실행 환경

- 컴퓨터 아키텍쳐나 os에 따라서 컴파일된 코드가 달라서 높은 이식성을 유지 못함
- JVM위에서는 어떤 아키텍쳐나 os에서도 실행이 가능하다

어셈블리 언어

- c언어는 컴파일러에 의해 assembler 코드를 생성한다.
- assembler는 cpu 제조회사에 종속되기 때문에 이식성이 떨어짐
    - gcc
    - visual c/c++ compiler

바이트 코드

- 자바 언어는 바이트 코드로 컴파일 된다.
- 바이트 코드는 JVM에서만 구동가능하다.
- 하드웨어나 os에 종속되지 않고 JVM에 종속된 바이트코드로 생성되기때문에 이식성이 높다

c언어 메모리 관리

- num1은 스택에 할당되고
- num2는 동적으로 메모리를 할당하면서 메모리의 힙 영역을 사용하게 된다.
- malloc으로 동적으로 할당한 메모리를 힙에서 반드시 해재해야 한다. (메모리에 계속 상주하면서 자원을 잡아먹음)

```
int num1 = 20;
int *numPtr2;
numPtr2 = malloc(sizeof(int)); //원하는 시점에 원하는 만큼 메모리를 동적할당
printf("%p\n", numPtr1);
printf("%p\n", numPtr2);
free(numPtr2);
```

Java 메모리 관리

- 프로그래머는 힙을 사용할 수 있는 만큼 자유롭게 사용하면 된다
- 메모리 관리를 JVM의 GC가 담당한다.
- 참조가 끊긴 메모리 영역을 GC가 알아서 수거해간다.

call by reference vs call by value

- call by value
  - 함수로 매개변수를 넘길때 값을 할당하여 사용
  - 매개변수로 전달한 값을 복사하여 함수 인자로 전달
  - 콜 했던 함수에서 데이터를 변경하여도 콜한 함수영역에서는 변하지 않음
- call by reference
  - 함수로 매개변수를 넘길때 참조를 넘기는 방식
  - 메모리 주소값 자체를 매개변수로 넘겨서 콜 했던 함수에서 조작한 데이터는 콜한 함수영역에서도 영향을 받는다.

call by reference vs call by value

java는 call by value이지만 객체의 클래스 변수를 함수에서 수정하면 호출한쪽의 객체의 값은 변한다. 하지만 기본은 call by value인것이지 call by reference는 아니다.

```
public static void changeName(User user){
    user = new User("2");
}

public static void main(String[] args){
    User human = new User("1");
    changeName(human);
    System.out.println(human); // 1
}
```

객체지향 vs 절차지향 ???

- Procedural Oriented Programming에서 Procedura은 절차가 아니라 프로시저(함수)
- 절차지향과 객체지향의 차이는 데이터 관리 방법
    - 객체지향에서 객체는 자신만의 데이터와 프로시져를 갖고 각 객체들간에 서로 연결되어 다른 객체의 기능 사용한다. (행위 중심으로 데이터및 데이터 관련 프로시저를 Object로 묶음)
    - 절차지향은 프로시저를 중심으로 구성