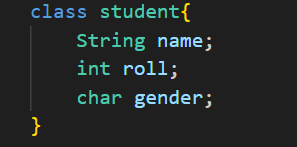
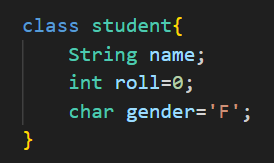
A class is a template for an object, and an object is an instance of a class.

A class creates a new data type that can be used to create objects.

This is how we declare a class.



Another way of declaring it is presetting some defaults. The code depicting that is as follows.



The main application of declaring defaults as observed in above case is to ensure that class by itself assigns some value to theses class elements in case they are not declared via objects.

When you declare an object of a class, you are creating an instance of that class.

Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

Objects are characterized by three essential properties: state, identity, and behavior.

The state of an object is a value from its data type. The identity of an object distinguishes one object from another.

It is useful to think of an object’s identity as the place where its value is stored in memory.

The behavior of an object is the effect of data-type operations.

Now this is how we at least declare reference to the objects in java.



Now a genuine question is where is this ‘Ashutosh’ variable of the class student. Well it exists in stack memory. Note that, the object is just instantiated here. It will access class elements and edit them in upcoming code.

If ‘Ashutosh’ is printed, the following will be the response.



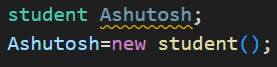
The 'new' keyword dynamically allocates(that is, allocates at run time)memory for an object & returns a reference to it.

This reference is, more or less, the address in memory of the object allocated by new.

This reference is then stored in the variable.

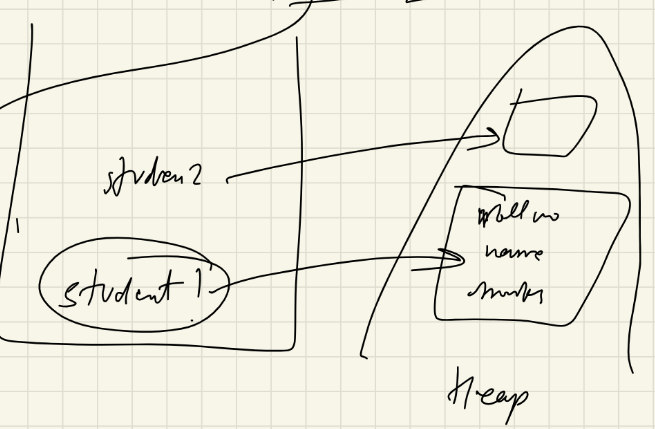
Thus, in Java, all class objects must be dynamically allocated.

This is how we allocate a class object.



The first line declares Ashutosh as a reference to an object of type student. At this point, Ashutosh does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to Ashutosh. After the second line executes, you can use Ashutosh as if it were a Box object. But in reality, Ashutosh simply holds, in essence, the memory address of the actual Box object.

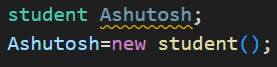
The Ashutosh reference is stored in stack whereas its features lies in the heap memory.



The key to Java’s safety is that you cannot manipulate references as you can actual pointers.

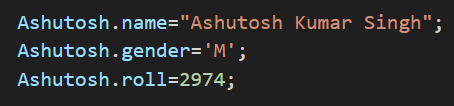
Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

**A Closer Look at new:**



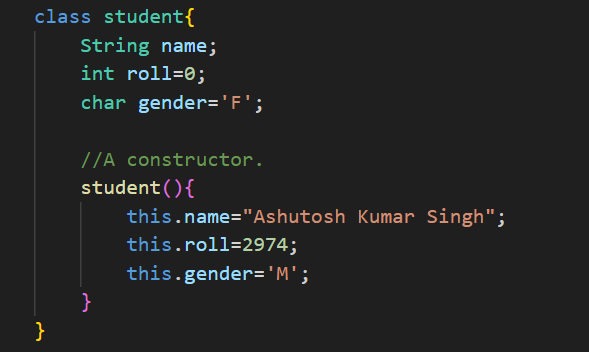
Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

Now lets see a code to see how to access an object and edit out its elements. Note that the dot operator plays a important role in this. The dot operator is also referred as separator.

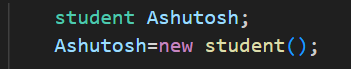


Now we take deep look into constructors. We have already seen how do we create class and object and then access objects via the dot operator to set the values elements of the class belonging to the object. Now we are going to do the same thing, but this time we are going to do it with constructors.

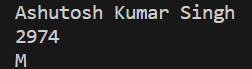
Constructors are generally declared inside the class object. Their return type is same as that of the class. A special keyword called ‘this’ plays a major role when it comes to usage of constructors for object creation. Lets see some snap of code to understand all that we are saying.



