변수

데이터의 저장과 참조를 위해 할당된 메모리 공간으로 자료형 선언을 통해 생성

변수 하나당 한 개의 데이터를 표현

컴파일러

자료형에 따라 메모리 공간을 할당

변수의 선언

자료형 변수 이름; ex) int value;

자료형 변수 이름=초기값; ex) int value = 1000;

|  |  |  |  |
| --- | --- | --- | --- |
| 자료형 | 데이터 | 메모리 크기 | 표현 가능 범위 |
| Boolen | 참과 거짓 | 1 바이트 | True, false |
| Char | 문자 | 2 바이트 | 모든 유니코드 문자 |
| Byte | 정수 | 1 바이트 | -128 ~ 127 |
| Short | 2 바이트 | -32768 ~ 32767 |
| Int | 4 바이트 | -2147483648 ~ 2147483647 |
| Long | 8 바이트 | -9223372036854775808 ~ 9223372036854775807 |
| Float | 실수 | 4 바이트 | +-(1.40X10^(-45) ~ 3.40X10^38) |
| Double | 8 바이트 | +-(4.94X10^(-324) ~ 1.79X10^308 |

기본자료형

정수형

기본으로 int 사용

부동 소수점형

기본으로 double 사용

Float 보다 double이 더 정확함

숫자 뒤에 f나 F가 붙으면 float으로 인식/ d나 D가 붙으면 double로 인식

오버플로우(overflow) – 가장 큰 값보다 더 큰 값을 저장할 때 발생

언더플로우(underflow) – 가장 작은 값보다 더 작은 값을 저장할 때 발생

//예시 작성

문자형

문자 인코딩에 유니코드 방식만을 사용

문자 자료형은 char사용

작은 따옴표 안에 한 개의 문자를 넣어서 표기

문자열

기본형 char을 사용 불가

참조 자료형 String사용

따옴표 속에 넣어서 표시

변수의 분류(클래스에 관해서는 6장 클래스에서 자세하게 다룰 예정)

|  |  |
| --- | --- |
| 분류 | 설명 |
| 지역변수(local variable) | 클래스 이외의 영역(메서드, 생성자, 초기화 블럭)에 선언  (클래스에 관해서는 6장에서 더 자세하게 다룰 예정)  변수 선언문이 수행되었을 때 메모리가 할당  메서드가 끝나면 소멸 |
| 매개변수(parameter) | 함수의 정의에서 전달받은 인수를 함수 내부로 전달하기 위해 사용하는 변수 |
| 인스턴스 변수(instance variable) | Class 내부에 위치  인스턴스가 생성될 때 메모리가 할당  객체를 생성해야만 사용 가능 |
| 클래스 변수(class variable) | Class 내부에 위치  클래스가 메모리에 올라갈 때 메모리가 할당(메모리가 딱 한번만 올라감) |

변수에 값 할당

= 연산자 사용

오른쪽의 것을 왼쪽에 할당

변수끼리 할당 가능

마지막에 할당된 값이 변수

연산자(표로 대체)

비교연산자

두개의 값을 비교한 후 결과로 true/false를 반환

|  |  |  |
| --- | --- | --- |
| > | op1>op2 | op1이 op2보다 크면 결과값이 true, 그 이외의 경우 false |
| < | op1<op2 | op1이 op2보다 작으면 결과값이 true, 그 이외의 경우 false |
| >= | op1>=op2 | op1이 op2보다 크거나 같으면 결과값이 true, 그 이외의 경우 false |
| <= | op1<=op2 | op1이 op2보다 작거나 같으면 결과값이 true, 그 이외의 경우 false |
| == | op1==op2 | op1와 op2가 같으면 결과값이 true, 그 이외의 경우 false |
| != | op1!=op2 | op1와 op2가 다르면 결과값이 true, 그 이외의 경우 false |

|  |  |  |
| --- | --- | --- |
| instaceof | op1 instanaceof 클래스 | op1이 클래스의 인스턴스인 경우 true |

