JAVA PROGRAMMING

Every .Java file is itself a class. Class name always start with a capital letter

[filename]

[program starts from here]

[commond line]

```
public class Demo {
        public static void main (String[] args) {
                System.out.println("Hello world!!");
        }
}
```

[available to all]

[System is a class and println is a fonction of it, out is a variable]

Output → Hello world!!

System.out.println( ) will add a new line but if we use System.out.print ( ) will print in the same line

For input we have Scanner class of Java.util. class So we have to import java.util.Scanner

*Number class →
Most of the time, while working with numbers in Java, we use primitive data types (which cannot be further break). But, Java also provides various numeric wrapper sub classes under the abstract class number present in java lang package. There are mainly six ~~classes~~ subclasses under ~~[illegible]~~ Number class. These subclasses are Byte, integer, Double, Short, Float, long

*Note → Primitive Wrapper Classes are immutable in Java

* Literals →
Java literals are syntactic representations of boolean, character, numeric, or string data.

* Identifiers →
Identifiers are names of variable, methods, classes, packages, interfaces.

* Some more basics →
Octal values are denoted in Java by leading zero. Normal decimal numbers cannot have leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's range (0 - 7)

We signify a hexadecimal constant with a leading zero - x (0x or 0X). The range of hexadecimal digit is 0 to 15, so A to F are substituted ~~[illegible]~~ from 10 to 15.

When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type. An integer literal can always be assigned to a long variable. However, to specify a long literal, you will need to ~~explicitly~~ explicitly tell the compiler that the literal value is of type long. You do this by appending an upper- or lower case L to the literal

Beginning with JDK 7, you can also specify integer literals using binary. To do so, prefix the value with 0b or 0B. For example, this specifies the decimal value 10 using a binary literal: int x = 0b1010;

int x = 123_456_789;
the value given to x will be 123,456,789. The underscore will ignored.

Floating-points literals in Java default to double precision. To specify a float literal, you must append an F and f to the constant.

Hexadecimal floating-point literals are also supported, but they are rarely used. They must be in a form similar to scientific notation, but a P or p, rather than an E or e, is used. For example, 0x12.2P2 is a valid floating-point literal. The value following the P, called the binary exponent, indicates the power-of-two by which the number is multiplied. Therefore, 0x12.2P2 represents 72.5.

double num = 9_423_497_862.0
the value will be 9,423,497,862.0
double num = 9_423_497.1_0_9;
In this case, the fractional part is .109

Boolean literals are simple. There are only two logical values that a boolean value can have, true and false which internally map 1 and 0.

As we know in C/C++, strings are implemented as arrays of characters. However, this is not the case in Java. String are actually object types. Because Java implements strings as objects, Java includes extensive string handling capabi-lities that are both powerful and easy to use.

The remainder a%b is defined such that (a/b) * b + a%b is always equal to a.

Date........
Page........

```
System.out.println(-15 % 2);   // -1
System.out.println(15%2);   // 1
```

TYPE CONVERSION AND TYPE CASTING →

* Java's Automatic Conversions →

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

1. The two types are compatible
2. The destination type is larger than source ~~code~~ type

Java ~~of~~ also performs an automatic type conversion when storing a literal integer constant into variables of the type byte, short, long or char.

* Casting incompatible types →

Although the automatic types conversions are helpful, they ~~are~~ will not fulfill all needs. For example, what if you want to ~~be~~ assign an int value to a byte variable ? This conversion will ~~be~~ not be performed automatically, because a byte is smaller than an int. This kind of conversion is called narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type

For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced ~~modulo~~ modulo (the remainder of ~~the~~ an integer division by the) byte's range.

```
int a;
byte b;
b = (byte) a;
```

A different type of conversion will occur when a floating point value is assigned to an integer type: truncation

* Automatic type promotion in expressions→

```
int a = 257;
byte b = (byte) a;
```

Date........
Page........

When the value 257 is cast into a ~~variable~~ byte variable, the result is the remainder of the division of 257 by 256 (~~by~~ the range of a byte), which is 1 in the case.

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a*b /c;
```

The result of the intermediate term a*b easily exceeds the range of either of its byte operands. To handle this kind of problem, Java automatically promotes each byte, short, or char operand to int when evaluating an expression. This means that the subexpression a*b is performed using integers - not bytes.

```
byte b = 50;
b = b*2;   // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid byte value, back into a byte variable. However, because the operands were automatically promoted to int when the expression was evaluated, the result ~~is~~ has also been promoted to int

* Type promotion rules→
Java defines several type promotion rules that apply to expressions. They are →
① All byte, short and char values are promoted to int, as just described. ② If one operand is long, the whole expression is promoted to long. ③ If one operand is float, the entire expression is promoted to float. ④ If any operands are double, then ~~[illegible]~~ result is double.

```
class Promote {
    public static void main(String[] args){
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = 0.1234;
        double result = (f*b) + (i/c) -(d*s);
        System.out.println((f*b)+" " + (i/c)+" "+ (d*s));
        System.out.println("Result is = "+result);
    }
}
```

Let's look closely at the type promotions that occur in this line from program :

```
double result = (f*b) + (i/c) + (d*s);
```

In first subexpression, f*b , b is promoted to a float and the result of the subexpression is float. Next, in the subexpression i/c , c is promoted to int , and the result is of type int. Then , in d*s , the value of s is promoted to double and the type of the subexpression is double. Finally , these three intermediate values , float, int and double are considered. The outcome of float plus an int is a float. Then the resultant minus last double is promoted to double , which is the type for the final result of expression.