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:

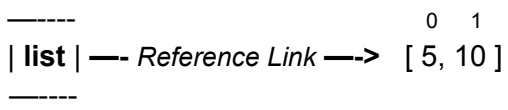
```
int x = 10;
```

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 ?

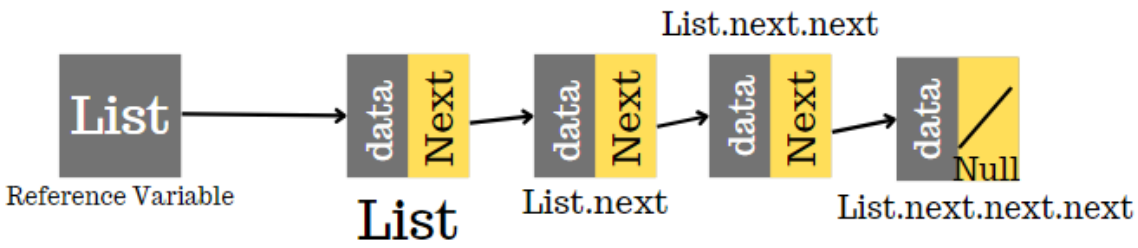
In most programming languages, variables declared within a certain block, function, or scope are only accessible within that scope. Once you exit that scope, the variable goes out of scope and is typically no longer accessible. It is effectively "wiped out" or removed from memory. It is no longer accessible,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

ArrayList Attention: Elements in Array/ArrayList would be shifted left or right when we **add** or **remove** elements. The important thing to remember is that the **Array that implements Queue** will not shift around because it's determined by Head and Tail Referemce.
So because of that, when we use Array with Queue, we will reuse Vacant Cells by having Circular Array so we won't waste Vacant spaces.

- When the **Queue** is full, how do I know ?
  check if **frontIndex** = **backIndex + 1**;

- For **Circular Array**, if it's full, when we element, we increment size of it (**backIndex)**, it will following the equation below:
  backIndex = (backIndex +1) % queue.length;

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.

Note! What is Instance ?

It's another word to call an object from a class.

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.

Iterator ? and Why ?

- Iterator is like a tool that we use to loop through ADT or any Data Structure that is more general, objects rather than number or something based on Index.
- For example , LinkedList, Deque, Tree, Hash, they are not going to have random access just like Array or ArrayList. So the only way to loop through Objects or ADT that don't have random access is by using Iterator.
- Iterators usually have an instance nextNode of Node type. nextNode is a pointer and can be seen as a current pointer. After it jumps to next element, i return data from the position it just left and set nextNode pointer = nextNode.next().

There are 2 popular methods inside the Iterator class. hasNext() to return true or false if next is empty or not. next() is to return data of current position of pointer and set pointer current position equal to next position of it nextNode = nextNode.next().

- We also have an iterator() -Be careful, this method is lower case "iterator", a method which returns a whole new Iterator class object, with nextNode set at the beginning place (brand new pointer).
- The other is getIterator(), it's similar to iterator() but instead of returning a new Iterator object, its return call method iterator().

Sorting (Algorithm)

Selection Sort

Array = {4, 3, 5, 6, 1}

- index i start at 0, stop when i < array.size() -1, i++
- MinIndex = i, and increasing as i. MinIndex is the position determined that min value will perform swap action within in turn.
- min is a variable store minimum value, if it found a smaller value, it set min = that value. But not yet actual swapping position, until looping through the whole list. After the looping of that turn is done, the final min value is the one that will be actually swapped position with the value at MintIndex position. So now the value of min will be swapped into the position of MintIndex and the value of Mindex will be swapped into the previous position of min.
    - **In order to swap, please create a temp variable to store data first.**

Selection Sort: starting with first position index = 0, so MinIndex = i so MinIndex = 0, now we have the int min variable = value at MinIndex. That means, min = 4.