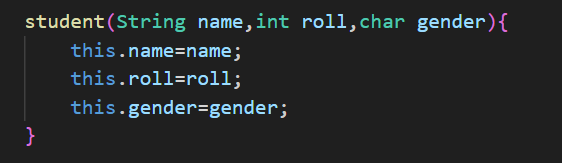
Yeah? So this is how we construct a constructor and assign object elements without using the dot operator exclusively. ‘this’ keyword here is to point reference to object name. In our case it ‘Ashutosh’. So this point to ‘Ashutosh’. Its very much like ‘this.name’ similar to ‘Ashutosh.name’. Lets see the main code to understand further with output.



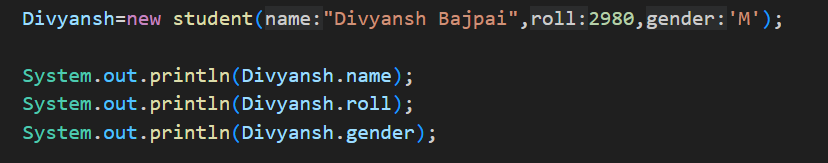
The ‘student()’ functions as the one calling the student constructor to assign values to class variables. Constructors with no inputs are called single/Independent constructors. The output is as follows.



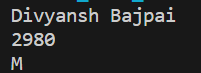
Lets see another instance of constructors with little bit of variations.



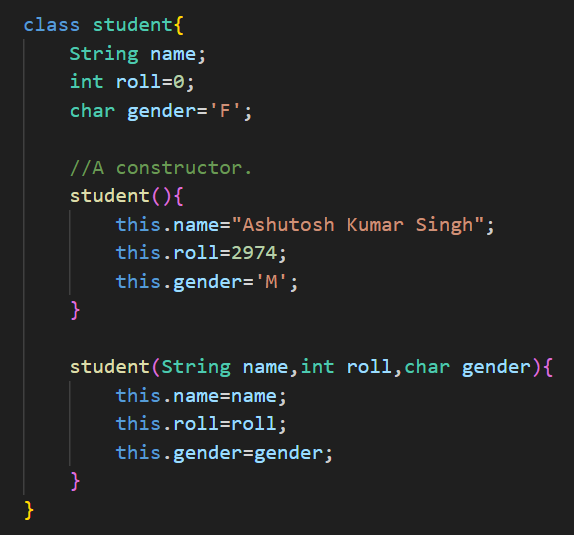
As observable this time we have 3 parameters being fed into the function. Corresponding mains call are.



The output can be observed as follows.

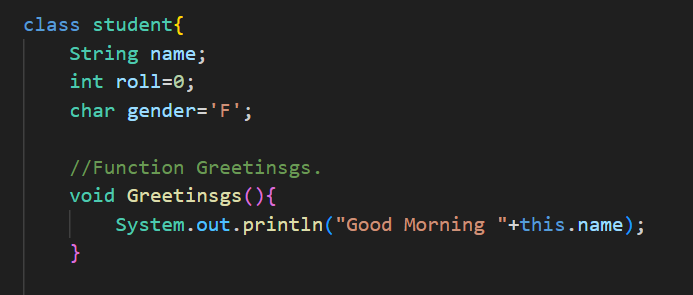


Now let move onto a special topic called, “Constructor Overloading”. This phenomenon takes place when within the same class, we declare two constructors with the same name but different parameter. The code example of this is as follows.

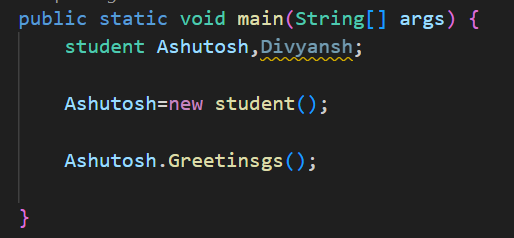


As observable here we have two constructors, in the same class, but the number of arguments they take are different.

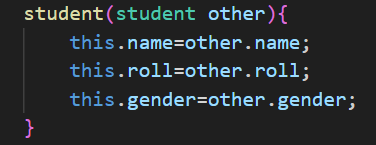
Now lets see few instances of functional programming within our artificial classes.



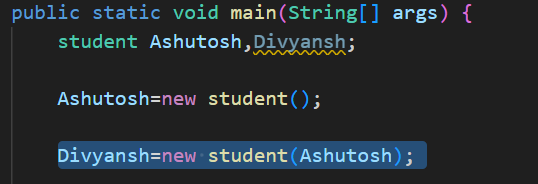
Lets see how do we call this function. Quite simple it is, <NameOfReferenceVariable>.<FunctionName>() from the main function. Example is as follows.



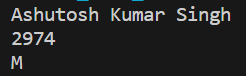
Lets say we want to duplicate a given object to another. Well we can do that via the following Constructor.



The input is as follows.

