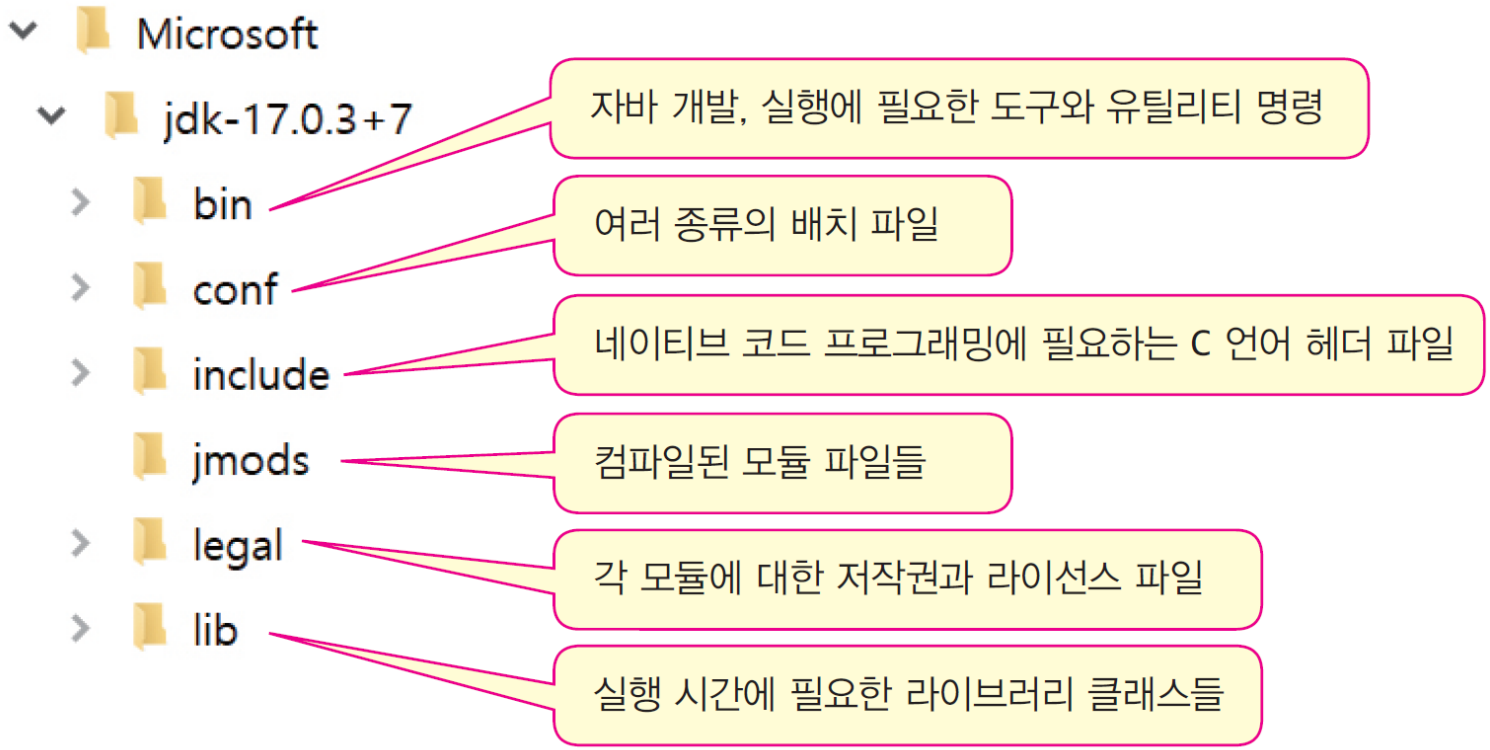



Java SE의 구성(출처: 오라클)

# JDK 설치 후 디렉터리 구조








Microsoft OpenJDK를 설치한 경우

# 자바의 배포판 종류





- 오라클은 개발 환경에 따라 다양한 자바 배포판 제공
- Java SE
  - 자바 표준 배포판(Standard Edition)
  - 데스크탑과 서버 응용 개발 플랫폼
- Java ME
  - 자바 마이크로 배포판
    - 휴대 전화나 PDA, 셋톱박스 등 제한된 리소스를 갖는 하드웨어에서 응용 개발을 위한 플랫폼
    - 가장 작은 메모리 풋프린트
  - Java SE의 서브셋 + 임베디드 및 가전 제품을 위한 API 정의
- Java EE
  - 자바 기업용 배포판
    - 자바를 이용한 다중 사용자, 기업용 응용 개발을 위한 플랫폼
  - Java SE + 인터넷 기반의 서버사이드 컴퓨팅 관련 API 추가

# 나는 누구?








(사진 출처 : 위키 백과)

# 자바 API





- 자바 API(Application Programming Interface)란?
  - JDK에 포함된 클래스 라이브러리
    - 주요한 기능들을 미리 구현한 클래스 라이브러리의 집합
  - 개발자는 API를 이용하여 쉽고 빠르게 자바 프로그램 개발
    - API에서 정의한 규격에 따라 클래스 사용
- 자바 패키지(package)
  - 서로 관련된 클래스들을 분류하여 묶어 놓은 것
  - 계층구조로 되어 있음
    - 클래스의 이름에 패키지 이름도 포함
    - 다른 패키지에 동일한 이름의 클래스 존재 가능
  - 자바 API(클래스 라이브러리)는 JDK내에 패키지 형태로 제공됨
    - 필요한 클래스가 속한 패키지만 import하여 사용
  - 개발자 자신의 패키지 생성 가능