So we got min = 4, then It loop through the list, it sees 4 > 3, then sets min = 3, and continues looping until the end of the list, it finds 3 > 1 then sets min now = 1. Now the loop has ended, and min = 1, so now it's time to actually swap the position of MinIndex = 0 with min position (at index 4). so now value 1 will be at index 0 and value 4 will be at index 4.

>>> Now Array = {1, 3, 5, 6, 4}

Next index++, index = 1, MinIndex = 1, mean min variable = value at index 1, it's 3. Then it starts looping through the end of the list and min = 3, so now there is no swap.

>>> Now Array = {1, 3, 5, 6, 4}

Next index++, MinIndex = 2, mean min = 5, looping through the list. It sees 5 > 4, so set min = 4. The list is ended, min = 4, so it actually does swapping positions of min = 4 and MinIndex = 5 (at index 2). Now value 4 will be at index 2 and value 5 will be at index 4.

>>> Now Array = {1, 3, 4, 6, 5}

Next index++, index = 3, MinIndex = 3, mean min variable = value at index 3, it's 6. Then it starts looping through the end of the list and sees 6 > 5, so set min = 5. The list is ended, min = 5. So now it actually does swapping positions of min = 5 (at index 4) and MinIndex = 6 (at index 3). Now value 5 will be at index 3 and value 6 will be at index 4.

>>> Now Array = {1, 3, 4, 5, 6}

Insertion Sort

Starting at index of 1, compare value at index 1 with value of previous index (index 0) **(LEFT).** If value at index 1 is smaller than value at index 0, swap position. Then starting at index 2, if the value of index 2 is already bigger than the value of index 1 then start the loop at index 3 and keep continuing until the array is sorted.

array = { 2,4, 1, 3, 5}

turn 1: start at index 1,value 4, Looping: 2 < 4, so do nothing.

>>> array = { 2,4, 1, 3, 5}

turn 2: start at index 2, value 1. Looping: 1 < 4, then swap. Now 1 < 2, another swap.

>>> array = { 2,1, 4, 3, 5}
>>> array = { 1,2, 4, 3, 5}

turn 3: start at index 3, value 3. Looping: 3 < 4, swap.

>>> array = { 1,2, 3, 4, 5}

turn 4: start at index 4, value 5 Looping:. Can't find any smaller values before (left side) of 5. So the array is sorted!!!

>>> array = { 1,2, 3, 4, 5}

MergeSort:

Divide array length into 2. Calling Recursion and split arrays until there is one element for each.
After that, merge them together. If left > right, copy elements from right into a sorted array and index of right++. If left < right, copy elements from left into sorted array and increase index of left ++. Use recursion to merge them together just like when we divide them.

! What we are trying to do is have another “empty array” which we can call it “sortedList” and then copy each element from the original array but in sorted positions.

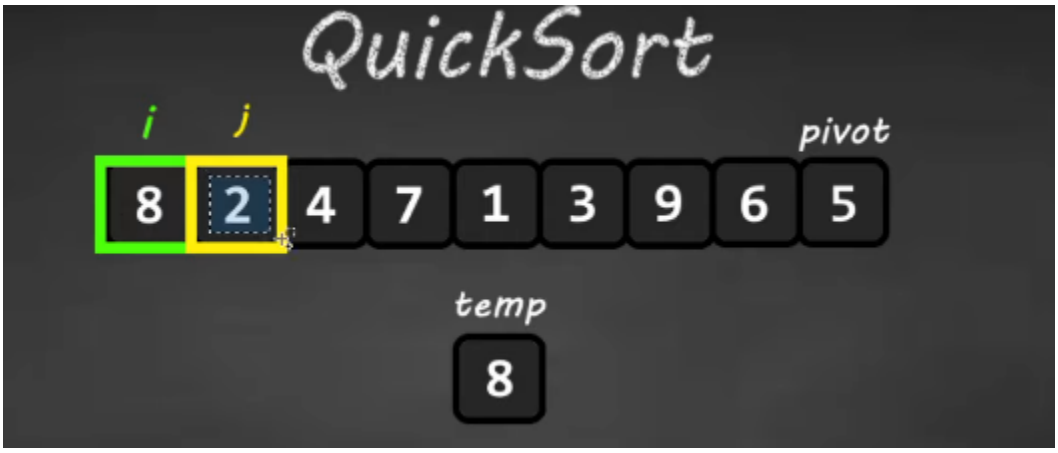
MergeSort:

QuickSort is divided and conquered just like MergeSort. This time, we can pick a pivot point. Pivot point divided array into 2 parts. The first step is making Pivot into this form below:

left side <= Pivot => right side

In this step, it's normal if the left and right side are not sorted yet.

Strategy of first step: we have i and j and temp variables. i is previous and j is current.



We start with j = first index and i = null;

If j > Pivot, ignore it. j++ only.

If j < Pivot, then increase i++. Then put the "i" value into temp. After that set i value = j value and then j value = temp (actually this is "i" value). Then increase j++.

Repeat the process until "j" reaches the Pivot point.

When "j" reaches the Pivot point. The new position of Pivot point is equal i++. Then we set temp value = that i + +. Then we set "i" = value of Pivot point and set "j" = temp ((actually this is "i" value).

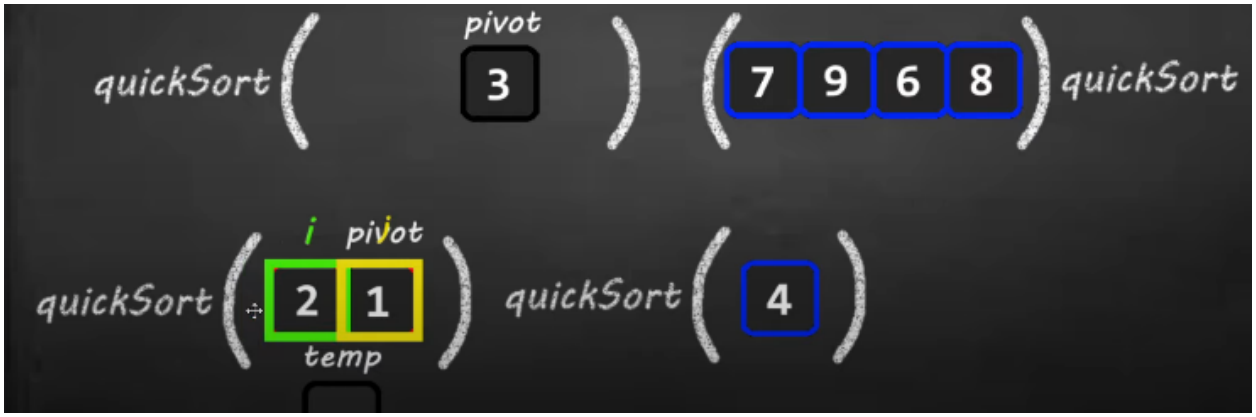
Completed first step and make array into this form left side <= Pivot => right side

Step 2: We create 2 partitions. One on the left and One on the right.

On the left side, it runs from the beginning of Array to Pivot point but not including Pivot Point.

On the right side, it runs from Pivot point but not including Pivot Point to the end of the array.

Completed 2nd step. Now we repeat those proccess with divided arrays of left side and right side. We do this by calling Recurvsion.



After sorting all elements inside smaller arrays in correct positions, we merge them together again.

Dummy Node ? or a “new” keyword: ?

- "What is a dummy node? A dummy node is a node that stores the reference link to a specific linked list. A dummy node doesn't store any value, just like the nodes inside the linked list. The actual node that contains data is the next node of the dummy node, which is commonly referred to as the 'head.' This node can be considered a representation of the linked list we are working on. Dummy nodes are often created inside custom Abstract Data Types (ADTs) we work on (Usually LinkedList).

- As for 'new,' it's used when we are working in another class and wish to create a new object or a new instance of an ADT or custom ADT object."

