ArrayList

ArrayList: The ArrayLists adjusts its size automatically when an element is added or removed. Hence, it is also known as a Dynamic Array.

ArrayList<Type> arrList = new ArrayList<>();

Adding Primitive Data Types: ArrayList can only store objects. To use primitive data types, we have to convert them to objects. In Java, Wrapper Classes are can be used to convert primitive types (int, char, float, etc) into corresponding objects.

Autoboxing: The conversion of primitive types into their corresponding wrapper class objects is called Autoboxing. Unboxing: The conversion of wrapper class objects into their corresponding primitive types is called Unboxing.

Method	Syntax	Usage
add()	arrList.add(index, element);	used to add a single element to the ArrayList.
get()	arrList.get(index);	used to access an element from an ArrayList.
set()	arrList.set(index, element);	used to replace or modify an element in the ArrayList.
remove(index)	arrList.remove(index);	removes the element at the specified position, i.e index, in the ArrayList.
remove(object)	arrayList.remove(obj);	removes the first occurrence of the specified element from the ArrayList if it is present. Remains unchanged if not present
clear()	arrList.clear()	It completely removes all of the elements from the ArrayList.
size()	arrList.size()	used to find the size of an ArrayList.
indexOf()	arrList.indexOf(obj);	returns the index of the first occurrence of the specified element in the ArrayList. Returns -1 if not present

Iterating over an ArrayList: Similar to iterating over an array, we can use the loops to iterate over an ArrayList.

```
ArrayList<String> players = new ArrayList<>();

players.add(0, "Bryant");

players.add("Wade");

for (String name : players)
```

```
    System.out.println(name);
}
// Output is:
Bryant
Wade
```

ArrayList Concatenation: The addAll() method is used to concatenate two ArrayLists. This method appends the second ArrayList to the end of the first ArrayList.

```
ArrayList<Integer> arrList1 = new ArrayList<>();
arrList1.add(5);
arrList1.add(10);
ArrayList<Integer> arrList2 = new ArrayList<>();
arrList2.add(25);
arrList2.add(30);
arrList1.addAll(arrList2);
System.out.println(arrList1);
// Output is:
[5, 10, 25, 30]
```

ArrayList Slicing: The subList() method is used for slicing of ArrayLists. It works similar to the copyOfRange() method in Arrays.

```
ArrayList<Integer> arrList = new ArrayList<>();
arrList.add(5);
arrList.add(10);
arrList.add(15);
arrList.add(20);
ArrayList<Integer> subArrList = new ArrayList<>(arrList.subList(1, 3));
System.out.println(subArrList);
// Output is:
[10, 15]
```

Conversion between Arrays and ArrayLists: Arrays.asList() method is used to convert Array to ArrayList

```
Arrays.asList(arr);
```

toArray() method of ArrayList is used to convert ArrayList into an Array

arrList.toArray(arr);

Frequency of an Element: The Collections.frequency() method is used to find the frequency with which an element occurs in the given ArrayList.

Integer[] arr = {3, 6, 2, 1, 2};

ArrayList<Integer> arrList= new ArrayList<>(Arrays.asList(arr));

int frequency = Collections.frequency(arrList, (Integer)2);

System.out.println(frequency); *// 2*

Reversing ArrayLists: We can reverse an ArrayList by using the Collections.reverse() method.

Interger[] arr = {1, 2, 3, 4};

ArrayList<Integer> arrList = new ArrayList<>(Arrays.asList(arr));

Collections.reverse(arrList);

System.out.println(arrList); *// [4, 3, 2, 1]*

Sorting an ArrayList: The Collections.sort() method can be used to sort the given ArrayList in two different ways

- **Ascending order**

Integer[] arr = {3, 6, 2, 1};

ArrayList<Integer> arrList= new ArrayList<>(Arrays.asList(arr));

Collections.sort(arrList);

System.out.println(arrList); *// [1, 2, 3, 6]*

- **Descending order: An ArrayList can be sorted in descending order by passing the argument Collection.reverseOrder() to the Collections.sort() method.**

Integer[] arr = {3, 6, 2, 1};

ArrayList<Integer> arrList= new ArrayList<>(Arrays.asList(arr));

Collections.sort(arrList, Collections.reverseOrder());

System.out.println(arrList); *// [6, 3, 2, 1]*

HashSet

HashSet: The HashSet is also an unordered collection of elements. The HashSet stores only unique elements and duplicate elements are not allowed.