비트 연산자

데이터를 비트 단위로 연산할 수 있는 연산자

정수형에만 사용 가능

|  |  |  |
| --- | --- | --- |
| |(or 연산자) | op1|op2 | 두개의 피연산자 중 하나의 값이 1이면 결과값이 1  그 외의 경우 결과값이 0 |
| &(and 연산자) | op1&op2 | 두개의 피연산자 모두 1일 경우 결과값 1  그 외의 경우 결과값이 0 |
| ^(xor 연산자) | op1^op2 | 두개의 피연자가의 값이 서로 다를 경우 결과값 1  그 외의 경우 결과값이 0 |

논리연산자

boolen형 조건식만 허용

&&연산이 ||연산보다 우선순위가 높음

|  |  |  |
| --- | --- | --- |
| &&( and 결합) | op1&&op2 | 두개의 피연산자 모두 true일떄 true결과값 출력 |
| ||(or 결합) | op1||op2 | 두개의 피연자중 하나가 true일 경우 true결과값 출력 |

삼항연산자

조건 ? 수식1 : 수식2

조건을 평가하여 true이면 수식1/false이면 수식2를 실행

증감연산자

증감연산자는 변수에 붙는 위치에 따라 출력되는 결과가 달라진다.

등가비교연산자

두 피연산자의 값이 같은 지 다른 지 비교하는 연산자

참조형에서도 사용가능

연산자 우선순위

|  |  |
| --- | --- |
| 우선순위 | 연산자 |
| 1 | ( ) [ ] { } |
| 2 | ++ -- ~ ! |
| 3 | \* / % |
| 4 | + - |
| 5 | >> >>> << |
| 6 | > >= < <= |
| 7 | == != |
| 8 | & |
| 9 | ^ |
| 10 | | |
| 11 | && |
| 12 | || |
| 13 | ? : |
| 14 | = |

기본 자료형 예제

**public** **class** DataTypeExample

{

**public** **static** **void** main(String[] args)

{

System.***out***.println("byte타입의 최소값 : " + Byte.***MIN\_VALUE***);