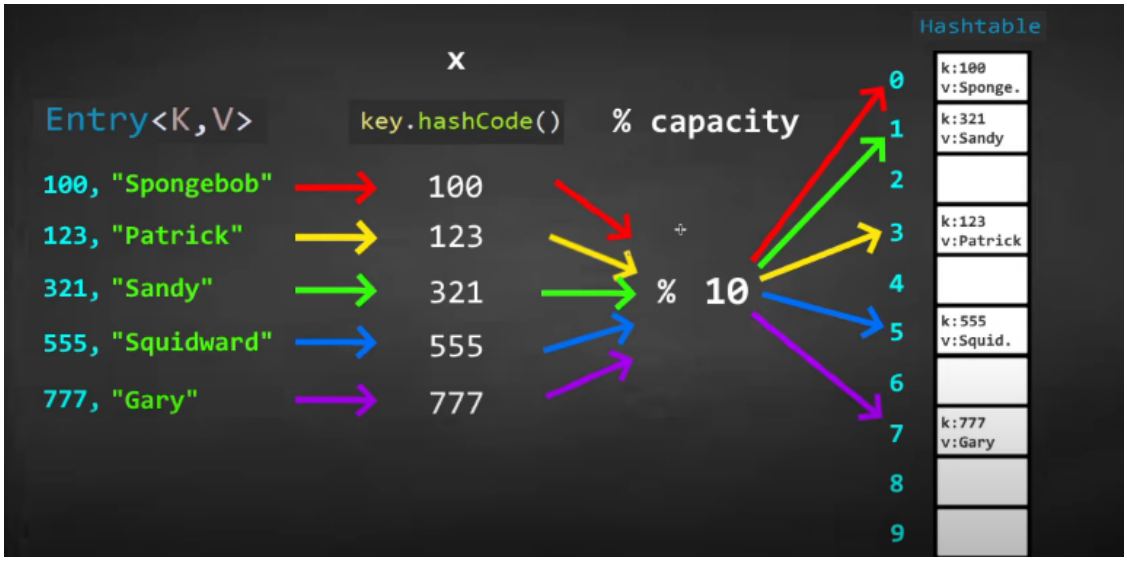
Primitive (char, int, boolean) ? or None-Primitive (Char, Integer, Boolean)?

- Primitive data type is used to store a flat value. For example variable int x = 10;
- Non Primitive data type is used to store an object or reference to an object such as Dog, Cat, LinkedList StockLedger, LinkedList BankAccountList.

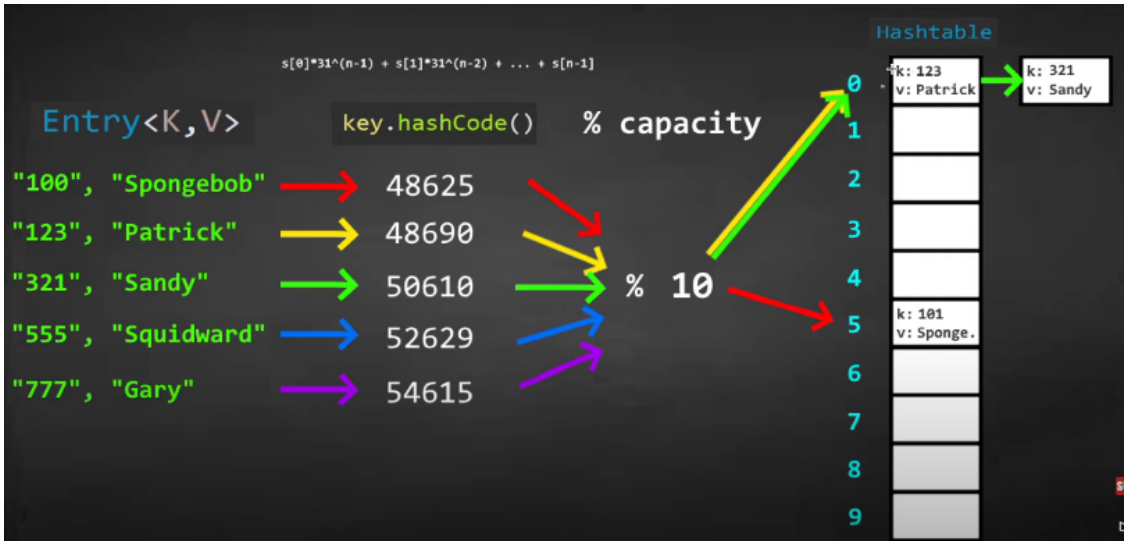
Dictionary:

HashTable: the table always has <k, v> represent for key=value. The standard mod(%) which is used to find index position is equal to array length. For example below, the array has 10 spaces. Then we use key(k) mod (%) length(n). For Spongebob value, it has the key “100”. The position would be 100%10 = 0. So 100=”Spongebob is at 0 index. If the position of another key has the same index, they call it Collision. One easy way to solve it is looping i + + until we see an empty space to put it in.

Attention! when we remove a key, we also remove the value.



Another way to solve Collision is to make that entry become a linkedlist and have the next node as different key-value pairs. So when we find it, it is going to loop the linkedlist of that entry to find the key and value we want. Just like the example of index 0, we can see index 0 has linkedlist with 2 nodes. This is called “Chaining”. The entry that has more than 1 key-value pairs at the same index is called “Bucket”.



When working with dictionaries or hash maps, here are some important key points and notes to keep in mind:

Key-Value Pair:

Dictionaries (or hash maps) store data in key-value pairs, where each key is associated with a specific value.

Keys:

Keys must be unique within the dictionary. Duplicate keys are not allowed.
Keys are used to retrieve values, so they should be chosen carefully for efficient look-up.

Hashing:

Dictionaries use a hashing function to convert keys into an index in the underlying data structure. This enables efficient retrieval of values based on their keys.

Efficiency:

Dictionaries provide fast look-up times, typically O(1), due to the use of hashing.

Hash Collisions:

Hash collisions occur when two different keys hash to the same index. This is resolved using techniques like separate chaining (linked lists at the same index) or open addressing (finding the next available index).

Data Retrieval:

To retrieve a value from a dictionary, use the key as an index, and the dictionary will return the associated value.

Adding and Removing Entries:

To add a key-value pair to the dictionary, use the put(key, value) method.
To remove a key-value pair, use the remove(key) method.

Data Structure:

Dictionaries are typically implemented as hash tables or similar data structures.

Key Immutability:

In many programming languages, keys are required to be immutable (unchangeable) to ensure consistent hashing.

Iteration:

You can iterate through the keys or values in a dictionary, depending on the programming language. This allows you to process all entries in the dictionary.

Handling Missing Keys:

Be prepared to handle cases where the key you're looking for is not present in the dictionary. Some languages offer methods like getOrDefault(key, defaultValue) to handle this.

Memory and Space:

Dictionaries can consume memory, so consider their size and the trade-off between memory usage and lookup speed.

Use Cases:

Dictionaries are useful for tasks like caching, storing configuration settings, implementing data structures (e.g., sets or graphs), and more.

Collisions and Load Factor:

Be aware of load factors that dictate when a dictionary should be resized to accommodate more entries efficiently.

Serialization:

Some languages provide mechanisms to serialize (convert to a stream of bytes) and deserialize (reconstruct from bytes) dictionaries for storage or network transfer.

Thread Safety:

If used in a multithreaded environment, consider synchronization to prevent data corruption.

Order (Unordered vs. Ordered):

Some dictionaries maintain the order of key-value pairs, while others do not. Be aware of the ordering behavior of the dictionary you're using.

Performance Trade-offs:

Consider the trade-offs between memory consumption, insertion speed, and lookup speed when choosing a dictionary implementation.