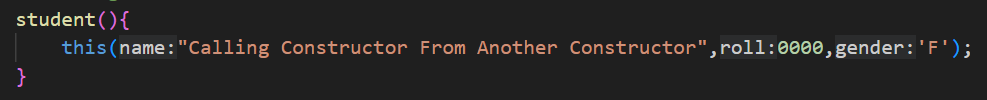


The corresponding output is as follows.



As observable the code worked successfully.

Another phenomenon surrounding the that we have not discussed is calling another constructor from a given constructor. Exampler code is as follows.



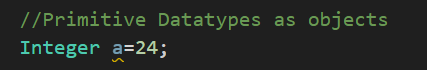
The input and output can be guessed.

You might be wondering why you do not need to use new for such things as integers or characters.

The answer is that Java’s primitive types are not implemented as objects.

Rather, they are implemented as “normal” variables. This is done in the interest of efficiency. Though when we see the usage of Strings and Arrays note how we use the ‘new’ keyword for their instantiation.

Now what is we want is to implement primitive datatypes as objects. For that we use special keywords such as Integer and what not. Examplar code depicting its implementation is as follows.

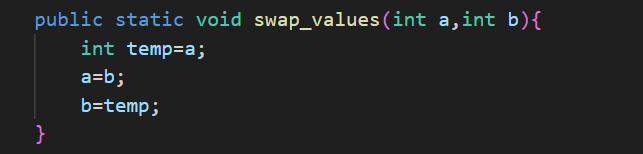


Write note how does this creates a difference. In Java whenever we pass primitive datatypes to a function, its pass by value. Any changes made to these variables in the function, does not have any effect on the variables globally.

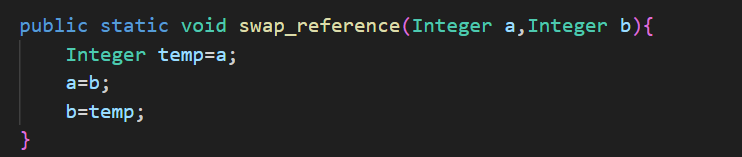
Its not the case with objects. Whether it be arrays, strings or Integer. Object when passed into the function as passed as references hence changes made to them in the function as observed globally.

Two contrasting code blocks for this are as follows.

Pass by value.



Pass by reference.

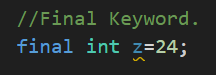


Next up lets discuss about “**final**” keyword.

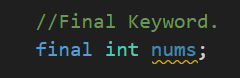
A final keyword can be used with both primitive datatypes and objects.

Though the effect that it has in either case is quite different. First lets discuss how does it work with primitive datatypes.

Declarations:

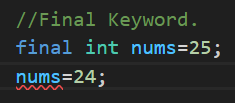


Note that we cannot have a null declaration in case we are using the final keyword. For instance:



The above should not happen.

Now what is the point of using the final keyword. Well in reference to primitive datatypes, whenever we declare when with final in conjunction, we see that it prohibits us from changing the value of that variable anywhere in the code. For instance:

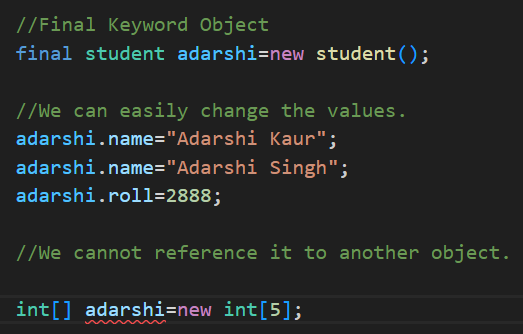


As observable, once we declare nums with 25, it shows error when we try to change it with 24.

It is important to understand that new allocates memory for an object during run time.

Now this was all about how final works with primitive datatypes. Now lets discuss that how does it work with objects.

In case we declare object with final keyword, it allows us to change there values, what it doesn’t allow us is re-reference that object variable to another object. For instance:



Its important to note that, if we do not define any of the above object elements, the default values given to them in the class block will rather be used. In case we to not declare the defaults too, we see that datatype specific defaults are used. For example, int by default uses 0, float, 0.0 and etc. Non-primitive datatypes such as arrays and strings use null, in case declaration has not been made.

Box b1 = new Box();

Box b2 = b1;

b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part

of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object

through b2 will affect the object to which b1 is referring, since they are the same object.

When you assign one object reference variable to another object reference variable, you are not creating a copy of the

object, you are only making a copy of the reference.

int square(int i){

return i \* i;

}

A parameter is a variable defined by a method that receives a value when the method is called. For example,

in square( int i), i is a parameter. An argument is a value that is passed to a method when it is invoked.

For example, square(100) passes 100 as an argument. Inside square( ), the parameter i receives that value.

NOTE:

Bus bus = new Bus();

lhs(reference i.e. bus) is looked by compiler & rhs (object i.e. new Bus()) is looked by jvm