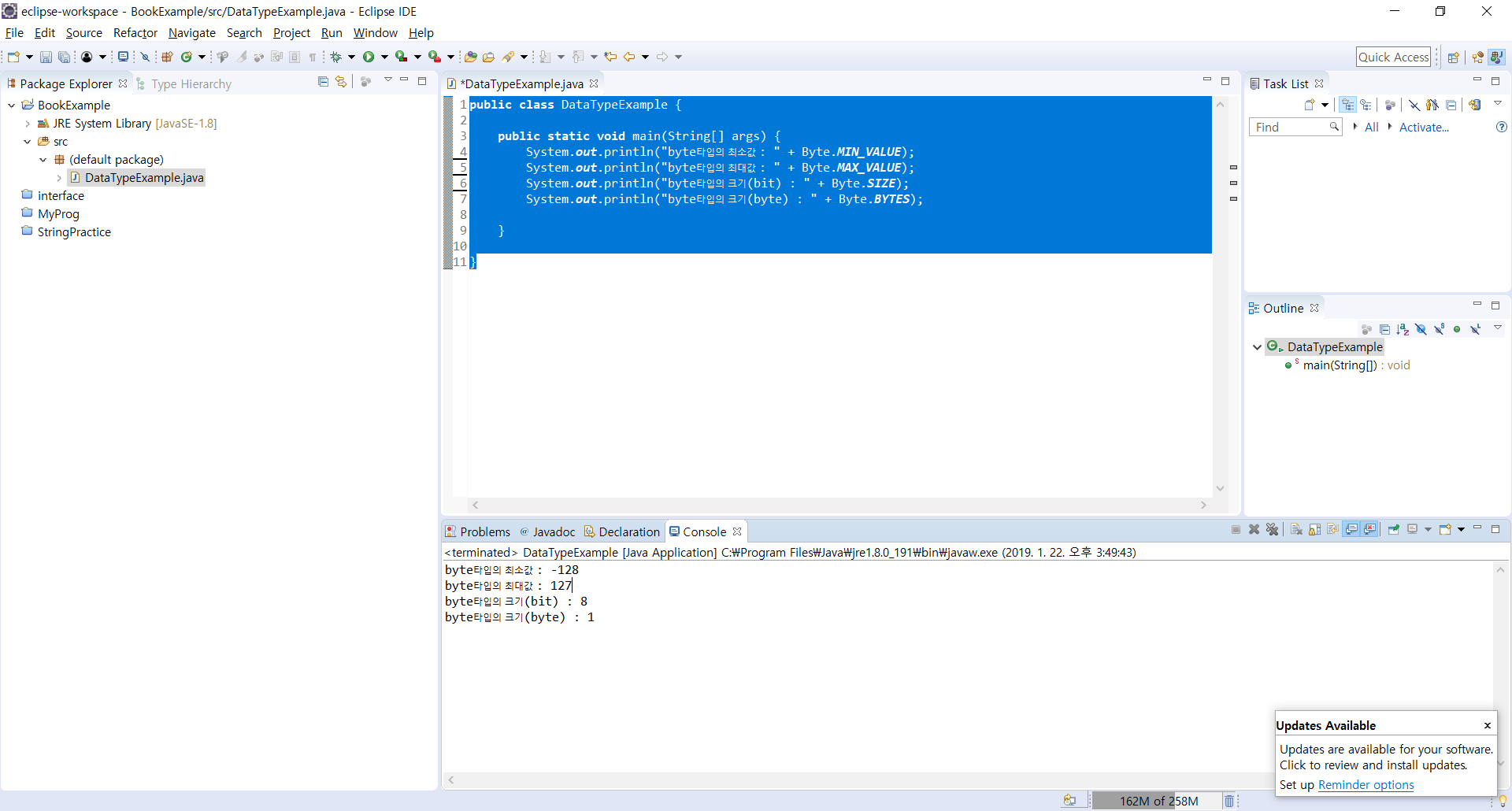
System.***out***.println("byte타입의 최대값 : " + Byte.***MAX\_VALUE***);

System.***out***.println("byte타입의 크기(bit) : " + Byte.***SIZE***);

System.***out***.println("byte타입의 크기(byte) : " + Byte.***BYTES***);

}

}



정수/실수 예제

**public** **class** Example

{

**public** **static** **void** main(String[] args)

{

{

**double** r;

r=3.14;

}

System.***out***.println(r);

}

}

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

r cannot be resolved to a variable

범위를 벗어난 변수를 사용 에러 발생

부동소수점 예제

//double과 float의 비교 연산시 문제

**public** **class** FloatPractice

{

**public** **static** **void** main(String[] args)

{

System.***out***.println(1.0==1.0f); //true

System.***out***.println(1.1==1.1f); //false

System.***out***.println(0.1==0.1f); //false

System.***out***.println(0.9==0.9f); //false

System.***out***.println(0.01==0.01f); //false

}

}

float은 4바이트 실수, double은 8바이트 실수 값을 저장할 수 있다. 여기에서 float과 double자료형의 실수 표현의 정밀도 차이 발생한다. 따라서 double과 float값을 비교할 땐 float형으로 형변환 하거나 정수로 변환하여 비교하여야 한다.