# Java 9부터 시작된 모듈 프로그래밍





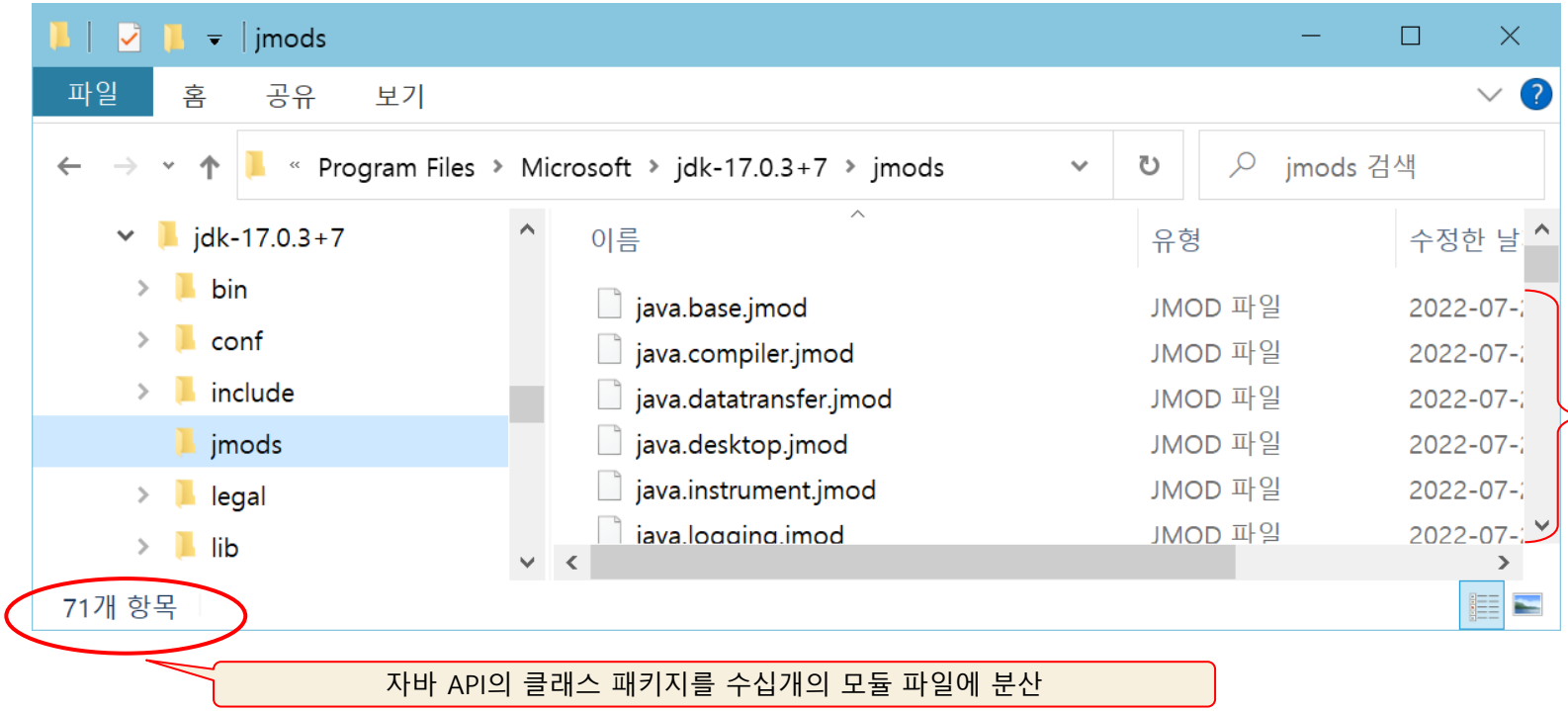
- 모듈화(modularity)
  - Java 9에서 정의된 새로운 기능, 2017년 9월 21일 출시
  - 모듈
    - 자바 패키지들과 이미지, XML 파일 등의 자원들을 묶은 단위
  - 모듈 프로그래밍
    - 자바 응용프로그램을 마치 직소 퍼즐(jigsaw)을 연결하듯이 필요한 모듈을 연결하는 방식으로 작성
- 자바 플랫폼의 모듈화
  - 실행 시간에 사용되는 자바 API의 모든 클래스들을 모듈들로 분할
  - 모듈화의 목적
    - 세밀한 모듈화, 자바 응용프로그램이 실행되는데 필요없는 모듈 배제
    - 작은 크기의 실행 환경 구성
    - 하드웨어가 열악한 소형 IoT 장치 지원
- 모듈 방식이 아닌, 기존 방식으로 자바 프로그래밍 해도 무관
  - 자바 플랫폼이 모듈 방식으로 바뀌었지만,
  - 굳이 응용프로그램을 모듈 방식으로 작성할 필요 없음
    - 모듈 설계자들도 이런 사실 강조

# 자바에서 제공하는 전체 모듈 리스트(Java SE)





- Java 9부터 플랫폼을 모듈화함
  - Java SE의 모든 클래스들을 모듈들로 재구성
  - JDK가 설치된 디렉터리 밑의 jmods 디렉터리에 있음



# 자바 온라인 API 문서





java,.base 모듈의
java.lang 패키지

String (Java SE 17 & JDK 17)

docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String.html

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP Java SE 17 & JDK 17

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

**Module** java.base
**Package** java.lang

## Class String

java.lang.Object
    java.lang.String

**All Implemented Interfaces:**

Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

# 자바 통합 개발 환경–이클립스(Eclipse)





- IDE(Integrated Development Environment )란?
  - 통합 개발 환경
  - 편집, 컴파일, 디버깅을 한번에 할 수 있는 통합된 개발 환경
- 이클립스(Eclipse)
  - 자바 응용 프로그램 개발을 위한 통합 개발 환경
  - IBM에 의해 개발된 오픈 소스 프로젝트
  - https://www.eclipse.org/downloads/ 에서 다운로드

# Tip: javadoc를 이용한 API 도큐먼트 생성



- javadoc.exe
  - 자바 소스 파일로부터 API 도큐먼트 생성
  - 소스의 선언문과 /** 와 */ 사이에 주어진 정보를 바탕으로 HTML로 된 API 도큐먼트 생성.
  - 클래스, 인터페이스 생성자, 메소드, 필드 등을 기술
- 실행 방법 사례
  - javadoc HelloDoc.java
  - HelloDoc.html 파일 생성
    - HelloDoc 클래스를 설명하는 API 도큐먼트

```
/**
   javadoc 사용 예제를 위한 클래스
 */
public class HelloDoc {
   /**
      두 정수의 합을 구하는 메소드

      @param i 합을 구할 첫번째 정수형 인자
      @param j 합을 구할 두번째 정수형 인자
      @return 두 정수의 합을 리턴
    */

   public static int sum(int i, int j) {
      return i + j;
   }

   public static void main(String[] args) {
      int i;
      int j;
      char a;
      String b;
      final int TEN = 10;

      i = 1;
      j = sum(i, TEN);
      a = '?';
      b = "Hello";

      java.lang.System.out.println(a);
      System.out.println(b);
      System.out.println(TEN);
      System.out.println(j);
   }
}
```

# javadoc로 HelloDoc 클래스의 API 도큐먼트생성





명령 프롬프트

```
C:₩temp>javadoc HelloDoc.java
Loading source file HelloDoc.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_131
Building tree for all the packages and classes...
Generating .₩HelloDoc.html...
Generating .₩package-frame.html...
Generating .₩package-summary.html...
Generating .₩package-tree.html...
Generating .₩constant-values.html...
Building index for all the packages and classes...
Generating .₩overview-tree.html...
Generating .₩index-all.html...
Generating .₩deprecated-list.html...
Building index for all classes...
Generating .₩allclasses-frame.html...
Generating .₩allclasses-noframe.html...
Generating .₩index.html...
Generating .₩help-doc.html...

C:₩temp>
```

HelloDoc.html 파일 생성

C:₩temp₩HelloDoc.html — HelloDoc

## Class HelloDoc

java.lang.Object
    HelloDoc

public class HelloDoc
extends java.lang.Object

javadoc 사용 예제를 위한 클래스 ← 주목

### *Constructor Summary*

**Constructors**

| Constructor and Description |
|---|
| HelloDoc() |

### *Method Summary*

**All Methods** | **Static Methods** | **Concrete Methods**

| Modifier and Type | Method and Description |
|---|---|
| static void | main(java.lang.String[] args) |
| static int | sum(int i, int j)<br>두 정수의 합을 구하는 메소드 ← 주목 |

