new operator in Java

When you are declaring a class in java, you are just creating a new data type. A class provides the blueprint for objects. You can create an object from a class. However obtaining objects of a class is a two-step process :

* 1. **Declaration :**First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Below is general syntax of declaration with an example :
  2. **Syntax :**
  3. **class-name var-name;**
  4. **Example :**
  5. // declare reference to an object of class Box
  6. Box mybox;

A variable in this state, which currently references no object, can be illustrated as follows (the variable name, mybox, plus a reference pointing to nothing):



* 1. **Instantiation and Initialization :**Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the ***new*** operator. The ***new*** operator instantiates a class by dynamically allocating(i.e, allocation at run time) memory for a new object and returning a reference to that memory. This reference is then stored in the variable. Thus, in Java, **all class objects must be dynamically allocated.**

The ***new*** operator is also followed by a call to a class constructor, which initializes the new object. A [constructor](https://www.geeksforgeeks.org/constructors-in-java/) defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. In below example we will use the [default constructor](https://www.geeksforgeeks.org/g-fact-50/). Below is general syntax of instantiation and initialization with an example :

**Syntax :**

**var-name = new class-name();**

**Example :**

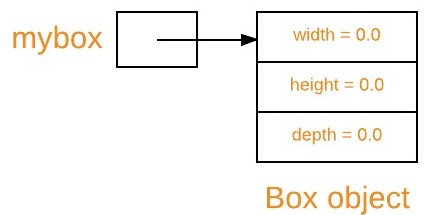
// instantiation via *new* operator and

// initialization via default constructor of class Box

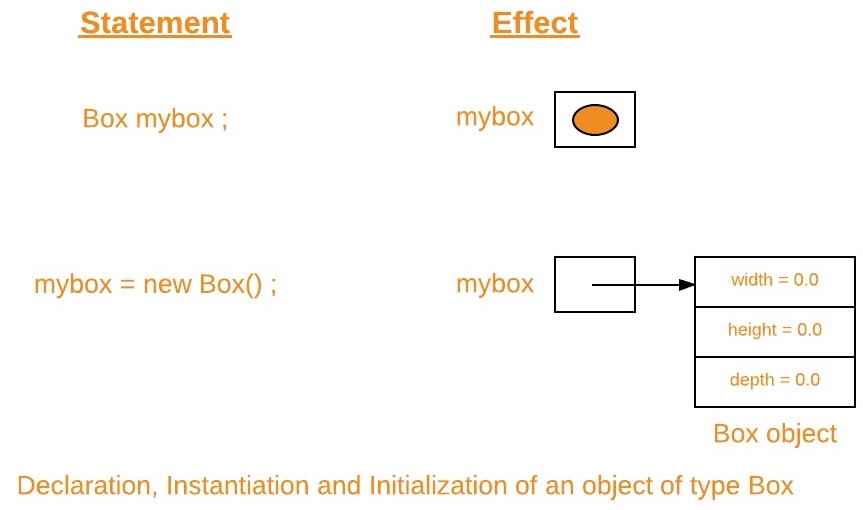
mybox = new Box();

Before understanding, how ***new*** dynamically allocates memory, let us see class *Box* prototype.

* 1. class Box
  2. {
  3. double width;
  4. double height;
  5. double depth;
  6. }
  7. A variable after second step, currently refer to a class object, can be illustrated as follows (the variable name, mybox, plus a reference pointing to Box object):



Hence declaration of a class variable, instantiation of a class and initialization of an object of class can be together illustrated as follows :



**Important points :**

1. The above two statements can be rewritten as one statement.
2. Box mybox = new Box();
3. The reference returned by the ***new*** operator does not have to be assigned to a class variable. It can also be used directly in an expression. For example:
4. double height = new Box().height;
5. Since [arrays are object in java](https://www.geeksforgeeks.org/arrays-in-java/), hence while instantiating arrays, we use ***new*** operator. For example:
6. int arr[] = new int[5];
7. At this point, you might be wondering why you do not need to use ***new*** operator for [primitives data types](https://www.geeksforgeeks.org/data-types-in-java/). 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. For object versions of the primitive data types, refer [Wrapper Classes](https://www.geeksforgeeks.org/wrapper-classes-java/).
8. The phrase “instantiating a class” means the same thing as “creating an object.” When you create an object, you are creating an “instance” of a class, therefore “instantiating” a class.

**Assigning object reference variables**

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. Let us understand this with an example.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate assigning  // of object reference variables    // Box class  class Box  {      double width;      double height;      double depth;  }    // Driver class  public class Test  {      // Driver method      public static void main(String[] args)      {          // creating box object          Box b1 = new Box();            // assigning b2 to b1          Box b2 = b1;            // height via b1 and b2          System.out.println(b1.height);          System.out.println(b2.height);            // changing height via b2          b2.height = 20;            // height via b1 and b2          // after modification through b2          System.out.println(b1.height);          System.out.println(b2.height);      }    } |

Output :

0.0

0.0

20.0

20.0

**Explanation :**

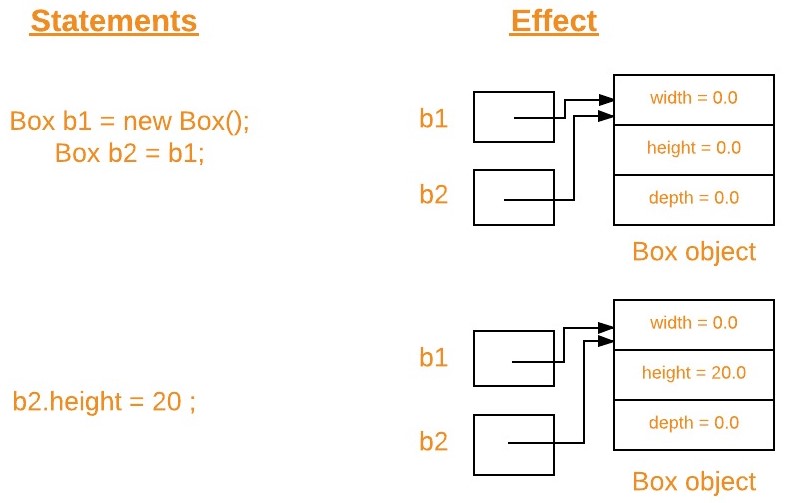
First let us understand what the following fragment does in above program.

Box b1 = new Box();

Box b2 = b1;

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **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. Same can be verified by output when we change *height* of box via **b2**.

This situation can be illustrated as follows :



**Note :**Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.For example :

Box b1 = new Box();

Box b2 = b1;

// ...

b1 = null;

Here, **b1** has been set to *null*, but **b2** still points to the original object.

**Passing object references variables to methods**

When we pass object reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. To know more with examples, refer [Passing and Returning Objects in Java.](https://www.geeksforgeeks.org/passing-and-returning-objects-in-java/)

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Attention reader! Don’t stop learning now. Get hold of all the important DSA concepts with the [**DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_dsa_content_bottom) at a student-friendly price and become industry ready.