**public** **class** FloatPractice

{

**public** **static** **void** main(String[] args)

{

**int** a = 10;

**float** b = 0.3f; //f를 붙여 주어야만 float으로 인식 실수형은 double이 기본

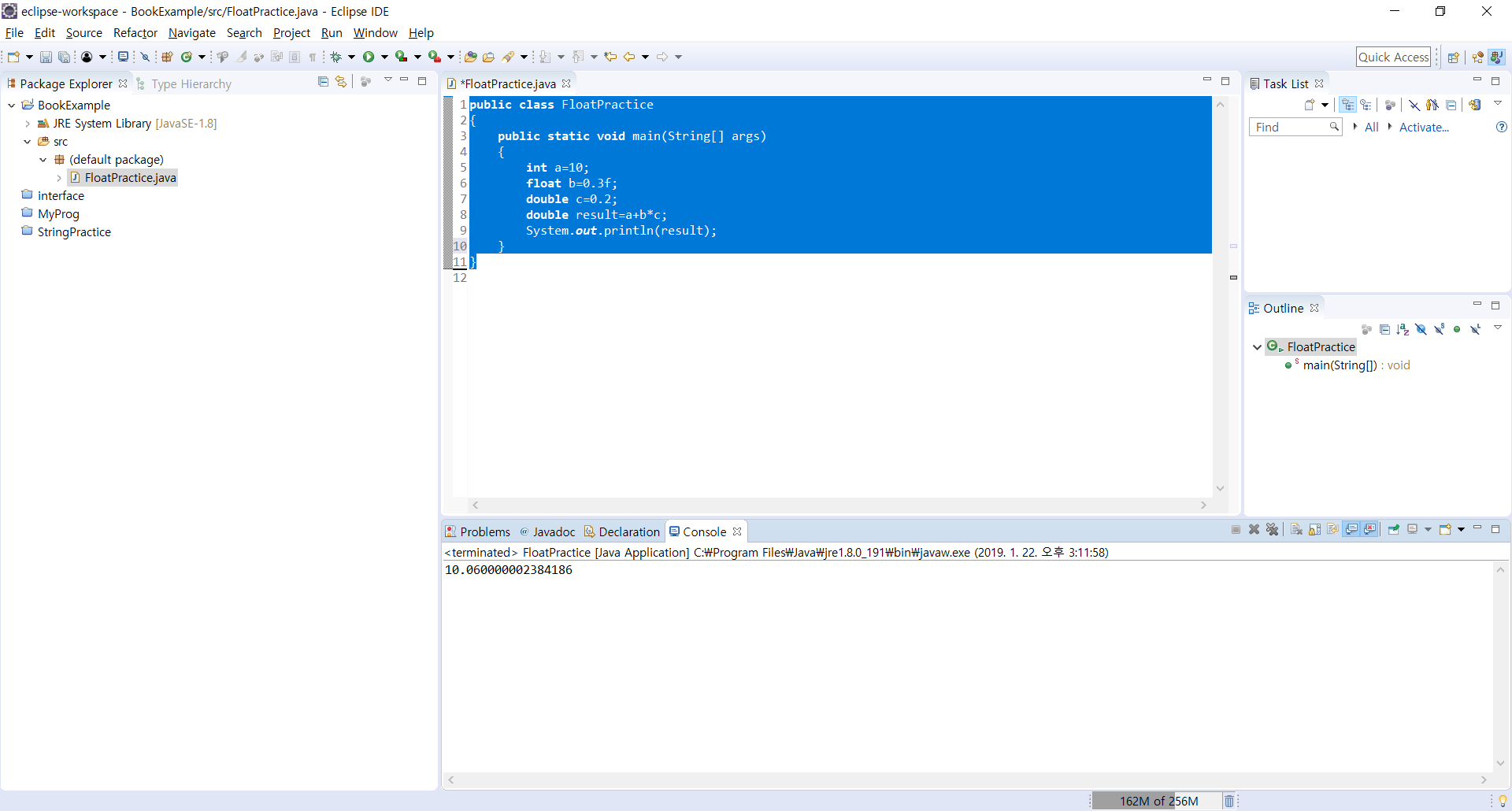
**double** c = 0.2;

**double** result= a + b \* c;

System.***out***.println(result);

}

}



정확한 계산은 10.06이다. double이 0.2를 정확히 표현하지 못한다.

