In Java, Array, LinkedList or other OOP. The variable when we create is not actually store the data in itself. The variable we created in Objects is storing a reference link to our data.

For Example:

Non Object Variable Declare:

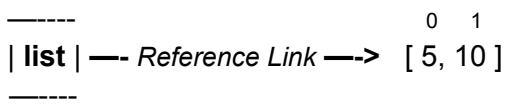
```
int x = 10;
```

In this case, variable **x** is storing a value of 10.

Object Variable Declare:

```
int[]  list = new int[];
list.add(5);
list.add(10);
```

In this case, the variable **list** is not storing value 5, 10 on itself. The actual picture is like this:



- **Array** is random access, so in a **list** variable we can randomly access any value.
- **LinkedList** is connected by previous or next node so the **list** variable can be seen as a pointer when looping the list.

Objects are LinkedList, ArrayList, Student Object, Car Object, etc.

ADT (Abstract Data Type)

It is about encapsulating data and operations into a single unit, ADTs provide a way to abstract the underlying implementation details and provide a clear interface for working with the data. It encapsulates our data structures such as ArrayList, LinkedList, etc. It's providing an easy way to access based on behaviors. Stack (LIFO), Queue (FIFO).

The purpose is to make it easier for users to access and manimupliate data without knowing how data structures work. Information hiding (Encapsulation), the internal workings of an object or data structure are kept hidden from the outside. This serves as a form of security, as it prevents users from directly accessing or altering the object's internal state, thereby avoiding unintended side effects and maintaining the integrity of the data. Another purpose is to make code easy to maintain and fix bugs.

Where is the local variable inside a function or a loop after work ?

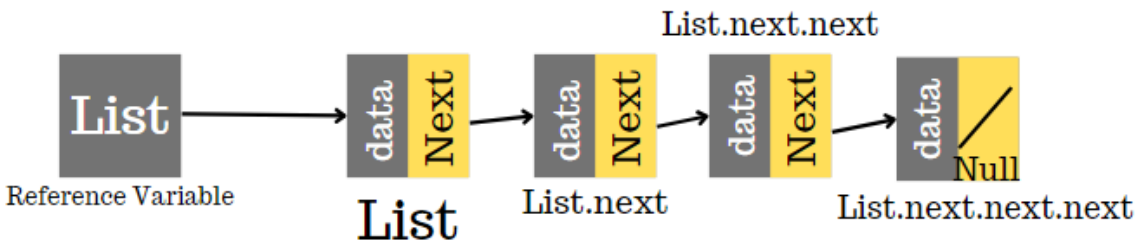
In most programming languages, variables declared within a certain block, function, or scope are only accessible within that scope. Once you exit that scope, the variable goes out of scope and is typically no longer accessible. It is effectively "wiped out" or removed from memory. It is no longer accessible, and its value is discarded

LinkedList Traverse:

We traverse the LinkedList with pointer (variable store the name of LinkedList)
To track elements in LinkedList when looping it, we can have local temporary variables such as *current, prev*. **"Current" variable** can be set equal to **Head/Node or the variable that stores reference to that LinkedList.**

After modifying, remember to reset the next node for LinkedList.

Illustration (Singly LinkedList)



Private and Public Variable, Objects

For example: LinkedList class

If **private** Node head;

Then if we want to add a new node as the first/head node, we can't do that. We can't do things such as *myList.head = new node* Instead, we need to creare ***public method*** like below:

```
Public void addFront( int data)  {
Node newNode = new Node(data);
newNode.next = head;
 head = newNode;
}
```

So when we have *variables in* **private mode**. You need to have a public **method** that is able to access that *variable* so users can work with that **variable** through the **public method** you provided.

Attention! Abstract Data Type

When we create a new ADT, a fundamental definition for new ADT. We don't want to write a method that allows our ADT to behave differently.

For example, **ADT Stack** is *Last In First Out*, we still can customize the ADT Stack and add methods **removeFront()** or **addFront()** but if so, we might break the rule of Stack and its fundamental definition, its behaviors. It can confuse users when they're working with ADT or Data Structures.

Why private variables ? Why public methods ?