# 자바 프로그램 개발





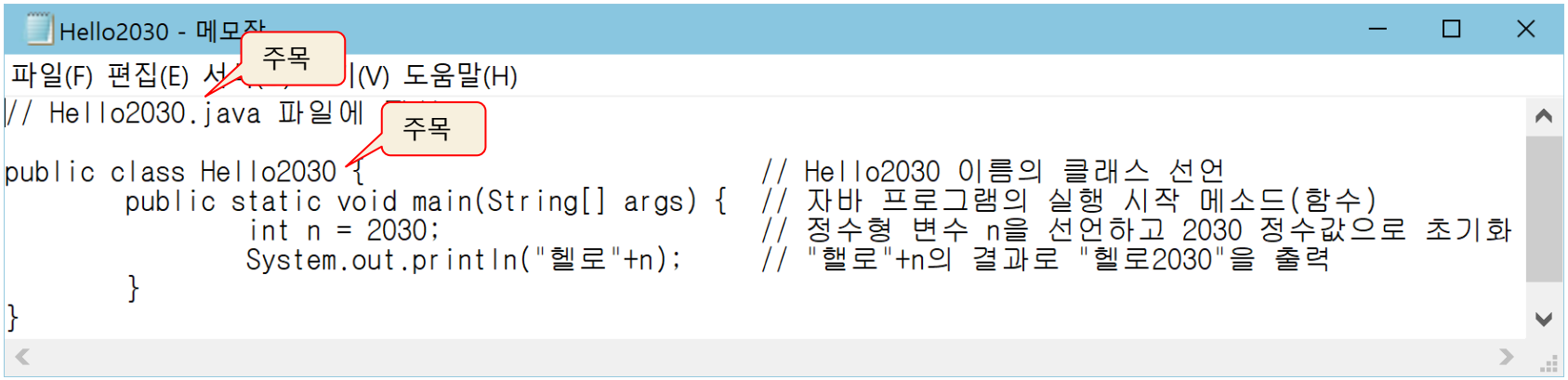
- public class Hello2030
  - 클래스 선언문
  - Hello2030 은 클래스 이름
  - 클래스는 {와 } 사이에 정의
  - 자바는 하나 이상의 클래스로 구성
- public static void main(String[] args)
  - 자바 프로그램은 main() 메소드에서 실행 시작
    - 실행을 시작하는 클래스에 main() 메소드가 반드시 하나 존재
- int n = 2030;
  - 지역 변수 선언
- System.out.println("헬로"+n);
  - 화면에 "헬로2030" 출력
  - System.out 객체는 JDK에서 제공됨

# 자바 소스 편집





- 어떤 편집기를 사용해도 무관
  - 메모장으로 작성한 샘플




- 작성 후 Hello2030.java로 저장
  - 반드시 클래스와 동일한 이름으로 파일 저장
    - C:₩Temp에 저장
  - 확장자 .java

# 자바 소스 컴파일 및 실행





Hello2030.java 는 C:₩Temp에 저장되어 있음

명령 프롬프트

C:₩Temp 디렉터리로 이동

```
C:₩>cd ₩Temp

C:₩temp>javac Hello2030.java

C:₩temp>dir Hello2030.*
 C 드라이브의 볼륨에는 이름이 없습니다.
 볼륨 일련 번호: 4443-1630

 C:₩temp 디렉터리

2017-06-23  오전 10:35               629 Hello2030.class
2017-06-23  오전 10:33               334 Hello2030.java
               2개 파일                 963 바이트
               0개 디렉터리  120,858,402,816 바이트 남음

C:₩temp>
```