반면

**public** **class** FloatPractice

{

**public** **static** **void** main(String[] args)

{

**int** a = 10;

**float** b = 0.3f;

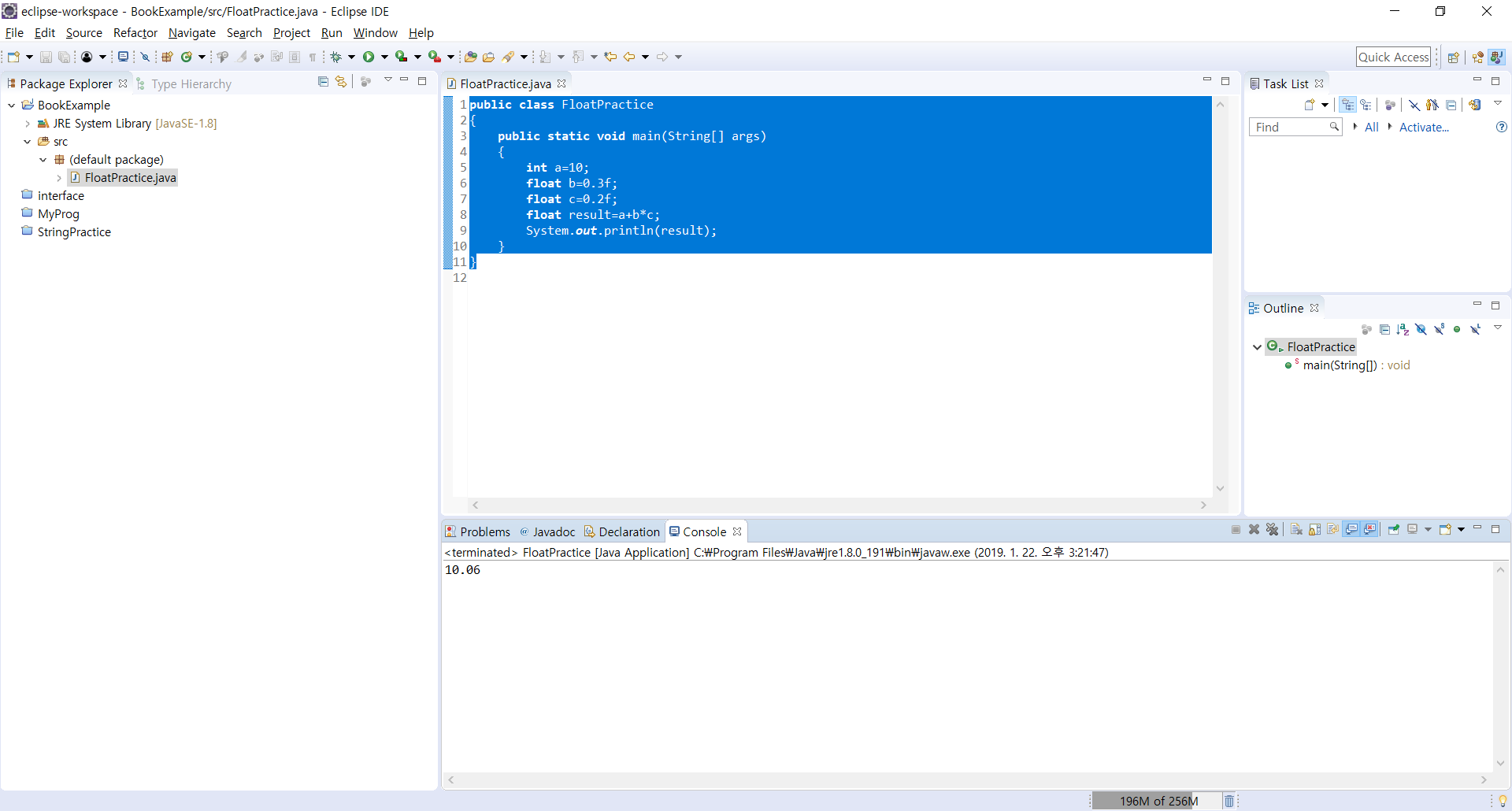
**float** c = 0.2f;

**float** result = a + b \* c;

System.***out***.println(result);

}

}



전부 float으로 선언하는 경우 위와 같은 문제점은 발생하지 않는다.

