First Program:

→ Every file which ends with .java is a class

→ A file must have a class of name same as filename and all the code must be inside the class in the file, this is compulsory when the class is public.

Main.java

```
class Main{
}
```

→ By convention the first letter of class_name must be capital.

→ And the class must be public so that the class can be accessed by anywhere (from another file and so on).

Main.java

```
public class Main{
}
```

→ Inside this class we must have a function with name "main". resembles int main() in C/C++.

Main.java

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

To compile, we do:

```
javac Main.java
```

→ then we get a bytecode generated

To ~~exe~~ run:

```
java Main
```

→ then we get "Hello World!"

→ we have:

⇒ main(), it have to be present and it is the entry point of program.

⇒ public, so that main() can be accessed by anyone as main() is entry point so if we keep it private then it might create problem as someone ~~not~~ might not get access so we declare main() as "public".

⇒ static, as we know to access any member of class, we need object of the class so same for our main() function, so if we need to access the main() function, we would need an object of the class in which main function is present. But we ~~do~~ need main() as entry point so we want to access the main() without creating any object and for that we use "static".

agar mujhe main() se hi entry karna hain ~~aur~~ aur yeh static nhi hain ~~toh~~ it means mujhe ek obj create karke then us object ka use karke main() main ghusna hoga but tab toh main() entry point rahegahi nhi toh we declare it as static.

⇒ void, return type of main() function.

⇒ String[] args, this is a command line argument which is an array (collection of strings)

Note - we can say where we want to save our byte code

javac -d . Demo.java (to create in current dir)
javac -d .. Demo.java (to create in prev directory)

Note - how does our computer knows that where the compiler or jit or etc are present.

Then javac or jit, there are executable file which directs computer to the ~~as~~ appor compiler or etc.

⇒ shortcut for our main() is psvm
⇒ " " output " sout

→ Another thing we have in our file is "package package_name", here it is a package or a folder which can contain many files so we can use the same package for any number of file.

```
package pack_name;
        ↑
   a folder / a package
```

⇒ System.out.println(), Java has provided some basic functions for printing, input taking, debugging and etc. System is a class inside System file, it (this class) have several class fields and methods which are useful to us. It cannot be instantiated since it is a final class.

println() is a function which is defined in such a manner that we need to give a string to it which get printed out, is a variable which is of type printstream and out has a variable called println() and println() is also of type printstream. (Printstream is a class)

And default is command line of printing or displaying but we can change it if we want by doing out = some value

→ and if we want to print something without displaying it to the next line then we can do:

```
System.out.print();
```

eg-

```
System.out.println("Hey");
System.out.println("People");
        ⇓
Hey
People
```

```
System.out.print("Hey");
System.out.println("People");
        ⇓
Hey People
```

→ and System.out is the standard output stream.

⇒ For taking input,

syntax:

```
Scanner input = new Scanner( );
```

- Scanner — this is a class
- input — object we created
- new — new keyword
- Scanner( ) — class as constructor
- inside ( ) we mention that from where we will take the input. Can be a file or etc

→ `Scanner input = new Scanner(System.in);`

- input — this will read what we want to take as input
- System.in — this means from keyboard we are taking inputs as "in" indicates keyboard

→* For using Scanner class, we need to import java.util.Scanner.

eg-

```
Scanner input = new Scanner (System.in);
System.out.println (input.next());
```

// in first line, we ask input to take value from keyboard and then in second line "input.next()" means that the o/p must be string and then that particular string will get displayed in command line.

→ for taking integer input, we have .nextInt()

→ " " string until space, " " .next()

→ " " line of strings, " " .nextLine()

Primitives : (Data type which we can't break into any other data type comes under primitive)

→ integer as int // int is primitive

→ String as String // String is not primitive
(starts with double quotes)

→ char as char // char is primitive
(starts with single quotes)

→ float as float // primitive
eg - float mark = 98.67f;
↑
mentioning "f" is compulsory

→ double as double // for large decimal values

→ long as long // for large integer values (add "L" after)

→ boolean as boolean // either true or false

⇒ instead of having int, we have long and instead of having float, we have double, this is due to size, we'll know in bitwise section.

⇒ why we have "f"?
by default all decimal value are of double type and so to define it as float we add "f" in the last.

⇒ why we have "L"?
by default we have integer type and so we use "L"

Note: All the primitive data type don't have any class and so all this starts with small case.
We do have some wrapper class which gives us more function to our primitives.