For example: **if we have bankAccount class with Balance is private:**

```
public class BankAccount {
    private double balance; // Private variable

    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }
```

If we want to add $300 into the balance variable, we can't do it by having **user_01.balance = 300** directly. Instead, we need to do it by calling **user_01.deposit(300)**. The method **deposit()** will check any conditions (if have), verify steps such as **balance = balance + deposit amount** or it could be **balance += deposit amount**.

What if balance is public variable

```java
public class BankAccount {
    public double balance; // Not private

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

Then the user can access balance, cheating on it. They can do something such as **user_01.balance = 1,000,000,000.00**. The program will let users modify their *variable freedom* without going through any condition checking steps or verify steps. This is not security.

Similar to data structures such as LinkedList. If the **head** is *public*.
for example: if our list = [3] -> [4] -> [5] -> [6] -> [7]. **head** is 3.
Then when we set head = new Node(7, head)
explain code:
" A new node with value of 7 would have the next pointer to the current head node. Now the head variable (store reference to the first node object) has now referenced a node with a value of 7. "

Then **head** is now 7, before 7 such as *3, 4, 5, 6 are lost connection now*. In **LinkedList**, when there is v*ariable that has no pointer point to*, then **it is no longer accessible, similar** to **Remove()** in LinkedList which *cuts off connection to that variable* we don't want. So we did destroy the structure of LinkedList, we now lost 3,4,5,6. So be careful with public head;

Primitive and Non-Primitive Data Type (A general rule):

- **Primitive data types (e.g., int, double, char, etc.)** store their values directly in the variable. For example **int x = 5**, **x** *store value of 5 into itself.*

- **Non-primitive data types (<Objects>, <Integer>, <Double>, <Char>)** store references to the memory location where the object's data is stored. For example **Integer x = 5**, **x** is a variable that is used to *reference a memory location where the value int 5 is stored.* Special case is String. String is non-primitive data.

Why Add() & Offer(), Element() & Peek(), Remove() & Poll(() ?

So for **Offer(), Peek(), Poll()**. It's about showing users a status of the data structure they are working on rather than showing Restriction / Limited of that data structure such as **Add() , Element(), Remove().**

For example:
If the Array is full, **Offer()** will notify users that the Array is full already, no more space! We're so Sorry!

```
>>> Return null;
```

Restriction : (**Array[50]** is not allowed to have more than 50).
For **Add()**, it will throw exception errors and let users know that the Array limit has reached maximum and can't add anymore.

```
>>> Exception in thread "main" java.util.NoSuchElementException
    at java.base/java.util.LinkedList.getFirst(LinkedList.java:318)
    at java.base/java.util.LinkedList.element(LinkedList.java:727)
    at ElementExample.main(ElementExample.java:21)
```

Attention! ArrayList Occupied & Vacant

Occupied are cells that stored values, Vacants are cells that are still available for future value.

We need a size **variable** for ArrayList because for Vacant, there are zeros, so if our list contains value zero as the last element in the list. The next slot of it would be the first Vacant index (also zeros) so we can't distinguish which zero is the actual value which is zero of the Vacants cells.

|----------Occupied Cells----------||----------Vacant Cells----------|

25	46	38	35	0	0	0	0	0	0

Orange Box is the reason why we need to have **size** variable and **size++** everytime we add value so we can know how many values we are currently having.

When we add new value into our list, we can do following below:

```
elementData[size] = value;
```

The reason we do this is because the size number is equal to the first empty Vacant Cell index. So if we have a size of 5 then the index 5 would be the next empty slot to store future new values. (Index is starting from 0 so our values in the list would have positions 0, 1, 2, 3, 4)

Attention! size should be private and only be access through public int size() method.

Attention! final keyword:

This keyword makes the variable unchangeable so it is fixed.

static or non-static ?

- **Static method1()** is called by using class.method1(). For example Cat.method1()

- **Non-static method2()** is called by creating a class object.
  Ex: Cat cat1 = new Cat();
  cat1.method2();

static is for shared data from all instances of the class. For example , we have static catCount(). So it can count how many cat objects we created.

Calling the static method using the instance is possible but not recommended.