따라서 double을 사용한 실수의 정확한 계산을 위해선 정수 형태로 계산을 한 뒤, 소수점을 나중에 반영하는 것이다.

**public** **class** FloatPractice

{

**public** **static** **void** main(String[] args)

{

**int** a=10;

**int** d = a\*100;

**int** b=3;

**int** c=2;

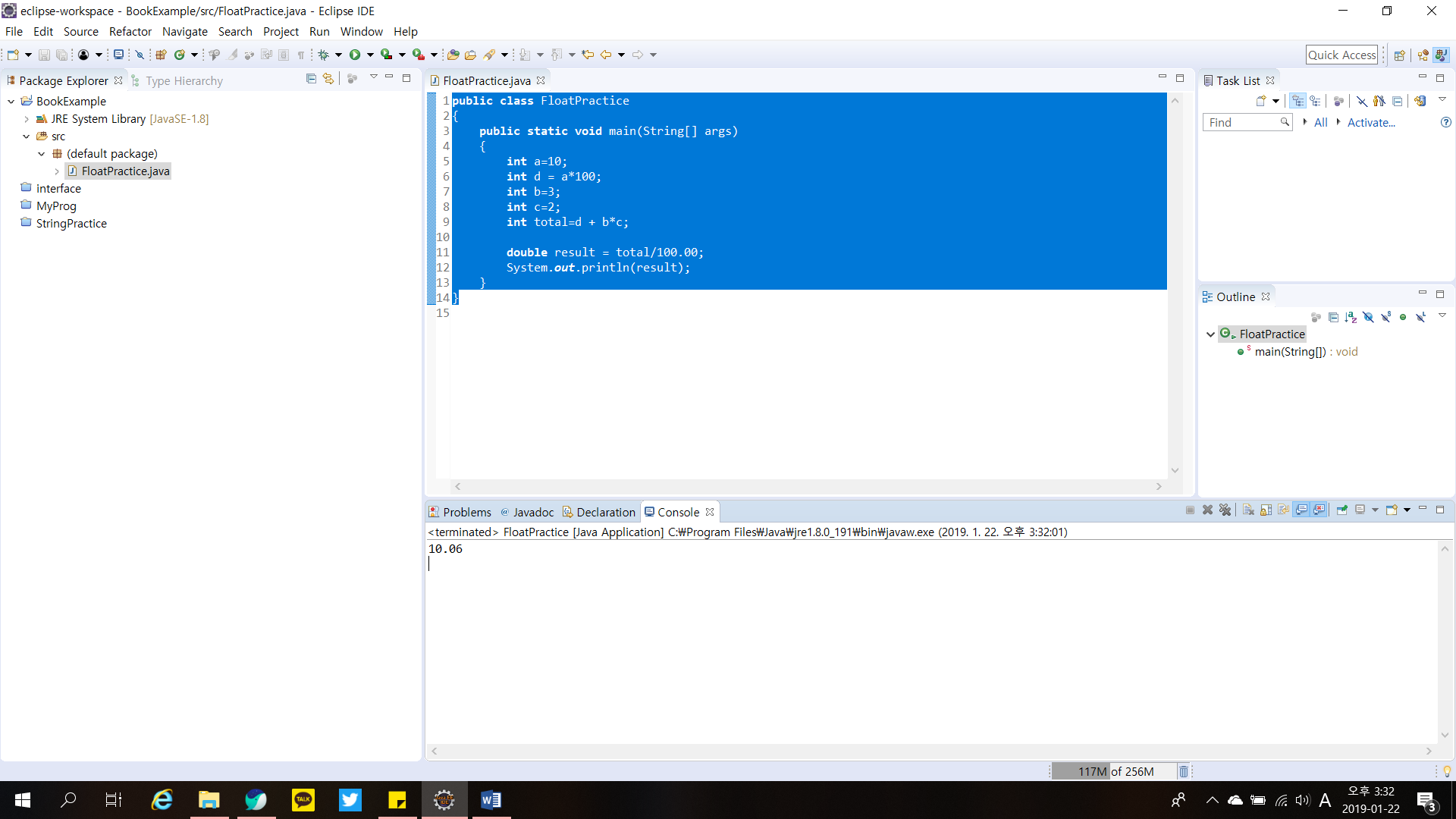
**int** total=d + b\*c;

