**Data structure defines how data is organized, managed, and stored. These data elements (data structures) provide an efficient way of storing and organizing data in the computer making possible utilizing them more efficiently.**

Java is one of the most popular programming languages. It is high-level and object-oriented, and a Java program can be run infinitely on any platform that supports Java because it follows the "Write Once Run Anywhere" (WORA) principle.

With that said, the data structure in Java can be done in many ways, and it can be confusing for beginners and seasoned software developers alike.

Since choosing the right data structure in Java is crucial to efficiency and speed, let's break down all of your options and when you might use what in this quick Java tutorial.

**Linear Data Structures**

The following data structures are all linear, which means elements are sorted and searched sequentially, one after the other. There is one first element, one last element, and every element in between has a "next" and "previous."

The downside is that linear data structures can become resource-intensive to search in large datasets.

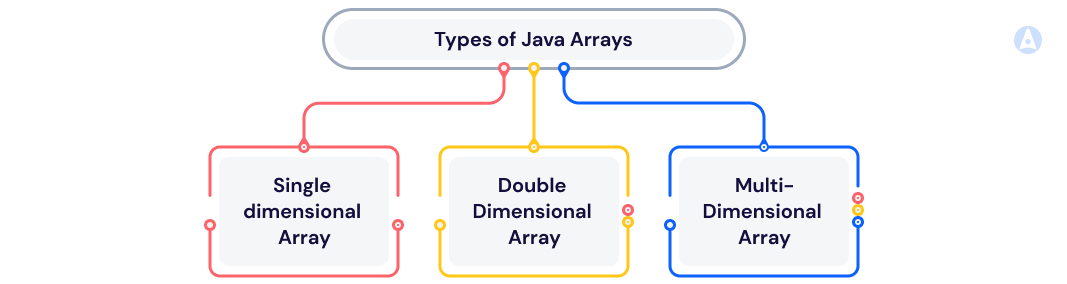
**Arrays**

[Arrays](https://www.adservio.fr/post/java-arraylist-vs-open-arraylist-performance) represent one of the simplest types of data structure in Java. An array is a linear data structure, or "list," where elements are stored.

With this data structure, you can access elements at random using the index. The array can also be sorted using techniques like quick sort or merge sort, but the biggest downside is that the size of the array is fixed.

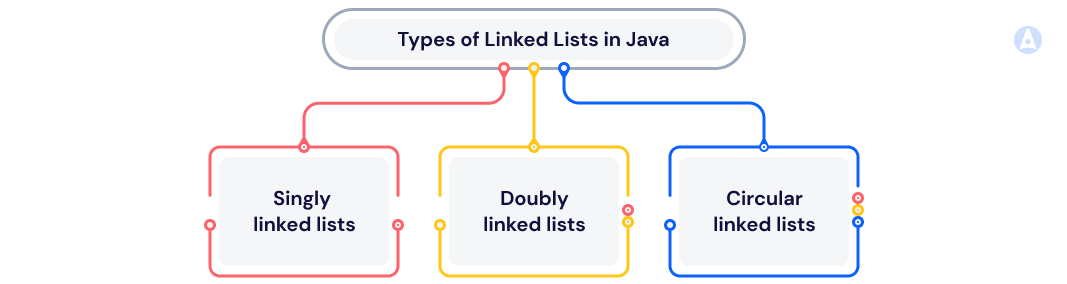
You may also avoid arrays because it's hard to add and delete elements once the array is made. Plus, a lot of the array is wasted if capacity and occupancy change over time.

Lastly, arrays store elements on contiguous memory, which requires special allocation. All in all, you might use arrays if you need to store things linearly and especially if you need to search the elements frequently.



**Linked List**

Another common data structure in Java is the [Linked List](https://www.geeksforgeeks.org/linked-list-in-java/), which is also linear. Unlike an array, a Linked List does not require contiguous memory, and it can use whatever memory is available.



Also, unlike an array, you cannot access elements at random, so you must work around the order of elements with this data structure.

However, it is dynamic in size, so it doesn't waste any capacity.

‍

**Singly Linked List**(Uni-Directional)



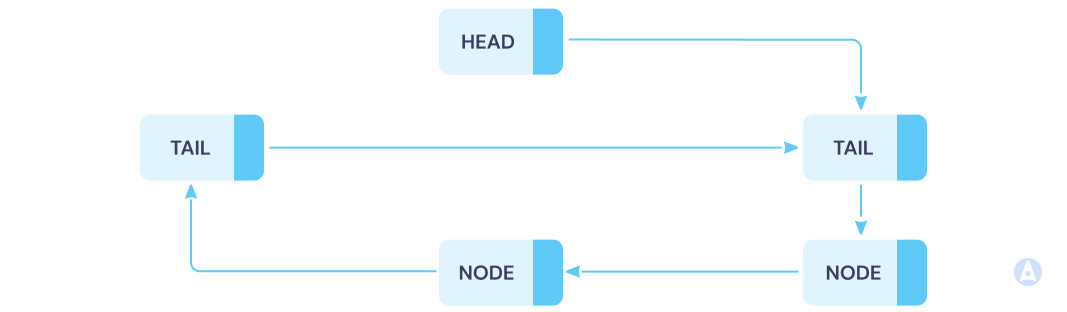
‍

**Doubly Linked List**(Bi-Directional)



‍

**Circular Linked List**



‍

You can easily insert and delete new elements as you need, and the memory allocation for a Linked List is much more efficient than with an array.