HashSet<Type> hset = new HashSet<>();

Method	Syntax	Usage
add()	hset.add(element);	to add a single element to the HashSet.

Method	Syntax	Usage
remove()	hset.remove(element);	removes an element from the HashSet.
clear()	hset.clear()	removes all the elements from a HashSet.
contains()	hset.contains(element);	checks if an element is present in a given HashSet.
size()	hset.size()	used to find the size of a HashSet.

Iterating Over a HashSet: Similar to iterating over an Array or ArrayList, we can use the for-each loop to iterate over a HashSet.

```
HashSet<String> players = new HashSet<>();

players.add("Rahul");

players.add("Rohit");

players.add("Virat");

players.add("Sachin");

for (String name : players)

   System.out.println(name);

// Output is:

Rohit

Rahul

Virat

Sachin
```

HashSet Operations

Union: The addAll() method can be used to perform the union of two sets. Syntax: hset1.addAll(hset2);

```
HashSet<Integer> hset1 = new HashSet<>();

HashSet<Integer> hset2 = new HashSet<>();

hset1.add(3);

hset1.add(32);

hset1.add(8);

hset2.add(8);

hset2.add(32);

hset2.add(30);
```

hset1.addAll(hset2);

System.out.println(hset1);

// Output is:

on: [32, 3, 8, 30]

Intersection: The retainAll() method can be used to perform the intersection of two sets. Syntax: hset1.retainAll(hset2);

HashSet<Integer> hset1 = new HashSet<>();

HashSet<Integer> hset2 = new HashSet<>();

hset1.add(3);

hset1.add(32);

hset1.add(8);

hset2.add(8);

hset2.add(32);

hset2.add(30);

hset1.retainAll(hset2);

System.out.println(hset1);

// Output is:

[32, 8]

Difference: The removeAll() method can be used to find the difference between two sets. Syntax: hset1.removeAll(hset2);

HashSet<Integer> hset1 = new HashSet<>();

HashSet<Integer> hset2 = new HashSet<>();

hset1.add(3);

hset1.add(32);

hset1.add(8);

hset2.add(8);

hset2.add(24);

hset2.add(30);

hset1.removeAll(hset2);

System.out.println(hset1);

//Output is:

[32, 3]

SuperSet: A superset of any given set is defined as the set which contains all the elements present in the given set. The containsAll() can be used to check if the given set is the superset of any other set.

hset1.containsAll(hset2);

Here, containsAll() checks if all the elements in hset2 are present in hset1, i.e, it checks if hset1 is a superset of hset2.

Subset: A subset of any given set is defined as the set which contains atleast one element present in the given set. The containsAll() can also be used to check if the given set is the subset of any other set.

hset2.containsAll(hset1);

Here, containsAll() checks if all the elements in hset1 are present in hset2, i.e, it checks if hset1 is a subset of hset2.

Converting to ArrayList: Conversion of HashSet to an ArrayList is done by passing the HashSet as an argument to the constructor of ArrayList.

ArrayList<Type> arrList = new ArrayList<>(hset);

HashMap

HashMap: The HashMap is also an unordered collection of elements. HashMap stores the data in key/value pairs. Here, keys should be unique and a value is mapped with the key. HashMap can have duplicate values.

HashMap<KeyType, ValueType> hmap = new HashMap<>();

Method	Syntax	Usage
put()	hmap.put(key, value);	used to add/update an element to the HashMap.
get()	hmap.get(key);	used to access the value mapped with a specified key in a HashMap.
replace()	hmap.replace(key, newValue);	replaces the old value of the specified key with the new value.
remove()	hmap.remove(key);	used to remove an element from a HashMap.
clear()	hmap.clear();	to remove all the elements from a HashMap.
keySet()	hmap.keySet();	returns a HashSet of all the keys of a HashMap.
values()	hmap.values();	to get all the values mapped to the keys in a HashMap.
entrySet()	hmap.entrySet();	used to get the elements of a HashMap.