컴파일

클래스 파일 생성

Hello2030.class 실행

명령 프롬프트

.class 확장자를 붙이지 않는다.

```
C:₩temp>java Hello2030
헬로2030

C:₩temp>
```

실행 결과

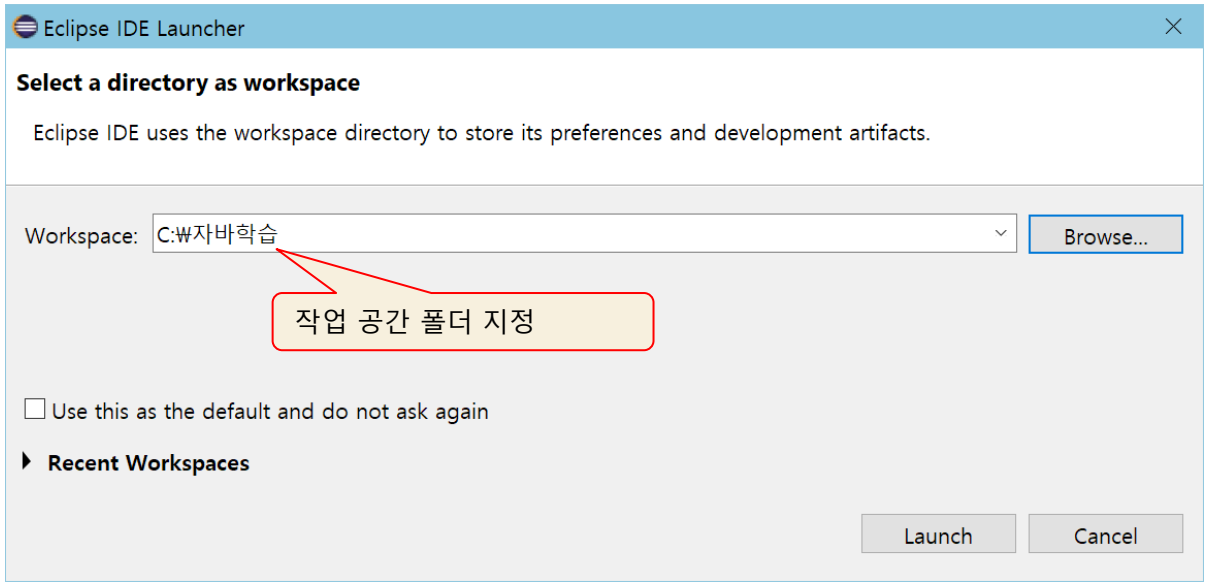
# 이클립스 실행







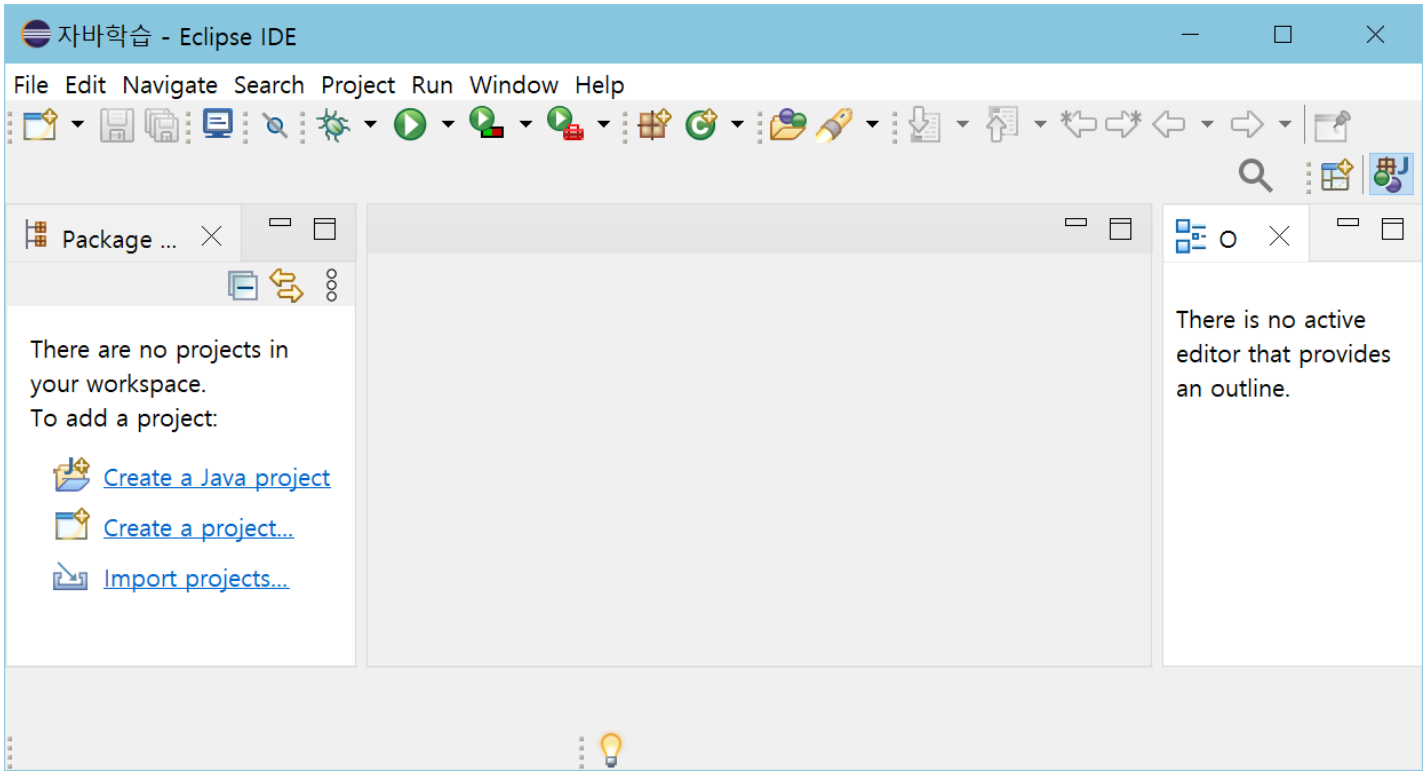

이클립스 시작 화면



# 이클립스의 사용자 인터페이스



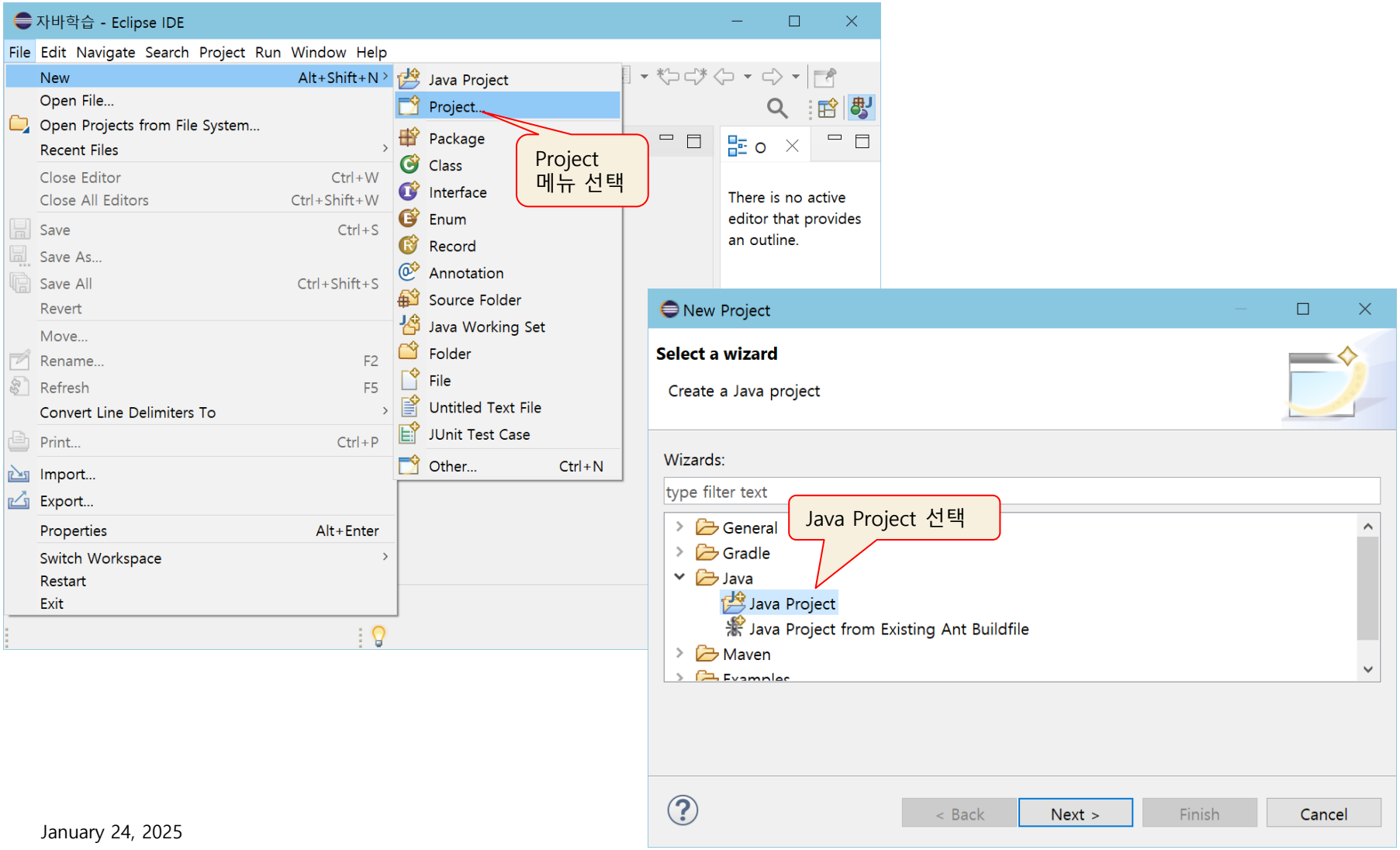




# 프로젝트 생성

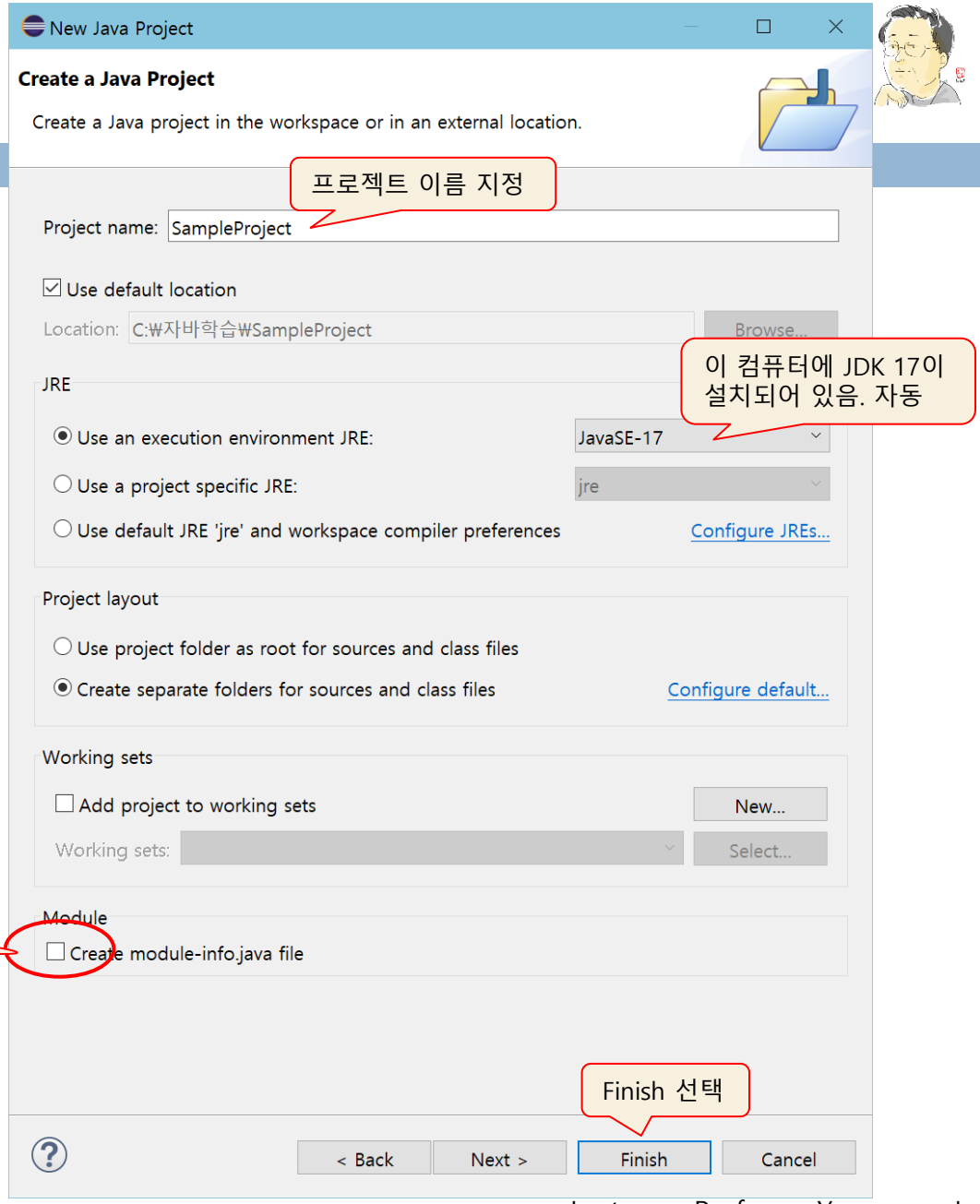





자바학습 - Eclipse IDE
File Edit Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Open Projects from File System...
Recent Files
Close Editor Ctrl+W
Close All Editors Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Import...
Export...
Properties Alt+Enter
Switch Workspace
Restart
Exit
Java Project
Project...
Package
Class
Interface
Enum
Record
Annotation
Source Folder
Java Working Set
Folder
File
Untitled Text File
JUnit Test Case
Other... Ctrl+N
Project 메뉴 선택