**double** result = total/100.00; //소수점 둘째 자리까지 표시함

System.***out***.println(result);

}

}



오버플로우 예제

**public** **class** OverflowExample {

**public** **static** **void** main(String[] args)

{

**int** a = 1000000;

**int** b = 1000000;

**long** c = a \* b;

System.***out***.println(c); //결과: -727379968

}

}

증감연산자 예제

**public** **class** OperatorExample {

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

**int** x = 1;

**int** y = x++;

System.***out***.println("후위 증감 연산자" + y);

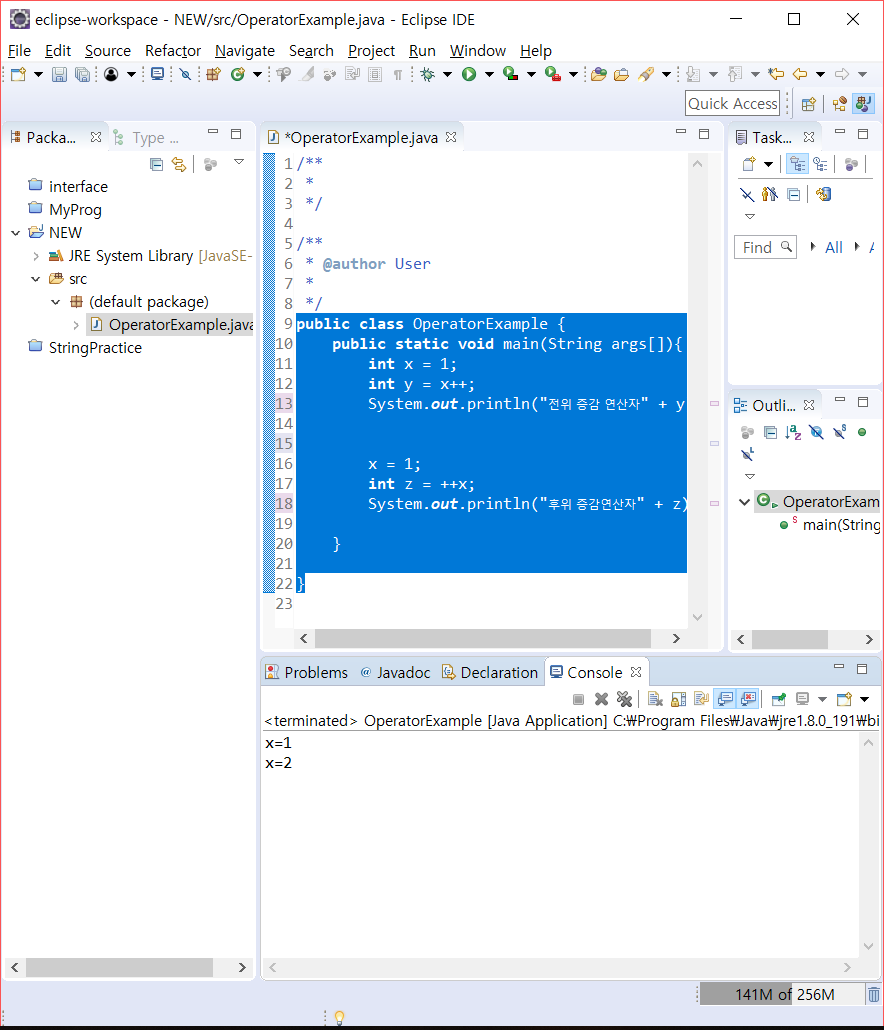
x = 1;

**int** z = ++x;

System.***out***.println("전위 증감연산자" + z);

}

}



전위 증감연산자 : 연산을 하기 전에 먼저 값을 +1

후위 증감연산자 : 연살을 한 후에 값을 +1