Method	Syntax	Usage
size()	hmap.size();	used to find the size of a HashMap.
containsKey()	hmap.containsKey(key);	It returns true if the HashMap contains the specified key. Otherwise false is returned.
containsValue()	hmap.containsValue(value);	It returns true if the HashMap contains the specified value. Otherwise false is returned.
putAll()	hmap2.putAll(hmap1);	used to copy all the elements from a HashMap to another HashMap.

Iterating a HashMap

```
HashMap<String, Integer> playersScore = new HashMap<>();
playersScore.put("Robert", 145);
playersScore.put("James", 121);
playersScore.put("Antony", 136);
playersScore.put("John", 78);
for (Map.Entry<String, Integer> entry : playersScore.entrySet())
   System.out.printf("%s:%d\n", entry.getKey(), entry.getValue());
// Output is:
James:121
Robert:145
John:78
Antony:136
```

Math Methods

Methods	Description
pow()	It calculates the exponents and returns the result.
round()	It rounds the specified value to the closest int or long value and returns it.
min()	Returns the numerically lesser number between the two given numbers.
max()	Returns the numerically greater number between the two given numbers.
abs()	Returns the absolute value of the given number

Type Conversions

Type Conversion: Type conversion is a process of converting the value of one data type (int, char, float, etc.) to another data type. Java provides two types of type conversion :

1. **Implicit Type Conversion**
2. **Explicit Type Conversion (Type Casting)**

Implicit Type Conversion: Java compiler automatically converts one data type to another data type. This process is called Implicit type conversion.

```
int value1 = 10;

float value2 = value1;

System.out.println(value1); // 10

System.out.println(value2); // 10.0
```

Explicit Type Conversion: In Explicit Type Conversion, programmers change the data type of a value to the desired data type. This type of conversion is also called Type Casting since the programmer changes the data type.

```
float x = 10.0f;

System.out.println((int)x); // 10
```

Type Conversion using Methods

Converting any Primitive Type to String: Any data type can be converted to String using the string method valueOf().

```
int num = 2;

float floatNum = 2.34f;

char ch = 'a';

System.out.println(String.valueOf(num)); // 2

System.out.println(String.valueOf(floatNum)); // 2.34

System.out.println(String.valueOf(ch)); // a
```

We may also use the toString() method of the corresponding wrapper class to convert primitive data types to String. Example 1:

```
int a = 10;

String str = Integer.toString(a);

System.out.println(str); // 10
```

Example 2:

char a = 'A';

String str = Character.toString(a);

System.out.println(str); *// A*

Converting String to any Primitive Type: We can convert String to int in Java using Integer.parseInt() method.

String str = "21";

int num = Integer.parseInt(str);

System.out.println(num); *// 21*

Similarly, for other primitive data types as given the table below,

Primitive Data Type	Syntax
byte	Byte.parseByte()
short	Short.parseShort()
int	Integer.parseInt()
long	Long.parseLong()
float	Float.parseFloat()
double	Double.parseDouble()
boolean	Boolean.parseBoolean()

Converting char to int: We can convert char to int in Java using Character.getNumericValue() method.

char ch = '3';

int num = Character.getNumericValue(ch);

System.out.println(num); *// 3*

Converting int to char: We can convert int to char in Java using Character.forDigit() method.

int num = 3;

char ch = Character.forDigit(num, 10);

System.out.println(ch); *// 3*

Getting Unicode Value of a Character: Using Explicit Type Conversion, we convert char to unicode value of type int.

char ch = 'A';

System.out.println((int)ch); ***// 65***

Getting Character Representation of a Unicode Value: we have to explicitly typecast the int value to char.

int unicodeValue = 65;

char ch = (char)unicodeValue;

System.out.println(ch); ***// A***