There is no active editor that provides an outline.
New Project
Select a wizard
Create a Java project
Wizards:
type filter text
General
Gradle
Java
Java Project
Java Project from Existing Ant Buildfile
Maven
Java Project 선택
< Back
Next >
Finish
Cancel

# 프로젝트 생성

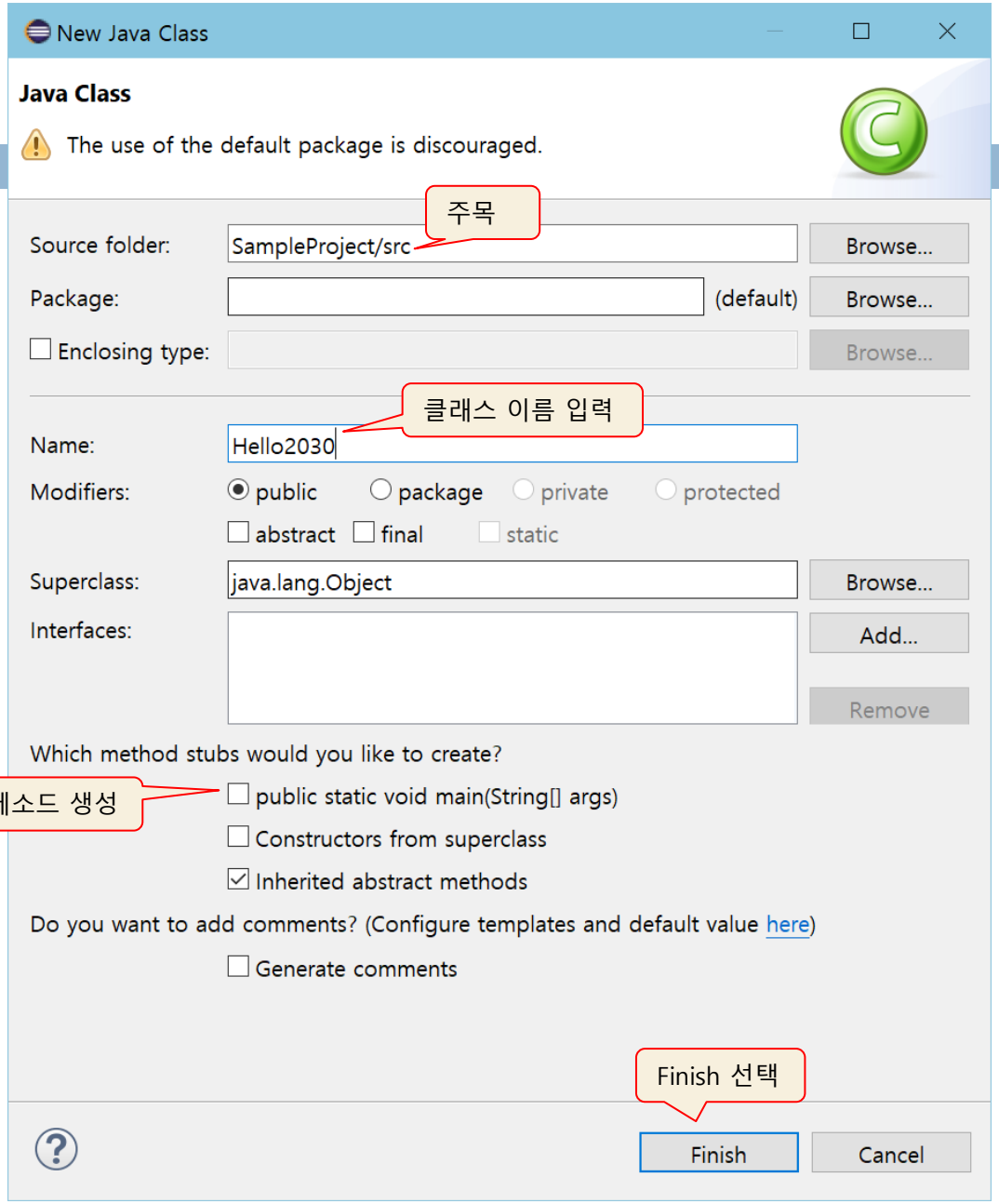



New Java Project
Create a Java Project
Create a Java project in the workspace or in an external location.
프로젝트 이름 지정
Project name: SampleProject
Use default location
Location: C:₩자바학습₩SampleProject
Browse...
이 컴퓨터에 JDK 17이 설치되어 있음. 자동
JRE
Use an execution environment JRE: JavaSE-17
Use a project specific JRE: jre
Use default JRE 'jre' and workspace compiler preferences
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets
New...
Working sets:
Select...
Module
Create module-info.java file
모듈 프로그래밍 안하기 위해 체크 해제 상태 그대로
Finish 선택
< Back
Next >
Finish
Cancel