Just remember, you're giving up random access, which can cause problems. Lastly, the "head node" (generally the first element in the list) is the key, and you'll lose the entire list if you lose that.

**Stack**

Imagine that each element is literally being stacked one on top of another, and you understand the stack data structure. This structure follows a last in, first out (LIFO) approach, so the most recent element is at the top of the stack.

When working with a Java Stack, you can use the basic "push" and "pop" operations to move elements in and out, but you can also use search, empty, and peek to manage the elements.

**Queues**

Queues follow the first in, first out (FIFO) method of organization. Using the queue, elements are added to the end of the list and deleted from the start of the list.

The **PriorityQueue** and **LinkedList** are the most common classes used for queues.

Queues are best utilized for storing elements before processing, and the FIFO principle ensures that they are processed in a given order.

Depending on your needs, you may use a circular queue for better memory utilization or a deque (double-ended queue) where insertions and deletions can take place from both ends.

**Non-Linear Data Structures**

The following data structures are all non-linear, which means there is no single sequential path connecting all the elements.

Instead, multiple paths may connect elements, allowing Java to efficiently accommodate a greater variety of applications and representing development complexity.

**Binary Tree**

The [binary tree data structure](https://www.edureka.co/blog/java-binary-tree) in Java is not linear but rather, hierarchical. This means that the top-most element is the "root" of the tree, and all other nodes connect to it.

Each node in the tree can have up to two children. The best part about this data structure in Java is that you can access elements randomly.

The file system hierarchy of this structure allows for simple relationships among data to be easily represented. Plus, you can efficiently insert data and search the nodes with ease.

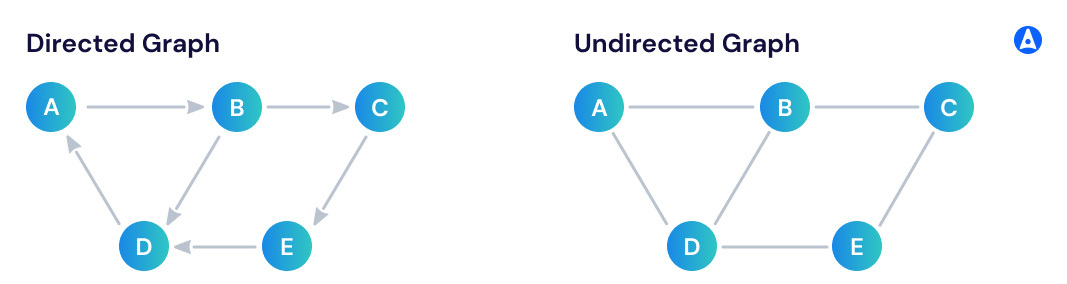
However, sorting data can be challenging, and it's quite inflexible. With that said, there are many variations of the binary tree, like the binary search tree.

**Graph**

The graph data structure in Java is easy to imagine because it is quite literally a graph created by a grouping of edges and vertices.

The graph is represented using V(G) for the vertices and E(G) for the edges, with the "G" standing for graph. Altogether, you end up with G(V, E) to summarize the graph.

Graphs are great because they can be disjointed or connected, directed or undirected, and so on. They're flexible and powerful when it comes to finding connections and identifying the shortest path, which means they're great for efficiency.



This structure is best reserved for medical science applications and situations where you're working on a very complex app, like LinkedIn or Facebook.

Circuit networks also do well with the graph data structure, but it's paramount that your team follows the best practices.

**Hash**

The [hash data structure](https://www.geeksforgeeks.org/hashing-in-java/)in Java uses the special hash function, which maps elements to a specific address where they are stored.

Variations include the **hashmap and HashSet**. The primary advantage of this data structure is that you get "constant-time" access. When collisions occur, two techniques—chaining and open addressing—are used to resolve them.

When fetching data using the hash data structure in Java, you will need to use the Hash function. The great news is that hash is fantastic for helping you fetch data quickly, and it's a highly efficient way to store elements.

However, with the need for collision resolution, there is a level of complexity created when working with the hash data structure in Java.

**Choosing the Right Data Structure in Java**

As you can see, there are a handful of Java data structures to choose from when creating a Java application.

What's more, almost all of the data structures listed here have many variations. For instance, things like queues and stacks can be created using an array or Linked List as the foundation.

When choosing a data structure in Java, follow these tips:

* If things like last in, first out (LIFO) versus first in, first out (FIFO) confuse you, take a step back. This will enable you to better understand Javascript before diving into things like priority queue, hash tables, abstract data types, and other elements you need to know to choose the best data structure.
* Compare all of your options. If you've never worked with some of these data structures before, do your homework, so you understand what each one does and how it functions.
* Think about your top priorities. Often, choosing a data structure means compromising between simplicity, flexibility, and accessibility. It's up to your team to decide what is most important.
* When a structure doesn't work, fix it. Adopting a data structure in Java only to realize you made the wrong decision is a terrible feeling, but trying to keep pushing forward when things aren't going right will only waste more time.