If s1, s2, and s3 were initialized as:

```java
s1.rollno = 1;   s1.student = "Nandeesh";  s1.marks = 98;
s2.rollno = 2;   s2.student = "Neha";      s2.marks = 95;
s3.rollno = 3;   s3.student = "Prajwal";   s3.marks = 90;
```

Then the **actual data structure in memory** will look like:

```
students → [ (1, "Nandeesh", 98), (2, "Neha", 95), (3, "Prajwal", 90) ]
```

Key Takeaways

✔ `new Student[3]` **only creates space for 3 references**, not actual objects.
✔ Each index of `students[]` **stores a reference (pointer) to a Student object**.
✔ `students[0] = s1;` makes `students[0]` **point to the same object as s1**, meaning changes to `students[0]` will reflect in `s1` and vice versa.

Example: Accessing Elements

Now, if you write:

```java
System.out.println(students[0].student);  // Output: Nandeesh
System.out.println(students[2].marks);   // Output: 90
```

It works **just like accessing an object**, because `students[0]` holds a reference to `s1`.

Would you like a **diagram** to visualize how memory is structured here? 🚀

The following lines of code create an **array of Student references** and assign existing `Student` objects to the array. Let's break it down step by step.

Step 1: Creating an Array of Student References

```java
Student students[] = new Student[3];
```

- This **allocates memory** for an array that can hold 3 `Student` references.
- However, at this point, the elements of `students[]` are **null** (i.e., no actual `Student` objects are created yet).

Memory state after this step:

```
students → [ null, null, null ]
```

Step 2: Assigning Existing Student Objects to Array

```java
students[0] = s1;
students[1] = s2;
students[2] = s3;
```

- These statements assign the references of previously created `Student` objects (`s1`, `s2`, `s3`) to the corresponding indexes in the array.
- Now, each array element **points to an actual Student object**.

Memory state after assignments:

```
students → [ s1, s2, s3 ]
```