// taking input of various primitive data type

```
Scanner input = new Scanner(System.in);

int roll_no = input.nextInt();
System.out.println("Your roll no :" + roll_no);
```

```
54
Your roll no : 54
```

Note : any object on assignment is literal and reference to it is identifier.

eg — `int a = 10;`
identifier literal

Note :

eg — `int a = 234_000_000;`
`System.out.println(a);`
⇓
234000000 { since "_" gets ignored

```
String name = input.nextLine();
System.out.println(name);
```

```
My name is Dwipshikha
My name is Dwipshikha
```

```
float marks = input.nextFloat();
System.out.println(marks);
```

```
564.675862
564.67584
```
// floating point error

```
10
10.0
```
// typecasting happened

Typecasting and conversion:

→ Automatic ~~type casting~~ type conversion would happen if both the types are compatible.

eg —

```
float num = input.nextFloat();
System.out.println(num);
```

```
65
65.0
```
// here we gave integer ip
// type conversion

// But if we do:

```
'a'
// error
```
// here we gave a char input

Main point to note: In Typecanversion, the destination data type must be greater than the source data type. Since int < float, so it works.

// if we do:

```
int num = input.nextInt();
```

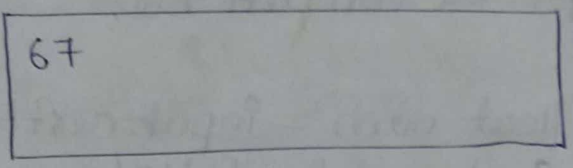
```
56.789
//error
```
// here destination type is smaller than source type.
// here we gave float as input

But type conversion (automatic) may not be helpful in some cases (suppose we given float value to store it in int).

And we have to ask for this explicitely which is known as narrowed conversion as we are explicitely asking for it.

This narrowed conversion is called Type casting, where the destination type can be smaller than the source type.

eg- int num = (int)(67.56f); // giving float
System.out.println(num);



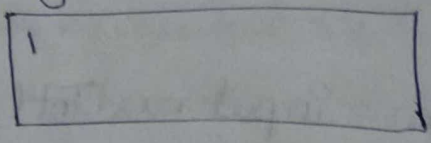
// type casting

Automatic type promotion in expression:

Now as we learned about typecasting still we have to keep in mind that the source type's size must be smaller or equal to destination.

Let us see an eg of opp to this point,

```
int a = 257;
byte b = (byte)(a); // 257 % 256 = 1
System.out.println(b);
```



Here since byte can take value upto 256 and so it can't store a (a = 257) even if we do typecasting, so what it does is, it takes modulo of the assigned

value with its size and the remainder will get stored in the variable or ~~right~~ left side of typecasting.

But when we do as:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a*b / c;
System.out.println(d);
```

```
20
```

→ here we can see, a and b are bytes which holding value less than there limit but a*b = 40 × 50 = 2000, which >> size of byte then how it get stores ina variable of type byte, since we know

byte * byte ⟶ byte, so 2000 also must be in byte but ~~this~~ is impossible. But we can see, we are ~~getting~~ correct answer.

So, what happens is that, in Java we have a thing called automatic type promotion, where

since a*b is a subexpression so byte a * byte b and then the result is stored in int type automatically since it is a subexpression and then this a*b = 2000 is divided by c, we get 20 which is in limit of size of byte and int and the result gets stored in d which is of int type.

in short, (expectation) byte a * byte b ⟹ byte (in general)
(reality) but byte a * byte b ⟹ int (because of automatic type promotion provided by Java)

so now if we do;

```
byte b = 50;
b = b * 2;  // this gives error
```

as (b*2) is now int and we can't assign int to a byte value automatically.

→ There are some rules for automatic typepromotion

i) All the byte, short and char values are promoted to int

ii) Any operand is of float then the result of the operation will be float.

If any of the oprand is double then the result of the operation will be double.

eg- 3 * 5.f345f ⇒ float value

eg-

```
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);
//(f+b) ⇒ float (according rule (ii))
// (i/c) ⇒ int ( " " " )
//(d * s) ⇒ double ( " " " " )
// float + int - double ⇒ double ( " " " )
System.out.println((f+b) + " " + (i/c) + " " +
(d * s));
system.out:println(result);
```

Statement and loops:

→ if (condition) {
    // code
}

If the condition is true then only code will run otherwise not.

→ initialization
while (condition) {
    // code // update
}

Same thing.

eg- int count = 1;
while (count != 5) {
    System.out.println(count);
    count++; // shorthand for count = count + 1;
}

→ for (initialization; condition; update) {
    // code
}

eg- for (int count = 1; count != 5; count++) {
    System.out.println(count);
}

Note: when we know, how many times the loop will run; use use for loop and otherwise use use while loop.