# 클래스 생성



File->New->Class 메뉴 선택

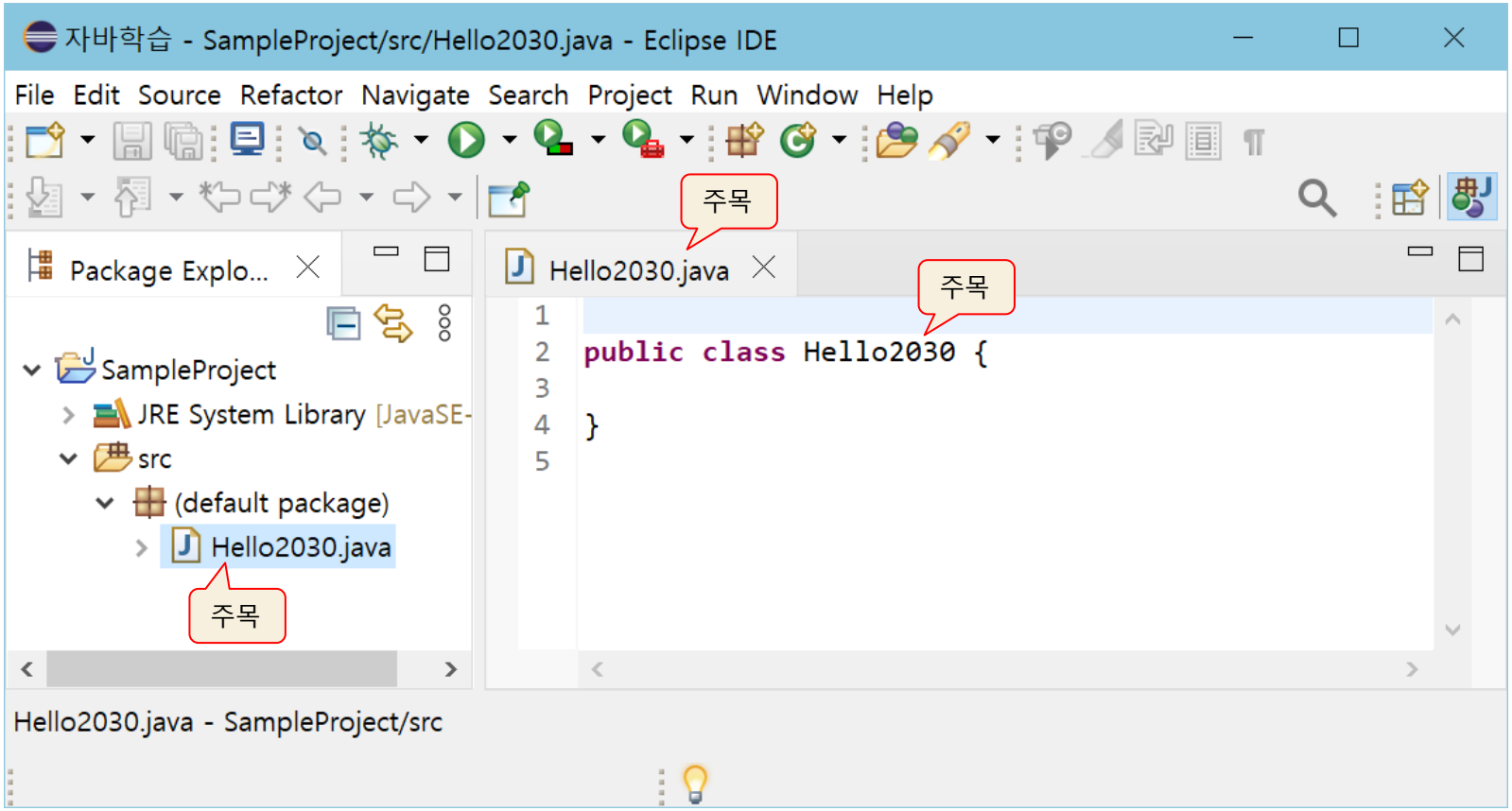



main()을 체크하면 자동으로 main() 메소드 생성

# 생성된 자바 소스



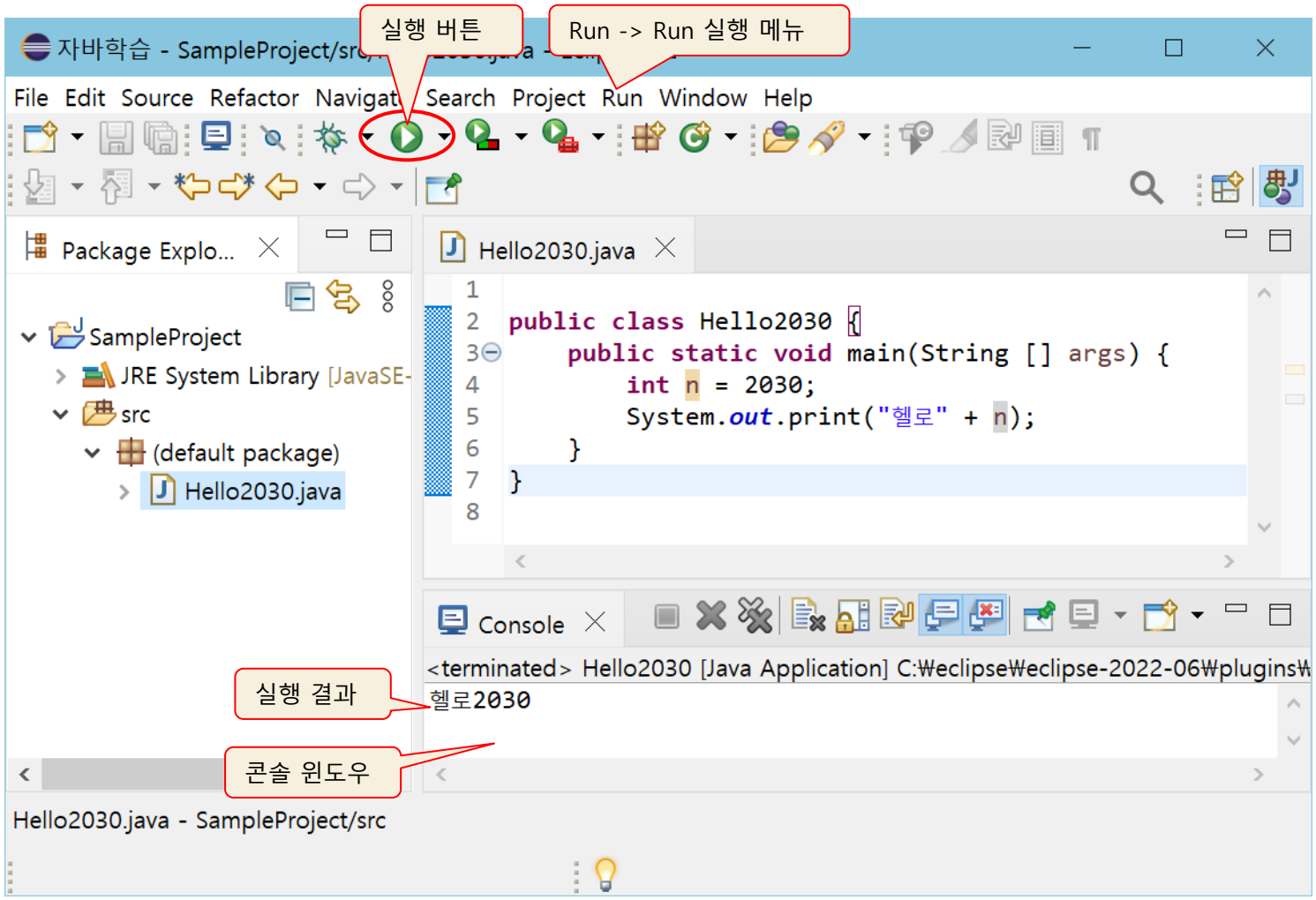



자바학습 - SampleProject/src/Hello2030.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
주목
Package Explo...
Hello2030.java
주목
SampleProject
JRE System Library [JavaSE-
src
(default package)
Hello2030.java
주목
1
2 public class Hello2030 {
3
4 }
5
Hello2030.java - SampleProject/src

# 소스 편집과 컴파일 및 실행





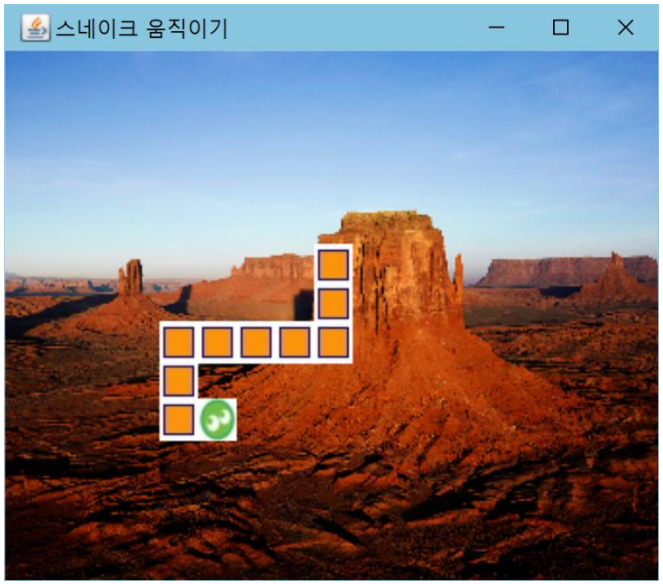


# 자바 응용의 종류 : 데스크톱 응용프로그램





- 가장 전형적인 자바 응용프로그램
  - PC 등의 데스크톱 컴퓨터에 설치되어 실행
  - 자바 실행 환경(JRE)이 설치된 어떤 컴퓨터에서도 실행
    - 다른 응용프로그램의 도움 필요 없이 단독으로 실행

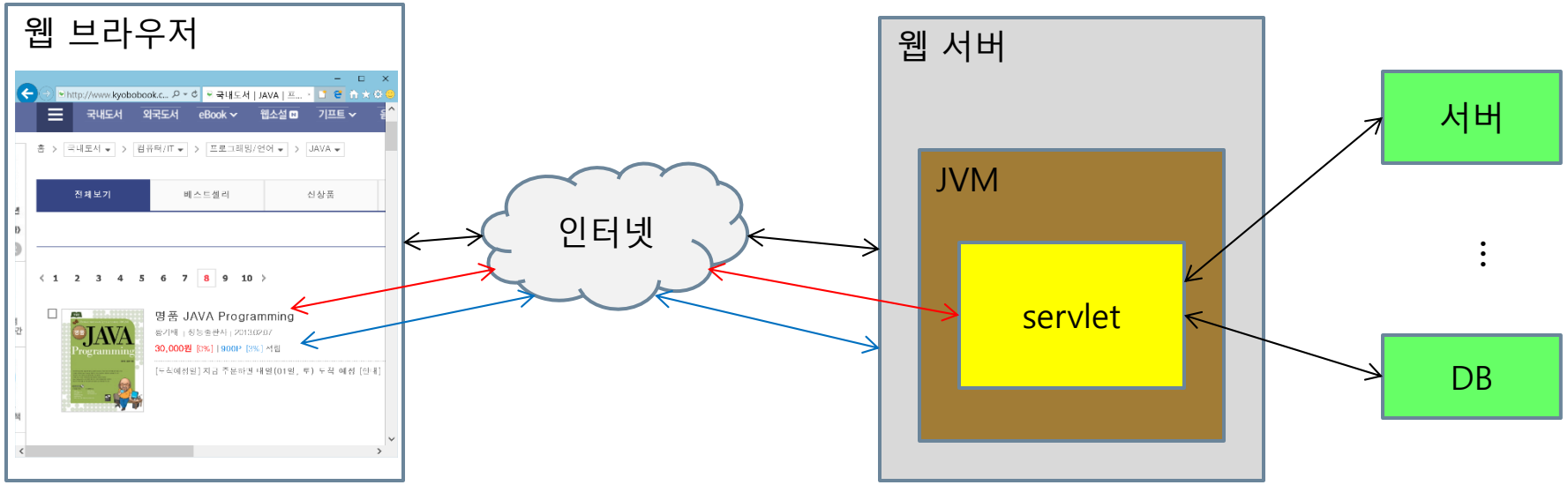


# 자바 응용의 종류 : 서블릿 응용프로그램





- 서블릿(servlet)
  - 웹 서버에서 실행되는 자바 프로그램
    - 서블릿은 웹브라우저에서 실행되는 자바스크립트 코드와 통신
  - 데이터베이스 서버 및 기타 서버와 연동하는 복잡한 기능 구현 시 사용
  - 사용자 인터페이스가 필요 없는 응용
  - 웹 서버에 의해 실행 통제 받음

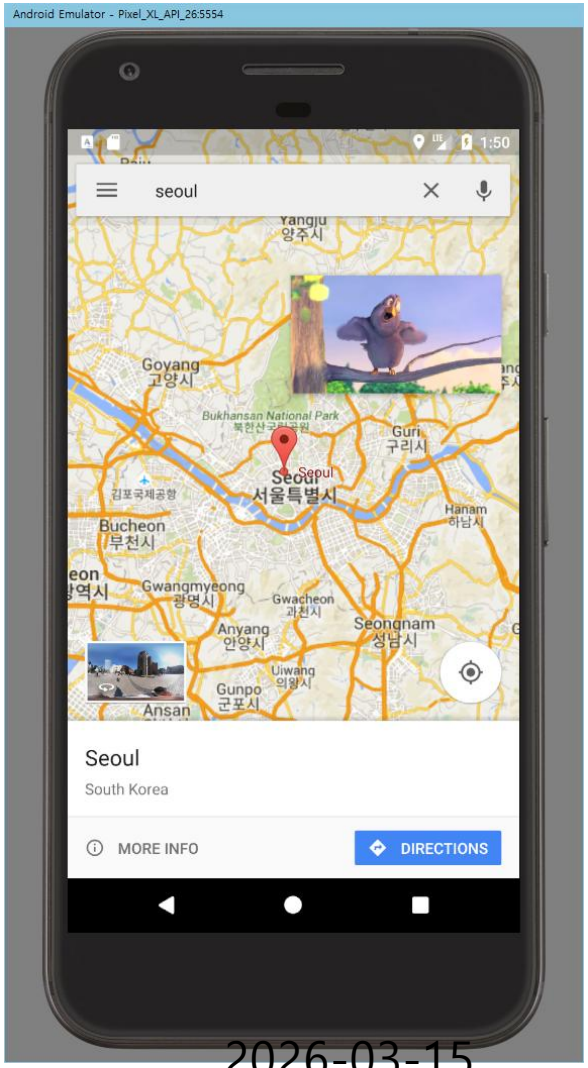


# 자바 모바일 응용 : 안드로이드 앱





- 안드로이드
  - 구글의 주도로 여러 모바일 회사가 모여 구성한 OHA(Open Handset Alliance)에서 만든 무료 모바일 플랫폼
  - 개발 언어는 자바를 사용하나 JVM에 해당하는 Dalvik은 기존 바이트 코드와 호환성이 없어 변환 필요



# 자바의 특성(1)



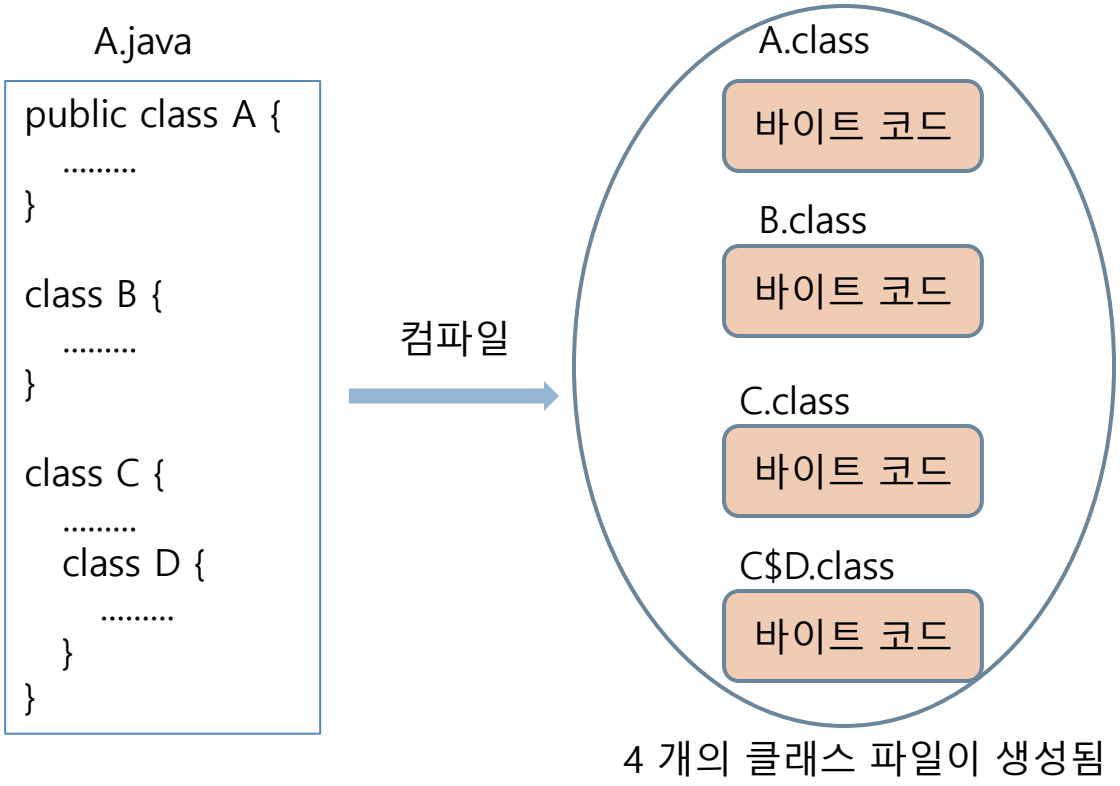


- 플랫폼 독립성
  - 자바 프로그램은 플랫폼에 상관없이 어디서든지 실행
- 객체지향
  - 상속성, 다형성, 캡슐화
- 클래스로 캡슐화
  - 클래스 내에 모든 변수(필드), 함수(메소드) 구현해야 함
  - 클래스 안에서 새로운 클래스(내부 클래스) 작성 가능
- 소스(.java)와 클래스(.class) 파일
  - 하나의 소스 파일에 여러 클래스 작성 가능
    - public 클래스는 하나만 가능
    - 소스 파일의 이름과 public으로 선언된 클래스 이름은 같아야 함
  - 컴파일된 클래스 파일(.class)에는 클래스는 하나만 존재
    - 다수의 클래스를 가진 자바 소스(.java)를 컴파일하면 클래스마다 별도 클래스 파일(.class) 생성

# 소스 파일과 클래스, 클래스 파일의 관계








4 개의 클래스 파일이 생성됨

# 자바의 특징(2)





- 실행 코드 배포
  - 실행 코드 : 한 개의 class 파일 또는 다수의 class 파일로 구성
  - 여러 폴더에 걸쳐 다수의 클래스 파일로 구성된 경우
    - jar 파일 형태로 배포 가능
  - main() 메소드
    - 자바 응용프로그램의 실행은 main() 메소드에서 시작
  - 하나의 클래스 파일에 하나 이상의 main() 메소드가 있을 수 없음
  - 각 클래스 파일이 main() 메소드를 포함하는 것은 상관없음
- 패키지
  - 관련된 여러 클래스를 묶어 관리하는 단위
  - 패키지는 폴더 개념
    - 예) java.lang.System은 java₩lang 디렉터리의 System.class 파일
- 멀티스레드
  - 자바는 운영체제의 도움 없이 자체적으로 멀티스레드 지원
    - 반면, C/C++ 등에서는 멀티스레드 운영체제 API를 호출

# 자바의 특징(3)





- garbage 컬렉션
  - 자바는 응용 프로그램에서 메모리 반환 기능 없음, 메모리 할당 기능(new)만 있음
    - 개발자의 부담 대폭 감소
  - garbage : 할당 후 사용되지 않는 메모리
  - 자바 가상 기계가 자동으로 가비지 회수
- 실시간 응용 시스템에 부적합
  - 자바 응용프로그램은 실행 도중 예측할 수 없는 시점에 가비지 컬렉션 실행
  - 일정 시간(deadline) 내에 반드시 실행 결과를 내야만 하는 실시간 시스템에는 부적합
- 자바 프로그램은 안전
  - 타입 체크가 매우 엄격
  - 포인터의 개념 없음
- 프로그램 작성이 쉬움
  - 포인터 개념이 없어 부담 적음
  - 다양하고 강력한 라이브러리가 많음
- 실행 속도를 개선하기 위해 JIT 컴파일러 사용
  - 자바의 느린 실행 요인 : 인터프리터 방식으로 바이트 코드 실행
  - JIT(Just in Time) 컴파일링 기법으로 개선
    - 실행 도중 바이트 코드를 해당 CPU의 기계어 코드로 컴파일, 해당 CPU가 기계어를 실행